Computer Science at Kent

Implementation and Application of Functional Languages
19th International Symposium, IFL 2007

Olaf Chitil (Ed.)

Freiburg, Germany, 27th-29th September 2007

Technical Report No. 12-07
September 2007

Published by the Computing Laboratory,
University of Kent, Canterbury, Kent, CT2 7NF, UK

Preface

The 19th International Symposium on Implementation and Application of Functional Languages (IFL 2007) is held at Freiburg, Germany, on the 27th to the 29th September 2007. Local organiser is the Programming Languages Group of the Department of Computer Science of the University of Freiburg.

IFL brings together researchers active in the area of functional programming, with an emphasis on the implementation and application of the same. IFL provides an annual open forum for researchers who wish to present and discuss new ideas and concepts, work in progress, preliminary results, etc. IFL has been held throughout Europe in the Netherlands, United Kingdom, Germany, Sweden, Spain, Ireland and Hungary. This year for the first time IFL is co-located with the International Conference on Functional Programming (ICFP). A record number of 44 papers have been submitted for these draft proceedings. By the time of printing 73 researchers had registered for attendance at the symposium.

Following tradition, two proceedings are to be published: the draft proceedings used at the symposium (this document), released as a technical report of the Computing Laboratory of the University of Kent, and the post-symposium proceedings based on revised papers. The draft proceedings are un-refereed and provide a useful reference to the delegates at the symposium. All participants who give talks at the symposium are invited to submit revised papers for review after the symposium, to normal conference standards. The post-symposium proceedings of selected revised papers will be published by Springer-Verlag in its Lecture Notes in Computer Science (LNCS) series.

Olaf Chitil
Programme Chair
University of Kent
September 2007

Local Organisers

Markus Degen
Peter Thiemann
Stefan Wehr

Supported by Deutsche Forschungsgemeinschaft (DFG)

Table of Contents

Index

Termination and complexity bounds for SAFE programs*

Salvador Lucas
Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
slucas@dsic.upv.es

Ricardo Peña
Sistemas Informáticos y Computación
Universidad Complutense de Madrid
ricardo@sip.ucm.es

Abstract. Safe is a first-order eager functional language with facilities for programmer-controlled destruction and copying of data structures and is intended for compile-time analysis of memory consumption. In Safe, heap and stack memory consumption depends on the length of recursive calls chains. Ensuring termination of Safe programs (or of particular function calls) is therefore essential to implement these features. Furthermore, being able to giving bounds to the chain length required by such terminating calls becomes essential in computing space bounds. In this paper, we investigate how to analyze termination of Safe programs by using standard term rewriting techniques, i.e., by transforming Safe programs into term rewriting systems whose termination can be automatically analyzed by means of existing tools. Furthermore, we investigate how to use proofs of termination which combine the dependency pairs approach together with polynomial interpretations to obtain suitable bounds to the length of chains of recursive calls in Safe programs.

1 Introduction

Safe [23, 25, 21] is a first-order eager functional language with facilities for programmer controlled destruction and copying of data structures and is intended for compile time analysis of memory consumption. In Safe, the allocation and deallocation of compiler-defined memory regions for data structures is associated to function application. So, heap memory consumption depends on the length of recursive calls chains. In order to compute space bounds for such chains it is essential to compute bounds to their lengths and, in turn, to previously ensure termination of such functions.

In this paper we investigate how to use rewriting techniques for proving termination of Safe programs and giving appropriate bounds to the number of recursive calls of Safe programs as a first step to compute space bounds. In particular, we introduce a transformation for proving termination of Safe programs by translating them into Term Rewriting Systems.

* Work partially supported by the EU (FEDER) and the Spanish MEC, under grant TIN 2004-7943-C04. Salvador Lucas was partially supported by the EU (FEDER) and the Spanish MEC grant HA 2006-0007, and by the Generalitat Valenciana under grant GV06/285. Ricardo Peña was partially supported by the Madrid Region Government under grant S-0505/TIC/0407 (PROMESAS).

Both termination and complexity bounds of programs have been investigated in the abstract framework of Term Rewriting Systems (TRSs [3, 22]). A suitable way to prove termination of programs written in declarative programming languages like Haskell or Maude [8] is translating them into (variants of) term rewriting systems and then using techniques and tools for proving termination of rewriting. See [11, 13] for recent proposals of concrete procedures and tools which apply to the aforementioned programming languages.

Polynomial interpretations have been extensively investigated as suitable tools to address different issues in term rewriting [3]. For instance, the limits of polynomial interpretations regarding their ability to prove termination of rewrite systems were first investigated in [15] by considering the *derivational complexity* of polynomially terminating TRSs, i.e., the upper bound of the lengths of arbitrary (but finite) derivations issued from a given term (of size n) in a terminating TRS. Hofbauer has shown that the derivational complexity of a terminating TRS can be better approximated if polynomial interpretations over the reals (instead of the more traditional polynomial interpretations over the naturals) are used to prove termination of the TRS [14].

Complexity analysis of first order functional programs (or TRSs) has also been successfully addressed by using polynomial interpretations [4–7]. The aim of these papers is to classify TRSs in different (TIME or SPACE) complexity classes according to the (least) kind of polynomial interpretation which is (weakly) compatible with the TRS. Recent approaches combine the use of *path orderings* [10] to ensure both termination together with suitable polynomial interpretations for giving bounds to the length of the rewrite sequences (which are known finite due to the termination proof), see [6]. Polynomials which are used in this setting are *weakly monotone*, i.e., if $x \geq y$ then $P(\ldots, x, \ldots) \geq P(\ldots, y, \ldots)$. This is in contrast with the use of polynomials in proofs of polynomial termination [18], where *monotony* is required (i.e., whenever $x > y$, we have $P(\ldots, x, \ldots) > P(\ldots, y, \ldots)$). However, when using polynomials in proofs of termination using the dependency pair approach [1], monotony is not longer necessary and we can use weakly monotone polynomials again [9, 20]. The real advantage is that, we can now avoid the use of path orderings to ensure termination: with the same polynomial intepretation we can both prove termination and as we show in this paper, obtain suitable complexity bounds. Furthermore, since the limits of using path orderings to prove termination of rewrite systems are well-known, and they obviously restrict the variety of programs they can deal with, we are able to improve on the current techniques.

2 Preliminaries

A binary relation R on a set A is *terminating* (or well-founded) if there is no infinite sequence $a_1\ R\ a_2\ R\ a_3 \cdots$. Given $f : A^k \to A$ and $i \in \{1, \ldots, k\}$, we say that R is monotonic on the i-th argument of f (or that f is i-monotone regarding R) if $f(x_1, \ldots, x, \ldots, x_k)\ R\ f(x_1, \ldots, y, \ldots, x_k)$ whenever $x\ R\ y$, for all $x, y, x_1, \ldots, x_k \in A$. We say that R is *monotonic* regarding f (or that f is R-monotone) if R is i-monotonic on the i-th argument of f for all i, $1 \leq i \leq k$.

A transitive and reflexive relation $\succeq$ on A is a quasi-ordering. A transitive and irreflexive relation $>$ on A is an ordering.

Terms and term rewriting Throughout the paper, $\mathcal{X}$ denotes a countable set of variables and $\mathcal{F}$ denotes a signature, i.e., a set of function symbols $\{f, g, \ldots\}$, each having a fixed arity given by a mapping $ar : \mathcal{F} \to \mathbb{N}$. The set of terms built from $\mathcal{F}$ and $\mathcal{X}$ is $\mathcal{T}(\mathcal{F}, \mathcal{X})$. A *context* is a term $C[\,]$ with a 'hole' (formally, a fresh constant symbol). A rewrite rule is an ordered pair (l, r), written $l \to r$, with $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $l \notin \mathcal{X}$ and $Var(r) \subseteq Var(l)$. A TRS is a pair $\mathcal{R} = (\mathcal{F}, R)$ where R is a set of rewrite rules. Given $\mathcal{R} = (\mathcal{F}, R)$, we consider $\mathcal{F}$ as the disjoint union $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ of symbols $c \in \mathcal{C}$, called *constructors* and symbols $f \in \mathcal{D}$, called *defined functions*, where $\mathcal{D} = \{root(l) \mid l \to r \in R\}$ and $\mathcal{C} = \mathcal{F} - \mathcal{D}$. Given a TRS $\mathcal{R}$, a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ rewrites to s, written $t \to_{\mathcal{R}} s$, if $t = C[\sigma(l)]$ and $s = C[\sigma(r)]$, for some rule $\rho : l \to r \in R$, context $C[\,]$ and substitution σ. A TRS $\mathcal{R}$ is terminating if $\to_{\mathcal{R}}$ is terminating.

A *conditional, oriented* TRS (CTRS), has rules of the form $l \to r \Leftarrow C$, where $C = s_1 \to t_1, \ldots, s_k \to t_k$ is called an oriented condition. A 3-CTRS satisfies $var(r) \subseteq var(l) \cup var(C)$ for every conditional rule. It is deterministic if the variables of the righthand side t_i of every condition $s_i \to t_i$ of C are introduced before they are used in the lefthand side s_j of a subsequent condition $s_j \to t_j$.

Term orderings and algebraic interpretations A term ordering is a pair $(\succeq, >)$ of relations over the set $\mathcal{T}(\mathcal{F}, \mathcal{X})$ of terms over the signature $\mathcal{F}$ and variables $\mathcal{X}$. The relation $\succeq$ is a quasi-ordering and $>$ is an ordering and they satisfy $> \circ \succeq \subseteq >$ or $\succeq \circ > \subseteq >$. The relation R is *stable* if, whenever $t\, R\, s$, we have $\sigma(t)\, R\, \sigma(s)$ for all terms t, s and substitutions σ. A term ordering $(\succeq, >)$ is said to be (1) well-founded if $>$ is well-founded; (2) stable if both $\succeq$ and $>$ are stable; (3) weakly monotonic if $\succeq$ is monotonic; and (4) strictly monotonic if both $\succeq$ and $>$ are monotonic. A *reduction pair* is a well-founded, stable, and weakly monotonic term ordering.

Term orderings can be obtained by giving appropriate *interpretations* to the function symbols of a signature. Given a signature $\mathcal{F}$, an $\mathcal{F}$-algebra is a pair $\mathcal{A} = (A, \mathcal{F}_A)$, where A is a set and $\mathcal{F}_A$ is a set of mappings $f_A : A^k \to A$ for each $f \in \mathcal{F}$ where $k = ar(f)$. For a given valuation mapping $\alpha : \mathcal{X} \to A$, the evaluation mapping $[\alpha] : \mathcal{T}(\mathcal{F}, \mathcal{X}) \to A$ is inductively defined by $[\alpha](x) = \alpha(x)$ if $x \in \mathcal{X}$ and $[\alpha](f(t_1, \ldots, t_k)) = f_A([\alpha](t_1), \ldots, [\alpha](t_k))$ for $x \in \mathcal{X}$, $f \in \mathcal{F}$, $t_1, \ldots, t_k \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. Given a term t with $\mathcal{V}ar(t) = \{x_1, \ldots, x_n\}$, we write $[t]$ to denote the *function* $F_t : A^n \to A$ given by $F_t(a_1, \ldots, a_n) = [\alpha_{(a_1,\ldots,a_n)}](t)$ for each tuple $(a_1, \ldots, a_n) \in A^n$, where $\alpha_{(a_1,\ldots,a_n)}(x_i) = a_i$ for $1 \leq i \leq n$.

A *quasi-ordered* $\mathcal{F}$-algebra, is a triple $(A, \mathcal{F}_A, \succeq_A)$, where $(A, \mathcal{F}_A)$ is an $\mathcal{F}$-algebra and $\succeq_A$ is a quasi-ordering on A. Then, we can define a stable quasi-ordering $\succeq$ on terms given by $t \succeq s$ if and only if $[\alpha](t) \succeq_A [\alpha](s)$, for all $\alpha : \mathcal{X} \to A$. We can also define a stable ordering $>$ on terms by $t > s$ if $[\alpha](t) \succ_A [\alpha](s)$, for all $\alpha : \mathcal{X} \to A$, where $\succ_A$ is the strict part of $\succeq_A$, i.e., $\succ_A = \succeq_A - \preceq_A$.

3 The SAFE language

Safe was first introduced in [23] as a research platform to investigate analyses related to sharing of data structures and to memory consumption. Currently it is provided with a type system guaranteeing that all well-typed programs will be free of dangling pointers at runtime, in spite of the memory destruction facilities provided by the language. More information can be found in [25] and [21].

There are two versions of Safe: full-Safe, in which programmers are supposed to write their programs, and Core-Safe (the compiler transformed version of full-Safe), in which all program analyses are defined.

Full-Safe syntax is close to Haskell's. The main difference is that Safe is eager and (at this moment) first-order. Safe admits two basic types (*booleans* and *integers*), algebraic datatypes (which are introduced by means of the usual **data** declarations), and the definition of functions by means of conditional equations with the usual facilitites for pattern matching, use of **let** and **case** expressions, and **where** clauses. No recursion is possible inside **let** expressions and **where** clauses and no local function definition can be given.

A Safe program consists of a sequence of (possibly recursive) function definitions together with a main expression. The only free variables allowed in function's bodies are function's formal arguments and the names of the previously defined functions, and the only free variables allowed in the main expression are the names of the defined functions.

Additionally, the programmer can specify a *destructive* pattern matching operation by using symbol ! after the pattern. The intended meaning of this operator is the destruction of the cell which is associated to the constructor symbol thus allowing its reuse later.

The merge-sort program of Figure 1 uses a constant heap space to implement the sorting of the list. This is a consequence of the destructive constant-space versions *splitD* and *mergeD* of the funtions which respectively split a list into two pieces and merge two sorted lists. Types which are shown in the program above are inferred by the compiler. A symbol ! in a type signature indicates that the corresponding data structure is destroyed by the function. Variables ρ are polymorphic and indicate the region where the data structure 'lives'.

3.1 Core-Safe syntax

The Safe compiler first performs a *region inference* which determines which region have to be used for each construction. A function has at most two associated memory regions: a *working* region which can be addressed by using the reserved identifier *self* and an *output* region which is passed as a parameter. For this reason, the low-level syntax, called Core-Safe requires an additional parameter r both in some function calls and in expressions such as $(C\ \overline{x_i}^n)@r$ which denotes a construction, and $x@r$ which denotes the *copy* of the structure with root labeled x into a region r. The compiler also *flattens* the expressions in such a way that the application of functions is made on constants or variables only. Also, **where** clauses are translated into **let** expressions, and boolean conditions

$$
\begin{array}{l}
splitD :: \forall a, \rho.Int \to [a]!@\rho \to \rho \to ([a]@\rho, [a]@\rho)@\rho \\
splitD\ 0\ xs! \qquad\quad = ([\,], xs!) \\
splitD\ n\ [\,]! \qquad\quad = ([\,], [\,]) \\
splitD\ n\ (x : xs)! = (x : xs_1, xs_2) \\
\quad \textbf{where}\ (xs_1, xs_2) = splitD\ (n-1)\ xs \\
mergeD :: \forall a, \rho.[a]!@\rho \to [a]!@\rho \to \rho \to [a]@\rho \\
mergeD\ [\,]!\ ys!\ = ys! \\
mergeD\ xs!\ [\,]!\ = xs! \\
mergeD\ (x : xs)!\ (y : ys)! \\
\quad \mid x \leq y \quad\ = x : mergeD\ xs\ (y : ys!) \\
\quad \mid otherwise = y : mergeD\ (x : xs!)\ ys \\
msortD :: \forall a, \rho.[a]!@\rho \to \rho \to [a]@\rho \\
msortD\ xs \\
\ \mid n \leq 1 \quad\ = xs! \\
\ \mid otherwise = mergeD\ (msortD\ xs_1)\ (msortD\ xs_2) \\
\quad \textbf{where}\ \ (xs_1, xs_2) = splitD\ (n\ `div`\ 2)\ \ xs \\
\qquad\qquad\ n \qquad\quad\ = length\ xs
\end{array}
$$

Fig. 1. Mergesort program in full-SAFE

in the guards are translated into **case** expressions. Bound variables are also conveniently renamed to avoid name clashes.

The syntax of Core-Safe is shown in Figure 2. We use the notation $\overline{x_i}^n$ to abbreviate the sequence $x_1 \ldots x_n$. Note that constructions can only occur on *binding expressions be* inside **let** expressions. The normal form of an expression is either a basic constant c, or a pointer p to a construction. We assume the existence of a heap which keeps track of the correspondence between pointers and constructions. The complete operational semantics can be found in [25].

Function *splitD* defined in the Safe program above is translated into the following Core-Safe definition:

```
splitD n xs r = case n of
                0 -> ([]@r,xs!)@r
                _ -> case! xs of
                     []     -> ([]@r,[]@r)@r
                     : x xx -> let z = let n' = n-1 in splitD n' xx @r in
                               let xs1 = case z of (ys1,ys2) -> ys1 in
                               let xs2 = case z of (zs1,zs2) -> zs2 in
                               let xs1' = (: x xs1)@r in
                               (xs1', xs2)@r
```

4 Transformation from Core-SAFE to CTRS

In this section we describe a transformation of Core-SAFE programs into conditional term rewriting systems. For the purpose of the transformation, we can even simplify the Core-SAFE syntax, because information concerning destructive patterns and regions is not relevant for termination purposes. Thus, we use

$$
\begin{array}{lcll}
prog & \rightarrow & dec_1; \ldots; dec_n; e & \\
dec & \rightarrow & f\ \overline{x_i}^n\ r = e & \{\text{single-recursive, polymorphic function}\} \\
 & | & f\ \overline{x_i}^n = e & \{\text{This does not return a new data structure}\} \\
a & \rightarrow & c & \{\text{atom is a basic constant}\} \\
 & | & x & \{\text{or a variable}\} \\
e & \rightarrow & a & \\
 & | & x@r & \{\text{copy}\} \\
 & | & x! & \{\text{reuse}\} \\
 & | & (f\ \overline{a_i}^n)@r & \{\text{function application}\} \\
 & | & (f\ \overline{a_i}^n) & \\
 & | & \textbf{let}\ x_1 = be\ \textbf{in}\ e & \{\text{non-recursive, monomorphic}\} \\
 & | & \textbf{case}\ x\ \textbf{of}\ \overline{alt_i}^n & \{\text{read-only case}\} \\
 & | & \textbf{case!}\ x\ \textbf{of}\ \overline{alt_i}^n & \{\text{destructive case}\} \\
alt & \rightarrow & C\ \overline{x_i}^n \rightarrow e & \{\text{algebraic datatype alternative}\} \\
 & | & \_ \rightarrow e & \{\text{default alternative for literal } \textbf{case}\} \\
be & \rightarrow & C\ \overline{a_i}^n@r & \{\text{constructor application}\} \\
 & | & e &
\end{array}
$$

Fig. 2. Syntax of Core-Safe

the simplified syntax in Figure 3. We assume that each **case** expression in a function definition has a unique integer label k. The transformation is defined by means of the following auxiliary functions:

1. *trP* which takes a sequence of Core-Safe function definitions and returns a CTRS.
2. *trF* which takes a function definition and returns a set of conditional rewrite rules.
3. *trR* which given an expression e, or a binding expression be, the set V of its free variables, and a condition $C = s_1 \rightarrow t_1, \ldots, s_k \rightarrow t_k$ consisting of atomic (rewrite) conditions $s_i \rightarrow t_i$, returns the right-hand side of a rule together with its conditional part, and a possibly empty set of conditional rewrite rules. In fact, we treat C as a list; if the conditional part $C = []$ then the generated right-hand side has no conditional part.
4. *trL* which, given a expression e and the set V of its free variables yields a left part of a condition, and a sequence of atomic conditions to its left.

Assume that $var(V)$ assigns the variables in V to a given term t in a fixed ordering. The transformation is given in Figure 4. Our running example would be transformed into the following CTRS:

```
splitD(n,xs) -> case1(n,n,xs)
case1(0,n,xs) -> Tup(Nil,xs)
case1(S(x),n,xs) -> case2(xs,n)
case2(Nil,n) -> Tup(Nil,Nil)
case2(Cons(x,xx),n) -> Tup(xs1',xs2) <= n-1 -> n', splitD(n',xx) -> z,
                       case3(z) -> xs1, case4(z) -> xs2, Cons(x,xs1) -> xs1'
case3(Tup(ys1,ys2)) -> ys1
case4(Tup(zs1,zs2)) -> zs2
```

$$
\begin{array}{lcll}
prog & \to & dec_1; \ldots; dec_n; e & \\
dec & \to & f\ \overline{x_i}^n = e & \{\text{A single version of function declaration}\} \\
a & \to & c & \{\text{basic constant}\} \\
 & | & x & \{\text{variable (replaces reuse and copy expressions)}\} \\
e & \to & a & \\
 & | & f\ \overline{a_i}^n & \{\text{A single version of function application}\} \\
 & | & \textbf{let}\ x_1 = be\ \textbf{in}\ e & \{\text{non-recursive, monomorphic}\} \\
 & | & \textbf{case}\ x\ \textbf{of}\ \overline{alt_i}^n & \{\text{a single version of case}\} \\
alt & \to & C\ \overline{x_i}^n \to e & \\
 & | & \_ \to e & \\
be & \to & C\ \overline{a_i}^n & \{\text{constructor application (region is irrelevant)}\} \\
 & | & e &
\end{array}
$$

Fig. 3. Simplified Core-SAFE

$$
\begin{array}{l}
trP(\overline{def_i}^n) \stackrel{\text{def}}{=} \bigcup_{i=1}^n trF(def_i) \\
trF(f\ \overline{x_i}^n = e) \stackrel{\text{def}}{=} f(x_1, \ldots, x_n) \to trR(e, fv(e), [\,]) \\
trR(c, V, C) \stackrel{\text{def}}{=} c \Leftarrow C \\
trR(x, V, C) \stackrel{\text{def}}{=} x \Leftarrow C \\
trR(C_r\ \overline{a_i}^n, V, C) \stackrel{\text{def}}{=} C_r(a_1, \ldots, a_n) \Leftarrow C \\
trR(f\ \overline{a_i}^n, V, C) \stackrel{\text{def}}{=} f(a_1, \ldots, a_n) \Leftarrow C \\
trR(k : \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n, V, C) \stackrel{\text{def}}{=} \\
\quad \{case_k(x, var(V)) \Leftarrow C\} \cup \\
\quad \{case_k(C_i(x_{i1}, \ldots, x_{in_i}), var(V)) \to trR(e_i, fv(e_i), [\,]) \mid i \in \{1..n\}\} \\
trR(\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2, V, C) \stackrel{\text{def}}{=} trR(e_2, fv(e_2), C +\!\!+ [, trL(e_1, fv(e_1)) \to x_1]) \\
trL(e, V) \stackrel{\text{def}}{=} trR(e, V, [\,]) \text{ if } e \in \{c, x, C_r\ \overline{a_i}^n, f\ \overline{a_i}^n, \textbf{case}\} \\
trL(\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2, V) \stackrel{\text{def}}{=} [trL(e_1, fv(e_1)) \to x_1,] +\!\!+ trL(e_2, fv(e_2))
\end{array}
$$

Fig. 4. Transformation from Core-SAFE to CTRS

Proposition 1. *Every Core-SAFE program $\mathcal{P}$ is transformed into an oriented, left-linear, non-overlapping, deterministic 3-CTRS $trP(\mathcal{P})$ which is, therefore, confluent.*

Proof. The resulting system is an oriented CTRS just by inspection of the generated rules.

For every defined symbol f, a single rule is generated with all argument variables $\overline{x_i}^n$ distinct. For every **case** expression labelled k, a non-overlapping set of rules $case_K$, one for every data constructor C_i, is generated. Each rule introduces distinct pattern variables $\overline{x_{ij}}^{n_i}$. So, the CTRS is non-overlapping and left-linear.

By induction on the calls to $trR(e, V, C)$, it is easy to show that $fv(e) \subseteq V \cup var(C)$. From here, and by inspection of the rules $l \to r \Leftarrow C$ generated, we conclude that the system is 3-CTRS. Finally, the last rule of trL satisfies

$fv(e_2) \subseteq V \cup \{x_1\}$. By induction on the number of simple conditions included in C we can prove that every condition C is deterministic. □

Proposition 2. *Given a Core-SAFE program $\mathcal{P}$ and its transformed 3-CTRS $\mathcal{R} = trP(\mathcal{P})$ the main expression e of $\mathcal{P}$ terminates according to Safe semantics if and only if the term t_e associated to e terminates in $\mathcal{R}$. Furthermore, in every term (except the last one, if it exists) of the reduction sequence of t_e there is only one redex.*

Proof. It simultaneously uses the small-step semantics of SAFE (not shown in this paper, see [24]) and the CTRS rules. It proceeds by induction on the depth k of the definition of the function symbols in the initial term. Then, by cases on the expressions e in function's bodies and, when the expression e is a **let** or a **case**, by induction on the number of operational semantics steps reducing e to normal form. □

Now we can apply standard transformations from deterministic 3-CTRS into unconditional TRSs [22, Def. 7.2.48]. If $\mathcal{R}$ is a 3-CTRS, let us call $U(\mathcal{R})$ to the TRS resulting from the transformation. For instance, in our running example $U(\mathcal{R})$ would be the following TRS:

```
splitD(n,xs) -> case1(n,n,xs)
case1(0,n,xs) -> Tup(Nil,xs)
case1(S(x),n,xs) -> case2(xs,n)
case2(Nil,n) -> Tup(Nil,Nil)
case2(Cons(x,xx),n) -> U1(n-1,x,xx)
U1(n',x,xx) -> U2(splitD(n',xx),x)
U2(z,x) -> U3(case3(z),z,x)
U3(xs1,z,x) -> U4(case4(z),x,xs1)
U4(xs2,x,xs1) -> U5(Cons(x,xs1),xs2)
U5(xs1',xs2) -> Tup(xs1',xs2)
case3(Tup(ys1,ys2)) -> ys1
case4(Tup(zs1,zs2)) -> zs2
```

Proposition 3. *For every Core-SAFE program $\mathcal{P}$,the TRS $U(trP(\mathcal{P}))$ satisfies the following properties:*

1. *It consists of non-overlapping rules. Moreover, all the lefthand sides are of the form $f(p_1, \ldots, p_n)$ where the p_i are flat patterns.*
2. *The righthand sides have at most a nesting depth of 1. In the worst case they are of the form $g(e_1, \ldots, e_n)$ with g being a function symbol and all the e_i being either variables, flat patterns, or terms $f(a_1, \ldots, a_m)$ with f being a function symbol and the a_j variables or basic constants.*

Proof. Straightforward by considering Proposition 1 for the original CTRS and the definition of the $U(\mathcal{R})$ transformation. □

It is a standard result [22, Prop. 7.2.50] that the termination of $U(\mathcal{R})$ implies the termination of $\mathcal{R}$.

5 Termination and complexity bounds

Termination of rewriting can be proved by using the dependency pairs approach [1]. Given a TRS $\mathcal{R} = (\mathcal{C} \uplus \mathcal{D}, R)$ the set $\mathsf{DP}(\mathcal{R})$ of *dependency pairs* for $\mathcal{R}$ is given as follows: if $f(t_1, \ldots, t_m) \to r \in R$ and $r = C[g(s_1, \ldots, s_n)]$ for some defined symbol $g \in \mathcal{D}$, and context $C[\cdot]$, and $s_1, \ldots, s_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, then $f^\sharp(t_1, \ldots, t_m) \to g^\sharp(s_1, \ldots, s_n) \in \mathsf{DP}(R)$, where $f^\sharp$ and $g^\sharp$ are new fresh symbols associated to f and g respectively. Termination of rewriting can be ensured by inspecting the *cycles* of the *dependency graph* associated to the TRS $\mathcal{R}$. The nodes of the dependency graph are the dependency pairs in $\mathsf{DP}(\mathcal{R})$; we refer the reader to [1] for details about how to build it.

An argument filtering π for a signature $\mathcal{F}$ is a mapping that assigns to every k-ary function symbol $f \in \mathcal{F}$ an argument position $i \in \{1, \ldots, k\}$ or a (possibly empty) list $[i_1, \ldots, i_m]$ of argument positions with $1 \leq i_1 < \cdots < i_m \leq k$. Argument filterings apply to terms to remove appropriate symbols and subterms (see [1]). Furthermore, when S is a subset of rules, we write $\pi(S)$ to denote the set $\{\pi(s) \to \pi(t) \mid s \to t \in S\}$.

The following results justify the use of reduction pairs (see Section 2) in proofs of termination using dependency pairs.

Theorem 1 (DP termination [1]**).** *A TRS $\mathcal{R}$ is terminating if and only if there is a reduction pair $(\succeq, \sqsupset)$ such that $\mathcal{R} \subseteq \succeq$ and $\mathsf{DP}(\mathcal{R}) \subseteq \sqsupset$.*

Proofs of termination using the dependency pair approach are usually achieved by considering the *dependency graph*, or rather the *cycles* in the dependency graph.

Theorem 2 (SCC termination [12]**).** *A TRS $\mathcal{R}$ is terminating if and only if for all cycles $\mathfrak{C}$ in the dependency graph there is an argument filtering $\pi_\mathfrak{C}$ and a reduction pair $(\succeq_\mathfrak{C}, \sqsupset_\mathfrak{C})$ such that $\pi(\mathcal{R}) \subseteq \succeq_\mathfrak{C}$, $\pi(\mathfrak{C}) \subseteq \succeq_\mathfrak{C} \cup \sqsupset_\mathfrak{C}$, and $\pi(\mathfrak{C}) \cap \sqsupset_\mathfrak{C} \neq \emptyset$.*

In general, the dependency graph of a TRS is *not* computable and we need to use some approximation of it (e.g., the *estimated* dependency graph, see [1]). According to Theorem 2, the important point in proofs of SCC termination is the generation of appropriate argument filterings $\pi_\mathfrak{C}$ and reduction pairs $(\succeq_\mathfrak{C}, \sqsupset_\mathfrak{C})$ for each cycle $\mathfrak{C}$ in the (estimated) dependency graph. In the following, we are interested in their generation by means of polynomial interpretations.

Proposition 4. *Given a Core-SAFE program $\mathcal{P}$, there is a bijection between cycles in the dependency graph of the TRS $\mathcal{R}_\mathcal{P} = U(trP(\mathcal{P}))$ and recursive calls in $\mathcal{P}$.*

Proof. Straightforfard by inspection of the transformation and by knowing the arcs in the estimated dependency graph require unification between the right part of a dependency pair and the left part of another pair. In our transformed system, a cycle is closed when the internal call $f^\sharp(t_1, \ldots, t_n)$ to a recursive Core-SAFE function f unifies with the dependency pair $f^\sharp(x_1, \ldots, x_n) \to r$ coming from the initial (and only) rule defining function f. □

So, in our running example, the only existing cycle in the dependency graph contains the following dependency pairs:

```
SplitD(n,xs)-> Case1(n,n,xs)
Case1(s(x),n,xs) -> Case2(xs,n)
Case2(cons(x,xx),n) -> U1(n-1,x,xx)
U1(n',x,xx) -> SplitD(n',xx)
```

which corresponds to the internal recursive call of *splitD*. Here, we capitalize the first letter of a function name f to indicate its associated symbol $f^\sharp$.

Let us assume that we use Theorem 1 and —by hand or by using a suitable tool— we find a polynomial interpretation of the TRS resulting from transforming a Core-Safe program. Let us call $[\![f^\sharp]\!]$ to the polynomial interpreting the symbol $f^\sharp$ associated to the Core-Safe function symbol f. This polynomial is guaranteed to remain non-negative for non-negative arguments and to decrease at each dependency pair. Moreover, in contrast to the polynomials obtained by using Theorem 2, there is a single polynomial interpreting each symbol $f^\sharp$.

Proposition 5 (polynomial bounds). *If $[\![f^\sharp]\!]$ is the polynomial (satisfying Theorem 1 and) interpreting the symbol $f^\sharp$ associated to a n-ary function symbol f in a Core-SAFE program and $x_1, \ldots, x_n$ are interpreted as the sizes of the input arguments to f, then the number $N(x_1, \ldots, x_n)$ of recursive calls to f with arguments $t_1, \ldots, t_n$ of sizes $x_1, \ldots, x_n$, respectively, is bounded by $[\![f^\sharp]\!](x_1, \ldots, x_n)$, i.e. $N(x_1, \ldots, x_n) \leq [\![f^\sharp]\!](x_1, \ldots, x_n)$*

Proof. The proposition is a consequence of Proposition 4 and of the fact that the polynomial obtained for DP-termination is an upper bound on the number of times a cycle of the depencency graph is traversed during the rewriting of a term. □

Consider the TRS $U(\mathcal{R})$ obtained in Section 4 for our running example. The following polynomial intepretation:

```
[pred](X) = 1/2.X
[S](X) = 2.X + 1
[splitD](X1,X2) = 0
[case1](X1,X2,X3) = 0
[0] = 0
[tup](X1,X2) = 0
[nil] = 0
[case2](X1,X2) = 0
[cons](X1,X2) = X2 + 2
[U1](X1,X2,X3) = 0
[U2](X1,X2) = 0
[case3](X1,X2) = 0
[SPLITD](X1,X2) = 2.X1 + 2.X2 + 1
[CASE1](X1,X2,X3) = X1 + X2 + 2.X3
[CASE2](X1,X2) = 2.X1 + X2
[UU1](X1,X2,X3) = 2.X1 + 2.X3 + 2
[UU2](X1,X2) = 1
[PRED](X) = 0
[CASE3](X1,X2) = 0
```

Safe function	Polynomial inferred
$length(x)$	$2x+1$
$splitD(n,x)$	$2n+2x+1$
$mergeD(x,y)$	$x+y+1$
$msortD(x)$	No proof obtained
$insert(x,t)$	$2t+1$

Fig. 5. Polynomials obtained for several Core-Safe functions

which is obtained by MU-TERM [19] proves DP-termination of $U(\mathcal{R})$.

The problem now is that it is unclear how to give a suitable definition of *polynomial* associated to a given defined symbol f. In principle, a symbol $f^\sharp$ can occur in *several* cycles $\mathfrak{C}$, thus leading to different polynomials $[\![f^\sharp]\!]_{\mathfrak{C}}$.

6 Case studies

We have applied Theorem 1 to the TRS's obtained by transforming the Core-Safe functions presented in Section 3 —including function `length` with the obvious definition— and have obtained the polynomials shown in Figure 5.

The last one *insert* is the function inserting an element in a binary search tree. Its full-Safe definition is as follows:

```
data Tree a = Empty | Node (Tree a) a (Tree a)

insert x Empty = Node Empty x Empty
insert x (Node l y r)
   | x < y  = Node (insert x l) y r
   | x == y = Node l y r
   | x > y  = Node l y (insert x r)
```

and the CTRS resulting from the *trP* transformation is:

```
insert(x,t) -> case1(t,x)
case1(Empty,x) -> Node(l0,x,r0) <= Empty -> l0, Empty -> r0
case1(Node(l,y,r),x) -> case2(c,l,y,r,x) <= lt(x,y) -> c
case2(False,l,y,r,x) -> case3(c',l,y,r,x) <= eq(x,y) -> c'
case2(True,l,y,r,x) -> Node(l',y,r) <= insert(x,l) -> l'
case3(False,l,y,r,x) -> case4(c'',l,y,r,x) <= gt(x,y) -> c''
case3(True,l,y,r,x) -> Node(l,y,r)
case4(False,l,y,r,x) -> error
case4(True,l,y,r,x) -> Node(l,y,r') <= insert(x,r) -> r'
```

From the above results, and interpreting the argument variables as characterizing the size of the corresponding data structures, we are glad to see that the bounds obtained are rather accurate. For instance, if an argument x is of type list and we interpret x as its length, the polynomial $x+y+1$ accurately bounds the number of recursive calls to $mergeD(x,y)$. To see whether interpreting argument variables as sizes is correct or not we must pay attention to the

interpretation given by Proposition 5 to data constructors. During the execution of a function f, the formal arguments of f will be replaced by actual ones and these consist just of ground terms formed by data constructors. By knowing the polynomial interpretation obtained for these constructors, we can know the polynomial associated to the whole term representing the actual data structure passed to f as actual argument. In the examples above, we have obtained the following interpretation for the list data constructors:

$$[\![Nil]\!] = 0$$
$$[\![Cons(x, xs)]\!] = k + xs$$

being $k = 1$ or $k = 2$. For binary trees, we have obtained:

$$[\![Empty]\!] = 0$$
$$[\![Node(l, x, r)]\!] = 1 + l + r$$

Then, the polynomial associated to a complete list will be related to its length and the one associated to a binary tree will coincides with its cardinality. This circumstance allows us to interpret argument variables as sizes (at least in the examples we have tried so far).

The polynomials obtained for *length* and *splitD* are less accurate, but at least they show an accurate linear dependency with their argument sizes. The bound for *insert* is also accurate as the binary tree needs not be balanced: in the worst case, the number of recursive calls grows linearly with the tree size.

We have not obtained a termination proof for *msortD*. We must be prepared for that due to the incompleteness of any termination proving algorithm. Apparently, the current TRS termination proving technology is not able to detect that the sizes of the lists passed as arguments to *msortD* in the two recursive calls are strictly smaller that the list of the external call. Manipulating the rules, it is possible to obtain a termination proof for $U(trF(msortD))$ but it is not clear that this manipulation can always be obtained automatically.

7 Hierarchical composition of SAFE programs

When proving termination and complexity bounds of Safe programs, two strategies can be applied:

1. Either the whole program is transformed into a TRS, and then this is submitted to a termination prover tool such as MU-TERM.
2. Or else each function is separately analyzed for termination, assuming that the functions possibly called from the analized one in turn terminate.

Approach (1) is more realistic in the sense that the TRS exactly corresponds to the original Core-Safe program. In particular, constructor and function symbols are global to the whole program and the polynomials obtained for them, in case of success, guarantee that *every term* will be finitely rewritten. That is, every well-formed main expression using those symbols will terminate.

However, programs can be huge and the time needed by the termination tool will probably increase more than linearly with program size. So, it is worthwhile

to investigate the modularity properties of the TRS obtained from the transformation of Safe programs. Intuitively, if we get a polynomial bounding the number of recursive calls of a particular function f, this is a property which depends on the *definition* of f (and, of the definition of all the functions used by f) but not on its *use* in enclosing contexts. So, we expect that the polynomial of a function f, once obtained, will remain stable along the function definitions following that of f in the Safe text. In this case, f's polynomial would not need to be inferred again when analyzing the functions that follows f.

Data constructors are a different matter as they miss a definition. Our experiments tell us that almost always they get a polynomial which clearly indicates the *size* of the data structure starting at this constructor, as it has been mentioned in Section 6. Then, we believe that it is desirable to force a fixed interpretation for the constructors, and this interpretation should convey the intuitive notion of size for the corresponding data structure.

In this vein, when inferring the polynomial for a particular function f, we could force the interpretation of the functions defined previously to f, to the polynomials obtained for them. In this way, the termination tool will have to infer only the polynomials for the new defined symbols introduced by the transformation of f, and the amount of work would approximately be proportional to the complexity of f.

We should then prepare our termination tool for this mixed working mode, i.e. for receiving some polynomials as given and then inferring the rest.

8 Related Work

We have already cited in the introduction the works ([4–7]) aiming to classify TRS's in time and space complexity classes by using polynomial interpretations.

In the area of programming languages, there have been some attempts to infer complexity space bounds by using specialized type systems. The two following works compute linear space bounds of first order functional programs:

- Hughes and Pareto [17] incorporate in Embedded-ML the concept of region and their sized-types system is able to *type-check* heap and stack linear bounds from annotations given by the programmer.
- More recently, in a proof carrying code framework, Hofmann and Jost [16] have developed a type system to *infer* linear bounds on heap consumption. The underlying machinery is a Linear Programming system which solves the restrictions generated during type inference.

Related to the latter there has been the successful EU funded project *Mobile Resources Guarantees* [2] which, in addition to inferrring space bounds, produces formal certificates of this property. These certificates can be verified by a proof-checker. A follow-on project is the Netherlands funded one AHA [26], which tries to extend the above results to space bounds beyond linear ones.

Our approach seems promising with respect to these works in that any polynomial can be inferred by current termination proving tools. Polynomic space

bounds can be easily obtained by multiplying the polynomial bounding the number of recursive calls and the space needed at each recursive call. The latter will be in general another polynomial given that inner calls can consume polynomic space in the *self* region.

9 Conclusions and Future Work

The experiments reported in this paper encourages us to continuing the exploration of the approach of using TRS termination tools to infer polynomial bounds on the number of recursive calls of real programs.

However, much work remains to be done. In particular, we consider using DP-termination as a first approach to the problem due to the following reasons:

1. It forces *all* dependency pairs of the TRS to be strictly oriented while in fact only one strictly oriented pair per cycle is required for SCC-termination. The consequence is that termination proofs will fail more often.
2. Bounding the *total* number of recursive calls is sometimes too a rough bound on the length of recursive calls chains, due to the fact that a function may be multiple-recursive.

SCC-termination seems more promising in both repects but it requires a correct way of composing the polynomials obtained for the different cycles of the TRS. It is not clear that the least upper bound of all the polynomials gives always the correct bound. We conjecture that adding the polynomials obtained for a given symbol $f^{\#}$ will always be a safe bound.

Another path to explore is to provide the termination proving tool with some help from the programmer in order to prove termination, and to infer correct polynomials, in cases, such as the *msortD*, function where the tool fails to obtain a proof.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.
2. D. Aspinall, S. Gilmore, M. Hofmann, D. Sanella, and I. Stark. Mobile Resources Guarantees for Smart Devices. In *Proceedings of the Int. Workshop CASSIS'05*, pages 1–26. LNCS 3362, Springer, 2005.
3. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, Cambridge, 1998.
4. G. Bonfante, A. Cichon, J.-Y. Marion, and H. Touzet. Complexity classes and rewrite systems with polynomial interpretation. In G. Gottlob, E. Grandjean, and K. Seyr, editors, *12th International Workshop on Computer Science Logic, CSL '98*, volume 1584 of *Lecture Notes in Computer Science*, pages 372–384. Springer-Verlag, 1998.
5. G. Bonfante, A. Cichon, J.-Y. Marion, and H. Touzet. Algorithms with polynomial interpretation termination proof. *Journal of Functional Programming*, 11(1):33–53, 2001.

6. G. Bonfante, J.-Y. Marion, and J.Y. Moyen. Quasi-interpretations and Small Space Bounds. In J. Giesl, editor, *16th International Conference on Rewriting Techniques and Applications, RTA'05*, volume 3467 of *Lecture Notes in Computer Science*, pages 150–164. Springer-Verlag, 2005.
7. A. Cichon and P. Lescanne. Polynomial interpretations and the complexity of algorithms. In D. Kapur, editor, *11th International Conference on Automated Deduction, CADE'92*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 139–147. Springer-Verlag, 1992.
8. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.
9. E. Contejean, C. Marché, A.-P. Tomás, and X. Urbain. Mechanically proving termination using polynomial interpretations. *Journal of Automated Reasoning*, 34(4):315–355, 2006.
10. N. Dershowitz. Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17:279–301, 1982.
11. F. Durán, S. Lucas, J. Meseguer, C. Marché, and X. Urbain. Proving operational termination of membership equational programs. *Higher-Order and Symbolic Computation*, page to appear, 2007.
12. J. Giesl, T. Arts, and E. Ohlebusch. Modular termination proofs for rewriting using dependency pairs. *Journal of Symbolic Computation*, 34:21–58, 2002.
13. Jürgen Giesl, Stephan Swiderski, Peter Schneider-Kamp, and René Thiemann. Automated termination analysis for haskell: From term rewriting to programming languages. In Frank Pfenning, editor, *RTA*, volume 4098 of *Lecture Notes in Computer Science*, pages 297–312. Springer, 2006.
14. D. Hofbauer. Termination Proofs by Context-Dependent Interpretations. In A. Middeldorp, editor, *12th International Conference on Rewriting Techniques and Applications, RTA'01*, volume 2051 of *Lecture Notes in Computer Science*, pages 108–121. Springer-Verlag, 2001.
15. D. Hofbauer and C. Lautemann. Termination proofs and the length of derivations. In N. Dershowitz, editor, *3rd International Conference on Rewriting Techniques and Applications, RTA'89*, volume 355 of *Lecture Notes in Computer Science*, pages 167–177. Springer-Verlag, 1989.
16. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 185–197. ACM Press, 2003.
17. R. J. M. Hughes and L. Pareto. Recursion and Dynamic Data-Structures in Bounded Space; Towards Embedded ML Programming. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming, ICFP'99*, ACM Sigplan Notices, pages 70–81, Paris, France, September 1999. ACM Press.
18. D.S. Lankford. On proving term rewriting systems are noetherian. Technical report, Louisiana Technological University, 1979.
19. S. Lucas. MU-TERM: A Tool for Proving Termination of Context-Sensitive Rewriting. In Vincent van Oostrom, editor, *RTA*, volume 3091 of *Lecture Notes in Computer Science*, pages 200–209. Springer, 2004.
20. S. Lucas. Polynomials over the reals in proofs of termination: from theory to practice. *RAIRO Theoretical Informatics and Applications*, 39(3):547–586, 2005.
21. M. Montenegro, R. Peña, and C. Segura. An inference algorithm for guaranteeing safe destruction. In *Proceedings of the 8th Symposium on Trends in Functional Programming, TFP'07, New York, April 2-4*, pages 1–16, Chapter XIV, 2007.
22. E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, 2002.

23. R. Peña and C. Segura. A first-order functional language for reasoning about heap consumption. In *Proceedings of the 16th International Workshop on Implementation of Functional Languages, IFL'04*, pages 64–80, 2004.
24. R. Peña and C. Segura. Formally Deriving a Compiler for SAFE. In *Proceedings of the 18th International Sympsium on Implementation and Application of Functional Languages, IFL'06, Budapest*, pages 429–446, 2006.
25. R. Peña, C. Segura, and M. Montenegro. A sharing analysis for SAFE. In *Proceedings of the 7th Symposium on Trends in Functional Programming, TFP'06, Nottingham (UK), March 2006. Also in Selected papers of TFP'06, to be published by Intellect*, pages 205–221, 2006.
26. M. van Eekelen, O. Shkaravska, R. van Kesteren, B. Jacobs, E. Poll, and S. Smetsers. AHA: Amortized Space Usage Analysis. In *Proceedings of the 8th Symposium on Trends in Functional Programming, TFP'07, New York, April 2-4*, pages 1–16, Chapter XVI, 2007.

Graph Parser Combinators

Steffen Mazanek, Mark Minas

Universität der Bundeswehr, München, Germany,
{steffen.mazanek|mark.minas}@unibw.de

Abstract. Graphs are a central data structure in computer science. A set of graphs can be described by a graph grammar in a manner similar to a string grammar known from the theory of formal languages. Such a graph grammar can be used, for instance, to describe correct diagrams of a particular visual language, and the membership of a particular diagram can be checked using a graph parser. Unfortunately, graph parsing is known to be computationally expensive in general. There are quite simple graph languages that crush most general-purpose graph parsers.
In this paper we present graph parser combinators, a new approach to graph parsing inspired by the well-known string parser combinators. The basic idea is to define primitive graph parsers for elementary graph components and a set of combinators for the construction of more advanced graph parsers. Using graph parser combinators efficient special-purpose graph parsers can be composed conveniently.

Keywords: functional programming, graph parsing, parser combinators

1 Introduction

Graphs are a central data structure in computer science. In the context of functional programming languages they are used to represent terms (with sharing of common subterms) such that term rewriting is, in fact, graph rewriting [1]. Furthermore graphs are heavily used for modeling and specification. For instance, the second author has specified visual languages using hypergraph grammars [2].

A hypergraph is a generalization of a graph where edges do not necessarily connect exactly two nodes, but are allowed to connect an arbitrary number of nodes as determined by the type of the edge. Hypergraph grammars are used to define a particular hypergraph language in analogy to string grammars known from formal language theory. We introduce hypergraphs formally in Sect. 3.

Checking whether a given hypergraph belongs to a particular hypergraph language or not can be done by using a hypergraph parser. However, even for restricted kinds of grammars hypergraph parsing is NP-complete [3, p. 142 ff.]. Thus a general-purpose hypergraph parser cannot be expected to run in polynomial time for arbitrary grammars. There are quite simple hypergraph languages that crush most general-purpose hypergraph parsers. And even if a graph language can be parsed in polynomial time by a general-purpose parser, a special-purpose parser tailored to the language is likely to outperform it.

For this reason we propose a new approach to graph parsing: Graph Parser Combinators. We have been inspired by the work of Hutton and Meijer [4] who have proposed monadic parser combinators for string parsing. The basic principle of such a parser combinator library is that primitive parsers are provided that can be combined into more advanced parsers using a set of powerful combinators. For example, in string settings there are the sequence and choice combinators that can be used to emulate a grammar. However, a wide range of other combinators are also possible. For instance, in parser combinator libraries there often are combinators `many` (applies a given parser multiple times and collects the results) and `sepBy` (similar to `many` except that a given separator has to be interspersed).

Parser combinators are very popular, because they integrate seamlessly into the rest of the program and hence the full power of the host language can be used. Unlike Yacc [5] no extra formalism is needed to specify the grammar. Another benefit is that parsers are first-class values within the language. For example, we could construct lists of parsers or pass them as function parameters. The possibilities are only restricted by the potential of the host language.

Having all these benefits in mind it seems to be quite interesting to see how parser combinators can be adopted to graph settings. The discussion of this idea is the main contribution of this paper:

- We propose a framework and a set of graph parser combinators in Section 4 that can be used to implement efficient special-purpose graph parsers straightforwardly.
- Then we go on to demonstrate the practical use of these combinators by applying them to a real-world example, namely the visual language VEX (visual expressions). This example is introduced in Section 2 and the corresponding parser is constructed in Section 5.

A first Impression

At this point we provide a toy example to give an impression of what a parser constructed using our combinators is going to look like.

An important advantage of the combinator approach is that a more operative description of a language can be given. For example, the language of the strings $\{a^n b^n c^n | n \in \mathbb{N}\}$ is not context-free. Hence a general-purpose parser for context-free languages cannot be applied at all, although parsing this language actually is very easy: "Take as many *a*s as possible, then take the same number of *b*s and finally take the same number of *c*s."

Using PolyParse [6], a widely-available parser combinator library for strings maintained by Wallace, a parser for this string language can be defined as shown in Fig. 1a. The type of this parser determines that there may be a user-state `s` (not used in this example), that the tokens have to be characters and that the result is a number. If the given word is not a member of the language one of the calls of `exactly` fails.

Note, that the given parser uses the `do`-notation, syntactic sugar Haskell [7] provides for dealing with monads. Monads in turn provide a means to simulate

```
abc::Parser s Char Int
abc =
 do
 as←many (char 'a')
 let na=length as
 exactly na (char 'b')
 exactly na (char 'c')
 return na
```

(a) String parser

```
abcG::NGrappa s String Int
abcG n =
 do
 (n',as)←chain (edge "a" 0 1) n
 let na = length as
 (n'',_)←exactChain na (edge "b" 0 1) n'
 (n''',_)←exactChain na (edge "c" 0 1) n''
 return na
```

(b) Graph parser

Fig. 1: Parsers for the string and the graph language $a^n b^n c^n$

state in Haskell. In parser settings they are used to hide the list of unconsumed input. Otherwise all parsers in a sequence would have to pass this list as a parameter explicitly.

In order to motivate our approach to graph parser combinators we provide the graph equivalent to the previously introduced string parser `abc`. Strings generally can be represented as hypergraphs straightforwardly. For instance, Fig. 2 provides the graph representation of the string "aabbcc".

A graph parser for this graph language can be defined using our combinators in a manner quite similar to the parser given above as shown in Fig. 1b. The main differences between the implementations of `abc` and `abcG` are:

- A starting node has to be passed as a parameter.
- Parsers also return the number of the node where they finish.
- Tentacle numbers identifying connections have to be provided as parameters of `edge`.

Interestingly a context-free hypergraph grammar can be defined that generates the string hypergraphs $a^n b^n c^n$ (see [3, p. 109]). However, parsing according to this graph grammar is polynomial whereas the given parser is linear in complexity.

Although many details have been omitted here, we hope that the idea behind graph parser combinators is understandable already. In this paper we present the framework and the combinators that can be used to conveniently implement parsers like `abcG`.

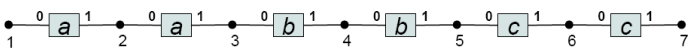


Fig. 2: The string graph "aabbcc"

2 A running example: VEX

In this section we introduce the visual language VEX as our running example. Visual languages and graphs are highly related, because graphs are a very natural means of describing complex situations on an intuitive level. Graphs in particular appear to be well suited as an intermediate data structure in visual language editors. For instance, in editors generated using the diagram editor generator DiaGen [2], visual components are mapped to hyperedges and a hypergraph parser is used to check whether the diagram is a member of the particular language. This mapping is described in more detail in Section 3.

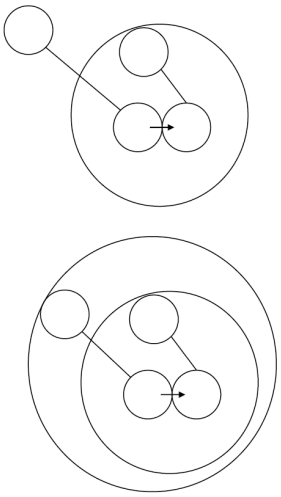


Fig. 3: The VEX expressions $\lambda x.(y\ x)$ and $\lambda x.\lambda y.(x\ y)$

VEX [8] is a language for the visualization of lambda expressions. In Fig. 3 two exemplary VEX diagrams are given that represent the lambda terms $\lambda x.(y\ x)$ and $\lambda x.\lambda y.(x\ y)$. A VEX diagram basically consists of a set of circles, lines and arrows, whose layout determines the represented lambda term.

A λ-abstraction is represented by a circle c_1 containing a smaller circle c_2 (the abstraction's parameter) internally tangent to it. The body of the abstraction is given by the components contained in c_1. A line between a variable v and c_2 indicates that v is bound to the parameter of the abstraction.

Function application is expressed by two circles externally tangent to each other. An arrow between these circles indicates the direction of application. Per default application associates to the left. However, a different order of application can be determined by a particular numbering scheme.

Circles that do not contain other circles represent variables. Variables have to be connected by a line either with the parameter of an abstraction or with a free variable circle. Free variable circles are used to identify occurrences of free variables within an expression and must not be contained in another circle. It is possible but not necessary to specify names of abstraction parameters and free variables.

We do not want to go into more detail here – [8] provides a full and more precise description of the syntax. In the following we will use the language VEX to clarify the notions of graphs, graph grammars and parsing. Furthermore the benefits of our parser combinators are demonstrated by constructing a special-purpose parser for VEX.

3 Definition of Graphs

In this section we give a formal definition of hypergraphs and Haskell types for their representation.

Following [3] a hypergraph consists of a set of hyperedges and a set of nodes. A hyperedge is an atomic item with a fixed number of tentacles, called the type of the hyperedge. It can be embedded into a hypergraph by attaching each of its tentacles to a node. Directed graphs are a special case of this notion, i.e. they are hypergraphs whose edges are distinguished by exactly two tentacles, the first representing the source of the edge and the second the sink, respectively.

In Fig. 4 the hypergraph model of Fig. 3 is shown. Each visual component is mapped to a particularly labeled hyperedge represented by a rectangular box. For instance, the free variable circle in the upper left is mapped to a hyperedge labeled "freevar". The filled black circles represent nodes that we have additionally marked with numbers. A line between a hyperedge and a node indicates that the node is visited by that hyperedge. The binding of the variable to the free variable circle in the upper diagram is mapped to an edge "bind" that links the "freevar" and "var" edges.

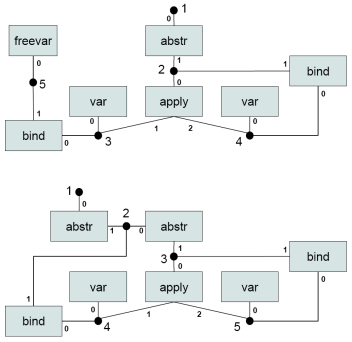


Fig. 4: Hypergraph representation of the VEX expressions of Fig. 3

The small numbers close to the hyperedges are the tentacle numbers. Without these numbers the image may be ambiguous, since different tentacles usually play different roles. For instance, the tentacle with number 1 of "abstr" hyperedges always has to be attached to the body of the abstraction and the tentacles 1 and 2 of "apply" hyperedges to a function and its argument, respectively.

Before we provide Haskell types that represent the graph data structures we have to introduce our graph model more formally, in particular because it differs from standard definitions as found in, e.g., [3, Sect. 2.2], that do not introduce the notion of a *context*.

Let C be a set of labels and $type : C \to \mathbb{N}$ a typing function for C. In the following a hypergraph H over C is a finite set of tuples (e, lab, ns), where e is a (hyper-)edge[1] number unique in H, $lab \in C$ is an edge label and ns is a sequence of node numbers such that $type(lab) = |ns|$ (length of sequence). The nodes represented by the node numbers in ns are called *incident* to edge e. A tuple (e, lab, ns) we call a *context* (in analogy to [9]).

The position of a particular node n in the sequence of nodes within a context of an edge e represents the tentacle of e that n is attached to. Hence the order of

[1] We call hyperedges just edges and hypergraphs just graphs if it is clear from the context that we are talking about hypergraphs.

nodes matters. The same node number also may occur in more than one context indicating that the edges represented by those contexts are connected via this node.

Note, that our notion of hypergraphs is slightly more restrictive than usual, because we do not allow isolated nodes. In particular the nodes of H are given by $V = \bigcup_{(e,l,ns)\in H} ns$, i.e., they are derivable from the set of contexts and not determined in advance. In fact, in many hypergraph application areas isolated nodes simply do not occur. For example, in the context of visual languages diagram components may be represented by hyperedges and nodes may just represent connection points of diagram components, i.e., each node is attached to at least one edge. So we ignore this issue at the moment.

The following Haskell code introduces the basic data structures for representing nodes, edges and (hyper-)graphs altogether:

```
module Graph where

type Node = Int
type Edge = Int
type Tentacle = Int
type Context lab = (Edge, lab, [Node])
type Graph lab = [Context lab]
```

For the sake of simplicity, we represent nodes and edges by integer numbers. We declare a graph as a list of contexts, where each context represents a labeled hyperedge including its incident nodes represented by their numbers.

The first VEX graph given in Fig. 4 can then be represented as the following Haskell term:

```
vex1::Graph String
vex1 = [(0,"abstr",[1,2]),(1,"apply",[2,3,4]),(2,"var",[4]),
        (3,"bind",[4,2]),(4,"var",[3]),(5,"bind",[3,5]),(6,"freevar",[5])]
```

The node numbers occurring in this term correspond directly to the node numbers given in the figure. The edges are numbered uniquely.

Note, that the given Haskell declaration does not ensure *per se* that only valid hypergraphs can be constructed. For instance, it has to be checked that the number of incident nodes corresponds to the type of the particular edge.

We additionally provide a function `matchContext` that searches for a particular `Context` and returns both this context and the remaining graph:

```
matchContext::(Context lab→Bool)→Graph lab→(Maybe (Context lab), Graph lab)
matchContext _ [] = (Nothing, [])
matchContext test (c:g)
  | test c = (Just c, g)
  | otherwise = let (c', g') = matchContext test g in (c', c:g')
```

In this section we have declared Haskell data types that represent a graph as a list of so-called contexts. The fact, that we use the data type list as a container may seem weird at first sight and indeed this is not the most efficient representation. However, using this standard data structure will pay off in the next section.

4 Parsing Graphs

In this section we introduce our graph parser combinators. However, first the notion of a parser in graph settings has to be clarified.

4.1 Graph Grammars and Parsers

A set of hypergraphs, i.e. a hypergraph language, can be defined using a graph grammar – an extension of formal language theory to graph settings. A widely known kind of graph grammar are hyperedge replacement graph grammars (HRGG) as described in [3]. Here, a nonterminal hyperedge of a given hypergraph is replaced by a new hypergraph that is glued to the remaining graph by fusing particular nodes. Formally, such a HRGG G is a quadruple $G = (N, T, P, S)$ that consists of a set of nonterminals $N \subset C$, a set of terminals $T \subset C$ with $T \cap N = \emptyset$, a finite set of productions P over N and a start symbol $S \in N$. To describe VEX we have to extend this notion with so-called embeddings – special productions to insert an additional hyperedge in an already derived hypergraph.

The graph grammar for VEX can be defined as $G_V = (N_V, T_V, P_V, Vex)$ where $N_V = \{Vex, Lambda, Freevar\}$, $T_V = \{freevar, apply, abstr, var, bind\}$ and P_V contains the productions given in Fig. 5. These productions are notated very similar to BNF known from string grammars, i.e. left-hand side *lhs* and right-hand side *rhs* are separated by the symbol `::=` and several *rhs* of one and the same *lhs* are separated by vertical bars. By convention we use lower case letters to label terminal hyperedges; in contrast labels of nonterminal hyperedges start with a capital letter.

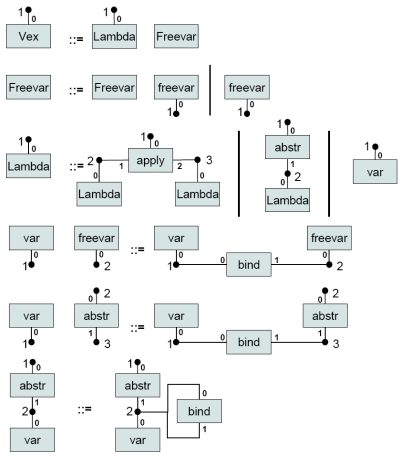


Fig. 5: Graph grammar for VEX

Note, that the grammar of VEX is not context-free, i.e. there are productions with more than one edge at their *lhs*, although the graph language is quite simple. In particular the embedding of the "bind" edge cannot be defined in a context-free manner such that the derivation is not a tree, but a DAG. The given grammar also does not ensure that there has to be exactly one "bind" hyperedge connected to each variable. Such restrictions usually have to be expressed by so-called application conditions (see, e.g., [2]).

A general-purpose graph parser for HRGGs gets passed a particular HRGG and a graph as parameters and constructs a derivation tree of this graph according to the grammar. This can be done, for instance, in a way similar to the well-known algorithm of Cocke, Younger and Kasami [10] in string settings (all HRGGs can be transformed to the graph equivalent of the string notion

Chomsky Normal Form). However, such a parser could not deal with our VEX grammar, because of the embeddings.

Another problem regarding this grammar is caused by the "Freevar" productions. Since the terminal "freevar" hyperedges are not connected in a specific way a lot of different derivations are possible. It is simply not clear, in which order the different "freevar" edges have to be derived. A general-purpose parser usually performs poorly on such highly ambiguous grammars.

Having this in mind we can state, that a general-purpose parser capable of parsing VEX hypergraphs has to be quite powerful. Hence the language VEX highly motivates the construction of a special-purpose parser.

4.2 A Combinator Library

The main contribution of this paper is the introduction of graph parser combinators, a library that simplifies the construction of special-purpose parsers as motivated above. Our design goals have been:

- Intuitive look and feel, i.e. short training period for people already familiar with parser combinators.
- Straightforward translation of a grammar to a parser.
- Simple parsers for simple languages even if the grammar of the language is complicated.[2]
- Sufficient performance for practically relevant applications.

We do not start from scratch implementing our graph parser combinator library. Rather we use PolyParse [6] as a base, a light-weight monadic parser combinator library already mentioned in the introduction.

In string settings a parser basically is a function that takes a string and returns a list of results including their remaining strings. However, since it is very useful to lex the input first, most parsing libraries have generalized the parser type to take a list of tokens instead of characters. As our graphs are lists of contexts we can just instantiate the token type to `Context` and use the already existing framework, that has in particular appeared to be well-suited for practical applications. Since PolyParse is widely-used in string settings, this approach has the additional benefit that many users are already familiar with its usage.

So, in the following we use the module `PolyState`, a part of [6], as a base for our module `Grappa`[3]:

```
module Grappa where
import Text.ParserCombinators.PolyState
import Graph
```

[2] A good example is the language of the string graphs $\{a^n b^n c^n | n \in \mathbb{N}\}$. The HRGG describing this language is quite complicated despite the simplicity of the language.

[3] The name `Grappa` is a shorthand for GRAPh PArser (in fact, we distill the most out of a graph instead of wine mash).

First we define our basic graph parser type:

```
type Grappa s lab a = Parser s (Context lab) a
```

The type `Parser` that we instantiate is defined as follows in module `PolyState`:

```
newtype Parser s t a = P (s → [t] → (EitherE String a, s, [t]))
```

Thereby `EitherE` is a type similar to `Either` that additionally provides support for different gradations of failing. Note, that in addition to the token list, a state of type `s` is carried along. For instance, we will need such a state when parsing VEX to keep track of variable bindings. The type parameter `t` defines the type of the tokens. The parse result has to be of type `a`. Thus the type `Grappa` describes the class of parsers whose tokens are given by contexts as introduced in Section 3.

There is an instance of the type class `Monad` for the type `Parser` defined in module `PolyState`. Hence in the following we also can use the convenient `do`-notation.

A main difference between graph parsing and string parsing is that in graph settings we normally do not know where to actually start parsing. There is no such thing as a first character/token. Thus, most of our parsers will need a start node as a parameter explicitly. For convenience, we define an additional type `NGrappa` that hides this node parameter:

```
type NGrappa s lab a = Node→Grappa s lab a
```

Most combinators will return a result of type `NGrappa`. The following function converts such a `NGrappa` to a normal `Grappa` by trying every node of the graph as a starting point successively.

```
nGrappaToGrappa::NGrappa s lab a→Grappa s lab a
nGrappaToGrappa ng = do
                     g←getTokens
                     oneOf (map ng (nodes g))
```

Using `getTokens` the current token list, i.e. graph, is queried (but not consumed). Thereafter the standard combinator `oneOf` is used that tries a list of parsers one after the other until finally a particular parser succeeds. We construct this parser list by mapping `ng` to all nodes of the graph.

Of course this function has to be used with care – at least if performance is an issue. In particular it does not choose the biggest parse but the first successful one. However, its declaration demonstrates how flexibly parser combinators can be composed.

Now we can declare our first primitive parser, `context`, that searches for and consumes the first context passing a particular test. This choice is deterministic for performance reasons:

```
context::(Context lab→Bool)→Grappa s lab (Context lab)
context test = P (λs ts→
  case matchContext test ts of
   (Nothing,_)→(Left (False,"No context fits."),s,ts)
   (Just c, ts')→(Right c,s,ts'))
```

Note, that there are only very few parsers that need to access the internals of the `Parser` type directly; `context` is such a parser. If no matching context can be found it fails (`Left`) softly (indicated by the boolean value `False`) returning a proper error message. Otherwise it returns the particular context removing it from the graph to be processed further.

Most of the time we are only interested in contexts (hyperedges), that are connected to a particular node. In this case the function `connContext` is handy, since it additionally checks if the context is connected to a given node `n` via a particular tentacle `t`, i.e. `ns!!t==n`:

```
connContext::(Context lab→Bool)→Tentacle→NGrappa s lab (Context lab)
connContext test t n = context (λc@(_,_,ns)→test c && ns!!t==n)
```

And if we additionally demand the edge to have a particular label we can use `labContext` or `connLabContext`, respectively:

```
labContext lab = context (hasEdgeLab lab)
connLabContext lab = connContext (hasEdgeLab lab)
```

The following parser `edge` can be used to process a particularly labeled connected context and additionally returns the node that is attached via a particular tentacle. This node can be understood as the link to a successive context.

```
edge::(Eq lab)⇒lab→Tentacle→Tentacle→NGrappa s lab (Node,Context lab)
edge lab inT outT n = do
                      c@(_,_,ns)←connLabContext lab inT n
                      return (ns!!outT,c)
```

Our library provides some more helpful primitive parsers. However, we omit their definitions here and rather switch the focus to combinators.

A first important combinator is `chain`. It applies a given `NGrappa` as long as possible assuring proper connections between the different parses. Its result is the list of these partial results. Note, that the connecting nodes are especially important, because they have to be provided as starting points for the particular parser. Therefore the active node has to be passed through, i.e. the parser has to also return the node where it stops.

```
chain::NGrappa s lab (Node,a)→NGrappa s lab (Node,[a])
chain p n = do {
            (n',x)←p n;
            (n'',xs)←chain p n';
            return (n'',x:xs)
            } `onFail` return (n,[])
```

The string combinator `exactly` used in the introduction can be carried over to graph settings quite easily, too:

```
exactChain::Int→NGrappa s lab (Node,a)→NGrappa s lab (Node,[a])
exactChain 0 p n = return (n,[])
exactChain num p n = do
                     (n',x)←p n
                     (n'',xs)←exactChain (num-1) p n'
                     return (n'',x:xs)
```

Note, that with `chain` and `exactChain` all combinators used in the definition of our motivating example `abcG`, i.e. the parser for the string graphs $\{a^n b^n c^n | n \in \mathbb{N}\}$, are introduced already.

In addition several parser combinators known from string parsing can be reused for graph parsing straightforwardly, e.g. `oneOf`. Furthermore, there are some combinators that have to be used with care. Their semantics changes in graph settings, because they do not maintain proper connections (unlike to string settings the order of tokens, i.e. contexts, does not represent a particular kind of connection anymore). For instance, the combinator `many` can be used in graph settings just to parse more or less independent subgraphs or star shapes.

And finally there are combinators that should not be used in most cases. The combinators `next` and `satisfy`, for instance, depend on the head of the token list. They are frequently used in string settings, however, in graph settings there is no such thing as a next token.

There are a lot more primitive parsers and combinators imaginable. However, the ones presented in this section give a good first insight. Furthermore, they are in particular needed to tackle our example VEX.

5 Parsing VEX

In this section we construct a parser for the example language VEX using the combinators defined in the previous section. The purpose of the presentation of this parser is twofold. First, we demonstrate how the combinators have to be used. And second, we intend to show, that special-purpose graph parsers can be defined very straightforwardly.

In Fig. 6 the parser for VEX diagrams is presented. Our goal is to map a VEX graph to its underlying λ-term. Hence we define the type `Lambda`, that represents λ-terms straightforwardly and that is going to be the result type of the parser. Note, that `Lambda` is a kind of tree and not a DAG as one might expect, because variable bindings are resolved by proper naming.

VEX is an example where we really need a parser state. In particular we store a number used to construct a name for the next fresh variable to be bound in an abstraction. Further on the state contains a lookup table that maps node numbers to variable names. The type `VexState` represents these requirements.

The top-level parser `vex` first consumes all (`many`) "freevar" edges using the parser `freevar`. Thereby the lookup table is extended properly introducing fresh names for the new variables. Note, that the "freevar" edges are consumed in the order of their occurrence within the list of contexts, i.e. we are not interested in the fact, that they may also be parsed in other orders. Finally the remaining graph is parsed using `lambda`.

The parser `lambda` does accept either an application (`apply`), an abstraction (`abstr`) or a variable (`var`). An application consists of a hyperedge labeled "apply" that is connected to two subgraphs via its tentacles 1 and 2, that have to be parseable with `lambda` again. The overall result then is composed by the

```
data Lambda = Abstr String Lambda |
              Apply Lambda Lambda |
              Var String

type VexState = (Int,[(Node,String)])

vex::NGrappa VexState String Lambda
vex n = do
        many freevar
        lambda n

freevar::Grappa VexState String ()
freevar = do
          (_,_,[n])←labContext "freevar"
          stUpdate (λ(nn,vt)→(nn+1,(n,"v"++show nn):vt))

lambda::NGrappa VexState String Lambda
lambda n = oneOf [apply n, abstr n, var n]

var,abstr,apply::NGrappa VexState String Lambda
apply n = do
          (_,_,ns)←connLabContext "apply" 0 n
          l1←lambda (ns!!1)
          l2←lambda (ns!!2)
          return (Apply l1 l2)
abstr n = do
          (_,_,ns)←connLabContext "abstr" 0 n
          oldstate@(nn,vt)←stGet
          stUpdate (const (nn+1,(ns!!1,"v"++show nn):vt))
          l←lambda (ns!!1)
          stUpdate (const oldstate)
          return (Abstr ("v"++show nn) l)
var n = do
        connLabContext "var" 0 n
        (_,_,ns)←connLabContext "bind" 0 n
                  `adjustErrBad`
                 const ("No bind at node "++show n)
        (_,vt)←stGet
        case lookup (ns!!1) vt of
          Nothing→fail $ show (ns!!1) ++
                              " out of scope!"
          Just v→return (Var v)

vexParse::Node→Graph String→Either String Lambda
vexParse n = (λ(res,_,_)→res) ∘ runParser (vex n) (1,[])
```

Fig. 6: A parser for VEX diagrams

application of the data constructor `Apply` to the results yielded by parsing these subgraphs.

Parsing an abstraction means introducing a new variable in the lookup table that can be released after parsing its body. The state can be queried and changed using the `PolyState` functions `stGet` and `stUpdate`, respectively.

Hyperedges labeled "var" represent variables only if there also is a "bind" edge to a node that can be mapped to a variable name via the lookup table. Otherwise a severe error has to be raised using `adjustErrBad`, because the whole parse cannot succeed anymore if a variable is not bound.

Note, that at "apply" edges there are two directions to further process the graph that both have to yield a valid subgraph. This is a main difference to string parsing where the next token is always predetermined. We have dealt with this issue by parsing an application depth-first beginning with its left side. Since the diagram language VEX does not support sharing of common subexpressions we can consume recognized input unconditionally anyhow.

The parser for VEX graphs can be called using the function `vexParse` that applies the function `runParser` defined in `PolyState` to an initial state and just returns the result or an error message. For our examples we get:

```
Vex> vexParse 1 vex1
Right λv2→(v1 v2)
```

```
Vex> vexParse 1 vex2
Right λv1→(λv2→(v1 v2))
```

6 Performance

In [3, p. 142 ff.] it is proved that parsing of HRGGs is NP-complete. Thus a general-purpose graph parser cannot be expected to run in polynomial time in general. And indeed, even the quite simple language VEX does cause a general-purpose parser to run in exponential time in the worst case. This occurs if there is a high number of "freevar" edges, since these edges can be derived in so many different ways/orders ($n!$). The parser presented in Section 4, however, is not affected by this issue: We parsed 100.000 "freevar" edges in about one second.

We have also measured the execution time to parse the string graphs $a^n b^n c^n$ for several n using a general-purpose parser. As already mentioned it is possible to describe this graph language with a HRGG. It turns out that these string graphs can be parsed in polynomial time regarding this grammar, however, performance is worse than one might expect for such a simple language.

Using a special-purpose algorithm like the one presented as a motivating example in the introduction, we could parse such a string graph of length 10.000 in less than a second. Admittedly this comparison does not consider that we have to pass the start node explicitly. If we do not have the starting node and all nodes have to be tried, we get at the very most a factor $3 * n$. This still outperforms a general-purpose algorithm.

We could further improve the performance by using an efficient set implementation for the representation of graphs instead of lists. So we can definitely say, that special-purpose parsers constructed using our graph parser combinators are superior from a performance point of view.

7 Related Work

To our best knowledge graph parser combinators have not been considered up to now. So in this section we shortly sketch several related approaches to parsing in general and dealing with graphs in functional languages.

Besides PolyParse [6] there are other parser combinator libraries that are also widely-used. For instance, Parsec [11] is well-known for its high performance and good error reporting capabilities. The main difference between Parsec and PolyParse is that Parsec is predictive by default only backtracking where an explicit `try` is inserted whereas in PolyParse backtracking is the default except where explicitly disallowed by a `commit`. We have to gain more experience with graph parser combinators to judge which approach is better suited in our settings.

In string settings the complement of the parser combinator approach is parser generation. Thereby a grammar is given, e.g. in EBNF, from which a real parser in a particular language can be generated. The tool Happy [12] can be used to generate such a Haskell parser. However, the advantage of a parser generator for strings – namely that efficient parsers can be generated for nearly arbitrary context-free grammars – does not count that much in graph settings because of the NP-completeness.

At this point we also have to mention the work of Erwig [9], because our declarations are mainly inspired by his approach. Erwig criticized the imperative style of the algorithms described in, e.g., [13] and proposed a new approach: looking at graphs as inductively defined data types. So he presented a graph declaration where nodes are added inductively one at a time. Incident edges are represented as a part of their so-called contexts. Unfortunately his library [14] does not generalize to hypergraphs and also does not support graph rewriting and parsing.[4]

To our knowledge there only is one other approach that aims at the combination of functional programming and graph transformation. At the time of writing a textbook is work in progress that provides an implementation of the categorical approach to graph transformation with Haskell [16]. Since graphs satisfy the laws of a category a higher level of abstraction is used to implement graph transformation algorithms. The benefit of this approach is its generality since it just depends on categories with certain properties. However, up to now parsers are not considered. From an efficiency point of view this approach of course also is restricted by the NP-completeness of the problem domain.

Finally we want to pick up visual languages again, since they are a very important area of application for graph grammars as we have demonstrated by our running example. Hypergraphs as a model for visual languages have been elaborated in-depth in the work of the second author. In particular the practical relevance of hypergraph parsing has been proven by the implementation of the diagram editor generator DiaGen, that allows the generation of diagram editors out of hypergraph grammars (see, e.g., [2]).

[4] In fact, Erwig discussed termgraph rewriting in [15], however, the presented algorithm is tailored to the problem and cannot be generalized straightforwardly.

8 Conclusion and Further Work

In this paper we have introduced graph parser combinators, a new approach to the construction of efficient special-purpose graph parsers.

We have demonstrated that our library is efficient and extensible. It allows the construction of graph parsers in a for Haskell programmers familiar manner. Thus in combination with the also implemented general-purpose hypergraph parser we have taken an important first step towards a functional graph transformation library.

As we have backed up by examples our library in its present form is perfectly usable already. Nevertheless a lot of work remains to be done.

First of all we have to provide a solid theoretical foundation. Up to now disturbing components – both edges and nodes – are just ignored by most of the combinators. For instance, our parser for VEX is not suited for checking whether a graph is a member of the language, but it is very useful to extract and parse a valid subgraph. We have to further research how to simplify the writing of correct parsers using our framework.

Another area of future research is the extension of the set of combinators. Compared to string settings for graph settings there are many more interesting patterns worth their own combinator. For instance, it would be very useful to have a combinator that recognizes a particular subgraph isomorphic to a given graph. However, in general such a combinator cannot be implemented efficiently. Further on it would be interesting to see how capabilities for error recovery could be added (like [17] in string settings).

And finally we have to switch the underlying data structure from lists to an efficient implementation of sets to provide fast random access to a particular context. Fortunately the interface to our library would not be affected by an internal change of the data structure.

All in all, we propose the following as an attractive approach to graph parsing:

- First, try using a general-purpose graph parser and see how it scales. If performance is good or not an issue everything is ok.
- However, if performance is a problem use the graph parser combinator library presented in this paper to construct an efficient special-purpose graph parser in no time.

Acknowledgements

We would like to thank Malcolm Wallace for his useful comments regarding PolyParse and Martin Erwig, whose work on graphs in functional settings inspired this paper and who gave us a lot of encouraging feedback.

References

1. Plump, D.: Term graph rewriting. In Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook of Graph Grammars and Computing by Graph Transformation. Vol. II: Applications, Languages and Tools. World Scientific (1999) 3–61
2. Minas, M.: Concepts and realization of a diagram editor generator based on hypergraph transformation. Science of Computer Programming **44**(2) (2002) 157–180
3. Drewes, F., Habel, A., Kreowski, H.J.: Hyperedge replacement graph grammars. In Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations. World Scientific (1997) 95–162
4. Hutton, G., Meijer, E.: Monadic parser combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham (1996)
5. Johnson, S.C.: Yacc: Yet another compiler compiler. Technical Report 32, Bell Laboratories, Murray Hill, New Jersey (1975)
6. Wallace, M.: polyparse `http://www.cs.york.ac.uk/fp/polyparse/`.
7. Peyton Jones, S.: Haskell 98 Language and Libraries. The Revised Report. Cambridge University Press (2003)
8. Citrin, W., Hall, R., Zorn, B.: Programming with visual expressions. In Haarslev, V., ed.: Proc. 11th IEEE Symp. Vis. Lang, IEEE Computer Soc. Press (5–9 1995) 294–301
9. Erwig, M.: Inductive graphs and functional graph algorithms. J. Funct. Program. **11**(5) (2001) 467–492
10. Kasami, T.: An efficient recognition and syntax analysis algorithm for context free languages. Scientific Report AF CRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Massachussetts (1965)
11. Leijen, D., Meijer, E.: Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht (2001)
12. Gill, A., Marlow, S.: Happy - the parser generator for haskell `http://www.haskell.org/happy`.
13. King, D.: Functional Programming and Graph Algorithms. PhD thesis, University of Glasgow (1996)
14. Erwig, M.: FGL - A Functional Graph Library `http://web.engr.oregonstate.edu/~erwig/fgl/haskell/`.
15. Erwig, M.: A functional homage to graph reduction. Technical Report 239, Fern-Universität Hagen (1998)
16. Schneider, H.J.: Graph transformations - an introduction to the categorical approach. `http://www2.cs.fau.de/~schneide/gtbook/` (2007)
17. Swierstra, S.D., Azero Alcocer, P.R.: Fast, error correcting parser combinators: a short tutorial. In Pavelka, J., Tel, G., Bartosek, M., eds.: SOFSEM'99 Theory and Practice of Informatics, 26th Seminar on Current Trends in Theory and Practice of Informatics. Volume 1725 of LNCS. (November 1999) 111–129

Encoding Iterators in Interaction Nets
(Extended Abstract)

José B. Almeida[1], Ian Mackie[2], Jorge Sousa Pinto[1], and Miguel Vilaça[1]

[1] Departamento de Informática / CCTC
Universidade do Minho, Braga, Portugal
[2] LIX, CNRS UMR 7161, École Polytechnique, 91128 Palaiseau Cedex, France

Abstract. We propose a method for encoding iterators (recursion operators) using interaction nets. The method can be used to obtain a visual notation for functional programs, and also to extend with recursion the many translations of the λ-calculus into interaction nets, which have been proposed as efficient implementation mechanisms. We exemplify the method with a number of list-processing examples that illustrate the application to practical functional programming. Our examples also show that the method seems to generate, from appropriate functional programs, many typical examples of interaction net programs.
Keywords: Recursion operators, interaction nets, visual programming.

1 Introduction

Interaction nets have been extensively used as an implementation mechanism for the λ-calculus. The main motivation for this approach is that it results in highly efficient evaluation strategies, made possible by the close control kept on the erasing and duplication of terms.

The use of visual notations for functional programs has long been an active research topic; the goal is to have a notation that can be used:

1. to input functional programs visually, and
2. to animate visually the execution of functional programs, with the obvious applications in debugging and education.

The existence of translations of functional programs into interaction nets allows for the use of this graphical formalism as a visual notation, as long as the translation allows for the correspondence between the programs and their visual representations to be immediately established. The interest of using interaction nets as a visual notation is that programs can be animated without leaving the interaction formalism: instead of resorting to a functional interpreter to run individual reductions of the program and then displaying the result of each reduction, a program can be animated by simply reducing the net.

In this paper we extend the ideas of applying interaction nets beyond the pure λ-calculus, to richer languages including recursive types and recursive function definitions based on recursion operators, such as iterators and primitive recursors (fixpoint operators have been studied elsewhere [1, 11]). Our approach is based on the following principles that directly take advantage of interaction net features:

- Pattern-matching at the language level is implemented by the matching mechanism in the framework. The inherent inability to match constructors at level deeper than 1 raises no problems, since recursion operators in general match only the top-level constructors.
- Each occurrence of a recursion operator gives rise to a new interaction symbol (a definition, in the programming view), and the operator parameters are internalized in the symbol's interaction rules.
- Rule application then corresponds to the expansion of a recursive definition. Recursion is captured very naturally by interaction rules in which the recursion operator symbol is reintroduced in the right-hand side.

To illustrate our ideas, we take the simply-typed λ-calculus with booleans, natural numbers, and their respective iterators. This system is very close to Gödel's System $\mathcal{T}$ [5]. To allow for the examples to have a more realistic programming flavour, list types (and a list iterator) are also included. We choose to implement normal-order evaluation for this language, and take the token-passing implementation of the λ-calculus [15] as the basis on which our ideas are incorporated. Technically the main contribution of this paper is thus an expansion of this implementation to a richer typed language, resulting in a highly intuitive way of representing visually programs and their evaluation.

An interesting feature of the work presented in this paper is that the interaction systems output by our encodings result in exactly the same definitions that one would obtain programming directly at the visual level with interaction nets. In this sense our work justifies semantically this functional subset of interaction nets. Moreover this provides further evidence that our approach is indeed an appropriate and natural way to represent functional programs visually.

Work in the area of Visual Functional Programming has addressed different aspects of visual programming. The Pivotal project [7] offers a visual notation (and Haskell programming environment) for data-structures, not programs. Visual Haskell [14] more or less stands at the opposite side of the spectrum of possibilities: this is a dataflow-style visual notation for Haskell programs, which allows programmers to *define* their programs visually (with the assistance of a tool) and then have them translated automatically to Haskell code. Kelso's VFP system [8] is a complete environment that allows to define functional programs visually and then reduce them step by step. Finally, VisualLambda [3] is a formalism based on graph-rewriting: programs are defined as graphs whose reduction mimics the execution of a functional program. As far as we know none of these systems is widely used. Visual Haskell and VisualLambda have in common that functions are represented as boxes with input ports for the arguments and an output port for the result; the contents of the box correspond to the body of the function. They differ in that Visual Haskell uses variables to refer to function arguments, while VisualLambda uses a purely graphical notation based on arrows. Kelso's VFP uses a notation without boxes, more inspired by the traditional representations of functional programs used in implementation-oriented abstract machines. In particular, it allows for named functions but also for λ-abstractions, and an explicit application node exists. Variables are used for arguments, as in Visual Haskell. Higher-order programming is a fundamental feature of functional programming. A function f can take function g as an argument and g can then be applied within the body of f. Expressing this feature is easy if variables are used as in Visual Haskell and VFP; in VisualLambda a special box would be used as a placeholder for g (in the body of f) to be instantiated later, and an arrow would link an input port in the box of f to the box of g. The work presented in this paper uses a pure visual representation of programs, without variables. In this aspect it resembles VisualLambda, however our work differs significantly from this in that no boxes are used, and all the graph-rewriting operations are *local* in the sense that only two nodes of the graph are involved in each step. A second difficulty arising from the higher-order nature of programs is that a (curried) function of two arguments may receive only its first argument and return as result a function. In a box-based representation this means that it must be possible for a box to lose its input ports one by one—a quite complicated process. Graph rewriting in general, and our approach in particular, treat this problem naturally as will become clear.

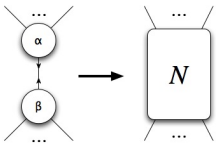
Structure of the Paper. Section 2 contains background material on interaction nets and encodings of the λ-calculus, and Section 3 defines the functional language used in the paper. Section 4 introduces the translation of functional programs into token-passing interaction nets. Section 5 discusses how the approach can be used with other translations of the λ-calculus into nets, and Section 6 presents a number of illustrative examples. Section 7 considers extensions of the language with other recursion operators. We conclude the paper in Section 8.

2 Background

Interaction nets [9] are constrained graph rewrite systems that can still encode all the computable functions. Interaction nets provide a model of computation in a graphical setting. Programs are represented as particular kinds of graphs, and computation is expressed as graph transformations. They are user-defined, in the same way as term rewriting systems, by giving a signature Σ (a set of symbols, which are nodes of the graph) and a set of rewrite rules R. An occurrence of a symbol will be called an *agent*. Each agent has a fixed arity. If the arity of an agent is n, then there are $n+1$ *ports* for the agent: a distinguished one, depicted by an arrow, called the principal port, and n auxiliary ports. Agents are represented graphically in the following way:

A net N built on a signature Σ is a graph (not necessarily connected) with agents at the vertices. The edges of the net connect agents together at the ports such that there is only one edge at every port, although edges may connect two ports of the same agent. The ports of an agent that are not connected to another agent are called the free ports of the net.

A pair of agents, say (α, β), connected on their principal ports is called an *active pair*, which is the interaction net analogue of a redex. An interaction rule replaces an occurrence of the active pair (α, β) by a net N. The rule has to satisfy a very strong condition: all the free ports are preserved during reduction, and moreover there is at most one rule for each pair of agents. The diagram below illustrates the idea, where N is any net built from the signature:



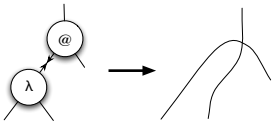
An interaction net system is therefore fully defined by the pair (Σ, R). We say that a net is in normal form if it does not contain any active pairs. We use the notation $\Longrightarrow$ for one-step reduction and $\Longrightarrow^*$ for its transitive reflexive closure. Additionally, we write $N \Downarrow N'$ if there is a sequence of interaction steps $N \Longrightarrow^* N'$, such that N' is a net in normal form. The strong constraints on the definition of interaction rules imply that reduction commutes (the one-step diamond property holds), and thus confluence is easily obtained. Consequently, any normalizing interaction net is strongly normalizing.

A number of different translations of the λ-calculus into interaction nets exist. These have in common some basic principles:

- If t is a λ-term then $\mathcal{T}(t)$ is a net in a fixed interaction net system $(\Sigma_{\mathcal{T}}, R_{\mathcal{T}})$.
- If t is a term then $\mathcal{T}(t)$ is an interaction net constructed with symbols from Σ_t. This net has at least one free port, corresponding to the *root* of the term, which will be drawn at the top of the net.
- Variables are translated simply as edges in $\mathcal{T}(t)$.
- If $x_1 \ldots x_n$ are free variables in t, then the net $\mathcal{T}(t)$ has n additional free ports (represented at the bottom) corresponding to each of the variables .
- $\mathcal{T}(\lambda x.t)$ is a net constructed structurally from $\mathcal{T}(t)$. Usually this introduces an abstraction symbol at the root of the term, with a port linked to the edge representing the bound variable x and a port linked to the root of the abstraction body net, $\mathcal{T}(t)$. A special case exists when $x \notin \mathcal{FV}(t)$, which is usually handled by introducing an erasing agent ε.
- $\mathcal{T}(tu)$ is a net constructed structurally from $\mathcal{T}(t)$ and $\mathcal{T}(u)$. Usually this introduces an application symbol (or net) with ports connected to the root ports of $\mathcal{T}(t)$ and $\mathcal{T}(u)$. A special case exists when a free variable occurs in both terms, since a single edge must represent this

variable at the bottom of the term. This is usually handled by introducing a copying agent c—its two auxiliary ports are connected to the edges representing the free variable in $\mathcal{T}(t)$ and $\mathcal{T}(u)$, and the edge connected to its principal port represents the variable in $\mathcal{T}(tu)$.

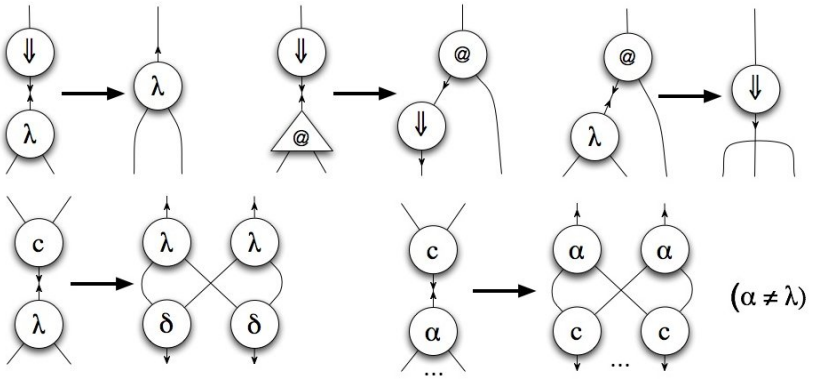
A straightforward translation can be used if the language is restricted to *linear* λ-terms. In this case just two agents @ and λ are needed that interact as:



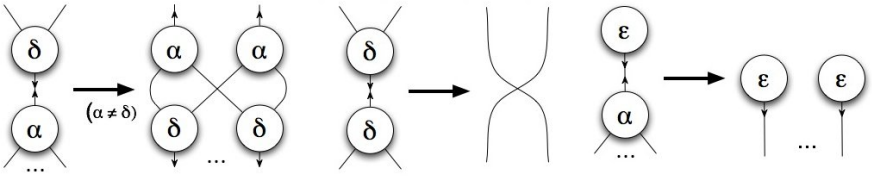
In the presence of non-linearity, some pairs of duplicating agents must annihilate when they interact, while others must duplicate each other. For this some extra machinery is required to handle the free variables in abstractions, such as a line of agents representing the 'boundary' of the term. These schemes have the advantage of imposing reduction strategies that cannot be defined using a term-based abstract machine; very efficient reduction strategies have been defined in this way. There are a number of interaction net encodings of the λ-calculus, which follow different strategies. To give just a sample: Gonthier, Abadi and Lévy [6] gave an implementation of optimal β-reduction. Mackie [12, 13] gave several systems, each giving a new strategy for reduction in the λ-calculus. [10] gave a method to implement existing strategies (head reduction). Correctness results differ between translations: stronger results establish a relation between the nets $\mathcal{T}((\lambda x.t)u)$ and $\mathcal{T}(t[u/x])$; weaker results establish relations between $\overline{\mathcal{T}(t)}$ and $\mathcal{T}(\overline{t})$, where $\overline{(\cdot)}$ denotes some notion of canonical form.

The *token-passing* encodings of [15] cope with non-linear terms by introducing a special evaluation agent (also called an evaluation *token*) that traverses the net imposing a call-by-name or call-by-value strategy. Applications are now represented by agents whose principal ports are located at the root of the terms, so in fact programs are represented by syntax trees, with additional edges corresponding to variables. The effect of the evaluation agent is to transform these into regular @ agents, whose principal port is turned towards the net representing the applied function.

The translation $\mathcal{T}_{\text{tp}}(\cdot)$ encodes terms in the system $(\Sigma_{\text{tp}}, R_{\text{tp}})$ where $\Sigma_{\text{tp}} = \{\Downarrow, @, @', \lambda, c, \varepsilon, \delta\}$ and R_{tp} consists of the rules:



and the standard rules for the eraser ε and duplicator δ are:



The arity of each symbol can be inferred from the rules. To emphasize the use of @′ as a syntactic symbol, we represent graphically the instances of @ and @′ respectively as:

The translation itself is shown in Figure 1, where $\mathcal{T}(.)$ stands for $\mathcal{T}_{\text{tp}}(\cdot)$. It generates nets containing no active pairs, so no reduction can happen. To start the reduction (corresponding to normal order evaluation), a $\Downarrow$ symbol must be connected to the root port of the term. Let $\Downarrow N$ denote the net obtained by connecting a $\Downarrow$ agent to the root port of N, then the following correctness result holds: $t \Downarrow z$ iff $\Downarrow \mathcal{T}_{\text{tp}}(t) \longrightarrow^* \mathcal{T}_{\text{tp}}(z)$, where the evaluation relation $\cdot \Downarrow \cdot$ is defined by the standard normal-order evaluation rules:

$$\frac{}{\lambda x.t \Downarrow \lambda x.t} \qquad \frac{t \Downarrow \lambda x.t' \qquad t'[u/x] \Downarrow z}{t\,u \Downarrow z}$$

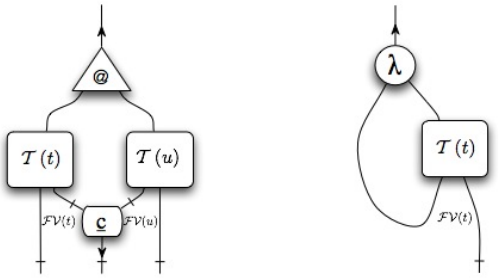


Fig. 1. The token-passing translation of λ-terms: the nets $\mathcal{T}(t\,u)$ and $\mathcal{T}(\lambda x.t)$. $\underline{\text{c}}$ denotes an arrays of c agents, one for each free variable occurring in both t and u. In $\mathcal{T}(\lambda x.t)$, a special case exists (not depicted) when the bound variable does not occur in the term: an ε agent must be connected to the λ agent instead.

3 The language BNL

In this paper we use the simply-typed λ-calculus extended with natural numbers, booleans, lists, and iterators for these recursive types. The language BNL is defined by the following syntax for types and terms (x, y range over a set of variables):

$$\begin{aligned} \tau &::= \mathbf{Bool} \mid \mathbf{Nat} \mid \mathbf{List}(\tau) \mid \tau \to \tau \\ t, u, v &::= x \mid \lambda x.t \mid t\,u \mid \mathsf{tt} \mid \mathsf{ff} \mid \mathsf{cond}\,t\,u\,v \\ &\quad \mid 0 \mid \mathsf{suc}(t) \mid \mathsf{iternat}\,(\lambda x.t)\,u\,v \mid \mathsf{nil} \mid \mathsf{cons}(t,u) \mid \mathsf{iterlist}\,(\lambda xy.t)\,u\,v \end{aligned}$$

and by the typing rules given by:

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x.t : \sigma \to \tau} \qquad \frac{\Gamma \vdash t : \sigma \to \tau \qquad \Gamma \vdash u : \sigma}{\Gamma \vdash tu : \tau}$$

$$\frac{}{\Gamma \vdash \mathsf{tt} : \mathbf{Bool}} \quad \frac{}{\Gamma \vdash \mathsf{ff} : \mathbf{Bool}} \quad \frac{}{\Gamma \vdash 0 : \mathbf{Nat}} \quad \frac{\Gamma \vdash n : \mathbf{Nat}}{\Gamma \vdash \mathsf{suc}(n) : \mathbf{Nat}} \quad \frac{}{\Gamma \vdash \mathsf{nil} : \mathbf{List}(\tau)}$$

$$\frac{\Gamma \vdash h : \tau \qquad \Gamma \vdash t : \mathbf{List}(\tau)}{\Gamma \vdash \mathsf{cons}(h,t) : \mathbf{List}(\tau)} \qquad \frac{\Gamma \vdash t : \mathbf{Bool} \qquad \Gamma \vdash V : \tau \qquad \Gamma \vdash F : \tau}{\Gamma \vdash \mathsf{cond}\, V\, F\, t : \tau}$$

$$\frac{\Gamma \vdash t : \mathbf{Nat} \qquad \Gamma \vdash \lambda x.S : \tau \to \tau \qquad \Gamma \vdash Z : \tau}{\Gamma \vdash \mathsf{iternat}\, (\lambda x.S)\, Z\, t : \tau}$$

$$\frac{\Gamma \vdash t : \mathbf{List}(\sigma) \quad \Gamma \vdash \lambda xy.C : \sigma \to \tau \to \tau \quad \Gamma \vdash N : \tau}{\Gamma \vdash \mathsf{iterlist}\, (\lambda xy.C)\, N\, t : \tau}$$

The reduction rules of BNL are as follows:

$$\begin{aligned}
(\lambda x.t)\, u &\longrightarrow t[u/x] \\
\mathsf{cond}\, V\, F\, \mathsf{tt} &\longrightarrow V \\
\mathsf{cond}\, V\, F\, \mathsf{ff} &\longrightarrow F \\
\mathsf{iternat}\, (\lambda x.S)\, Z\, 0 &\longrightarrow Z \\
\mathsf{iternat}\, (\lambda x.S)\, Z\, (\mathsf{suc}(n)) &\longrightarrow S[(\mathsf{iternat}\, (\lambda x.S)\, Z\, n)/x] \\
\mathsf{iterlist}\, (\lambda xy.C)\, N\, \mathsf{nil} &\longrightarrow N \\
\mathsf{iterlist}\, (\lambda xy.C)\, N\, \mathsf{cons}(h,t) &\longrightarrow C[h/x, (\mathsf{iterlist}\, (\lambda xy.C)\, N\, t)/y]
\end{aligned}$$

The normal-order (call-by-name) evaluation semantics are as follows. Note that terms with an outermost constructor are taken to be canonical forms.

$$\frac{}{\lambda x.t \Downarrow \lambda x.t} \qquad \frac{t \Downarrow \lambda x.t' \qquad t'[u/x] \Downarrow z}{t\, u \Downarrow z} \qquad \frac{}{0 \Downarrow 0} \qquad \frac{}{\mathsf{suc}(n) \Downarrow \mathsf{suc}(n)}$$

$$\frac{}{\mathsf{tt} \Downarrow \mathsf{tt}} \qquad \frac{}{\mathsf{ff} \Downarrow \mathsf{ff}} \qquad \frac{t \Downarrow \mathsf{tt} \qquad V \Downarrow z}{\mathsf{cond}\, V\, F\, t \Downarrow z} \qquad \frac{t \Downarrow \mathsf{ff} \qquad F \Downarrow z}{\mathsf{cond}\, V\, F\, t \Downarrow z}$$

$$\frac{t \Downarrow 0 \qquad Z \Downarrow z}{\mathsf{iternat}\, (\lambda x.S)\, Z\, t \Downarrow z} \qquad \frac{t \Downarrow \mathsf{suc}(n) \qquad S[\mathsf{iternat}\, (\lambda x.S)\, Z\, n/x] \Downarrow z}{\mathsf{iternat}\, (\lambda x.S)\, Z\, t \Downarrow z}$$

$$\frac{}{\mathsf{nil} \Downarrow \mathsf{nil}} \qquad \frac{}{\mathsf{cons}(u,v) \Downarrow \mathsf{cons}(u,v)} \qquad \frac{t \Downarrow \mathsf{nil} \qquad N \Downarrow z}{\mathsf{iterlist}\, (\lambda xy.C)\, N\, t \Downarrow z}$$

$$\frac{t \Downarrow \mathsf{cons}(u,v) \qquad C[u/x, \mathsf{iterlist}\, (\lambda xy.C)\, N\, v/y] \Downarrow z}{\mathsf{iterlist}\, (\lambda xy.C)\, N\, t \Downarrow z}$$

4 A Token-passing Encoding of BNL

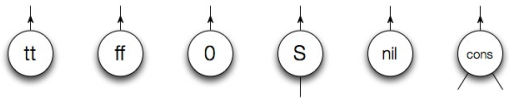
In this section we extend the token-passing call-by-name translation of the λ-calculus to BNL. We keep the translation of applications and abstractions, and propose a new way to encode recursive function definitions. Later it will be argued that this approach can be applied with other base translations of the λ-calculus.

Terms of inductively defined types can be represented in interaction nets in the natural way, as *trees* where each node corresponds to a constructor, with its principal port facing the parent node. For BNL we would have constructors: tt, ff, 0 and nil with arity 0; suc with arity 1; and cons with arity 2. A unique feature of interaction nets is that they allow for the representation of both programs and data in the same simple graphical formalism.

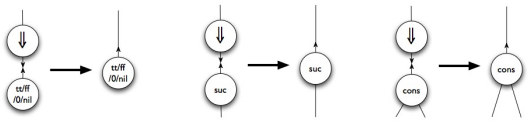
One key aspect of our approach is that each occurrence of an iterator will give rise to a new symbol, which will have one interaction rule for each different constructor of its argument type. This very simple form of pattern-matching, directly available in interaction nets, is sufficient for iterators and other recursion operators such as primitive recursors or accumulations (Section 7). The interaction rules of this agent internalise the iterator's parameters, so that the net $\mathcal{T}\,(\mathsf{iterlist}\,(\lambda xy.C)\,N\,\mathsf{cons}(h,t))$ one-step reduces to $\mathcal{T}\,(C[h/x,\mathsf{iterlist}\,(\lambda xy.C)\,N\,t/y])$, with an evaluation token on top to control normal-order evaluation.

A second key aspect is that each such new symbol will have auxiliary ports in a one-to-one correspondence with the free variables in the iterator term, since iterator terms are not restricted to be closed. The significance of this will be clear from the examples.

The interaction system will not be extended by introducing a fixed set of symbols; instead a new symbol will be introduced for *each occurrence of a recursion operator*, so a dedicated interaction system is generated for each term. Specifically, the interaction system in which the net $\mathcal{T}\,(t)$ will be evaluated is given as: $(\Sigma_t, R_t) = (\Sigma_{\mathrm{tp}} \cup \Sigma_{\mathrm{BNL}} \cup \Sigma_t^0, R_{\mathrm{tp}} \cup R_{\mathrm{BNL}} \cup R_t^0)$, where $(\Sigma_{\mathrm{tp}}, R_{\mathrm{tp}})$ has been defined in Section 2, the symbols in Σ_{BNL} can be depicted as:



and the rules in R_{BNL} are given below.



Finally, $(\Sigma_t^0, R_t^0) = \mathcal{S}\,(t)$, with the function $\mathcal{S}\,(\cdot)$ defined recursively as follows ($\cup$ is occasionally used to denote pairwise union). Note that we do not mention explicitly the arity of each agent, but this can be inferred from interaction rules involving the agent.

$\mathcal{S}\,(x) \doteq \mathcal{S}\,(\mathsf{tt}) \doteq \mathcal{S}\,(\mathsf{ff}) \doteq \mathcal{S}\,(0) \doteq \mathcal{S}\,(\mathsf{nil}) \doteq (\emptyset, \emptyset)$
$\mathcal{S}\,(\lambda x.t) \doteq \mathcal{S}\,(\mathsf{suc}(t)) \doteq \mathcal{S}\,(t)$
$\mathcal{S}\,(tu) \doteq \mathcal{S}\,(\mathsf{cons}(t,u)) \doteq \mathcal{S}\,(t) \cup \mathcal{S}\,(u)$
$\mathcal{S}\,(\mathsf{cond}\,V\,F\,b) \doteq (\{\mathsf{lt}_{V,F}^{\mathbf{Bool}}, \mathsf{lt}_{V,F}^{\mathbf{Bool}\prime}\} \cup \Sigma, R_{\mathsf{lt}_{V,F}^{\mathbf{Bool}}} \cup R)$,
where $(\Sigma, R) = \mathcal{S}\,(b) \cup \mathcal{S}\,(V) \cup \mathcal{S}\,(F)$, and $R_{\mathsf{lt}_{V,F}^{\mathbf{Bool}}}$ consists of the interaction rules included in Figures 2(a) and 2(b).

$\mathcal{S}\,(\mathsf{iternat}\,(\lambda x.S)\,Z\,n) \doteq (\{\mathsf{lt}_{S,Z}^{\mathbf{Nat}}, \mathsf{lt}_{S,Z}^{\mathbf{Nat}\prime}\} \cup \Sigma, R_{\mathsf{lt}_{S,Z}^{\mathbf{Nat}}} \cup R)$
where $(\Sigma, R) = \mathcal{S}\,(n) \cup \mathcal{S}\,(S) \cup \mathcal{S}\,(Z)$ and $R_{\mathsf{lt}_{S,Z}^{\mathbf{Nat}}}$ consists of the interaction rules included in Figures 2(a) and 2(c).

$\mathcal{S}\,(\mathsf{iterlist}\,(\lambda xy.C)\,N\,l) \doteq (\{\mathsf{lt}_{C,N}^{\mathbf{List}}, \mathsf{lt}_{C,N}^{\mathbf{List}\prime}\} \cup \Sigma, R_{\mathsf{lt}_{C,N}^{\mathbf{List}}} \cup R)$
where $(\Sigma, R) = \mathcal{S}\,(l) \cup \mathcal{S}\,(C) \cup \mathcal{S}\,(N)$ and $R_{\mathsf{lt}_{C,N}^{\mathbf{List}}}$ consists of the interaction rules included in Figures 2(a) and 2(d).

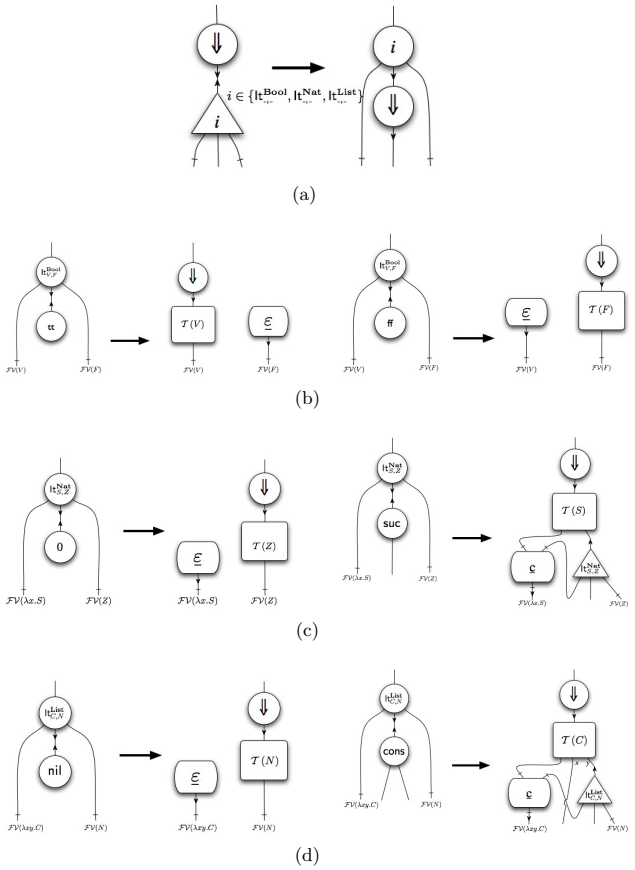


Fig. 2. Interaction rules for iterators

In Figures 2(b) to 2(d), $\underline{c}$ denotes an array of c agents and $\underline{\varepsilon}$ denotes an array of ε agents. The size of this array depends on the number of free variables in the corresponding terms. We may now define the translation from BNL programs into interaction nets. Iterator terms are naturally translated as syntax trees (like any other terms in the token-passing approach), which requires the introduction of a companion syntactic agent i' for each iterator agent i. As was the case for the encoding of the λ-calculus, we use the same names but depict the agents by triangles.

Definition 1. *The translation $\mathcal{T}(\cdot)$ is a function that given a* BNL *program t constructs an interaction net $\mathcal{T}(t)$ in the interaction system (Σ_t, R_t) as follows:*

- *If t is an abstraction or application, then $\mathcal{T}(t)$ is defined as in Figure 1.*
- *If t is one of* tt, ff, 0, *or* nil, *then $\mathcal{T}(t)$ is an instance of the corresponding symbol.*
- *If $t = \mathsf{suc}(n)$, then $\mathcal{T}(t)$ is constructed by connecting the auxiliary port of a* suc *agent to the root port of $\mathcal{T}(n)$.*
- *If $t = \mathsf{cons}(h, t')$, then $\mathcal{T}(t)$ is constructed by connecting the auxiliary ports of a* cons *agent to the root ports of $\mathcal{T}(h)$ and $\mathcal{T}(t')$.*
- *If $t = \mathsf{cond}\, V\, F\, b$ then $\mathcal{T}(t)$ is given by the net in Figure 3(a).*
- *If $t = \mathsf{iternat}\,(\lambda x.S)\, Z\, n$ then $\mathcal{T}(t)$ is given by the net in Figure 3(b).*
- *If $t = \mathsf{iterlist}\,(\lambda xy.C)\, N\, l$ then $\mathcal{T}(t)$ is given by the net in Figure 3(c).*

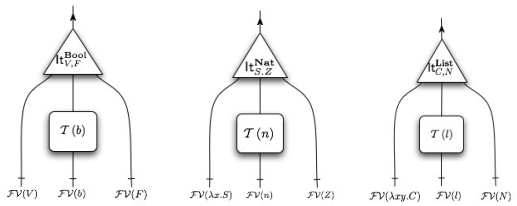


Fig. 3. Translations of iterators

Lemma 1. *Let t be a closed* BNL *term; then: $t \Downarrow z \implies \Downarrow \mathcal{T}(t) \longrightarrow^* \mathcal{T}(z)$.*

Lemma 2. *Let t be a closed* BNL *term and z a canonical form, then: $\Downarrow \mathcal{T}(t) \longrightarrow^* \mathcal{T}(z) \implies t \Downarrow z$.*

Proposition 1 (Correctness). *If t is a closed* BNL *term and z a canonical form, then: $t \Downarrow z \iff \Downarrow \mathcal{T}(t) \longrightarrow^* \mathcal{T}(z)$.*

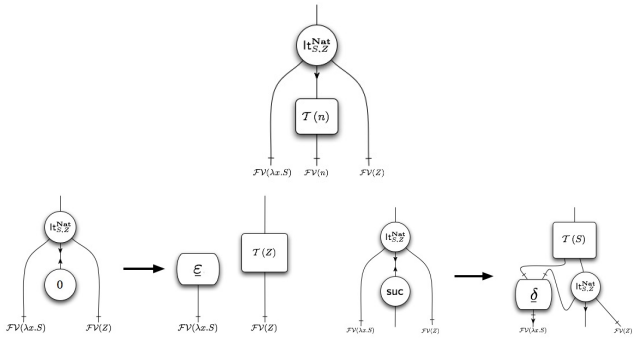
5 Handling Other Encodings

The token-passing translation of the λ-calculus has the advantage of implementing a simple evaluation order and maintaining a structure in the nets that is always immediately recognizable and understandable in terms of the evaluation semantics. As such it is totally appropriate for our goal of providing a visual representation for functional programs.

This translation is however not representative of most work in this area, which has concentrated on designing *efficient*, rather than simple, translations. These efficient translations are not controlled by an evaluation token, and in fact they produce nets already containing active pairs.

Let $\mathcal{T}(\cdot)$ be one such translation. $\mathcal{T}(tu)$ is constructed from $\mathcal{T}(t)$ and $\mathcal{T}(u)$ by introducing an application symbol @ with its principal port connected to the root port of $\mathcal{T}(t)$. Our treatment of

iterators can be adapted to this setting by simply removing the evaluator tokens and introducing the iterator agents with the principal port immediately facing the argument. For instance we have that $\mathcal{T}\,(\mathsf{iternat}\,(\lambda x.S)\,Z\,n)$ may be given by the following net, with the rules below.

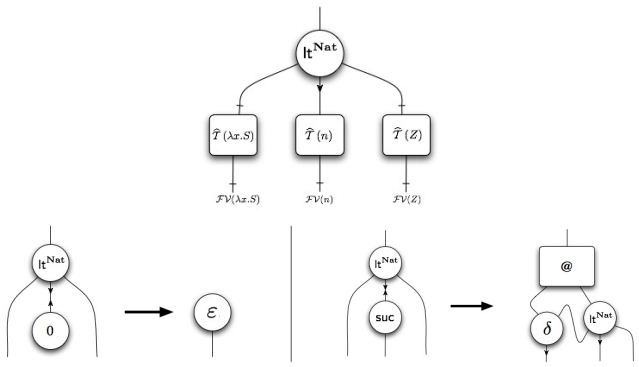


When the iterated function is a closed term, a correctness result can be easily established:

Lemma 3. *Let $\lambda x.S$ be a closed term, then*

1. $\mathcal{T}\,(\mathsf{iternat}\,(\lambda x.S)\,Z\,0) \longrightarrow \mathcal{T}\,(Z)$
2. $\mathcal{T}\,(\mathsf{iternat}\,(\lambda x.S)\,Z\,\mathsf{suc}(n)) \longrightarrow \mathcal{T}\,(S[\mathsf{iternat}\,(\lambda x.S)\,Z\,n/x])$

We remark that it is always possible to work with iterators with closed functions—thus this result applies to all programs. In general the correctness of the resulting translation of BNL has to be proved for each base translation of the λ-calculus. If such a result can be established, it still has to be studied if, and in what way, the reduction strategy imposed by the translation for the λ-calculus is modified by this treatment of recursion. An alternative approach exists, which is not suited for obtaining a visual representation of programs, but more suited for extending arbitrary translations without disrupting the underlying, presumably efficient reduction strategy. In this approach we add a single new symbol to the interaction system, for each recursion operator in the language, and extend the encoding accordingly. We exemplify this approach with the natural numbers iterator. We introduce a symbol $\mathsf{It}^{\mathbf{Nat}}$ and define $\widehat{\mathcal{T}}\,(\mathsf{iternat}\,(\lambda x.S)\,Z\,n)$ to be the following net, with the the rules below.



Observe that the iterator parameters are no longer internalised in the agent, and the interaction rule for successor simply encodes the application of the external function connected to agent. This

is a conservative extension in the sense that it does not disturb the reduction strategy imposed by the translation $\widehat{\mathcal{T}}(\cdot)$, since substitutions are no longer realized by the iterator reductions.

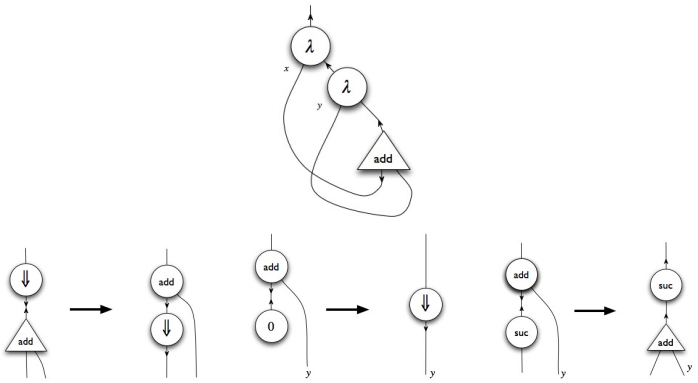
Using the fact that erasing and duplication for this system hold, i.e., if t is a closed term then the agent ε erases the net $\mathcal{T}(t)$ and the agent δ produces two copies of $\mathcal{T}(t)$.

Lemma 4. *Let $\lambda x.S$ be a closed term, then*

1. $\widehat{\mathcal{T}}\,(\mathsf{iternat}\,(\lambda x.S)\,Z\,0) \longrightarrow^* \widehat{\mathcal{T}}\,(Z)$
2. $\widehat{\mathcal{T}}\,(\mathsf{iternat}\,(\lambda x.S)\,Z\,\mathsf{suc}(n)) \longrightarrow^* \widehat{\mathcal{T}}\,((\lambda x.S)\,(\mathsf{iternat}\,(\lambda x.S)\,Z\,n))$

6 Examples

Example 1. $\mathsf{add} : \mathbf{Nat} \to \mathbf{Nat} \to \mathbf{Nat}$, defined by $\lambda xy.\mathsf{iternat}\,(\lambda r.\mathsf{suc}(r))\,y\,x$. The free variable y in the second argument of the iterator gives rise to an auxiliary port in the symbol $\mathsf{It}^{\mathbf{Nat}}_{\mathsf{suc}(r),y}$. It is encoded as the following net, where add stands for $\mathsf{It}^{\mathbf{Nat}}_{\mathsf{suc}(r),y}$, with the rules below.

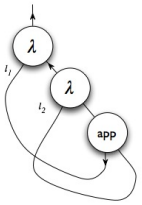


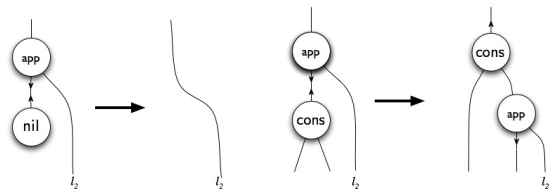
We now leave behind the token-passing translation and concentrate instead on the encoding of iterators, assuming a translation in the spirit of Section 5.

Example 2. The append function: $\mathsf{app} : \mathbf{List}(a) \to \mathbf{List}(a) \to \mathbf{List}(a)$, defined as

$$\lambda l_1 l_2.\mathsf{iterlist}\,(\lambda hr.\mathsf{cons}(h,r))\,l_2\,l_1$$

This is a very similar iteration to the previous example. The free variable in the second argument of the iterator gives rise to an auxiliary port in the symbol $\mathsf{It}^{\mathbf{List}}_{\mathsf{cons}(h,r),l_2}$. It is encoded as the following net, where app stands for $\mathsf{It}^{\mathbf{List}}_{\mathsf{cons}(h,r),l_2}$, with the rules below.

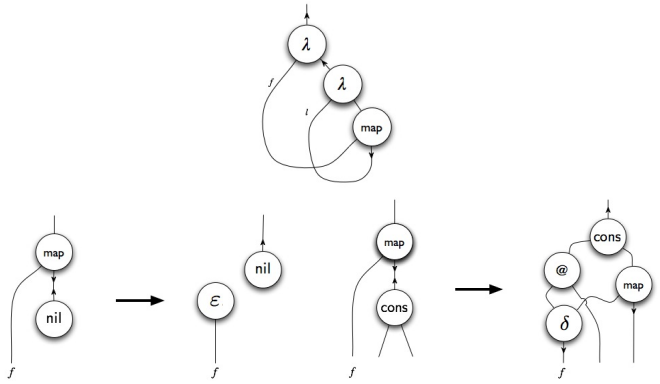




Example 3. The function $\mathsf{map} : a \to b \to \mathbf{List}(a) \to \mathbf{List}(b)$, defined as

$$\mathsf{map} = \lambda fl.\mathsf{iterlist}\,(\lambda hr.\mathsf{cons}(fh, r))\,\mathsf{nil}\,l$$

This example differs from the previous in that a free variable now occurs in the *first* argument of the iterator. Again this generates an auxiliary port in $\mathsf{It}^{\mathbf{List}}_{\mathsf{cons}(fh,r),\mathsf{nil}}$. Note also the introduction of a duplicator agent for this argument. The function is encoded as the following net, where $\mathsf{map} \equiv \mathsf{It}^{\mathbf{List}}_{\mathsf{cons}(fh,r),\mathsf{nil}}$, with the rules below.



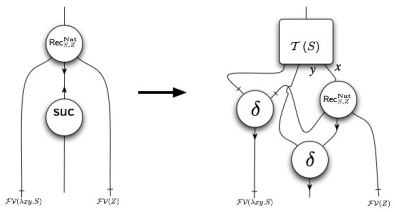
7 Extending the Language with New Operators

Primitive recursion requires a recursor for natural numbers, with the following syntax, typing and reduction rules: $t, u, v ::= \ldots \mid \mathsf{recnat}\,(\lambda x.Z)\,S\,t$,

$$\frac{\Gamma \vdash t : \mathbf{Nat} \qquad \Gamma \vdash \lambda xy.S : \tau \to \mathbf{Nat} \to \tau \qquad \Gamma \vdash Z : \tau}{\Gamma \vdash \mathsf{recnat}\,(\lambda xy.S)\,Z\,t : \tau}$$

$$\begin{aligned} \mathsf{recnat}\,(\lambda xy.S)\,Z\,0 &\longrightarrow Z \\ \mathsf{recnat}\,(\lambda xy.S)\,Z\,(\mathsf{suc}(n)) &\longrightarrow S[\mathsf{recnat}\,(\lambda xy.S)\,Z\,n/x, n/y] \end{aligned}$$

Essentially the extra computational power of the primitive recursor comes from the fact that it has access to its *argument*, in addition to the recursive result on that argument. The factorial function, for instance, can be defined in this way, but not with an iterator. Only a few modifications are required: an agent $\mathsf{Rec}^{\mathbf{Nat}}_{S,Z}$ must be used in the translation of the expression $\mathsf{recnat}\,(\lambda xy.S)\,Z\,t$ instead of $\mathsf{It}^{\mathbf{Nat}}_{S,Z}$, and the only difference in the corresponding interaction system (with respect to $\mathsf{It}^{\mathbf{Nat}}_{S,Z}$) is the interaction rule with the successor symbol, shown below, where we note that with an argument $\mathsf{suc}(n)$, the net representing n must now be duplicated.



Extending the language with other recursion operators is not only a matter of expressiveness, but also of convenience. We take as example the `foldl` list operator: even though it can be encoded with `foldr` (the Haskell list iterator), it is still convenient to have it in the language. For instance, a linear time, tail-recursive function for reversing lists can be written in the two following ways:

```
revt l = foldr (\h r a -> r(h:a)) id l []
revt l = foldl (\r h -> h:r) [] l
```

The latter is clearly preferable for its simplicity. The first version can be written in BNL as $\mathsf{revt} = \lambda l.\mathsf{iterlist}\,(\lambda xya.y\,\mathsf{cons}(x,a))\,(\lambda x.x)\,l\,\mathsf{nil}$. Applying the encoding of Section 4 results in the introduction of an agent $\mathsf{It}^{\mathbf{List}}_{(\lambda a.y\,\mathsf{cons}(x,a)),(\lambda x.x)}$. Naturally, the interaction rules for this agent introduce encodings of abstractions in their right-hand sides, which results in a quite complicated definition.

To accommodate the second, simpler definition, we now consider the extension of BNL with an operator corresponding to `foldl`.

$$t, u, v ::= \ldots \mid \mathsf{acclist}\,(\lambda xy.t)\,u\,v$$

with the following typing and reduction rules:

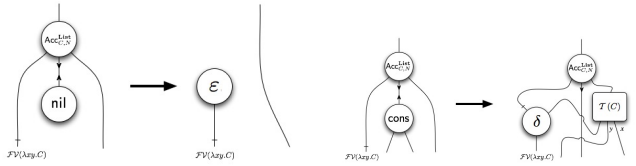
$$\frac{\Gamma \vdash t : \mathbf{List}(\tau) \qquad \Gamma \vdash \lambda xy.C : \sigma \to \tau \to \sigma \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathsf{acclist}\,(\lambda xy.C)\,N\,t : \sigma}$$

$$\begin{aligned} \mathsf{acclist}\,(\lambda xy.C)\,N\,\mathsf{nil} &\longrightarrow N \\ \mathsf{acclist}\,(\lambda xy.C)\,N\,\mathsf{cons}(h,t) &\longrightarrow \mathsf{acclist}\,(\lambda xy.C)\,C[N/x, h/y]\,t \end{aligned}$$

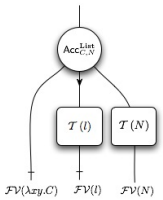
The function $\mathcal{S}\,(\cdot)$ is extended to create accumulator symbols as follows.

$$\begin{aligned} &\mathcal{S}\,(\mathsf{acclist}\,(\lambda xy.C)\,N\,l) = (\{\mathsf{Acc}^{\mathbf{List}}_{C,N}\} \cup \Sigma, R_{\mathsf{Acc}^{\mathbf{List}}_{C,N}} \cup R) \\ &\quad \text{where } (\Sigma, R) = \mathcal{S}\,(l) \cup \mathcal{S}\,(C) \cup \mathcal{S}\,(N) \end{aligned}$$

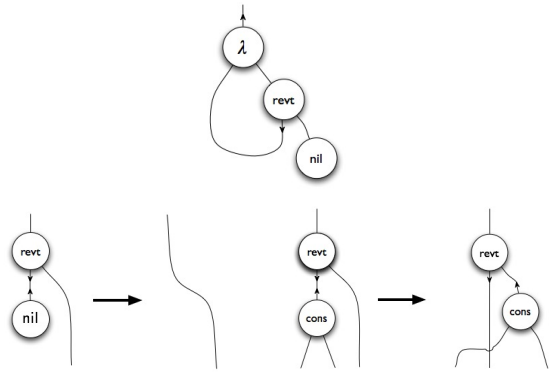
and $R_{\mathsf{Acc}^{\mathbf{List}}_{C,N}}$ consists of the following rules:



The translation is then extended as follows (we exemplify this in the setting without tokens for the sake of simplicity). $\widehat{\mathcal{T}}\,(\mathsf{acclist}\,(\lambda xy.C)\,N\,l)$ is defined to be the net



We remark that in the reduction rules for $\mathsf{acclist}\,(\lambda xy.C)\,N\,l$ the second argument N is not fixed throughout iteration; as such it cannot be internalized as part of the definition of the agent $\mathsf{Acc}^{\mathbf{List}}_{C,N}$. Instead the corresponding net is connected to an auxiliary port in that agent. The second version can now be written $\mathsf{revt} = \lambda l.\mathsf{acclist}\,(\lambda xy.\mathsf{cons}(y,x))\,\mathsf{nil}\,l$, and $\widehat{\mathcal{T}}\,(\mathsf{revt})$ is the following net, where revt stands for $\mathsf{Acc}^{\mathbf{List}}_{\mathsf{cons}(y,x),\mathsf{nil}}$, with the rules below.



8 Conclusions and Future Work

We have presented an approach to encoding in interaction nets functional programs defined with recursion operators, and given the full details of the application of this approach to the token-passing implementation of a normal-order language, which results in a very convenient visual notation for this language.

The approach can be extended to other base encodings; to richer sets of inductive types; and to other recursion operators. We are currently working on the encoding of a fixpoint operator for general recursion in the token-passing setting, and also on the encoding of co-iterator operators.

We are also in the process of developing a tool for visual functional programming. The tool consists of an evaluator for interaction nets together with a compiler module that translates programs to nets. It will allow users to type in a functional program, visualize it, and then follow its evaluation visually step by step. Initially the implementation will be based on the token-passing encodings with call-by-name, call-by-value, and call-by-need, for which we will adapt the encoding of the λ-calculus given in [16].

A different line of work is inspired by work of the datatype-generic programming community and the school of program calculation [2]. This prompts the investigation of visual fusion laws for instance, which is a topic closely related to notions of contextual equivalence for interaction nets [4].

References

1. A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages*, volume 45 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.
2. R. Bird and O. de Moor. *Algebra of Programming*. Prentice Hall, 1997.
3. L. Dami and D. Vallet. Higher-order functional composition in visual form. Technical report, 1996.
4. M. Fernández and I. Mackie. Operational equivalence for interaction nets. *Theoretical Computer Science*, 297(1–3):157–181, February 2003.
5. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.
6. G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*, pages 15–26. ACM Press, Jan. 1992.
7. K. Hanna. Interactive Visual Functional Programming. In S. P. Jones, editor, *Proc. Intnl Conf. on Functional Programming*, pages 100–112. ACM, October 2002.
8. J. Kelso. *A Visual Programming Environment for Functional Languages*. PhD thesis, Murdoch University, 2002.
9. Y. Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, Jan. 1990.
10. S. Lippi. Encoding left reduction in the lambda-calculus with interaction nets. *Mathematical Structures in Computer Science*, 12(6):797–822, 2002.
11. I. Mackie. The geometry of interaction machine. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, pages 198–208. ACM Press, January 1995.
12. I. Mackie. YALE: Yet another lambda evaluator based on interaction nets. In *Proceedings of the 3rd International Conference on Functional Programming (ICFP'98)*, pages 117–128. ACM Press, 1998.
13. I. Mackie. Efficient λ-evaluation with interaction nets. In V. van Oostrom, editor, *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA'04)*, volume 3091 of *Lecture Notes in Computer Science*, pages 155–169. Springer-Verlag, June 2004.
14. H. J. Reekie. *Realtime Signal Processing – Dataflow, Visual, and Functional Programming*. PhD thesis, University of Technology at Sydney, 1995.
15. F.-R. Sinot. Call-by-name and call-by-value as token-passing interaction nets. In P. Urzyczyn, editor, *TLCA*, volume 3461 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2005.
16. F.-R. Sinot. Token-passing nets: Call-by-need for free. *Electr. Notes Theor. Comput. Sci.*, 135(3):129–139, 2006.

Testing Erlang Refactorings with QuickCheck

Huiqing Li and Simon Thompson

Computing Laboratory, University of Kent, UK
{H.Li, S.J.Thompson}@kent.ac.uk

Abstract. Refactoring is a technique for improving the design of existing programs without changing their behaviour. Wrangler is a tool built at the University of Kent to support Erlang program refactoring; the Wrangler tool is written in Erlang.
In this paper we present the use of a novel testing tool, Quviq QuickCheck, for testing the implementation of Wrangler. QuickCheck is a specification-based testing tool for Erlang. With QuickCheck, programs are tested by writing properties in a restricted logic, and using the tool these properties are tested in randomly generated test cases.
This paper first gives overviews of Wrangler and Quviq QuickCheck, then discusses the various ways in which refactorings can be validated, and finally shows how QuickCheck can be used to test the correctness of refactorings in an efficient way.

1 Introduction

Refactoring [7] is a technique for transforming program source code in such a way that it changes the program's internal structure and organisation, but not external behaviour. The key characteristic that distinguishes refactoring from general code manipulation is its focus on structural change, strictly separated from changes in functionality. Functionality-preservation requires that refactorings do not introduce (or remove) any bugs. Refactorings typically have two apsects: *program analysis* is required to check that certain side-conditions are met by the program in question in order for the refactoring to preserve behaviour, and *program transformation* which carries out the actual program restructuring. In a slogan: *"Refactoring = Condition + Transformation"*.

Refactorings can be done manually, but this can be tedious and error-prone for small programs, and impractical for larger systesm. Software tools (*"refactoring engines"*) can help programmers perform refactorings automatically, and are available for a variety of languages, including Smalltalk, Java, C#, C++, Haskell and Erlang. With a refactoring tool, the programmer only needs to select which part of the program to be refactored and which refactoring to apply, and the tool will automatically check the side-conditions and apply the transformations throughout the whole program if the side-conditions are satisfied. Wrangler is the tool that we are implementing to support refactoring Erlang [1] programs, and this forms one aspect of 'Formally-Based Tool Support for Erlang Development'[1] [6], a joint research project between Universities of Kent and Sheffield.

[1] FORSE is supported by EPSRC, UK.

Implementing a practical and usable refactoring tool for a real world programming language is by no means trivial. A refactoring tool needs to get access to the program's syntax and static semantics (possibly including type information), to implement different kinds of program analysis and transformation, and to preserve the comments, and potentially, layout, of the transformed program. Among other criteria, such as efficiency, usability and completeness, the reliability of a refactoring tool is vital for it to be accepted in practice. A bug within a refactoring tool can introduce bugs in the refactored programs silently, and such bugs may be impossible to detect statically, if they result in a valid program which behaves differently from the original.

The correctness of refactorings implemented can be ensured from several aspects including, but not limited to, a clear specification clarifying the pre-conditions, transformation rules, and/or post-conditions of each refactoring; a verification that argues the correctness of the specification, and most importantly a thorough testing of the refactoring tool. A traditional way of testing refactoring tools is to create test cases manually. Each test case contains an input program, a refactoring command, and the expected result, which could be either the refactored version of the input program or the original input program (along with a failure message) depending on whether the side-conditions are satisfied. Then these tested cases are usually run with a Unit testing tool, such as EUnit [3] for Erlang. Writing test cases manually is tedious and hard to cover all possible refactoring scenarios. Incomplete test suite potentially leaves bugs in refactoring tools.

We present the technique of using Quviq QuickCheck [15], a tool developed by Quviq AB, to automate the testing of Wrangler. Instead of writing small test programs, we use real-world available Erlang programs as our refactoring input programs. For example, one of the Erlang programs we have used is Wrangler itself, which currently contains 25 modules. Quviq QuickCheck tests running code against formal specification, using controllable random test case generation combined with automated test case simplification to assist error diagnosis. With Quviq QuickCheck, we automate the generation of refactoring commands and the checking of refactoring outputs. Refactoring commands are generated randomly using the information stored in the annotated abstract syntax tree (AAST) of the input program. Along with the development of each refactoring, we write a collection of properties that the refactoring should satisfy. Failing to meet one or more of these properties indicates bugs in the implementation or properties. Each time the testing is run, it generates 100 random refactoring commands, applies each command to the input program, and checks that the properties being tested return true in every case. This way, we are able to integrate the specification and testing of refactorings very naturally.

The rest of the paper is structured as follows. In sections 2 and 3, we give introductions to Wrangler and Quviq QuickCheck. Section 4 gives an overview of the different ways in which refactoring engines can be tested, and in section 5 we explain our approach to testing Wrangler with QuickCheck, including the generation of refactoring commands, and the kind of general properties that we

use to test refactorings. In section 6, as an example, we illustrate the testing of *renaming a function*. In section 7, we give an evaluation of our approach; related work is presented in section 8, and conclusions are drawn in section 9.

2 Wrangler – An Erlang Refactorer

Wrangler [10, 11] is the tool that we are building to provide support for interactive refactoring of Erlang programs. The current version supports a small number of basic Erlang refactorings, including renaming variable, function and module names and generalisation of a function definition. More advanced refactorings are being added.

Wrangler is built on top of the Erlang `syntax-tools` package [13] which provides a representation of the Erlang AST within Erlang. `syntax-tools` allows syntax trees to be augmented with additional information as necessary. The Wrangler AST representation is annotated with a variety of information:

- Comments in the source code are inserted as attachments to the nodes in the AST at the appropriate place.
- Each function or variable name is associated with its actual source location and the location of its defining occurrence, thus reflecting the binding structure of the program.
- The start and end location of each syntactic entity in the source code is also stored in the augmented AST, allowing entities to be located by means of their position, as well as supporting pretty-printing facilities.
- Category information indicating the kind of syntax phrase the AST node represents, such as expression, function, pattern and so on is also included in the tree.
- Finally, free and bound variable information is also attached to the AST representation of each syntax phrase in the source code.

Wrangler is embedded in the Emacs editing environment; to manage communication between the refactoring engine and Emacs we make use of the functionalities provided by Distel [12], an Emacs-based user interface toolkit for Erlang,.

To perform a refactoring with Wrangler, the focus of refactoring interest has to be selected in the editor first. For instance, an identifier is selected by placing the cursor at any of its occurrences; an expression is selected by highlighting it with the cursor. Next, the user chooses the refactoring command from the *refactor* menu, and inputs the parameter(s) in the mini-buffer if required. The Wrangler tool checks that the focus item is suitable for the refactoring selected, that the parameters are valid, and that the refactoring's side-conditions are satisfied.

If all these checks are successful, then Wrangler will perform the refactoring, and update the program with the new result, otherwise it will give an error message, and abort the refactoring with the input program unchanged. The *undo* operation is supported by Wrangler; applying *undo* once will revert the program back to its state immediatlely before the last refactoring was performed.

Snapshots of Wrangler are given in Figures 1-2 with a particular refactoring scenario showing the generation of function `repeat/1` on the expression `io:format("Hello")`.

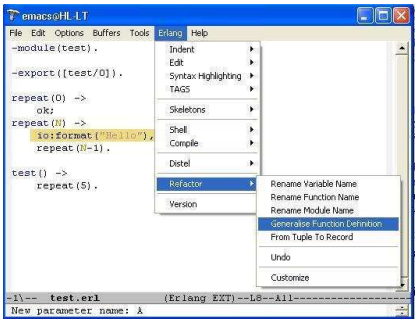


Fig. 1. A snapshot of Wrangler

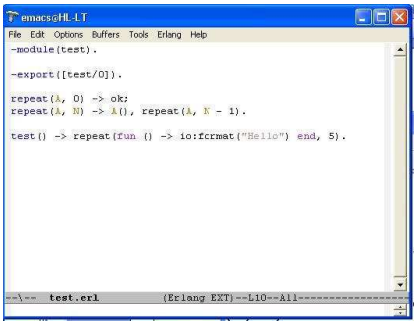


Fig. 2. A snapshot of Wrangler showing the result of generalisation

3 Quviq QuickCheck

Quviq QuickCheck is a property-based testing tool, developed from Claessen and Hughes' earlier QuickCheck tool for Haskell [4] re-designed for Erlang with a number of extensions, of which the most significant is an ability to simplify failing test cases automatically [15].

Quviq QuickCheck provides an API in Erlang that allows users to write *properties* that are expected to hold of programs; these properties are themselves expressed as Erlang source code. QuickCheck also defines a variety of *generators* and combining forms for generators by means of which the user can generate test data of the appropriate type and distribution for their needs..

As an example, consider the standard list reverse function. One property of this function is expressed thus:

```
prop_reverse() -> ?FORALL(Xs, list(int()),
                          list:reverse(list:reverse(Xs))== Xs).
```

As an abstract property, this says that reversing a list of integers twice has the result of returning the original list. In QuickCheck, the functions `int/0` and `list/1` are both data generators: `int/0` generates random integers, and `list/1` generates a list of elements generated by its argument. `?FORALL` is an Erlang macro: `?FORALL(X, Gen, Prop)` binds `X` to a value generated by `Gen` within the property `Prop`. The example property will be said to hold in QuickCheck if `list:reverse(list:reverse(Xs))== Xs)` holds for all values of `Xs` generated by `list(int())`.

The property is checked by running 100 random test cases generated by the generators, and reports success if all test pass. If any test case fails, the (first such) failing case will be printed. A failing test case indicates bugs in either the implementation under test or the written properties. For example, testing the following property against the standard function `lists:delete/2`

```
prop_list_delete()->
    ?FORALL(I, int(),
        ?FORALL(List, List(int()),
            not (lists:member(I, lists:delete(I, List)))))
```

against the standard function `lists:delete/2` might report

```
Failed! After 37 tests.
-8
[5, -8, 12, -8, 9]
```

as `lists:delete(I, List)` only removes the first occurrence of I in List. Once a counterexample has been found, the *shrinking* functionality provided by Quviq QuickCheck will allows QuickCheck to minimize the failing case as much as possible. For the above example, the length of the counterexample data will be reduced, and the output above would be augmented by

```
Shrinking......(6 times)
-8
[-8,8]
```

By writing properties in this style, a QuickCheck user can build up a formal specification, which is then checked against the implementation by QuickCheck. The mutual testing of implementation and specification ensures the correctness of both.

In comparison with traditional automated testing, as provided by systems such as EUnit [3], which runs the same set of tests repeatedly, QuickCheck allows the user to run many different test with little effort, therefore has the potential to find more bugs. It is, of course, possible to re-run tests simply by re-using a seed value within the random generation, and so to ensure that regression testing takes place if required.

The API provided by QuickCheck contains functions for generating both simple and complex test data, according to distributions described by the user, as well as providing macros for writing and testing properties. In the following sections, an explanation will be given when an API function or macro is used.

4 Validating Refactoring Engines

Refactorings and refactoring engines can be validated in a number of different ways. In this section we present an overview of the various approaches and their pros and cons, before explaining our approach in more detail in the next section.

In checking whether the result of a refactoring has preserved behaviour, the result naturally needs to compile and run without errors; in the remainder of this section we assume that the results are also checked for being compilable as well as being tested in various ways we discuss.

4.1 Regression testing of refactored programs

The most popular means of validating refactorings in current use is to ensure that refactored code meets all the tests that the original version met. As OO refactoring has been identified as one of the central characteristics of an extreme programming approach, it is reasonable to assume that the test data will already be in place, and so the advantage of this approach is that cost of testing the refactored code is small. This approach means that the refactored code has the same warranty as the original code.

The approach has two limitations. First, the coverage of the code is necessarily partial, and so it is possible that bugs have been introduced in the untested parts of the code. Also, the testing cost can be higher in cases where the test cases have themselves to be refactored: for instance, if a function is generalised, then it is necessary to add an extra datum to the test data for each function call.

4.2 Testing the old and new programs

A variant of the previous approach tests the two versions of the program against each other: on input data taken from an existing test suite, the outputs from two versions of the program can be compared directly. This approach is lower cost in the case where there is no pre-exisiting test data, since it is not necessary explicitly to state the output values corresponding to the various input data. A disadvantage is that any framework needs to accommodate the co-existance of two versions of the code under testt.

4.3 Programs as data

In contrast to the earlier approaches, it is possible to see the refactoring as a program, and so to supply it with a set of input programs and the corresponding output programs that are expect to result. Two variants of the check are possible:

- It is possible to analyse the abstract syntax tree (AST) resulting from the transformation, and to compare this with the expected result. This neglects the layout of the refactored program.
- In contrast, it is possible to specify the source code to be expected, with a given program layout. This is a stronger test than the former; since it not only prescribes the AST but also its particular layout, but this approach is appropriate when refactoring code is expected to be laid out in a way that will make it recognisable to its author.

This was the approach that we used first, using the Haskell package HUnit for testing HaRe (the Haskell refactorer) [9], and EUnit for testing Wrangler.

In our experience, the main disadvantage of writing test cases under this approach is that it is very tedious, and hard to cover all the refactoring scenarios especially when both the implementation and the test cases are written by the same people. Hence we did not gain sufficient assurance about the correctness of the refactorings implemented.

Other variants of this approach involve a degree of random generation; we will explore our particular approach in the next section, and discuss related work in section 8.

4.4 Program verification

Rather than using testing, it is possible to write formal proofs of correctness for refactoring engines. Two approaches suggest themselves:

- It is possible to produce, program by program, separate proofs of equivalence between the original and the refactored programs. Such proofs might be generated by tactic-based proof descriptions, or result from a proof planning process.
- Alternatively, the formal theorem proved can itself contain a quantifier over all programs of a certain form (which are the input to the refactoring in question). Preliminary work under this approach is to be found in Li's thesis [8] and the forthcoming thesis of Sultana [14].

This section has summarised various approaches to validating refactoring engines; we next look at our particular work.

5 Testing Wrangler with QuickCheck

Before adopting QuickCheck as the test engine of refactorings, we used the unit testing approach, as discussed in the previous section. We concluded that this mechanism was not ideal, and so to improve the testing of Wrangler, we have experimented with the idea of using Quviq QuickCheck as the test engine.

Under this approach, a collection of properties are written along with the implementation of each refactoring. These properties specify the conditions that must be met by the program after the refactoring, in order for the transformation to be behaviour-preserving. From the formal specification point of view, these properties can be viewed as the post-conditions of a refactoring. While there are some general properties which apply to most of the refactorings, for example, all the programs after a refactoring must compile successfully, some properties are particular to individual refactorings, especially those involving structural changes to the program. Writing properties along with the implementation of refactorings, we were able to make testing an integral part of the refactoring development process.

Properties are tested on the refactored version of the input program. While occasionally we have written a few small input programs to test a particular case, mostly we use real-world Erlang programs as the testing code base. Before the testing of a specific refactoring, the code base should be carefully examined to make sure that enough refactoring scenarios are covered in the program. For example, to test a refactoring involving the communication between processes, we should choose those programs that contain plenty use of process communications; and to test a refactoring that transforms a tuple to a record, we need to make sure that tuples and records are reasonably used in the test program.

Once the test program has been chosen, refactoring commands are automatically generated using the information stored in the annotated abstract syntax tree (AAST) of the test program. Each run of the testing generates 100 refactoring commands. Both the generation of refactoring commands and the creation of properties make use of the Wrangler infrastructure API, which provides programmer access to the infrastructure on which Wrangler is built. As the infrastructure has been more thoroughly tested, we trust its robustness in this exercise. Alternatively, we can also test an API function exposed by the infrastructure using the same approach.

More about the generation of refactoring commands and the creation of properties are discussed in the following two sub-sections. Following that, as an example, testing the *renaming a function* refactoring is examined in more detail.

5.1 Generation of Refactoring Commands

In Wrangler, a refactoring command normally contains the refactoring name, the name of the source file under refactoring, the focus of the refactoring which can be a location/range in the program source code, and some user inputs. For example, the refactoring *renaming a function* has the following interface:

```
rename_fun(FileName, SrcLoc={Line, Col}, NewName, SearchPaths)
```

where `FileName` is the name of the file containing the definition of the function to be renamed; `SrcLoc`, which is a tuple containing a line and a column number, represents the location of one of the occurrences of the function name in the source; `NewName` is the new function name, and `SearchPaths` specifies where to search for those files that could possibly use this function; this is needed when the function to be renamed is exported the module in which it is defined.

As another example, the refactoring *generalisation of a function definition* has the following interface:

```
generalise_fun(FileName, Range={StartLoc, EndLoc}, ParName)
```

where `FileName` is the name of the source file containing the definition of the function to be generalised; `Range` represents the start and end location of the selected expression in the source, and `ParName` is the new parameter name. A function can be generalised by making an identified sub-expression of its right-hand

side into a value passed into the function via a new formal parameter. In Wrangler, this refactoring only affects the current module, therefore the `SearchPaths` is not needed.

Next, we return to the 'renaming' example to explain how refactoring commands can be generated. If a specific file is used as the input program, then the `FileName` is fixed, otherwise a file can be randomly chosen from a directory for each refactoring command. Given a directory that contains some Erlang source files, the following function serves to select an Erlang file from it.

```
gen_filename(Dir) ->
    {ok,Files} = file:list_dir(Dir),
    ErlFiles = [F|| F <-Files, filename:extension(F)==".erl"],
    oneof(ErlFiles).
```

where the function `oneof/1` is a QuickCheck API function which generates a value using a randomly chosen element of a list of generators; in this example, all the list elements are constant generators.

Instead of generating source locations using the integer generators provided by Quviq QuickCheck, the value of `SrcLoc` is generated based on the location information stored in the AAST representation of the chosen Erlang file. As discussed earlier, in the AAST representation of an Erlang file, each occurrence of a function name is associated with its location in the source, the name of the module in which it is defined, as well as its defining location in that module.

To generate a source location, we first collect all those locations which are associated with the occurrences of the function names defined in this file, then choose one from the collection randomly. This way, we can make sure that selected location points to a function name defined in the current module. In order to test the case when the user deliberately points to a location in the source which does not correspond to a function name defined in the module, we can always add a fake location to the collection of real ones, or generate one at random. Again, the QuickCheck API function `oneof/1` is used.

Some refactorings ask the user to input a new name. For example, to rename a function, the user needs to input the new function name; and to generalise a function definition, the user has to input a new variable name. In order to cover test cases in which name conflict/shadow occurs, identifier names are generated from both pre-created names and those used in the program source.

The following function generates refactoring commands for *renaming a function.*

```
rename_fun_commands(Dir) ->
        ?LET(FileName, gen_filename(Dir),
        {FileName,
         oneof(collect_fun_locs(FileName)),
         oneof(collect_names(FileName)),
         Dir}).
```

In the above function, `Dir` specifies where to look for Erlang files to refactor; `?LET` is a macro provided by Quviq QuickCheck (`?LET(Pat, G1, G2)` generates a value from `G1`, binds it to `Pat`, then generates a value from `G2` which may refer to the variables bound in `Pat`); function `collect_fun_locs/1` adds all the locations where a locally defined function name occurs in the selected Erlang file to a list of default locations; `collect_name/1` adds all the function names that occur in the source to a list of pre-created identifiers, and as last, we assume that `Dir` is the only directory to search for those files that would possibly affected by the refactoring.

Suppose that the testing directory is `"c:/wrangler-0.1/test"`, which has three Erlang files, the following shows part of the refactoring commands generated by the above function in one run of QuickCheck.

```
1% {"test.erl",{3,1},module,"c:/wrangler-0.1/test"}
1% {"refac_rename_fun.erl",{243,64},halt,"c:/wrangler-0.1/test"}
1% {"refac_qc.erl",{184,48},ordsets,"c:/wrangler-0.1/test"}
1% {"test.erl",{5,39},"DDD","c:/wrangler-0.1/test"}
1% {"refac_qc.erl",{366,30},get_pos,"c:/wrangler-0.1/test"}
1% {"refac_rename_fun.erl",{117,33},purge_module,"c:/wrangler-0.1/test"}
```

As an example, the first command should rename the function whose name occurs at the location: {`line: 3, column: 1`} in file `test.erl` to the new name `module`, and search the directory `"c:/wrangler-0.01/test"` for files in which the function is used, if the function is exported. The percentage at the beginning of each line shows the proportion of the total represented by the command.

5.2 Properties

Formally specified or not, each refactoring comes with a set of pre-conditions, which embody when a refactoring can be applied to a program without changing its meaning; a set of transformation rules which states how the program should be transformed to fulfil the refactoring while keeping the program's semantics unchanged; and a collection of post-conditions which articulate some properties the program should hold after the refactoring has been done. While the pre-condition checking and transformation rules are always explicitly implemented, the checking of post-conditions are normally ignored by the developers of refactoring tools as we assume that the pre-conditions and transformation rules together should guarantee the post-conditions.

With the QuickCheck testing approach, we can test most of these post-conditions explicitly. Ideally, one post-condition that applies to any refactoring is that the input program and its refactored version should have the same semantics; however whether two programs have the same semantics is in general not decidable. Furthermore, even the two programs do have the same semantics, the refactor still might not have performed the anticipated change on the program correctly. For example, a buggy refactoring could return the input program unchanged

without an error message. Therefore, instead of checking two programs having the same semantics, we test a number of properties which are decidable.

There are a couple of basic properties that should hold by all the refactorings:

- first, the refactoring engine should not crash, i.e. the refactorer should not terminate with an uncaught exception;
- second, if the refactoring has finished without giving an error message, then the refactored version of the program should compile successfully (Wrangler only refactors programs that compile).

Refactorings are normally bi-directional. Given a refactoring that transforms a program from P to P', we can generally find another refactoring that transforms program P' to P. For example, renaming an entity in a program from A to B, then renaming it back to A, should produce the original program; as another example, first generalising a function definition over an expression, then specialising the function on the newly added parameter with the expression should always produce the original function. This feature of refactoring allows us to write properties that embody mutual testing of refactorings.

During the implementation of Wrangler, we always try to separate the precondition checking part from the transformation part. One of the benefits of doing this is that it allows the mutual testing of condition-checking and transformation. For example, performing the transformation with the knowledge that some of necessary side-conditions are not satisfied should either make the refactoring engine crash or violate some post-conditions in the case that the transformation (apparently) succeeds.

Apart from those general post-conditions that apply to most of refactorings, each refactoring also has it own particular post-conditions, especially those concerning structural change of the program, as different refactorings change the program structure in different ways. For some refactorings, there may also be special constraints that should hold during the transformation. For example, some refactorings are supposed to keep the program's module interface unchanged; while others are supposed to keep some particular function interfaces unchanged. All the constraints can be expressed as QuickCheck properties.

In the following section, we again take the *renaming a function* refactoring as an example to illustrate how properties can be specified and tested.

6 An Example: Testing *Renaming a Function*

Renaming a function is one of the most basic, but very useful, refactorings, supported by almost all the existing refactorers. This refactoring renames a user-selected function name to a new name and updates all the references to it. When the renamed function is exported by the module, this function could potentially affect every module in the program. Suppose the old and new function names (with arity) are `bar/n` and `foo/n` respectively, then the side-conditions on *renaming a function* are as follows.

1. The new name should be a lexically valid function name, otherwise the transformed program will not compile.
2. No binding for `foo/n` may exist in the same scope. This condition avoids *name conflict* in the scope where `bar/n` is defined, and violating this condition will result in the transformed program failing to compile.
3. No binding for `foo/n` may intervene between the binding of `bar/n` and any of its uses, and the binding to be renamed must not intervene between existing bindings and the uses of `foo/n`.
   This condition avoids *name capture*, and violating this condition will lead to the binding structure of the program being changed silently. ('Binding structure' here refers to the association of uses of identifiers with their definitions in a program, and is determined by the scope of the identifiers).
4. Callback functions should not be renamed.

To check the correctness of the implementation, we focus on the defining properties depending on whether the refactoring succeeds or not. If the refactoring completes without giving an error message, we then test the following properties.

- Renaming the new function back to its original name should affect the same set of Erlang files in the application, and produce the original program. This property also implies the condition that the refactored version of the program should compile without errors.
- The function-level binding structure of the refactored version of the program should be the same as, or isomorphic to, that of the original program.
  Unlike some functional languages that allow nested function definitions, Erlang has a very straightforward function defining structure. In Erlang, all named functions are top-level functions. The function-level binding structure of an Erlang program can be represented as a list of tuples:

  $$B = [\{\{M_1, Loc\}, \{M_2, Id, A\}\}]$$

  and $\{\{M_1, Loc\}, \{M_2, Id, A\} \in B$ if and only if the function name Id, which occurs in module M_1 at location Loc, refers to the function defined in M_2 whose name is Id and arity is A.
  Suppose the function `bar/1` defined in module `N` is renamed to `foo/1`, and the binding structures of the program before and after the refactoring are B and B' respectively, then replacing all the occurrences of $\{N, \texttt{foo}, 1\}$ in B' with $\{N, \texttt{bar}, 1\}$ should produce B.
- The programs before and after the refactoring should have the same set of callback functions if which functions are callback functions has been explicitly specified in the program.

If the refactoring fails because of one of the side-conditions fails, then the necessity of the side-condition can be also be tested. For example

- Transforming the program when the side-condtion 1 or 2 does not hold should produce a program that does not compile.

– Transforming the program when the side-condition 3 does not hold should produce a program that compiles but has a different function-level binding structure.

A simplified version of the top-level function for testing *renaming a function* is given in figure 3. To make it easier to read, we have omitted the part that handles client modules, however this should not affect the idea expressed by this function.

7 Evaluation of Approach

A number of other refactorings have been tested using this approach, including *renaming a variable name*, and *generalisation of a function definition*. We actually started to use Quviq QuickCheck after the first preliminary release of Wrangler, which was tested on a number of small test cases using EUnit, and was also manually tested on a large code base.

Even so four bugs were found within the first release of Wrangler in a short time. All these bugs escaped the pre-release testing due to the incomplete coverage of the testing suite. Among these bugs, one silently changed the binding structure of the program when the *generalisation* refactoring is applied, and was detected by the reversibility property we wrote for this refactoring; the other three bugs were all caught by the very basic properties, for example, one bug caused the refactoring engine to crash because of an unmatched case clause; and another caused the refactored code fail to compile because of the improper handling of generalisation on operators.

From our experience so far, the advantages of the QuickCheck approach are as follows:

– We are able to make the development of refactorings and their testing very closely integrated. The meaning of each refactoring was further clarified by the mutual testing of the implementation and the specification.
– Once properties have been written, many different test cases can be run with very little effort, instead of repeating the same set of tests cases every time. As any Erlang programs can serve as the test program, we can run the testing on as many test programs, especially large programs, as possible.
– Because of the random generation of refactoring commands, and the large amount of tests we can run, more refactoring scenarios will be covered, therefore increasing the possibility of finding more bugs. At this point, one might think of the exhaustive testing of refactorings. While it is possible to enumerate all the possible refactoring commands when the input program is very small, it is not practical with large input programs due to the huge amount of refactoring commands that could be generated.

While properties can be written separately from the implementation of refactorings, these properties normally make use of the infrastructure on which the refactorings are built, therefore familiarity with the infrastructure is essential for the testing using this approach.

```
qc_rename_fun(Dir) ->
  F = ?FORALL(C, (rename_fun_commands(Dir)),
  begin
    [FileName, SrcLoc, NewName, SearchPaths] = C,
      %% backup the current version of the program.
    file:copy(FileName, "temp.erl"),
      %% get the function name (with arity) to be renamed.
    Mod, FunName, Arity = pos_to_fun_name(FileName, SrcLoc),
      %% calculate the binding structure of the current program.
    B1 = fun_binding_structure(FileName),
      %% get the name of the callbacks functions if there is any.
    CallBacks = get_callback_funs(FileName),
      %% apply the refactoring command to the source.
    Res = apply(refac_rename_fun, rename_fun, C),
    case Res of
        %% ChangeFiles contains the names of those files
        %% that have been affected by this refactoring.
      ok, ChangedFiles ->  %% refactoring completed successfully.
        B2 = fun_binding_structure(FileName), %% new binding structure.
          %% get the name of the callback functions if there is any.
        CallBacks1 = get_callback_funs(FileName),
        C1 = [FileName,NewName, Arity, FunName, SearchPaths],
          %% rename the function back to its original name.
          %% we cannot use location as it might have been changed.
        ok, ChangedFiles1 = apply(refac_rename_fun, rename_fun_1, C1),
          %% property1: renaming in both directions affect the same set of files.
        prop1 = ChangedFiles == ChangedFiles1,
          %% property2: rename twice should returns to the original file.
        Prop2 = pretty_print(FileName) == pretty_print("temp.erl"),
          %% property 3: B1 and B2 are isomorphic.
          %% rename/3 replaces Mod, FunName, Arity with Mod, NewName, Arity in B1
        Prop3 = B2== rename(B1, Mod, FunName, Arity, Mod, NewName, Arity),
          %% property 4: the same set of callback functions.
        Prop4 = CallBacks == CallBacks1,
          %% recover the original program for the next refactoring command.
        file:copy("temp.erl", FileName),
        Prop1 and Prop2 and Prop3 and Prop4;
      error,ErrorMsg ->  %% refactoring failed with an error message.
          %% carry out the transformation even though the side-conditions
          %% do not held; do_rename_fun/4 transforms the program.
       _Res = apply(refac_rename_fun, do_rename_fun, C),
       case ErrorMsg of
         1, _R1 ->     %% failed for side-condition 1;
                       %% the transformed program should not compile.
                   file:copy("temp.erl", FileName),
                   error, _Reason = get_AST(FileName), true;
         2, _R2 ->     %% failed for side-condition 2;
                   file:copy("temp.erl", FileName),
                   error, _Reason = get_AST(FileName), true;
         3, _R3 ->     %% failed for side-condition 3;
                       %% the transformed program should compile, but the new
                       %% binding structure is not isomorphic to the original one.
                   ok, _AST = get_AST(FileName),
                   B2 = fun_binding_structure(FileName),
                   file:copy("temp.erl", FileName),
                   B2 /= rename(B1, Mod,FunName, Arity,Mod, NewName, Arity)
       end end end),
    qc:quickcheck(F).
```

Fig. 3. The top-level function for testing *renaming a function*

8 Related Work

A number of case studies regarding to using Quviq QuickCheck or its predecessor as the test engine have been done and reported, among which one to test an industrial implementation of the Megaco protocol, and faults that have not been detected by other testing techniques were found [2]. This case study also shows the power of shrinking provided by Quviq QuickCheck, and one example is that a test case consisting of a sequence of 160 commands was reduced to just seven. Because of the simplicity of the refactoring commands in our case, we have not exploited the benefit of shrinking so far.

The most closely related work on the automated testing of refactorings is the approach of Daniel *et. al.* [5]. The core of this approach is ASTGen, a library for generating abstract syntax trees (ASTs) for Java programs. ASTGen allows the developer to write *imperative generators* whose executions produce abstract syntax trees(ASTs) for refactoring engines. To test a refactoring, a developer writes a generator whose execution produces thousands of programs with structural properties that are relevant for the specific refactoring being tested. Several kinds of properties (oracles) have also been created to automatically check that the refactoring engine transformed the generated program correctly. Compared with approach, our approach is more lightweight, however a developer does need to make sure that the testing code base covers enough structure features and refactoring scenarios for the refactoring under testing.

9 Conclusion

Refactoring tools ought to allow program developers to quickly and safely refactor their program, especially large programs. However, a robust and safe refactoring tool is hard to develop, and most refactoring tools still contain bugs even after extensive testing. While unit testing does help to find bugs in refactoring tools, it is tedious to manually write test programs, and the coverage of the test cases is hard to guarantee, and it is even harder to test refactoring tools on large systems.

We have explored the idea of using Quviq QuickCheck to automate the testing of refactorings. In this approach, the correctness of refactorings is tested against specifications written in Erlang. Once a test program has been chosen, we automated the generation of refactorings commands and the checking of refactoring outputs. Within a short time, a number of bugs were found in the first release of Wrangler using this approach.

We envisage exploring a number of further ideas for automated testing of refactorings using QuickCheck.

– One of the options followed by Daniel *et. al.* in [5] is to compare the effect of two refactoring engines, namely Eclipse and NetBeans for Java. We will explore this option for Wrangler and the refactoring engine built by the group at Eötvös Loránd University, Budapest [11].

– We have not addressed the behaviour checking of programs; it would nevertheless be possible to extend our work to check the results of refactorings against their original version using randomly-generated input values.
– We have assumed the correctness of our infrastructure library; it would be instructive to express and then to test crucial properties of the functions in this library.

We also intend to provide an API to help the specification of properties in the context of refactorings, and we would also like to adopt this approach to test our Haskell refactoring tool, HaRe.

References

1. Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.
2. Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing Telecoms Software with Quviq QuickCheck. In Phil Trinder, editor, *Proceedings of the Fifth ACM SIGPLAN Erlang Workshop*. ACM Press, 2006.
3. Richard Carlsson and Mickaël Rémond. Eunit: a lightweight unit testing framework for erlang. In *ERLANG '06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, pages 1–1, New York, NY, USA, 2006. ACM Press.
4. Koen Claessen and John Hughes. QuickCheck: a Lightweight Tool for Random Testing of Haskell Programs.
5. Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *ESEC/FSE 2007: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, New York, NY, USA, September 2007. ACM Press.
6. FORSE. Formally-Based Tool Support for Erlang Development. http://www.cs.kent.ac.uk/projects/forse/.
7. Martin Fowler. Refactoring Home Page. http://www.refactoring.com.
8. Huiqing Li. *Refactoring Haskell Programs*. PhD thesis, Computing Laboratory, University of Kent, Canterbury, Kent, UK, September 2006.
9. Huiqing Li, Claus Reinke, and Simon Thompson. Tool Support for Refactoring Functional Programs. In Johan Jeuring, editor, *ACM SIGPLAN Haskell Workshop, Uppsala, Sweden*, August 2003.
10. Huiqing Li and Simon Thompson. A Comparative Study of Refactoring Haskell and Erlang Programs. In Massimiliano Di Penta and Leon Moonen, editors, *Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006)*, pages 197–206. IEEE, September 2006.
11. Huiqing Li, Simon Thompson, László Lövei, Zoltán Horváth, Tamás Kozsik, Anikó Víg, and Tamás Nagy. Refactoring Erlang Programs. In *The Proceedings of 12th International Erlang/OTP User Conference*, Stockholm, Sweden, November 2006.
12. Luke Gorrie. Distel: Distributed Emacs Lisp (for Erlang).
13. Richard Carlsson. Erlang Syntax Tools. http://www.erlang.org/doc/doc-5.4.12/lib/syntax_tools-1.4.3.
14. Nik Sultana. Verification of Refactorings in Isabelle/HOL. Master's thesis, Computing Laboratory, University of Kent, UK, September 2007.
15. Thomas Arts and John Hughes. Erlang/Quickcheck. In Ninth International Erlang/OTP User Conference.

Call Graphs, Dominator Trees, and Lambda Lifting

Marco T. Morazán and Ulrik P. Schultz

Seton Hall University, South Orange, NJ, USA
morazanm@shu.edu
University of Southern Denmark, Odense, Denmark
ups@mmmi.sdu.dk

Abstract. The process of lambda lifting flattens a program by lifting all local function definitions to the global level. Optimal lambda lifting computes the minimal set of extraneous parameters needed by each function as is done by the $O(n^3)$ equation-based algorithm proposed by Johnsson. In contrast, modern lambda lifting algorithms have used a graph-based approach to compute the set of extraneous parameters needed by each function. Danvy and Schultz proposed an algorithm that reduced the complexity of lambda lifting from $O(n^3)$ to $O(n^2)$. Their algorithm, however, is an approximation of optimal lambda lifting. Morazán and Mucha proposed an optimal graph-based algorithm at the expense of raising the complexity to $O(n^3)$. Their algorithm, however, suggested that dominator trees might be used to develop an $O(n^2)$ algorithm. This article explores the relationship between the call graph of a program, its dominator tree, and lambda lifting by developing algorithms for successively richer sets of programs. The result of this exploration is an $O(n^2)$ optimal lambda lifting algorithm.

1 Introduction

The process of lambda lifting flattens a program by lifting all local function definitions to the global level. In order to perform this program transformation the free variables of a function, f, and a subset of the free variables transitively needed by its callees, must be added as formal parameters to f before it can be lifted to the global level. That is, f must be made scope insensitive before it can be moved to the global level. Free variables must be explicitly passed to f, because at runtime the lifted version of f does not have the benefit of a closure to store the bindings of the free variables.

This program transformation technique is important for restructuring functional programs written for the web [7], for partial evaluators [1], and for efficient compilation [10]. Lambda lifting and its inverse lambda dropping [2] are also important for improving the performance of compiled programs by providing a mechanism through which the number of parameters of a function can be optimized for the target machine. For example, functions with a large number of parameters (which are handled poorly by most compilers) can be transformed to

have fewer parameters [2]. Danvy and Schultz also point out that in the context of teaching lambda lifting and lambda dropping are useful by offering different views of programs that help students understand lexical scoping and block structure [2].

The computation of the set of free variables needed by a lifted function makes lambda lifting difficult. Modern graph-based approaches [3, 9] tackle the problem by transforming the call graph of a program into a directed acyclic graph that is used to propagate free variables. The algorithm developed by Danvy and Schultz [3] improves the complexity of Johnsson's [8] lambda lifting algorithm from $O(n^3)$ to $O(n^2)$. Their algorithm, however, is not optimal because it may unnecessarily increase the arity of lifted functions. The algorithm developed by Morazán and Mucha [9] makes graph-based lambda lifting optimal at the cost of increasing its complexity to $O(n^3)$.

In this article, we first review Johnsson's (J), Danvy's and Schultz's (DS), and Morazán's and Mucha's (MM) lambda lifting algorithms. At the end of this review, we present a new insight that simplifies the presentation and the implementation of graph-based lambda lifting by using a depth-first traversal instead of a breadth-first traversal to propagate free variables. The article then explores the relationship between call graphs, dominator trees, and lambda lifting. The result of this exploration is an optimal $O(n^2)$ lambda lifting algorithm. Although the discussion is technically intricate at some points, the resulting algorithm is simple and elegant. The presentation of all algorithms assumes that all variable names are unique. Programs for which this does not hold can easily be transformed by generating a fresh identifier for repeated identifiers [4]. The article ends with some concluding remarks and directions of future work.

2 Lambda Lifting Algorithms

In this section, we first describe previous lambda lifting algorithms. We then present a new insight that simplifies the implementation of graph-based lambda lifting by using a depth-first traversal instead of a breadth-first traversal. The section ends with an illustrative example.

2.1 Johnsson's Algorithm

In the J-algorithm, the source program is traversed top-down to compute the required (i.e. minimal) set of extraneous parameters needed by each function. For any given function, f, the equation for the required set of free variables of f, R_f, is given by:

$$R_f = FV_f \cup ((\cup_{g \in FF_f} R_g) \cap SV_f)$$

where FV_f is the set of free variables directly referenced by f, FF_f is the set of functions referenced by f, and SV_f is the set of variables defined in f's enclosing scope. Mutually recursive functions give rise to a system of mutually recursive equations which is solved by traversing down the parse tree. Once R_f is known

it is used to compute the minimal set of free variables for functions declared further down the program's parse tree.

2.2 Danvy's and Schultz's Graph-Based Lambda Lifting

To perform lambda lifting in quadratic time, a program is represented as a call graph. Each node in this graph represents a function. An edge from f to g means that there is a reference to g in the body of f. Mutually recursive functions give rise to strongly connected components (akin to Johnsson's mutually recursive equations). Danvy and Schultz observed that a function, f, in a strongly connected component can be given as extraneous parameters the set of free variables lexically visible to f found in the union of the free variables of the functions that constitute the component. Therefore, strongly connected components can be coalesced in the call graph of a program to yield a directed acyclic graph that is traversed to propagate free variables between nodes.

2.3 Morazán's and Mucha's Graph-Based Algorithm

Morazán and Mucha observed that using strongly connected components to propagate free variables may result in an approximation of the required set of extraneous parameters needed by lifted functions. Unnecessary extraneous parameters may be added to lifted functions for two reasons. The first reason is that functions can be members of a strongly connected component that contains nested strongly connected components and that also contains functions defined at different levels in the program. Suppose f is defined at level n in the parse tree of a program and that there are m disjoint sets of functions (modulo f), $D_{1...m}$, defined at any level greater than n (i.e. in the parse tree of the program f is an ancestor of these functions) such that f dominates all paths from functions in D_i to functions in D_j, $i \neq j$. In such a scenario, f may declare variables that are free[1] for functions in D_i that are not needed as extraneous parameters by functions in D_j and viceversa. This may occur, for example, when f is contained in two independent loops (modulo f).

The second reason is that a variable, x, declared by f that is free in D_i may not be needed as an extraneous parameter by all the functions in D_i. For example, let f and g be members of the same loop such that x is known to be free in g and is declared by f. The variable x only needs to be carried by successors of g if there is a path, that does not contain f, from g to another function where x is directly referenced. This follows from the observation that the successors of g do not need to make x available to any other function if such a path does not exist. Thus, these successors do not require x as an extraneous parameter.

The MM-algorithm is an improvement of the DS-algorithm that reduces the arity of lifted functions by computing the minimal set of extraneous parameters needed by each lifted function, as is done by the J-algorithm, based on the observations above. Extraneous parameters are reduced by splitting the strongly

[1] We call such free variables *local* to the strongly connected component.

```
(define (f x)
  (define (g ...) (... (i...)...))
  (define (h ...) (...x...))
  (define (i ...(h...)...))
 (...(g...)...(h...)...))
```

Fig. 1. Sample Scheme-like Code.

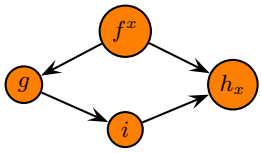


Fig. 2. Call Graph for Figure 1.

connected components of a call graph that contain functions defined at different levels in the program's parse tree into multiple components based on its nested strongly connected components and by ignoring edges into a dominating function that are internal to any such component after the split. Splitting strongly connected components into multiple components guarantees that free variables local to a component (e.g. declared by the dominating function) do not propagate between nested strongly connected components. Ignoring internal edges into the dominating function of a nested strongly connected component guarantees that a free variable local to a component is not propagated beyond the last function that references it in a loop. This occurs, because the removal of such edges eliminates the loop and, therefore, these functions no longer constitute a strongly connected component and do not have to receive the same set of extraneous parameters.

2.4 A Simplifying Insight

The graph-based lambda lifting algorithms developed to date use the reversed breadth-first ordering of the nodes of an acyclic graph to ensure that a node is only processed once all of its successors in the call graph have been processed. Successor nodes must be processed first, because the required set of free variables of predecessor nodes depends on them. The use of this ordering, however, requires that special attention be paid to calls from functions appearing late in the breadth-first ordering to functions appearing early in the breadth-first ordering.

To illustrate the problem consider the Scheme-like code in Figure 1 and its diamond-shaped call graph in Figure 2. In this graph the function f declares x (noted as right superscript) and x is free in h (noted as a right subscript). The breadth-first ordering of the nodes is: $\{f, g, h, i\}$[2]. There are no strongly connected components and, thus, nothing to coalesce. Having an acyclic graph means that free variables ought to be propagated from callees to callers in a reversed breadth-first order. For our example that order is: $\{i, h, g, f\}$. If free variables are simply propagated from callees to callers nothing propagates from i to g, from h the free variable x propagates to i and nothing propagates to f, and nothing propagates from g to f. The end result would be that x is identified as a free variable for h and i, but not for g which also needs x as a free variable.

[2] The breadth-first ordering could also be $\{f, h, g, i\}$, but this is irrelevant for our purposes.

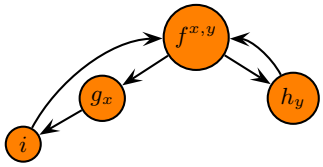


Fig. 3. Strongly Connected Component Functions Require Different Free Variables.

To avoid this pitfall, the DS-algorithm unifies local free variables with those of the immediate successors in the graph. Thus, x propagates from i to g when g is processed.

We observe that if the graph is acyclic, as is always the case after coalescing strongly connected components, then a depth-first traversal of the graph can be used to propagate free variables: every time the process pops back from a node to its antecessor free variables are propagated. This ensures that all successors are processed before a caller is processed. For the call graph in Figure 2, a depth-first traversal follows the path $f \rightarrow g \rightarrow i \rightarrow h$. The free variable x is propagated back through this path from h to i and finally to g. The depth-first traversal then proceeds down the path $f \rightarrow h$ and nothing is propagated from h to f before terminating. Although the result is the same as using the reversed breadth-first ordering, this process is more elegant and simplifies the implementation of lambda lifting.

Propagation using a depth-first traversal instead of a reversed breadth-first ordering is still proportional to the number of function calls and the number of declared variables in the program. Therefore, the DS-algorithm is still $O(n^2)$ when using a depth-first traversal for propagation.

2.5 An Illustrative Example

Consider the call graph displayed in Figure 3 in which each node is labeled with the name of a function. As before, the superscript at the right of each function is the set of variables declared by the function and the subscript at the right of each function is the set of free variables referenced by the function. Assume that g, h, and i are local to f.

J-Algorithm The free variables of each function are computed as the lambda lifting process descends down the parse tree of the program. First, at the top-most level of the parse tree, the free variables of f are computed by solving the following equation:

$$R_f = FV_f \cup \ ((\cup_{g \epsilon FF_f} \ R_g) \ \bigcap \ SV_f)$$

Since $FV_f = SV_f = \emptyset$, we may conclude that $R_f = \emptyset$.

At the next level of the parse tree, the free variables equations to solve are:

$$
\begin{aligned}
R_g &= FV_g \cup \ ((\cup_{j \epsilon FF_g} R_j) \cap SV_g) \\
&= \{x\} \cup \{R_i \cap \{x, y\}\} \\
&= \{x\} \cup \{\{FV_i \ \cup ((\cup_{j \epsilon FF_i} R_j) \cap SV_i)\} \cap \{x, y\}\} \\
&= \{x\} \cup \{\{\emptyset \cup \{R_f \cap \{x, y\}\}\} \cap \ \{x, y\}\} \\
&= \{x\} \cup \ \{\emptyset \cup \{\emptyset \cap \{x, y\}\} \cap \ \{x, y\}\} \\
&= \{x\} \\
R_h &= FV_h \cup ((\cup_{j \epsilon FF_h} R_j) \cap SV_H) \\
&= \{y\} \cup \{R_f \cap \{x, y\}\} \\
&= \{y\} \cup \{\emptyset \cap \{x, y\}\} \\
&= \{y\} \cup \emptyset \\
&= \{y\} \\
R_i &= FV_i \cup ((\cup_{j \epsilon FF_i} \ R_j) \cap SV_i) \\
&= \emptyset \cup \{FV_f \cap \{x, y\}\} \\
&= \emptyset \cup \{\emptyset \cap \{x, y\} \\
&= \emptyset \cup \emptyset \\
&= \emptyset
\end{aligned}
$$

Notice that x is not identified as an extraneous parameter for h and that y is not identified as an extraneous parameter for g and i. Furthermore, x is not identified as an extraneous parameter for i. This occurs, because the extraneous parameters needed by f, an ancestor of g, h, and i in the program's parse tree, are computed before the extraneous parameters needed by g, h, and i. Thus, the members of FF_f are not explored during the computation of R_g, R_h, and R_i and do not contribute extraneous parameters to g, h, and i.

DS-Algorithm The DS-algorithm coalesces the strongly connected components in the call graph of a program. For the call graph in Figure 3 this means that all the functions are coalesced into one node. The union of all the free variables of the functions in the node (i.e. f, g, h, and i) is taken. For each function, the lexically visible variables in this union become parameters to the lifted functions. That is, $\{x, y\}$ are identified as extraneous parameters for g, h, and i.

In contrast with the results obtained with the J-algorithm, notice that more extraneous parameters are identified for g, h, and i. This occurs because the coalesced strongly connected component has a function (i.e. f) that declares variables that are free in other functions in the strongly connected component and that dominates all the paths between mutually exclusive subsets of functions (i.e. $\{g, i\}$ and $\{h\}$). Coalescing such a strongly connected component is equivalent to not computing the extraneous parameters of *ancestor* functions in the parse tree before computing the extraneous parameters of any descendants as is done in the J-algorithm. Thus, the descendants of an an ancestor function can all unnecessarily contribute extraneous parameters to each other.

MM-Algorithm If a strongly connected component contains functions defined at different levels in the parse tree, the MM-algorithm splits strongly connected components based on nested strongly connected components and ignores edges to the dominating function of the strongly connected component. The call graph in Figure 3 is split into two components displayed in Figure 4.

Fig. 4. MM-algorithm components for the call graph in Figure 3.

This disconnected graph is used to propagate free variables. Notice that the dominating ancestor function, f, is a member of two components, thus, preventing its descendants from unnecessarily contributing free variables to each other. By ignoring the edges into f in Figure 3, the nested strongly connected components cease themselves to be strongly connected. During the depth-first traversal for the propagation of free variables, y is not unnecessarily propagated to g and i, and x is not unnecessarily propagated to h and i. Notice that within the loop formed by $\{f, g, i\}$ only g requires and receives x as an extraneous parameter. This algorithm yields the same results as the J-algorithm.

3 Call Graphs and Dominator Trees

The key lessons that must be highlighted from the previous section are:

1. **J-algorithm**: The set of extraneous parameters for an ancestor function in a parse tree must be known before finalizing the computation of the set of extraneous parameters for any descendant function.
2. **DS-algorithm**: Lambda lifting can be done using a graph-based approach. Furthermore, functions in a strongly connected component of a call graph that do not have references to any free variables local to the component can be coalesced. These functions all require the same set of extraneous parameters.
3. **MM-algorithm**: Strongly connected components of a call graph that have an ancestor function that dominates the paths between disjoint sets of functions in the component must be split in order to avoid descendant functions from unnecessarily contributing free variables to each other. Furthermore, simple loops on a dominating function must be dissolved in order to avoid unnecessary propagation of free variables.
4. **New Observation**: Once an acyclic graph is obtained for a graph-based approach a depth-first traversal can be used to simplify the process of propagating free variables.

The MM-algorithm repeatedly computes strongly connected components in order to avoid the unnecessary propagation of local free variables. The splitting of a strongly connected component is always done around an ancestor function that dominates all paths between disjoint sets of functions within the strongly connected component when the strongly connected component contains functions defined at different levels in the parse tree of the program. This observation suggests that dominator trees can be used to perform lambda lifting.

```
(define (f x y z)
  (define (g a b)
    (define (h c d)
      (define (i e) (...(h e e)...(g e e)...))
      (...(i (+ c d))...))
    (...(h a b)...))
  (...(g (* x y) z)...))
```

Fig. 5. Code Fragment to Illustrate Lowest Upward Dependence

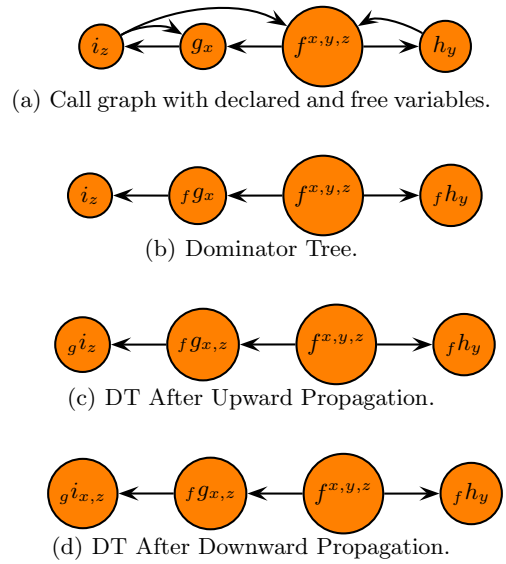
A defining property of a dominator tree is that an ancestor function always appears before its descendants. Thus, a dominator tree tells us for which functions the complete set of free variables must be computed first. For our purposes, an interesting feature of the dominator tree of a call graph is that independent loops dominated by a function are represented as different branches out of the dominating function which precludes the need to dissolve simple loops. Dominator trees, therefore, can be used as the basis of a graph used to propagate free variables. Since dominator trees can be computed in linear time [11], the need to repeatedly compute strongly connected subcomponents, which makes the MM-algorithm cubic, can be eliminated to reduce the complexity of lambda lifting.

The dominator tree, however, does not capture all dependencies between functions needed for lambda lifting. We classify these missing dependencies as *vertical* and *horizontal* dependencies, described in Sections 4 and 5 respectively. Vertical dependencies capture dependencies arising due to recursion between ancestors and descendants in the dominator tree. Horizontal dependencies capture dependencies arising between functions that do not have a vertical dependence in the dominator tree. Vertical dependencies are annotated on the dominator tree and are used to drive the propagation of free variables throughout the tree. Horizontal dependencies are added to the dominator tree, which necessitates coalescing the strongly connected components to obtain a directed acyclic graph. The resulting coalesced graph is used to make all functions scope insensitive by propagating free variables using a depth-first traversal.

4 Vertical Function Dependencies

We define a *downward* vertical dependence as the dependence that exists between a function and a descendant. A call to any local function, g, must be preceded, at some point during the computation, by a call to g's ancestors in the dominator tree of the program. Any extraneous parameters that g contributes to its ancestors can be propagated up the dominator tree.

We define an *upward* vertical dependence as the dependence that exists between a function, g, and a function, f, which is an ancestor of g. The function g may depend on several of its ancestors in the dominator tree of which we are interested in the one that has the maximum depth. We define the lowest upward



(a) Call graph with declared and free variables.

(b) Dominator Tree.

(c) DT After Upward Propagation.

(d) DT After Downward Propagation.

Fig. 6. Call Graph, Dominator Tree, and Propagation Steps.

vertical dependence of g, LD_g, as the function with the maximum depth in the dominator tree that g depends on. LD_g, if it exists, is the ancestor of g with the maximum depth that is either called by g or is reachable from any of g's descendants in the dominator tree. For example, consider the fragment of code in Figure 5. The dominator tree for this code is simply the chain: $f \to g \to h \to i$. Observe that i calls two of its ancestors, g and h, in the dominator tree. Since h has the maximum depth, we have that $LD_i = h$. The function h does not call any of its ancestors, but it depends on its ancestor g which is called from i. Since g is the only ancestor of h that is reachable from the subtree rooted at h (i.e. $h \to i$), we have that $LD_h = g$. Finally, $LD_g = LD_f = \emptyset$ because none of the ancestors of g or f are reachable from the subtrees of the dominator tree rooted at these functions.

To start exploring lambda lifting algorithms let us restrict our observations to the class of programs in which all dependencies are vertical (this restriction will be removed in the next section). Upward vertical dependence is not captured by a dominator tree, but can be computed, for example, as free variables are propagated up the dominator tree to identify the extraneous parameters contributed by downward vertical dependence. Along with free variables, the set of reachable ancestor functions can be propagated up the dominator tree.

Clearly, all extraneous parameters for g contributed by its descendants will reach g during an upward propagation. The following theorem establishes that LD_g, if it exists, contains all the extraneous parameters needed by g that are contributed by its ancestors in the dominator tree.

Theorem 1. *The set of extraneous parameters needed by LD_g contains all the extraneous parameters needed by g from its ancestors.*

Proof. Let DT be the dominator tree for a call graph, CG, and let h be LD_g. Assume x is an ancestor-contributed extraneous parameter needed by g that is not a member of the set of extraneous parameters needed by h. If x is defined by an ancestor of h, then x must be a member of the set of extraneous parameters needed by h which contradicts our assumption. This follows from observing that h must carry x in order to make it available to g. If x is defined by h or a descendant of h, then there must exist a path in CG from g to a function where x is a known free variable that does not contain the function that declares x. All the functions on this path must be descendants of h which means that $LD_g \neq h$. This contradicts our assumption and completes the proof that x must be a member of the set of extraneous parameters needed by h. Q.E.D

To illustrate how vertical dependencies can be used in lambda lifting consider Figure 6. Figure 6(a) displays a call graph annotated with variable declarations and free variable references and Figure 6(b) displays its dominator tree. Free variables needed by functions due to downward vertical dependence can be propagated up the dominator tree using a depth-first traversal. After this is done, the variable z has been propagated from i to g. In addition during this propagation step, the LD_i of each function i is computed by also propagating relevant upward vertical dependencies. LD_i is g and LD_h is f, because these are leaves with no successors and their LD is simply the ancestor that they directly reference. Nodes pass the set of reachable ancestors back up the tree along with their free variables. In this manner, LD_g becomes f as it is the ancestor of g with the largest depth that is reachable form the subtree rooted at g. The result of this step is displayed in Figure 6(c) in which the subscript to the left of each function name is its lowest dependence function. Finally, free variables need to be propagated down the dominator tree to satisfy upward vertical dependencies. This propagation proceeds in a breadth-first order propagating to function i the free variables needed by LD_i. A breadth-first order propagation is required to guarantee that the extraneous parameters of ancestor functions are known before the extraneous parameters of any descendant function are computed (which satisfies the key lesson highlighted from the J-algorithm). During this step the variable x is propagated from g to i. The result of this propagation step is displayed in Figure 6(d).

5 Horizontal Function Dependencies

A function may not only depend on functions that are its ancestors and its descendants in the dominator tree. For example, siblings in the dominator tree may call each other. We define a *horizontal* dependence as a reference to a function that is not an ancestor or a descendant in the dominator tree. The free variables of a horizontal dependence must also be propagated from the callee to the caller. Horizontal dependencies are not captured by the dominator tree

of a call graph and, thus, a dominator tree must be augmented into a graph to capture horizontal dependencies.

To convert a dominator tree into a graph that captures horizontal dependencies, the dominator tree is augmented with the edges between functions in the call graph that do **not** have a vertical dependence. We call this graph an EDT (**E**xtended **D**ominator **T**ree) graph and the new edges are called lateral edges. If the resulting EDT graph does not contain any cycles then it only has *simple* horizontal dependencies. Otherwise, it has *complex* horizontal dependencies. Clearly, the EDT graph for a program that only has functions with vertical dependencies is its dominator tree.

First, we highlight some important properties of EDT graphs. Second, we extend our lambda lifting algorithm to handle the class of programs that have simple horizontal dependencies. Finally, we extend our lambda lifting algorithm to handle arbitrary programs that may contain complex horizontal dependencies.

5.1 Important Properties of EDT Graphs

Formally, the set of lateral edges, E_l, in an EDT graph formed from the dominator tree, DT, is defined as:

$$E_l = \{(f, g) | f \text{ is not the parent of } g \text{ in } DT \wedge g \text{ is not an ancestor of } f \text{ in } DT\}.$$

The set E_l endows the EDT graph with important properties outlined by the following theorems. After establishing the validity of these properties we will point out their significance for lambda lifting.

Theorem 2. *If (f,g) $\in E_l$, then the parent of g, p_g, in the dominator tree, DT, dominates f.*

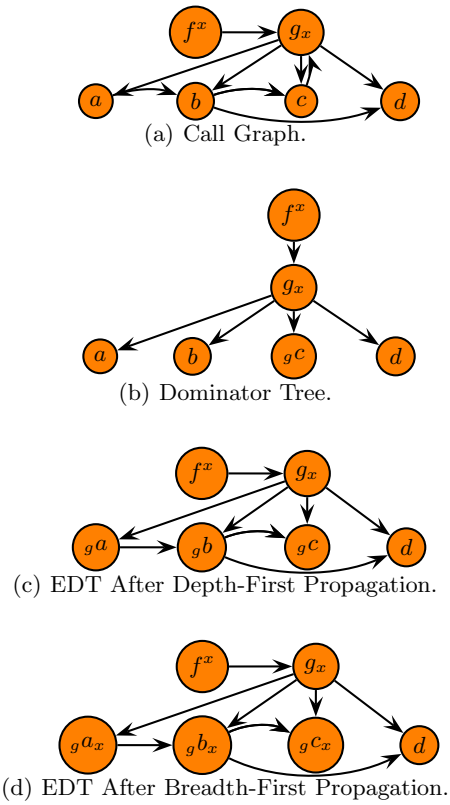
Proof. Let G be the EDT graph obtained by only extending DT with the lateral edge from f to g and let r be the root function of DT. If there is a path in G from r to g that contains f and that does not contain p_g, then p_g does not dominate all paths from r to g. This means that DT can not be the dominator tree which contradicts our assumption. Q.E.D.

Having established that the parent of the called function for a lateral edge in the EDT graph dominates the caller, we can now establish that all the ancestors of the called function dominate the caller. The proof simply exploits the fact that domination is a transitive property.

Theorem 3. *If (f,g) $\in E_l$, then all ancestors of g in the dominator tree, DT, dominate f.*

Proof. Theorem 2 establishes that the parent of g dominates f. All other ancestors of g dominate its parent. Therefore, all of g's ancestors dominate f. Q.E.D.

The significance of Theorems 2 and 3 for lambda lifting is that the existence of a lateral edge from f to g in an EDT graph means that LD_g, if it exists, dominates f. Therefore, LD_g may also be LD_f. This occurs when none of the nodes in the dominator tree path from the parent of g to f are LD_f. This means that in addition to free variables, LD information must be propagated from callees to callers across lateral edges.



(a) Call Graph.

(b) Dominator Tree.

(c) EDT After Depth-First Propagation.

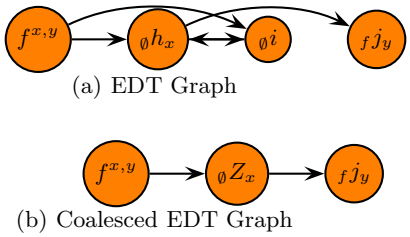
(d) EDT After Breadth-First Propagation.

Fig. 7. Call Graph and Dominator Tree with Simple Horizontal Dependence

5.2 Simple Horizontal Dependencies

When an EDT graph has only simple horizontal dependencies it suffices to first propagate free variables and lowest dependence information between functions using a depth-first traversal (akin to propagating up the dominator tree) and then to propagate free variables in breadth-first order exploiting lowest dependence information (akin to propagating down the dominator tree). The correctness of the first propagation follows from observing that free variables are propagated from callees to callers and from Theorem 3 that guarantees lowest dependence information can safely be propagated across lateral edges.

To illustrate the use of horizontal dependence information in the absence of strongly connected components consider the call graph in Figure 7(a). The dominator tree for this graph is displayed in Figure 7(b). Extending the dominator tree with edges between functions that do not have a vertical dependence results in the original call graph without the edge from c to g. Figure 7(c) displays the results of propagating free variables and LD information after a depth-first



(a) EDT Graph

(b) Coalesced EDT Graph

Fig. 8. EDT with a Strongly Connected Component Formed by Simple Lateral Edges.

traversal. No free variables propagate between the functions in this step, but LD_c, g, propagates to become LD_b and LD_a. Figure 7(d) displays the results of propagating free variables in a breadth-first order by exploiting LD information. Each function receives the free variables of its LD function.

5.3 Complex Horizontal Dependencies

The augmentation of the dominator tree, however, may lead to an EDT graph that is no longer acyclic. That is, the resulting graph may contain strongly connected components. This occurs, for example, when two siblings in the dominator tree are mutually recursive. In the presence of strongly connected components, it no longer suffices to simply propagate free variables and LD information using a depth-first traversal. The problem is that such a traversal does not guarantee that all successors of a node are processed first.

Strongly connected components must be coalesced, but as learned from the MM-algorithm sets of functions that include a dominating function and the functions it dominates should not be coalesced. That is, functions that have a vertical dependence should not be coalesced. This observation suggests that within a strongly connected component only functions at the same level in the dominator tree can be coalesced together. Notice that a function at level n in the dominator tree can not declare any variables that are free in other functions at level n. This means that it is safe to coalesce these functions together, because none of these functions will unnecessarily contribute free variables to each other.

The goal, therefore, is to coalesce strongly connected components in an EDT graph without loosing vertical dependence information. To achieve this it is helpful to distinguish between two types of edges in an EDT graph. The first kind of edge is a *simple lateral edge* which occurs between functions at the same level of the dominator tree. These edges can only occur between siblings. Any strongly connected components formed solely by simple lateral edges can be coalesced in the EDT graph, because among the siblings in each component there is no dominating function. If an EDT graph is created by solely adding simple lateral edges to the dominator tree, then after coalescing strongly connected components the EDT graph is acyclic. Thus, lambda lifting can proceed as described in section 5.2 by making a coalesced node's free variables the union of the free variables

of the functions in the strongly connected component and by making the node's LD function be the $max_f(LD_g)$, where g is a function in the strongly connected component. To illustrate this concept consider the EDT graph displayed in Figure 8(a). The functions i and h have no known upward vertical dependencies at this time, LD_j is f, and x is free in h. The strongly connected component formed by $\{h, i\}$ can be coalesced into a node, say, Z. The set of free variables of Z is $\{x\}$ and LD_Z is $\emptyset$. The result of this transformation is displayed in Figure 8(b). The computation of required free variables can now proceed as described in section 5.2. After the depth-first propagation the set of free variables of Z is $\{x, y\}$ and $LD_Z = f$. Nothing propagates during the breadth-first propagation (because f has no free variables). After the propagation steps, we have that $\{x, y\}$ are the required free variables for h and i which is precisely what is needed.

The second kind of edge is an *upward lateral edge* which exists between functions at different levels of the dominator tree. These edges always occur from a function, g, at level n to a function, f, at level $n - i$, where $i \geq 1$, such that f is not an ancestor of g in the dominator tree[3]. The existence of such an edge, means that g needs the free variables of f. Notice, however, that f may not need all of g's free variables despite being in the same strongly connected component. The free variables of g not needed by f are those that are local to the strongly connected component and that are not lexically visible nor declared by f. All of these variables must be declared by a function with a depth greater than or equal to the depth of f in the dominator tree. Notice that this set of functions may include siblings of f in the dominator tree. There is no dominating function between siblings. An incoming upward lateral edge means that these siblings need the same free variables. This follows from observing that they all need as free variables the variables declared by common ancestors in the dominator tree that are free in the strongly connected component. Therefore, we have that the siblings of a function in the dominator tree, like f that has an incoming upward lateral edge, that are in the same strongly connected component can be coalesced with f without local free variables being unnecessarily propagated during lambda lifting. Coalescing these strongly connected components in this manner preserves vertical dependence information and provides a directed acyclic graph that can be used to compute the free variables needed by each function in an arbitrary program.

To illustrate the use of horizontal dependence information in the presence of strongly connected components created by upward lateral edges consider the call graph in Figure 9(a). Its dominator tree is displayed in Figure 9(b). The first step is to extend the dominator tree with simple lateral edges. The resulting graph is displayed in Figure 10(a). Five simple lateral edges have been added to extend the dominator tree. These additions have formed a strongly connected component that contains the functions j and k. These functions are coalesced to form a new node S. The set of free variables for S is obtained from the union of the free variables of j and k. The set of variables declared by S is obtained from the union of the declarations of j and k. The resulting graph is displayed

[3] There can not exist any edges in the other (i.e. downward) direction from f to g.

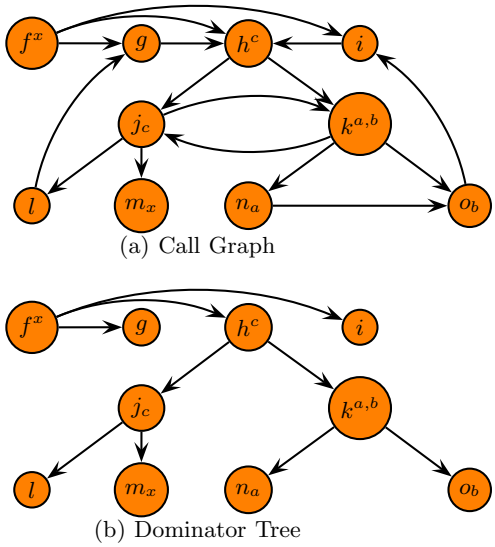


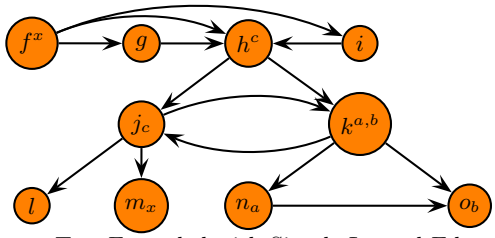
Fig. 9. Call Graph and Dominator Tree to Illustrate the Creation of an EDT Graph.

in Figure 10(b). The graph in Figure 10(b) is now extended with upward lateral edges. If a function on either side of an edge has been coalesced then the coalesced node replaces the function. The result of this extension adds edges from l to g and from o to i. The result is displayed in Figure 10(c). This graph now has a strongly connected component formed by $\{g, h, i, S, l, n, o\}$. Function g has an incoming upward lateral edge and, therefore, it is coalesced with its siblings h and i that are also members of the strongly connected component. Given that i, a function with an incoming lateral edge, has been coalesced there is no need for further action with it. No other functions have an incoming upward lateral edge which means the graph is now acyclic. The finalized EDT graph is displayed in Figure 10(d) in which Q represents the coalesced functions $\{g, h, i\}$. This graph can now be used to propagate free variables and LD information as done in section 5.2.

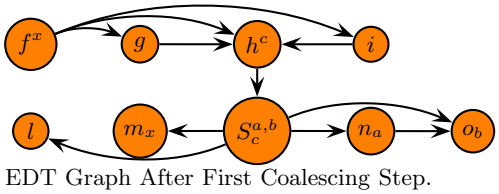
6 The Algorithm, Complexity, and Correctness

6.1 The New Lambda Lifting Algorithm

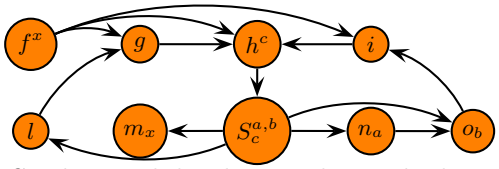
The new lambda lifting algorithm builds a directed acyclic EDT graph from the call graph of a program. Propagation of free variables then proceeds in two steps: the first using a depth-first traversal and the second using a breadth-first traversal. The steps in the algorithm can be outlined as follows:



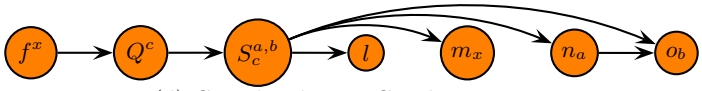
(a) Dominator Tree Extended with Simple Lateral Edges.



(b) EDT Graph After First Coalescing Step.



(c) EDT Graph Extended with Upward Lateral Edges.



(d) Completed EDT Graph.

Fig. 10. EDT Graph Creation Steps.

1. Build the call graph, CG, of the program from its parse tree.
2. Build the dominator tree, DT, for CG.
3. Extend DT with simple lateral edges and coalesce strongly connected components to obtain an acyclic graph EDT'.
4. Extend EDT' with upward lateral edges and compute strongly connected components. Coalesce functions that have an incoming upward lateral edge with their dominator tree siblings that are members of the same component. The resulting graph is the directed acyclic EDT'' graph.
5. Use EDT'' to propagate free variables and LD information using a depth-first traversal.
6. Use EDT'' to propagate free variables using LD information using a breadth-first traversal.
7. For each function, f, make f scope insensitive by adding its complete set of free variables as parameters to f and as arguments to each reference to f.
8. Remove block structure by floating each function to the global level.

6.2 Complexity and Correctness

For a program P, let i be the number of functions, let e be the number of function calls, let v be the number of variables declared, and let n be the size of the program (i.e. $i + e + v$). Step 1 is proportional to $O(e + i)$ or simply $O(n)$. Step 2 is $O(n)$ [11]. For steps 3 and 4 extending a graph with edges is $O(e+i)$ or simply $O(n)$. The computation of strongly connected components is $O(n)$ [5, 6] and their coalescing is $O(n^2)$. For step 5, the propagation of free variables and LD information is $O(e * (i + v))$, or simply $O(n^2)$, assuming the union operation is done in linear time. A similar line of reasoning holds for step 6. Step 7 is $O(v + i + e)$ or simply $O(n)$. Finally, step 8 is $O(n)$. This means that optimal lambda lifting is done in $O(n^2)$. Since lambda lifting can generate an output program of size $O(n^2)$, the time complexity of this algorithm is optimal [3].

The correctness of the algorithm hinges on correctly computing the set of required variables for each function. The required set of free variables for a function, f, depends on the free variables f directly references and on a subset of the free variables transitively needed by the functions f calls. The computation of the latter subset is achieved by never coalescing a dominating function with any functions it dominates. This leads to a graph in which the breadth-first propagation in Step 6 completes the computation of the required free variables of any ancestor function in the parse tree before any successor function as done the J-algorithm. Thus, preventing free variables local to a strongly connected component to be unnecessarily propagated. The required set of free variables computed for each function is complete, because all functional dependencies are captured by the EDT graph. Lateral and downward vertical dependencies are captured by edges and upward vertical dependencies are captured by LD information.

7 Concluding Remarks

This article presents an optimal graph-based $O(n^2)$ lambda lifting algorithm. The algorithm is optimal in the sense that it computes the minimal set of free variables required by each function to make them scope insensitive. The new algorithm is superior to Johnsson's and to Morazán's and Mucha's algorithms by reducing the complexity of optimal lambda lifting from $O(n^3)$ to $O(n^2)$ and it is superior to Danvy's and Schultz's algorithm by being optimal. Nonetheless, this new algorithm owes a great deal of its creation to these predecessors. Considering that Johnsson's original algorithm first appeared in 1985, this newest algorithm has been over 20 years in the making. It is, indeed, a tribute to all these algorithms and to the work of the cited authors from whom we borrowed ideas and inspiration.

Future work includes the implementation of a closureless functional language that uses applicative-order evaluation. The main idea behind the design of this new langauge is to dynamically generate functions that are specialized based on the bindings of its free variables instead of allocating closures. Lambda lifting identifies for us the variables that are used to specialize functions.

8 Acknowledgements

The authors thank Olivier Danvy, Sven-Bodo Scholz, and Barbara Mucha for the discussions during and after IFL 2005 that initiated us down the path that lead to the new algorithm presented in this article. Marco T. Morazán also thanks TLTC at Seton Hall University for the support received through a Faculty Innovation Grant.

References

1. C. Consel. A Tour of Schism: A Partial Evaluation System for Higher-Order Applicative Languages. In *Proc. of the Symp. on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154. ACM Press, June 1993.
2. Olivier Danvy and Ulrik P. Schultz. Lambda-Dropping: Transforming Recursive Equations into Programs with Block Structure. *Theoretical Computer Science*, 248(1–2):243–287, 2000.
3. Olivier Danvy and Ulrik P. Schultz. Lambda-Lifting in Quadratic Time. *Journal of Functional and Logic Programming*, 2004(1), July 2004.
4. Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. The MIT Press, 2001.
5. Alan Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, 1985.
6. Ronald Gould. *Graph Theory*. The Benjamin/Cummings Publishing Company, Inc., 1988.
7. J. Matthews, R. Findler, P. Graunke, S. Krishnamurthi, and M. Felleisen. Automatically Restructuring Programs for the Web. *Automated Software Engineering*, 11(4):337–364, 2004.
8. Thomas Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proc. of a Conf. on Functional Prog. Lang. and Comp. Arch.*, pages 190–203. Springer-Verlag New York, Inc., 1985.
9. Marco T. Morazán and Barbara Mucha. Improved Graph-Based Lambda Lifting. In Hamid Arabnia, editor, *Proc. of the Int. Conf. on Prog. Lang. and Compilers*, pages 896–902. CSREA Press, June 2006.
10. Dino P. Oliva, John D. Ramsdell, and Mitchell Wand. The VLISP Verified PreScheme Compiler. *Lisp and Symbolic Computation*, 8(1-2):111–182, 1995.
11. Stephen Alstrup and Dov Harel and Peter W. Lauridsen and Mikkel Thorup. Dominators in Linear Time. *SIAM Journal on Computing*, 28(6):2117–2132, 1999.

To be or not to be... lazy*

Mercedes Hidalgo-Herrero[1] and Yolanda Ortega-Mallén[2]

[1] Dept. Didáctica de las Matemáticas
Facultad de Educación, Universidad Complutense de Madrid, Spain
mhidalgo@edu.ucm.es
[2] Dept. Sistemas Informáticos y Computación
Facultad CC.Matemáticas, Universidad Complutense de Madrid, Spain
yolanda@sip.ucm.es

Abstract. Laziness restricts the exploitation of parallelism because expressions are evaluated only under demand. Thus, parallel extensions of lazy functional languages, like Haskell, usually override laziness to some extent. The purpose of the present work is to analyze how and to which extent strictness should be introduced in a lazy language to design a parallel extension of it. Towards this end, we have considered different evaluation strategies mixing laziness and eagerness for the language Eden —a parallel extension of Haskell—, we have given formal definitions for each, and we have implemented them in an interpreter to be able to run examples with alternative evaluation models.
Although the study is based on Eden, the concepts involved and the conclusions that we have obtained can be transferred to other parallel and functional languages.

1 Introduction

Referential transparency permits the implementation of alternative orders of execution while retaining the functionality of a program. This property, inherent to pure functional languages, is a key factor for the exploitation of parallelism in the functional paradigm, ranging from a completely implicit parallelism — like for instance automatic parallelization— to an explicit parallelism where the programmer distributes the computation among a set of communicating processes that even may be located by the programmer himself at designated processors. In [13] an excellent classification of functional parallel approaches by level of control of parallelism can be found. Many of these approaches are parallel extensions of sequential functional languages. For instance, the lazy functional language Haskell [16] has been used as the basis of a large and various set of parallel and distributed languages (see [21] for a comprehensive survey of these). One of these parallel extensions of Haskell is the language Eden [2,14] that includes a set of *coordination* features to control the parallel evaluation of processes while keeping the high-level nature of the declarative paradigm.

* Work partially supported by the Spanish projects TIN2006-15578-C02-01, TIN2006-15660-C02-01 and PAC06-0008.

As a lazy language, Haskell adopts normal order evaluation, avoiding repeated computations by sharing reductions. This lazy approach restricts the exploitation of parallelism because expressions are evaluated only under demand. Thus, parallel versions of Haskell usually override laziness in some points:

Speculative work Some languages allow for the evaluation of parts of the code that have not been demanded yet. This does not necessarily change the underlying sequential lazy semantics, because the overall result of the program can be obtained even if some speculative task gets stuck; this is achieved by guaranteeing that the scheduler always prefers to evaluate the computations of the main process. In this case, speculation only influences the efficiency of the system.

Examples of this kind of speculative computation are the `par` operator defined in GpH [20], and the eager process creation in Eden.

Introducing strictness A more drastic way of overriding laziness is to force the evaluation of some portions of the code before the result is really needed. Thus, the underlying lazy semantics is modified.

Examples of this second form are the strict operator (`seq`) introduced in GpH [20,19], or forcing the reduction to normal form of the values that are to be transmitted through channels in Eden. Similarly, the transmission of lists in Caliban [11,18] is head-strict, and data-parallel versions of Haskell introduce strictness in the use of some predefined data types (mainly lists).

Mixed (lazy and strict) evaluation has already been analyzed in a sequential context (see [3,4] for a discussion on advantages and risks of this combination), but few work has been done to carefully analyze how and to which extent strictness should be introduced in a lazy language to design a parallel extension of it.

Towards this end, in [8,9] we have considered a few alternative evaluation models for the language Eden, and we have implemented an interpreter capable of dealing with all of them. We have then used this environment to analyze the influence of the evaluation strategies in the performance of some chosen parallel skeletons implemented in Eden. The purpose of the present paper is to achieve a more rigorous and complete comparative analysis, by extending the spectrum of evaluation strategies mixing laziness and strictness, and by presenting formal definitions for each evaluation model. Although the study is based on Eden —or more exactly on a simple calculus that includes the main coordination features of Eden—, the concepts involved and the conclusions that we have obtained can be transferred to other parallel and functional languages.

The paper is organized as follows: We start with a very brief introduction to Eden's coordination features, and we describe the calculus that we are going to use for our analysis. In Section 3 we discuss on the possible evaluation strategies, and we give a classification of these around three concepts. Then in Section 4 we present a distributed operational semantics for the calculus, and we formalize the evaluation strategies given before. In Section 5 we present a collection of

$E ::= x$		variable
$\mid$	$\backslash x.E$	λ-abstraction
$\mid$	$x_1 x_2$	application
$\mid$	$x_1 \,\texttt{\#}\, x_2$	process instantiation
$\mid$	$\texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ x$	local declaration

Fig. 1. Eden's restricted core syntax

examples that shows how the evaluation strategies may affect issues like termination or deadlock. We conclude with a summary discussion on the lazy-eager combinations.

2 Eden's coordination features

Coordination in Eden is based on two principal concepts: *explicit definition of processes* and *implicit stream-based communication* [10]. In the same way as there is a distinction between function definition and function application, Eden includes *process abstractions*, i.e. abstract schemes for process behavior, and *process instantiations* for the actual creation of processes. Additionally, *nondeterminism* is explicitly introduced and encapsulated within processes by means of a predefined process abstraction which is used to instantiate non-deterministic processes that fairly merge several input streams into a single output stream.

Figure 1 shows the restricted[3] (abstract) syntax of an untyped λ-calculus extended with recursive lets and process instantiation. This simple calculus captures the essence of Eden and proves to be sufficient for our purposes.

In the syntax description $x \in \mathit{Var}$ denotes variables and $E \in \mathit{Exp}$ represents expressions. The expression $\texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ x$ is an abbreviation of $\texttt{let}\ x_1 = E_1, \ldots, x_n = E_n\ \texttt{in}\ x$.

For simplicity we have identified process abstractions with one-argument functions, so that new processes are created with only one input and one output channels. When evaluating an expression $x_1 \,\texttt{\#}\, x_2$ inside a process p, a new child process q is created. When process q receives from its parent p the value of x_2 through its input channel, it evaluates $x_1\ x_2$ and returns to its parent the result via its output channel. The diagram in Figure 2 illustrates this behavior.

Apart from the process creation involved, a key difference between application and instantiation is the non-strictness of the former versus the eagerness of the latter.

3 Mixed evaluation strategies

Eden has been designed for distributed environments without shared memory between processes; therefore, bindings have to be copied from one heap to the

[3] Restricted syntax is considered to simplify the semantic rules, as in [12,1].

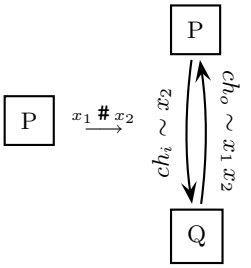


Fig. 2. Process creation in Eden

other when creating new processes or when communicating values. In this context the following questions can be stated:

- In the expression $x_1 \# x_2$, it is clear that x_2 has to be evaluated in the instantiating process. But what about x_1? Should the parent evaluate the expression before copying it in the child's heap?
- How should the free variables in a newly instantiated process be handled?
- What about the values communicated through the channels? To what extent should they be evaluated before being communicated? It is advisable to send the extra work related to the free variables —with an unknown degree of evaluation— to the receiver?
- Should an instantiation expression be copied from one heap to the another? Or it is more advisable to suspend the corresponding communication or instantiation?

All these questions are related to the computation distribution between processes: *How much work should do the parent/producer of a process/value, and how much work is left for the child/consumer?* This is a crucial point in any parallel language, and not particular to Eden, although the features of Eden maybe offer more possibilities for discussion.

It turns out that the alternatives can be expressed as different mixtures of *lazy* and *eager* evaluation. In fact, neither pure laziness nor eagerness are optimal, in the sense that, for each proposal, examples can be found showing that the opposite view would be much more efficient. Therefore, we want to keep the discussion under a methodological point of view. In other words, we seek that programs should be efficient, but also clear, safe and easy to write and verify.

3.1 Keystones of the evaluation strategies

We can organize the evaluation strategies around three concepts:

Process Abstraction Evaluation (PAE) In the case of a process instantiation the evaluation of the process abstraction can be done either by the parent process, or by the child. In the first case process instantiation could be more costly for the instantiating process, but the programmer has a greater control of the sharing of work between parent and child, that leads to the possibility of designing libraries of process abstractions to create "slaves" to get the "hard work" done. The performance of the processes created from these libraries is guaranteed, because it will not depend on the context were processes are created.

Evaluation Before Copy (EBC) When copying bindings from a heap process to another it may be required that every needed binding —corresponding to free variables in process/lambda abstractions— is previously evaluated. This corresponds to a strict semantics as can be found for ML [15], although there free variables in a λ-abstraction would have been evaluated before reducing the embedding expression to normal form. This option applies to two situations: (1) when creating the initial heap of a new process (EBC_p); (2) when communicating a value through a channel (EBC_v).

Instantiation Copy (IC) represents the copy of bindings —from one heap to another— corresponding to pending process instantiations. If the copy of instantiations is not permitted, then the action is blocked until the instantiation is resolved. Again this applies to process creation (IC_p) as well as to value communication (IC_v).

Therefore we have five issues (PAE,EBC_p,EBC_v,IC_p and IC_v), each with two options: parent/child for PAE, yes/no for the rest. This gives a total of $2^5 = 32$ combinations. Some of these can be discarded; for instance, if it is required that every needed binding should be evaluated before its copy (EBC = yes), this should imply the evaluation of pending instantiations too (i.e. IC = no), then the list is reduced to 18 options. Moreover, if it is required that the parent evaluates the process abstraction (PAE = parent) then it is reasonable that also the parent evaluates the needed bindings before they are copied to the heap of a new child (EBC_p = yes). This then reduces the set to 12 strategies. By separating the discussion relating to process creation from the options relative to communication, we can organize the strategies in a table with four entries (see table 1). For each entry we have to consider the three options permitted for communication.

4 A semantic distributed model

The semantic model that we will consider here has already been used to give a formal semantics for Eden [5,6,7]. As it is usual for parallel and concurrent languages [1,17], the model embodies two levels of transition systems: one lower level to handle the local behavior of processes, and an upper level to describe global effects on the system, namely process creation and communication.

The evaluation of an expression in the calculus presented before in general will require the creation of several parallel processes in the system. Each process

	PAE	**EBC**$_p$	**IC**$_p$
(1)	parent	yes	no
(2)	child	yes	no
(3)	child	no	yes
(4)	child	no	no

	EBC$_v$	**IC**$_v$
(a)	yes	no
(b)	no	yes
(c)	no	no

Table 1. Evaluation strategies

will, in turn, encompass a set of independently executing threads, each devoted to the production of one output of the process.

The semantics evolves through global steps. The tasks to be done comprises local parallel evolution of all the processes in the system, process creation, inter-process communication and thread state management (like for instance, thread unblocking and deactivation).

In our model the evaluation state of a process is represented by its heap of closures, i.e. the set of bindings of variables to expressions. Following [1], each binding is considered a potential thread, and has associated a label indicating its state: $x \overset{\alpha}{\mapsto} E$ where $\alpha ::= I|A|B$ corresponds, respectively, to Inactive (either not yet demanded or already completely evaluated), Active (or demanded and in execution), and Blocked (demanded but waiting for the value of another binding).

The set $dom(H)$ contains the left-side variables of a heap H. Besides, notation $H + \{x \overset{\alpha}{\mapsto} E\}$ (and also $\{x \overset{\alpha}{\mapsto} E\} + H$) means that the heap H is extended with the binding $x \overset{\alpha}{\mapsto} E$, and it is assumed that $x \notin dom(H)$. If $x \overset{\alpha}{\mapsto} E \in H$ then $H(x) = E$.

In the following, we will use x, y as program variables, while ch denotes a channel; sometimes we distinguish between input channels (from parent to child) ch_i and output channels (from child to parent) ch_o. We will use θ for referring to program variables as well as channels, and η will stand for a fresh renaming.

To evaluate a main expression E, the initial system consists only of a main process with an initial heap $H_0 = \{main \overset{A}{\mapsto} E\}$, where it is assumed that $main$ is a fresh variable.

For the purpose of the present work we only need to describe here the (global) semantic rules for process creation and communication. The interested reader is referred to [6,7], where the whole set of semantic rules can be found.

4.1 Process creation

New processes are created when evaluating `#`-expressions, by applying the rule given in Figure 3.

In our calculus processes are eagerly created when instantiations are found at the top-level, i.e. when an variable in a heap is directly bound to a `#`-expression, even if that binding is not active (i.e. demanded).

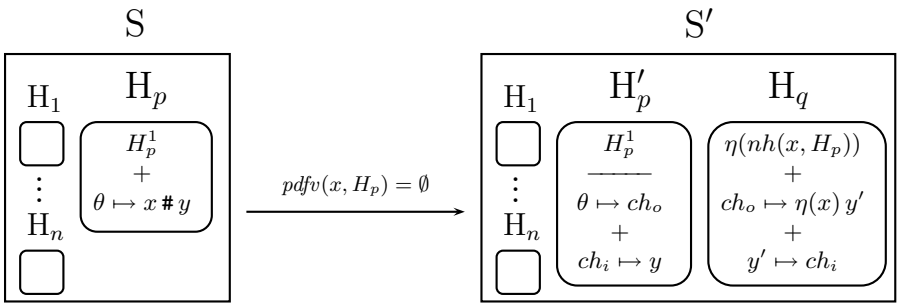


Fig. 3. Process creation

When creating a process, the thread evaluating the instantiation (at the *parent* side) is blocked on a fresh output channel, ch_o, corresponding to the initial thread in the new *child* process q. Correspondingly, the child process gets a thread which is blocked on a new input channel ch_i, that is served by a new thread in the parent (communication from parent to child).

As it has been mentioned before, the absence of a common shared heap requires that every binding needed for the evaluation of the free variables in the child process body is copied from the parent to the child heap, using for this purpose the function nh (needed heap), whose definition is independent of the semantic option:

$$
\begin{aligned}
&nh(E, \emptyset) && = \emptyset && \\
&nh(x, H) && = \emptyset && \text{if } x \notin dom(H) \\
&nh(x, \{x \overset{\alpha}{\mapsto} E\} + H) && = \{x \overset{I}{\mapsto} E\} + nh(E, H) && \\
&nh(\backslash x.E, H) && = nh(E, H) && \\
&nh(x_1\, x_2, H) && = nh(x_1, H) \cup nh(x_2, H) && \\
&nh(x_1 \,\#\, x_2, H) && = nh(x_1, H) \cup nh(x_2, H) && \\
&nh(\texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ x, H) && = nh(x, H) \cup (\bigcup_{i=1}^{n} nh(E_i, H)) &&
\end{aligned}
$$

Function $nh(E, H)$ collects all the bindings in H that are reachable from E. Moreover, in order to keep all the names distinct, even if they belong to different heaps, we rename —by means of η— the copied closures.

However, when the evaluation of the process body depends on a value to be communicated from some other process, the process creation is suspended until the necessary communications have taken place. Depending on the semantics option, a process creation may be further delayed because of other reasons which are detected by function *dfv* (demand of free variables), that will be explained in Section 4.3.

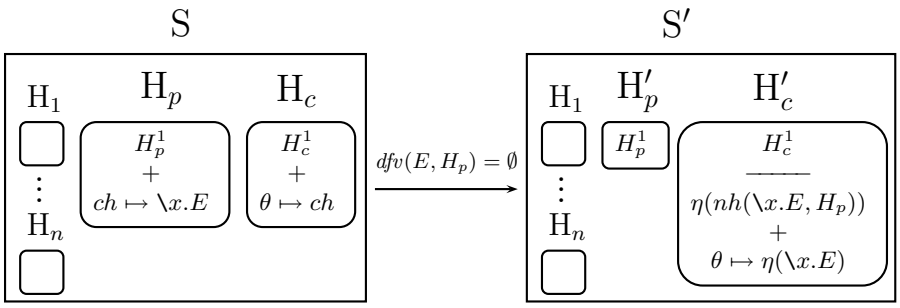


Fig. 4. Value communication

4.2 Communication

The rule for communication is given in Figure 4. When communicating a value —a λ-abstraction in our calculus—, every binding needed for the evaluation of the free variables in the value are to be copied from the producer's to the consumer's heap. Similarly to the case of process creation, this copy can take place only if there is no dependency on pending communications. The renaming (η) used for the heap is applied to the passed value too. A fresh renaming of the bound variables in the abstraction is also needed. Notice that the binding of the channel disappears when the communication has been completed.

Likewise to process creation, a communication can be suspended depending on the semantic option used. This is expressed again by function *dfv* described in Section 4.3.

4.3 Formalization of semantic options

Function *dfv* checks the circumstances that cause a process creation (or a communication) to be suspended:

- A pending communication.
- A pending process creation.
- A free variable not bound to a λ-abstraction.

However, whereas all the evaluation strategies take into account pending communications, pending process creations are only considered when IC=no, and the last condition only when EBC=yes. Consequently, three different versions of *dfv* are needed to express the evaluation strategies considered in table 1.

In order to take into account option PAE, in the process creation rule, function *pdfv* (previous to demand of free variables) is applied before proceeding with *dfv*:

$$
\begin{array}{lll}
\mathit{pdfv}(x, H) & = \emptyset & \text{if } x \notin \mathit{dom}(H) \\
\mathit{pdfv}(x, \{x \overset{\alpha}{\mapsto} \backslash x.E\} + H) & = \mathit{dfv}(E, H) & \text{if PAE=parent} \\
\mathit{pdfv}(x, \{x \overset{\alpha}{\mapsto} E\} + H) & = \{x \overset{\alpha}{\mapsto} E\} & \text{if PAE=parent} \wedge E \neq \backslash x.E' \\
\mathit{pdfv}(x, \{x \overset{\alpha}{\mapsto} E\} + H) & = \mathit{dfv}(E, H) & \text{if PAE=child}
\end{array}
$$

If PAE=parent then the process abstraction x_1 —for an instantiation expression $x_1 \,\#\, x_2$— must be evaluated before proceeding with the creation. In this case, and only if the corresponding expression is still unevaluated, i.e. it is not a λ-abstraction, *pdfv* returns a heap with a unique binding for x_1. Otherwise, *pdfv* just calls the function *dfv*.

I. EBC=yes ($\Rightarrow$ IC=no)

$$
\begin{array}{lll}
\mathit{dfv}^I(E, \emptyset) & = \emptyset & \\
\mathit{dfv}^I(x, H) & = \emptyset & \text{if } x \notin \mathit{dom}(H) \\
\mathit{dfv}^I(x, \{x \overset{\alpha}{\mapsto} \backslash x.E'\} + H) & = \mathit{dfv}^I(E', H) & \\
\mathit{dfv}^I(x, \{x \overset{\alpha}{\mapsto} E\} + H) & = \{x \overset{\alpha}{\mapsto} E\} & \text{if } E \neq \backslash x.E' \wedge \alpha \neq B \\
\mathit{dfv}^I(x, \{x \overset{B}{\mapsto} y\} + H) & = \{x \overset{B}{\mapsto} y\} \cup \mathit{dfv}^I(y, H) & \\
\mathit{dfv}^I(x, \{x \overset{B}{\mapsto} x_1\, x_2\} + H) & = \{x \overset{B}{\mapsto} x_1\, x_2\} \cup \mathit{dfv}^I(x_1, H) & \\
\mathit{dfv}^I(x, \{x \overset{B}{\mapsto} x_1 \,\#\, x_2\} + H) & = \{x \overset{B}{\mapsto} x_1 \,\#\, x_2\} \cup \mathit{dfv}^I(x_1, H) & \\
\mathit{dfv}^I(x, \{x \overset{B}{\mapsto} \mathit{ch}\} + H) & = \{x \overset{B}{\mapsto} \mathit{ch}\} & \\
\mathit{dfv}^I(\backslash x.E, H) & = \mathit{dfv}^I(E, H) & \\
\mathit{dfv}^I(x_1\, x_2, H) & = \mathit{dfv}^I(x_1, H) \cup \mathit{dfv}^I(x_2, H) & \\
\mathit{dfv}^I(x_1 \,\#\, x_2, H) & = \mathit{dfv}^I(x_1, H) \cup \mathit{dfv}^I(x_2, H) & \\
\mathit{dfv}^I(\texttt{let } \{x_i = E_i\}_{i=1}^n \texttt{ in } x, H) & = \mathit{dfv}^I(x, H) \cup (\bigcup_{i=1}^{n} \mathit{dfv}^I(E_i, H)) &
\end{array}
$$

When expression E is an variable x, it may be one of the following cases:

1. x is already bound to an abstraction: then dfv^I must gather the free variables corresponding to this value.
2. x is bound to another expression: then the binding for x is collected, but if this binding is blocked, then the following cases must be considered:
   (a) If it is blocked on another variable, then dfv^I continues with the binding for this second variable.
   (b) If it is blocked either on an instantiation or an application, then dfv^I continues checking the abstraction.

   Notice that if the binding is blocked on a channel, then dfv^I is not further invoked because this channel variable cannot appear in the left-hand-side of a binding inside the heap under consideration.

If the expression is not an variable, then dfv^I is invoked with its subexpressions.

II. EBC=no, IC=no

In this case dfv^{II} just detects dependencies on instantiation expressions and channels (communications).

$$
\begin{array}{lll}
dfv^{II}(E, \emptyset) & = \emptyset & \\
dfv^{II}(x, H) & = \emptyset & \text{if } x \notin dom(H) \\
dfv^{II}(x, \{x \overset{\alpha}{\mapsto} x_1 \,\#\, x_2\} + H) & = \{x \overset{\alpha}{\mapsto} x_1 \,\#\, x_2\} & \text{if } \alpha \neq B \\
dfv^{II}(x, \{x \overset{B}{\mapsto} x_1 \,\#\, x_2\} + x_1 \overset{I}{\mapsto} E + H) & = \{x \overset{B}{\mapsto} x_1 \,\#\, x_2, x_1 \overset{I}{\mapsto} E\} & \\
dfv^{II}(x, \{x \overset{B}{\mapsto} x_1 \,\#\, x_2\} + x_1 \overset{AB}{\mapsto} E + H) & = \{x \overset{B}{\mapsto} x_1 \,\#\, x_2\} & \\
dfv^{II}(x, \{x \overset{B}{\mapsto} x_1 \,\#\, x_2\} + H) & = \{x \overset{B}{\mapsto} x_1 \,\#\, x_2\} & \text{if } x_1 \notin dom(H) \\
dfv^{II}(x, \{x \overset{B}{\mapsto} ch\} + H) & = \{x \overset{B}{\mapsto} ch\} & \\
dfv^{II}(x, \{x \overset{\alpha}{\mapsto} E\} + H) & = dfv^{II}(E, H) & \text{if } E \neq x_1 \,\#\, x_2 \wedge E \neq ch \\
dfv^{II}(\backslash x.E, H) & = dfv^{II}(E, H) & \\
dfv^{II}(x_1\, x_2, H) & = dfv^{II}(x_1, H) \cup dfv^{II}(x_2, H) & \\
dfv^{II}(x_1 \,\#\, x_2, H) & = dfv^{II}(x_1, H) \cup dfv^{II}(x_2, H) & \\
dfv^{II}(\texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ x, H) & = dfv^{II}(x, H) \cup (\bigcup_{i=1}^{n} dfv^{II}(E_i, H)) &
\end{array}
$$

III. EBC=no, IC=yes

With this combination, the only reason to suspend a communication or a creation is a dependency on a channel (communication).

$$
\begin{array}{lll}
dfv^{III}(E, \emptyset) & = \emptyset & \\
dfv^{III}(x, H) & = \emptyset & \text{if } x \notin dom(H) \\
dfv^{III}(x, \{x \overset{B}{\mapsto} ch\} + H) & = \{x \overset{B}{\mapsto} ch\} & \\
dfv^{III}(x, \{x \overset{\alpha}{\mapsto} E\} + H) & = dfv^{III}(E, H) & \text{if } E \neq ch \\
dfv^{III}(\backslash x.E, H) & = dfv^{III}(E, H) & \\
dfv^{III}(x_1\, x_2, H) & = dfv^{III}(x_1, H) \cup dfv^{III}(x_2, H) & \\
dfv^{III}(x_1 \,\#\, x_2, H) & = dfv^{III}(x_1, H) \cup dfv^{III}(x_2, H) & \\
dfv^{III}(\texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ x, H) & = dfv^{III}(x, H) \cup (\bigcup_{i=1}^{n} dfv^{III}(E_i, H)) &
\end{array}
$$

The versions of *dfv* corresponding to the evaluation strategies given in Table 1 are expressed in Table 2.

5 Case study

In this section we include some examples that show to what extent the evaluation strategy may affect the behavior of a program.

	Global Rules	
	Process Creation	**Communication**
(1)(a)	dfv^I	dfv^I
(1)(b)	dfv^I	dfv^{III}
(1)(c)	dfv^I	dfv^{II}
(2)(a)	dfv^I	dfv^I
(2)(b)	dfv^I	dfv^{III}
(2)(c)	dfv^I	dfv^{II}
(3)(a)	dfv^{III}	dfv^I
(3)(b)	dfv^{III}	dfv^{III}
(3)(c)	dfv^{III}	dfv^{II}
(4)(a)	dfv^{II}	dfv^I
(4)(b)	dfv^{II}	dfv^{III}
(4)(c)	dfv^{II}	dfv^{II}

Table 2. Definition of *dfv* for each evaluation strategy

5.1 Termination

Example 1. Let us consider the following expression:

$$\begin{aligned}\texttt{let}\ & x1 = x2\,\texttt{\#}\,x3,\\ & x2 = \backslash x7.x7,\\ & x3 = \backslash x8.x4,\\ & x4 = \backslash x9.x5,\\ & x5 = x6\ x2,\\ & x6 = \backslash x10.x10\ x10\\ \texttt{in}\ & x1\end{aligned}$$

With an evaluation strategy that requires that free variables are to be evaluated to whnf before being copied (EBC=yes), the evaluation never terminates. During the evaluation, the following system[4] is reached, and from then on each global step returns the same system:

[4] The systems shown for the examples have been obtained with an interpreter. The implemented generation of free variables returns names in the style xn, where n is an increasing integer.

main (N. Children: 1)	**main.1** (N. Children: 0)
$ch_i \overset{I}{\mapsto} \backslash x8.x4$	$ch_o \overset{B}{\mapsto} x11$
main $\overset{\mathbf{B}}{\mapsto}$ **x1**	$x11 \overset{B}{\mapsto} ch_i$
x1 $\overset{\mathbf{B}}{\mapsto}$ $\mathbf{ch_o}$	$x13 \overset{I}{\mapsto} \backslash x12.x12$
$x2 \overset{I}{\mapsto} \backslash x7.x7$	
$x3 \overset{I}{\mapsto} \backslash x8.x4$	
$x4 \overset{I}{\mapsto} \backslash x9.x5$	
$x5 \overset{A}{\mapsto} x6\ x6$	
$x6 \overset{I}{\mapsto} \backslash x10.x10\ x10$	

It can be observed how the *main* variable is blocked and bound to another variable ($x1$) which is, in turn, blocked on the output channel for the child process, while the unique active thread ($x5$ in the main process) is bound to a self-application. As the parent cannot send the input data to its child, the latter remains blocked and cannot produce the expected result.

By contrast, if free variables are allowed to be copied unevaluated (EBC=no), then the evaluation comes to an end, and the obtained final system is:

main (N. Children: 1)	**main.1** (N. Children: 0)
main $\overset{\mathbf{I}}{\mapsto}$ **\x20.x23**	$x11 \overset{I}{\mapsto} \backslash x14.x17$
$x1 \overset{I}{\mapsto} \backslash x20.x23$	$x13 \overset{I}{\mapsto} \backslash x12.x12$
$x2 \overset{I}{\mapsto} \backslash x7.x7$	$x17 \overset{I}{\mapsto} \backslash x15.x18$
$x3 \overset{I}{\mapsto} \backslash x8.x4$	$x18 \overset{I}{\mapsto} x19\ x19$
$x4 \overset{I}{\mapsto} \backslash x9.x5$	$x19 \overset{I}{\mapsto} \backslash x16.x16\ x16$
$x5 \overset{I}{\mapsto} x10\ x10$	
$x6 \overset{I}{\mapsto} \backslash x10.\ x10\ x10$	
$x23 \overset{I}{\mapsto} \backslash x21.x24$	
$x24 \overset{I}{\mapsto} x25\ x25$	
$x25 \overset{I}{\mapsto} \backslash x22.x22\ x22$	

Notice that in this case the *main* variable is bound to a λ-abstraction, and all the threads in the final system are inactive.

5.2 Deadlock

In some contexts, and depending on the option chosen for IC, a deadlock is produced or not.

Example 2. Let us consider the following expression:

$$\begin{aligned} \texttt{let}\ & x1 = x1 \,\#\, x1, \\ & x2 = x3 \,\#\, x4, \\ & x3 = \backslash x5.x5, \\ & x4 = \backslash x6.\backslash x7.x1 \\ \texttt{in}\ & x2 \end{aligned}$$

When IC=no, the communication from the parent to the child cannot take place. Consequently, the system gets deadlocked:

main (N. Children: 1)	**main.1** (N. Children: 0)
$\mathbf{ch_i} \overset{\mathbf{I}}{\mapsto} \mathbf{\backslash x6.(\backslash x7.\ x1)}$	$\mathbf{ch_o} \overset{\mathbf{B}}{\mapsto} \mathbf{x8}$
$\mathbf{main} \overset{\mathbf{B}}{\mapsto} \mathbf{x2}$	$\mathbf{x8} \overset{\mathbf{B}}{\mapsto} \mathbf{ch_i}$
$\mathbf{x1} \overset{\mathbf{B}}{\mapsto} \mathbf{x1 \# x1}$	$x16 \overset{I}{\mapsto} \backslash x9.\ x9$
$\mathbf{x2} \overset{\mathbf{B}}{\mapsto} \mathbf{ch_o}$	
$x3 \overset{I}{\mapsto} \backslash x5.\ x5$	
$x4 \overset{I}{\mapsto} \backslash x6.(\backslash x7.\ x1)$	

Nevertheless, when IC=yes, the communication from the parent to the child is done successfully. The evaluation ends with a whnf value bound to the *main* variable:

main (N. Children: 1)	**main.1** (N. Children: 0)
$\mathbf{main} \overset{\mathbf{I}}{\mapsto} \mathbf{\backslash x13.(\backslash x14.\ x12)}$	$x8 \overset{I}{\mapsto} \backslash x11.(\backslash x19.\ x10)$
$x1 \overset{B}{\mapsto} x1 \,\#\, x1$	$x16 \overset{I}{\mapsto} \backslash x9.\ x9$
$x2 \overset{I}{\mapsto} \backslash x13.(\backslash x14.\ x12)$	$x10 \overset{B}{\mapsto} x10 \,\#\, x10$
$x3 \overset{I}{\mapsto} \backslash x5.\ x5$	
$x4 \overset{I}{\mapsto} \backslash x6.(\backslash x7.\ x1)$	
$x12 \overset{B}{\mapsto} x12 \,\#\, x12$	

Although three process instantiations remain (self)blocked, these are unimportant because the variables are not needed (speculative work).

5.3 Too costly children

Creating a new child process may not be profitable for the parent process. The following example illustrates this situation when EBC_v=yes.

Example 3. Let us consider the following expression:

$$\begin{aligned} \texttt{let}\ & x1 = x3\,x2, \\ & x2 = \backslash x6.x6, \\ & x3 = x2\,x2, \\ & x4 = \backslash x7.\backslash x8.x1\,x2, \\ & x5 = x2 \,\#\, x4 \\ \texttt{in}\ & x5 \end{aligned}$$

The new process is created in the first global step:

main (N. Children: 1)	**main.1** (N. Children: 0)
$ch_i \overset{A}{\mapsto} x4$	$ch_o \overset{A}{\mapsto} x11\,x9$
$main \overset{A}{\mapsto} x5$	$x9 \overset{B}{\mapsto} ch_i$
$x1 \overset{I}{\mapsto} x3\,x2$	$x11 \overset{I}{\mapsto} \backslash x10.x10$
$x2 \overset{I}{\mapsto} \backslash x6.x6$	
$x3 \overset{I}{\mapsto} x2\,x2$	
$x4 \overset{I}{\mapsto} \backslash x7.\backslash x8.x1\,x2$	
$x5 \overset{B}{\mapsto} ch_o$	

However, before the communication from the parent to the child takes place variables $x1$ and $x3$ must be evaluated. Afterwards, communication is carried out.

main (N. Children: 1)	**main.1** (N. Children: 0)
$main \overset{B}{\mapsto} x5$	$ch_o \overset{B}{\mapsto} x9$
$x1 \overset{A}{\mapsto} \backslash x6.x6$	$x9 \overset{A}{\mapsto} \backslash x12.\backslash x13.x16\, x17$
$x2 \overset{I}{\mapsto} \backslash x6.x6$	$x11 \overset{I}{\mapsto} \backslash x10.x10$
$x3 \overset{I}{\mapsto} \backslash x6.x6$	$x16 \overset{I}{\mapsto} \backslash x14.x14$
$x4 \overset{I}{\mapsto} \backslash x7.\backslash x8.x1\, x2$	$x17 \overset{I}{\mapsto} \backslash x15.x15$
$x5 \overset{B}{\mapsto} ch_o$	

Therefore, the parent has to do all the work to send to the child everything already evaluated, and the activity of the child is reduced to return to the parent the same value that it has received from it!

When EBC$_v$=no, a child process may result unworthy as well. For instance if the value communicated by the child has many free unevaluated variables that are used by the parent.

Example 4. Let us consider the following expression:

$$\begin{array}{ll} \texttt{let} & x1 = x3\, x2, \\ & x2 = \backslash x8.x8, \\ & x3 = x2\, x2, \\ & x4 = \backslash x9.\backslash x10.x1\, x2, \\ & x5 = x2\, \texttt{\#}\, x4, \\ & x6 = x5\, x3, \\ & x7 = x6\, x3 \\ \texttt{in} & x7 \end{array}$$

After the three first global steps, input/output communications between parent and child have been accomplished, and the following system is obtained:

main (N. Children: 1)	**main.1** (N. Children: 0)
$main \overset{B}{\mapsto} x7$	$x11 \overset{I}{\mapsto} \backslash x14.\backslash x15.x18\, x19$
$x1 \overset{I}{\mapsto} x3\, x2$	$x13 \overset{I}{\mapsto} \backslash x12.x12$
$x2 \overset{I}{\mapsto} \backslash x8.x8$	$x17 \overset{I}{\mapsto} x19\, x19$
$x3 \overset{I}{\mapsto} x2\, x2$	$x18 \overset{I}{\mapsto} x17\, x19$
$x4 \overset{I}{\mapsto} \backslash x9.\backslash x10.x1\, x2$	$x19 \overset{I}{\mapsto} \backslash x16.x16$
$x5 \overset{A}{\mapsto} \backslash x20.\backslash x21.x24\, x25$	
$x6 \overset{A}{\mapsto} x5\, x3$	
$x7 \overset{B}{\mapsto} x6\, x3$	
$x23 \overset{I}{\mapsto} x25\, x25$	
$x24 \overset{I}{\mapsto} x23\, x25$	
$x25 \overset{I}{\mapsto} \backslash x22.x22$	

but still many global steps (twelve steps to be precise) are needed for finishing the evaluation. And these steps do only involve computations in the parent process.

The last example shows a situation where a process creation is not profitable because PAE=parent.

Example 5. Let us consider the following expression:

$$\begin{array}{ll} \texttt{let} & x1 = x2\,\texttt{\#}\,x3, \\ & x2 = x3\,x4, \\ & x3 = \backslash x7.x7, \\ & x4 = x3\,x5, \\ & x5 = x3\,x6, \\ & x6 = x3\,x3 \\ \texttt{in} & x1 \end{array}$$

After evaluating the let-expression we obtain:

main (N° Hijos: 0)
$main \overset{A}{\mapsto} x1$
$x1 \overset{I}{\mapsto} x2\,\texttt{\#}\,x3$
$x2 \overset{I}{\mapsto} x3\,x4$
$x3 \overset{I}{\mapsto} \backslash x7.\ x7$
$x4 \overset{I}{\mapsto} x3\,x5$
$x5 \overset{I}{\mapsto} x3\,x6$
$x6 \overset{I}{\mapsto} x3\,x3$

The process creation is delayed until its abstraction is evaluated. This work is carried out by the parent and costs eleven global steps.

main (N° Hijos: 1)	**main.1** (N° Hijos: 0)
$ch_i \overset{A}{\mapsto} x3$	$ch_o \overset{A}{\mapsto} x18\,x8$
$main \overset{B}{\mapsto} x1$	$x8 \overset{B}{\mapsto} ch_i$
$x1 \overset{B}{\mapsto} ch_o$	$x18 \overset{I}{\mapsto} \backslash x9.x9$
$x2 \overset{A}{\mapsto} \backslash x7.x7$	
$x3 \overset{I}{\mapsto} \backslash x7.x7$	
$x4 \overset{I}{\mapsto} \backslash x7.x7$	
$x5 \overset{I}{\mapsto} \backslash x7.x7$	
$x6 \overset{I}{\mapsto} \backslash x7.x7$	

Afterwards, the evaluation developed by the child process only takes two further steps. Once again, the child has not carried out much work whereas the parent has done most of the computation.

6 Conclusions and future work

We could consider the combination PAE=parent, EBC_p=yes and EBC_v=yes (IC_p=no and IC_v=no), i.e. entry (1)(a) in Table 1, the most eager approach.

This evaluation strategies tends to be more efficient, because in many cases duplication of work is avoided, and the size of transmitted data is much smaller. It also benefits of a greater control of load balancing and of communications, as the size of the transmissions depends exclusively on the type of the value to be communicated; while in a context with EBC_v=no there is no way to determine the expected size of a transmission, as the "current state" of the free variables for the communicated value must be packed and sent to the consumer, and the evaluation of these may depend on very large objects.

The main argument against this eager strategy is that, as we have seen in the first example in Section 5, the evaluation of free variables in advance to creating a child may lead to a *loss of the normal order*, and this is a critical matter. As a consequence, we cannot replace equals by equals, as any functional programmer would expect.

As eagerness may lead to spend a lot of energy on useless work or even to endless loops, we can look for a way to provide the programmer with a mean to pass, when desired, unevaluated definitions as subexpressions of the process abstraction, in order to be (or not to be) evaluated by the child process. A natural way to do it is to encapsulate them within λ-abstractions. For example, if the programmer is interested in the child process —instead of the parent process— to evaluate a subexpression e_y bound to y, a free variable of the process abstraction, it can be encapsulated, $\lambda dummy.e_y$, and bound to y'; besides, the variable y must be substituted by $y'(\lambda x.x)$ in the abstraction. Thus, although the option EBC_p is yes, the parent will not evaluate e_y.

At the other extreme, we could consider the combination PAE=child, EBC_p=no, EBC_v=no, IC_p=yes, and IC_v=yes, i.e. entry (3)(b) in Table 1, the laziest one.

In the cases where a parent process does not share variables with its children, and the children themselves do not share variables between them, then the load balancing could be better under this strategy, as the parent has not to do all the work, but can divide it among its children. Moreover, as the parent does not need to evaluate the free variables, less time should be needed to create each child, although the real gain depends on some factors such as the work necessary to evaluate the free variables, the amount of graph to be packed, etc. We wonder how often this kind of situation occurs. We think that this problem can be solved methodologically if the programmer tries not to use free variables or, at least, free variables that do not require a big amount of work. In such cases, the performance is nearly the same for both options of EBC_p.

To gain efficiency in this approach, we can provide the programmer with some means of eagerly evaluating the free variables in the parent side. This would allow to have sharing and also to send less work and/or data when packing the closures for the child.

There are two ways of introducing eagerness:

- Sending free variables through channels.
- Using the functions `nf` (evaluation of an expression to normal form) and `seq` (strict sequential composition).

The problem with the first approach is that currying is lost. The second alternative is not as elegant as the former, but it preserves currying. The programmer only needs to force the evaluation of each free variable that it is desired to be evaluated before the creation of the process.

Acknowledgements This work is much indebted to the *veteran* members of the Eden Group at Madrid around the year 1999, and a lively and long-forgotten discussion on eagerness vs. laziness. We are also grateful to Fernando Rubio for contribute with the title, some ideas, and invaluable support.

References

1. C. Baker-Finch, D. King, and P. W. Trinder. An operational semantics for parallel lazy evaluation. In *ACM-SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 162–173, Montreal, Canada, September 2000.
2. S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden: Language definition and operational semantics. Technical Report 96/10, Reihe Informatik, FB Mathematik, Philipps-Universität Marburg, Germany, URL `http://www.mathematik.uni-marburg.de/~eden/`, 1996.
3. M. van Eekelen and M. de Mol. Reasoning about explicit strictness in a lazy language using mixed lazy/strict semantics. In *Draft Proceedings of the 14th International Workshop on Implementation of Functional Languages, IFL'02*, pages 357–373. Dept. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2002.
4. M. van Eekelen and M. de Mol. Proof tool support for explicit strictness. In *Proceedings of the 17th International Workshop on Implementation and Application of Functional Languages, (IFL'05 selected papers)*, pages 37–54. LNCS 4015, Springer, 2006.
5. A. de la Encina, L. Llana, F. Rubio, and M. Hidalgo-Herrero. Observing intermediate structures in a parallel lazy functional language. In *9th International ACM-SIGPLAN Symposium on Principles and Practice of Declarative Programming, PPDP'07*, pages 111–120. ACM Press, 2007.
6. M. Hidalgo-Herrero. *Semánticas formales para un lenguaje funcional paralelo.* PhD thesis, Dept. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2004.
7. M. Hidalgo-Herrero and Y. Ortega-Mallén. An operational semantics for the parallel language Eden. *Parallel Processing Letters (World Scientific Publishing Company)*, 12(2):211–228, 2002.
8. M. Hidalgo-Herrero, Y. Ortega-Mallén, and F. Rubio. Analyzing the influence of mixed evaluation on the performance of Eden skeletons. *Parallel Computing*, 32(7-8):523–538, 2006.
9. M. Hidalgo-Herrero, Y. Ortega-Mallén, and F. Rubio. Comparing alternative evaluation strategies for stream-based parallel functional languages. In *Proceedings of the 18th International Workshop on Implementation of Functional Languages, (IFL'06 selected papers)*, pages 55–72. LNCS 4449, Springer, 2007.
10. G. Kahn and D. MacQueen. Coroutines and networks of parallel processes. In *IFIP'77*, pages 993–998. Eds. B. Gilchrist. North-Holland, 1977.
11. P. Kelly. *Functional Programming for Loosely-Coupled Multiprocessors.* Pitman, 1989.

12. J. Launchbury. A natural semantics for lazy evaluation. In *ACM Symposium on Principles of Programming Languages, POPL'93*, pages 144–154. ACM Press, 1993.
13. R. Loogen. *Research Directions in Parallel Functional Programming*, chapter 3: Programming Language Constructs, pages 63–92. Eds. K. Hammond and G. Michaelson. Springer, 1999.
14. R. Loogen, Y. Ortega-Mallén, and R. Peña. Parallel functional programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.
15. R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, 1990.
16. S. L. Peyton Jones. *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, 2003.
17. J. H. Reppy. Concurrent ML: Design, application and semantics. In *Proceedings of Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 165–198. LNCS 693, Springer, 1993.
18. F. S. Taylor. *Parallel Functional Programming by Partitioning*. PhD thesis, Imperial College, 1997.
19. P. W. Trinder, K. Hammond, H. W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Funcional Programming*, 8(1):23–60, 1998.
20. P. W. Trinder, K. Hammond, J. Mattson Jr., A. Partridge, and S. L. Peyton Jones. GUM: a portable implementation of Haskell. In *Proceedings of Programming Language Design and Implementation, PLDI'96*, pages 78–88, Philadephia, USA, 1996. ACM Press.
21. P. W. Trinder, H. W. Loidl, and R. F. Pointon. Parallel and Distributed Haskells. *Journal of Functional Programming*, 12(4+5):469–510, 2003.

The Structure of the Essential Haskell Compiler, or Coping with Compiler Complexity

Atze Dijkstra, Jeroen Fokker and S. Doaitse Swierstra

Department of Information and Computing Sciences,
Universiteit Utrecht,
P.O.Box 80.089,
Padualaan 14, Utrecht, Netherlands,
{atze,jeroen,doaitse}@cs.uu.nl,
WWW home page: http://www.cs.uu.nl

Abstract. In this paper we describe the structure of the Essential Haskell Compiler (EHC) and how we manage its complexity, despite its growth from essentials to a full Haskell compiler. Our approach splits both language and implementation into smaller, manageable steps, and uses specific tools to generate parts of the compiler from higher level descriptions.

1 Introduction

Haskell is a perfect example of a programming language which offers many features improving programming efficiency by offering a sophisticated type system. As such it is an answer for the programmer looking for a programming language which does as much as possible of the programmer's job, while at the same time guaranteeing program properties like "well-typed programs don't crash". However, the consequence is that a programming language implementation is burdened by these responsibilities, and consequently becomes quite complex. Haskell thus also is a perfect example of a programming language for which compilers are complex. Testimony to this observation is the Glasgow Haskell Compiler (GHC) [15, 16, 18, 20], which simultaneously incorporates many novel features, is used as a reliable workhorse for many a functional programmer, and offers a research platform for language designers. As a result, modifying GHC requires much knowledge of GHC's internals.

In this paper we show how we deal with the complexity of compiling Haskell in the the Essential Haskell (EH) Compiler (EHC) [7, 8]. EH intends

- to compile full Haskell (the H in EH)
- to offer an implementation in terms of the essential, or desugared, core language constructs of Haskell (the E in EH)
- to provide a solid framework for research (i.e., extendable for experimentation) and education (another interpretation of the E in EH)

In particular the following areas require attention:

– **Implementation complexity** (Section 2) The amount of work a compiler has to do is a source of complexity. We organise the work as a series of smaller transformation steps [18, 22] between various internal representations.
– **Description complexity** (Section 3) The specification of parts of the implementation itself can become complex because low-level details are visible. We use domain specific languages which factor out such low-level details, so they are dealt with automatically.
– **Design complexity** (Section 4) Experiments with language features are usually done in isolation. We describe their implementation in isolation, as a sequence of language variants, building on top of each other.
– **Maintenance complexity** (Section 5) Actual compiler source, its documentation, and its specification tend to become inconsistent over time. We fight such inconsistencies by avoiding their main cause: duplication. Whenever two artefacts have to be consistent, we generate them from a common description.

In the next sections we explain how we deal with each of these complexities.

2 Coping with implementation complexity: transform

EHC is organised as a sequence of transformations between internal representations of the program being compiled. In order to keep the compiler understandable, we keep the transformations simple, and consequently, there are many. This approach is similar to the one taken in GHC [18, 19, 21]. All our transformations are expressed as a full tree walk over the data structure, using a tool for easily defining tree walks (see Section 3.1). At each step in which the representation changes drastically we introduce a separate data structure (or "language"). Fig. 1 shows these languages and the transformations between them:

– **HS** (Haskell) is a representation of the program text as parsed. It is used for desugaring, name and dependency analysis, and making binding groups explicit.
– **EH** (Essential Haskell) is a simplified and desugared representation. It is used for type analysis and code expansion of class system related constructs.
– **Core** is a representation in an untyped λ-calculus.
– **Grin** (Graph reduction intermediate notation) is a representation proposed by Boquist [5, 6] in which local definitions have been made sequential and the need for evaluation has been made explicit.
– **Silly** (Simple imperative little language) is a simple abstraction of an imperative language with an explicit stack and heap, and functions which can be called and tail-called.
– **C** is used here as a universal back-end, hiding the details of the underlying machine. Primitive functions are implemented here.
– **LLVM** (Low level virtual machine) is an imperative language which, other than C, is *intended* to be a universal back-end [14]. We have it under consideration as an alternative route to attain executable code.

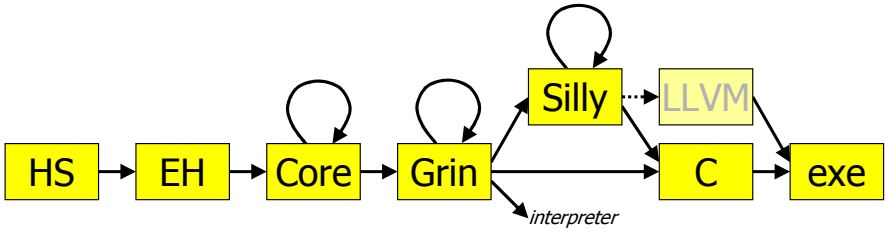


Fig. 1. Intermediate languages and transformations in the EHC pipeline

As can be seen from the figure, the compilation pipeline branches after the Grin stage, offering different modes of compilation:

- Grin code can be interpreted directly by a simple (and thus slow) interpreter.
- Grin code can be translated to C directly. In this mode, the program is represented in a custom bytecode format, stored in arrays, and executed by an interpreter written in C. Its speed is comparable to that of Hugs [1].
- Grin code can be translated to executable code via transformations which perform global program analysis, and generate optimized Silly code, which can be further processed through either the C or LLVM route.

The transformations between the languages mentioned above bring the program stepwise to a lower level of representation, until it can be executed directly. Most of the simplification work however is done by the transformations that are indicated by a loop in Fig. 1, i.e., for which the source and target language are the same. We strive to have many small transformations rather than a few complicated ones. To give an idea, we list a short description of the more important of these transformations. Some of these are necessary simplifications, others are optimisations that can be left out.

- Transformations on the Core language include:
  - Cleanup transformations: *Eta-reduction*, *Eliminating trivial applications*, *Inline let alias*, *Remove unnecessary letrec mutual recursion*
  - *Constant propagation* and *Rename identifiers to unique names*
  - Lambda lifting, split up in: *Full lazyness of subexpressions*, *Lambda/CAF globals passed as argument*, *Float lambda expressions to global level*
- Transformations on the Grin language include:
  - Transformations on separate modules: *Alias elimination*, *Unused name elimination*, *Eval elimination*, *Unboxing*, *Local inlining*
  - Transformations based on a global abstract interpretation that determines possible constructors of actual parameters: *Inline eval operation*, *Remove dead case alternatives and unused functions*, *Global inlining*
  - Transformations that remove higher-level constructs, such as splitting complete nodes into their constituent fields.
- Transformations on the Silly language include:
  - *Shortcut*: avoid unnecessary copying of local variables
  - *Embed*: map local variables to stack positions

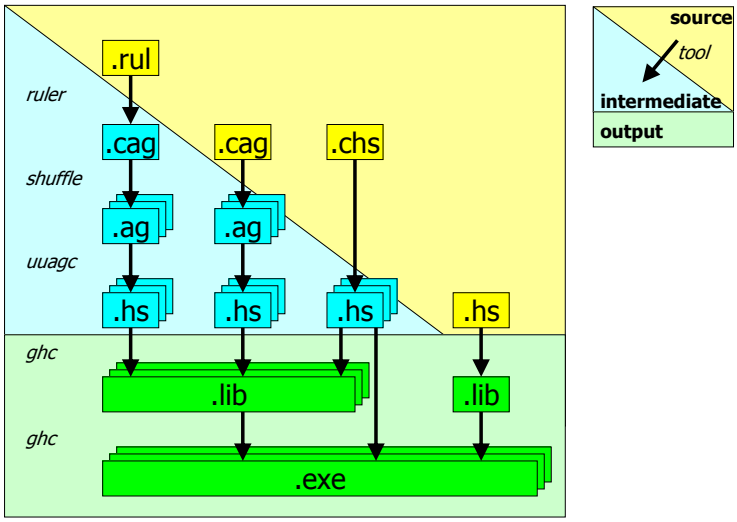


Fig. 2. Chain of tools used to build EHC

3 Coping with description complexity: use tools

Haskell is well suited as an implementation language for compilers, among others because of the ease of manipulating tree structures. Still, if one needs to write many tree walks, especially if these involve multiple passes over complicated syntax trees, the necessary mutually recursive functions tend to become hard to understand, and contain large pieces of boilerplate code. In the implementation of EHC we therefore use a chain of preprocessing tools, depicted in Fig. 2.

We use the following preprocessing tools:

- **UUAGC** (Utrecht University Attribute Grammar Compiler), which enables us to specify abstract syntax trees and tree walks over them using an attribute grammar (AG) formalism [2, 8, 9, 25].
- **Ruler**, a translator for an even more specialized language than AG, which enables a high-level specification of type inferencing, generating both AG code and LaTeX documentation [10].
- **Shuffle**, which deals with the compiler organisation and logistics of many different language features, and provides a form of literate programming.

In the remainder of this section we elaborate on the rationale of UUAGC (Section 3.1) and *Ruler* (Section 3.2). We illustrate their use with example code, which implements part of a Hindley-Milner type checker. In the section on UUAGC this is idealized toy code, but in the section on *Ruler* we show actual code taken from EHC for the same example. In Section 4 and 5 we continue with the rationale and use of *Shuffle*.

3.1 UUAGC, a system for specifying tree walks

Higer-order functional languages are famous for their ability to parameterize functions not only with numbers and data structures, but also with functions and operators. The standard textbook example involves the functions *sum* and *product*, which can be defined separately by tedious inductive definitions:

$$\begin{aligned}
&sum\ [\,] &&= 0\\
&sum\ (x : xs) &&= x + sum\ xs\\
&product\ [\,] &&= 1\\
&product\ (x : xs) &&= x * product\ xs
\end{aligned}$$

This pattern can be generalized in a function *foldr* that takes as additional parameters the operator to apply in the inductive case and the base value:

$$\begin{aligned}
&foldr\ op\ e\ [\,] &&= e\\
&foldr\ op\ e\ (x : xs) &&= x\ `op`\ foldr\ op\ e\ xs
\end{aligned}$$

Once we have this generalized function, we can partially parameterize it to obtain simpler definitions for *sum* and *product*, and many other functions as well:

$$\begin{aligned}
sum &= foldr\ (+)\ 0\\
product &= foldr\ (*)\ 1\\
concat &= foldr\ (+\!\!+)\ [\,]\\
sort &= foldr\ insert\ [\,]\\
transpose &= foldr\ (zipWith\ (:))\ (repeat\ [\,])
\end{aligned}$$

The idea that underlies the definition of *foldr* (capturing the pattern of an inductive definition by adding a function parameter for each constructor of the data structure), can also be used for other data types, and even for multiple mutually recursive data types. Functions that can be expressed in this way are called *catamorphisms* by Bird, and the collective extra parameters to *foldr*-like functions *algebras* [3, 4]. Thus, $((+), 0)$ is an algebra for lists, and $((+\!\!+), [\,])$ is another. In fact, every algebra defines a *semantics* of the data structure.

Outside circles of functional programmers and category theorists, an algebra is simply known as a "tree walk". In compiler construction, algebras could be very useful to define a semantics of a language, or bluntly said to define tree walks over the parse tree. This is not widely done, due to the following problems:

1. Unlike lists, which have a standard function *foldr*, in a compiler we deal with (many) custom data structures to describe the abstract syntax of a language, so we have to invest in writing a custom *fold* function first. Moreover, whenever we change the abstract syntax, we need to change the *fold* function, and every algebra.
2. Generated code can be described as a semantics of the language, but often we need additional semantices: pretty-printed listings, warning messages, and various derived structures for internal use (symbol tables etc.). This can be done in one pass by having the semantic functions in the algebra return tuples, but this makes them hard to handle.

3. Data structures for abstract syntax tend to have many alternatives, so algebras end up to be clumsy tuples containing dozens of functions.
4. In practice, information not only flows bottom-up in the parse tree, but also top-down. E.g., symbol tables with global definitions need to be distributed to the leaves of the parse tree to be able to evaluate them. This can be done by making the semantic functions in the algebra higher order functions, but this pushes the handling of algebras beyond human control.
5. Much of the work is just passing values up and down the tree. The essence of a semantics in the algebra is obscured by lots of boilerplate.

In short: the concepts of catamorphism and algebra apply here, but their encoding in Haskell is cumbersome and becomes prohibitively complex. Many compiler writers thus end up writing ad hoc recursive functions instead of defining the semantics by an algebra, or even resort to non-functional techniques. Others try to capture the pattern using monads [17]. Some succeed in giving a concise definition of a semantics, often using proof rules of some kind, but loose executability. For the implementation they still need conventional techniques, and the issue arises whether the program soundly implements the specified semantics.

To save the nice idea of using an algebra for defining a semantics, we use a preprocessor for Haskell [25] that overcomes the mentioned problems. It is not a separate language; we can still write auxiliary Haskell functions, and use all abstraction techniques and libraries. The preprocessor just allows a few additional constructs, which are translated into custom *fold*-like functions and algebras.

We describe the main features of the preprocessor here, and explain why they overcome the five problems mentioned above. For a start, the grammar of the abstract syntax of the language is defined in **data** declarations, which are like a Haskell **data** declaration with named fields, except that we do not have to write braces and commas and that constructor function names need not be unique. As an example, we show a fragment of EHC that represents a lambda calculus:

```
data Expr
   = Var   name :: Name
   | Let   decl :: Decl   body :: Expr
   | App   func :: Expr   arg  :: Expr
   | Lam   arg  :: Pat    body :: Expr
data Decl
   = Val   pat :: Pat   expr :: Expr
data Pat
   = Var   name :: Name
   | App   func :: Expr   arg :: Expr
```

The preprocessor generates corresponding Haskell **data** declarations (making the constructors unique by prepending the type name, like *Expr_Var*), and more importantly, generates a custom *fold* function. This overcomes problem 1.

For any desired value we wish to compute from a tree, we can declare a "synthesized attribute" (the terminology goes back to Knuth [12]). Attributes can be

defined for one or more data types. For example, we can define that for all three datatypes we wish to synthesize a pretty-printed listing, and that expressions in addition synthesize a type and a variable substitution map:

```
attr Expr Decl Pat syn listing  :: String
attr Expr          syn typ      :: Type
                       varmap   :: [(Name, Type)]
```

In the presence of multiple synthesized attributes, the preprocessor ensures that the semantic functions combine them in tuples, but in our program we can simply refer to the attributes by name. The attribute declarations of a single datatype can even be distributed over the program. This overcomes problem 2.

The value of each attribute needs to be defined for every constructor of every data type which has the attribute. These definitions of the semantics of the language are known as "semantic rules", and start with keyword **sem**. An example is:

```
sem Expr | Let
  lhs.listing = "let " ++ @decl.listing ++ " in " ++ @body.listing
```

This states that the synthesized *listing* attribute of a *Let* expression can be constructed by combining the *listing* attributes of its *decl* and *body* children and some fixed strings. The @ symbol in this context should be read as "attribute", not to be confused with Haskell "as-patterns". The keyword **lhs** refers to the parent of the children @*decl* and @*body*, i.e., the nameless *Expr* at the **l**eft **h**and **s**ide of the grammar rule. At the left of the = symbol, the attribute to be defined is mentioned (here the @ symbol may be omitted); at the right, any Haskell expression can be given. The example below shows the use of a **case** expression and an auxiliary function *substit*, applied to occurrences of child attributes. Also, it shows how to use the value of leaves (@*name* in the example), and how to group multiple semantic rules under a single **sem** header:

```
sem Expr
  | Var  lhs.listing =  @name
  | Lam  lhs.typ     = Type_Arrow (substit @body.varmap @arg.typ) @body.typ
  | App  lhs.typ     = case @func.typ of
                         (Type_Arrow p b) -> substit @arg.varmap b
```

The preprocessor collects and orders all definitions into a single algebra, replacing the attribute references by suitable selections from the results of the recursive tree walk on the children. This overcomes problem 3.

To be able to pass information downward during a tree walk, we can define "inherited" attributes. As an example, it can serve to pass an environment (a lookup table that associates variables to types), which can be consulted when we need to determine the type of a variable:

```
attr Expr inh env :: [(Name, Type)]
sem Expr
  | Var lhs.typ = fromJust (lookup @name @lhs.env)
```

The value to use for the inherited attributes can be defined in semantic rules higher up the tree. In the example, *Let* expressions extend the environment which they inherited themselves with the new environment synthesized by the declaration, in order to define the environment to be used in the body:

```
sem Expr
  | Let body.env = @decl.newenv ++ @lhs.env
```

The preprocessor translates inherited attributes into extra parameters for the semantic functions in the algebra. This overcomes problem 4.

In practice, there are many situations where inherited attributes are passed unchanged as inherited attributes for the children. For example, the environment is passed down unchanged at *App* expressions. This can be quite tedious to do:

```
sem Expr
  | App func.env = @lhs.env
        arg.env  = @lhs.env
```

Since the code above is trivial, the preprocessor has a convention that, unless stated otherwise, attributes with the same name are automatically copied. So, the attribute *env* that an *App* expression inherited from its parent, is automatically copied to the children which also inherit an *env*, and the tedious rules above can be omitted. This captures a pattern that is often addressed by introducing a *Reader* monad [11]. Similar automated copying is performed for synthesized attributes, so if they need to be passed unchanged up the tree, this does not need an explicit encoding, nor a *Writer* monad.

It is allowed to declare both an inherited and a synthesized attribute with the same name. In combination with the copying mechanisms, this enables us to silently thread a value through the entire tree, updating it when necessary. Such a pair of attributes can be declared as if it were a single "threaded" attribute. A useful application is to thread an integer value as a source for fresh variable names, incrementing it whenever a fresh name is needed during the tree walk. This captures a pattern for which otherwise a *State* monad would be needed.

The preprocessor automatically generates semantic rules in the standard situations described, and this overcomes problem 5.

3.2 Ruler, a system for specifying type rule implementations

With the AG language we can describe the part of a compiler related to tree walks concisely and efficiently. However, this does not give us any means of looking at such an implementation in a more formal setting. Currently a formal description of Haskell, suitable for both the generation of an implementation and use in formal proofs, does not exist. For EH we make a step in that direction with *Ruler*, which allows us to have both an implementation and a type rule presentation with the guarantee that these are mutually consistent.

With *Ruler* we describe type rules in such a way that both a LaTeX rendering and an AG implementation can be generated from such a common type rule description. We demonstrate the use of *Ruler* by showing *Ruler* code for the Hindley-Milner type inferencing of function application *App* (see previous section for this and other names for expression terms). We omit a thorough explanation of the meaning of these fragments, as our purpose here is to demonstrate how we can describe these fragments with one common piece of *Ruler* source text. Also we do not intend to be complete in our description; we point out those parts corresponding to the distinguishing features of the *Ruler* system.

From a single source, to be discussed below, *Ruler* can both generate a LaTeX rendering for human use in technical writing:

$$\frac{\begin{array}{c} v \text{ fresh} \\ \Gamma ; \mathcal{C}^k ; v \to \sigma^k \vdash^e e_1 : \sigma_a \to \sigma \rightsquigarrow \mathcal{C}_f \\ \Gamma ; \mathcal{C}_f ; \sigma_a \vdash^e e_2 : \_ \rightsquigarrow \mathcal{C}_a \end{array}}{\Gamma ; \mathcal{C}^k ; \sigma^k \vdash^e e_1\ e_2 : \mathcal{C}_a \sigma \rightsquigarrow \mathcal{C}_a} (\text{E.APP}_{HM})$$

and its corresponding AG implementation, for further processing by UUAGC:

```
sem Expr
  | App (func.gUniq, loc.uniq1)
                = mkNewLevUID @lhs.gUniq
        func.knTy = [mkTyVar @uniq1] `mkArrow` @lhs.knTy
        (loc.ty_a_, loc.ty_)
                = tyArrowArgRes @func.ty
        arg .knTy = @ty_a_
        loc .ty   = @arg.tyVarMp ⊕ @ty_
```

The given rule describes the algorithmic typing of a function application in a standard lambda calculus with the Hindley-Milner type system. The rule involves four judgements: three premises and a conclusion. All judgements but the one involving the freshness of a type variable have the same structure as these all relate various properties of expressions: the conclusion about the function application $e_1\ e_2$, the premises about the function e_1 and argument e_2.

Ruler exploits this commonality by means of the *scheme* of a judgement, which can be thought of as the type of a judgement:

```
scheme expr =
  holes [node e : Expr, inh valGam : ValGam, inh knTy : Ty
        , thread tyVarMp : C, syn ty : Ty]
  judgeuse tex valGam; tyVarMp.inh; knTy ⊢ .."e" e : ty ⇝ tyVarMp.syn
  judgespec valGam; tyVarMp.inh; knTy ⊢ e : ty ⇝ tyVarMp.syn
```

The scheme declaration for expressions *expr* defines a common framework for the judgements of each *expr* term, such as *App* and *Lam* (lambda expression):

– **holes**: names, types and modifiers of placeholders (or *holes*) for various properties, such as e and *valGam*
– **judgeuse tex** (unparsing): LaTeX pretty printing in terms of holes and other symbols, such as $\vdash$ and $\rightsquigarrow$
– **judgespec** (parsing): concrete syntax for specifying a complete judgement.

Modifiers **node**, **inh**, **syn**, and **thread** are required when generating an AG implementation, to be able to turn a rule into an algorithm. The **thread** modifier introduces two holes with suffix .**inh** and .**syn**, corresponding to an AG threaded attribute. For a LaTeX rendering these modifiers are ignored, but additional formatting is required to map identifiers to LaTeX symbols, for example:

$$
\begin{array}{ll}
\mathit{valGam} & \mapsto \Gamma \\
\mathit{ty} & \mapsto \sigma \\
\mathit{knTy} & \mapsto \sigma^k \\
\mathit{tyVarMp}.\mathbf{inh} & \mapsto \mathcal{C}^k
\end{array}
$$

We omit further discussion of lexical issues.

The rule for function application *App* now is defined by judgements introduced with the keyword **judge**:

$$
\begin{array}{l}
\mathbf{rule}\ e.\mathit{app} = \\
\quad \mathbf{judge}\ \mathit{tvarvFresh} \\
\quad \mathbf{judge}\ \mathit{expr} = \mathit{tyVarMp}.\mathbf{inh}; \mathit{tyVarMp}; (v \rightarrow \mathit{knTy}) \\
\qquad\qquad\qquad \vdash \mathit{eFun} : (\mathit{ty}.a \rightarrow \mathit{ty}) \rightsquigarrow \mathit{tyVarMp}.\mathit{fun} \\
\quad \mathbf{judge}\ \mathit{expr} = \mathit{tyVarMp}.\mathit{fun}; \mathit{valGam}; \mathit{ty}.a \\
\qquad\qquad\qquad \vdash \mathit{eArg} : \mathit{ty}.a \rightsquigarrow \mathit{tyVarMp}.\mathit{arg} \\
\quad - \\
\quad \mathbf{judge}\ \mathit{expr} = \mathit{tyVarMp}.\mathbf{inh}; \mathit{valGam}; \mathit{knTy} \\
\qquad\qquad\qquad \vdash (\mathit{eFun}\ \mathit{eArg}) : (\mathit{tyVarMp}.\mathit{arg}\ \mathit{ty}) \rightsquigarrow \mathit{tyVarMp}.\mathit{arg}
\end{array}
$$

For each judgement its scheme is specified (*expr* in the example). The **judgespec** of the corresponding scheme is used to check the concrete syntax and to bind the holes of the judgement to the concrete values specified by the judgement. From this rule definition a LaTeX rendering can straightforwardly be generated.

For the generation of an AG implementation we need information as specified by hole modifiers. In an AG implementation the structure of the tree drives the choice of which rule to apply. One of the holes needs to correspond to a node of such a tree; the modifier **node** specifies which. Other holes correspond to attributes, which have a direction: top-down (inherited, indicated by modifier **inh**) bottom-up (synthesized, indicated by **syn**) or both (indicated by **thread**).

The judgement with scheme *tvarvFresh* is an example of a judgement which does not fit into a tree structure as required by AG: it does not refer to a **node** hole. For such schemes, called *relation*s, an explicit AG implementation must be provided. We omit further discussion of relations.

Finally, *Ruler* also provides support for incremental language specification, which we discuss in Section 4.

	Haskell	extensions
1	λ-calculus, type checking	
2	type inferencing	
3	polymorphism	
4		higher ranked types, existentials
5	data types	
6	kind inferencing	kind signatures
7	records	tuples as records
8	code generation	GRIN
9	class system	
10		extensible records
11	type synonyms	
12		explicit parameter passing for implicit parameters *
13		higher order predicates *
14–19		*reserved for other extensions* *
20	module system	
95	class instance deriving *	
96		exception handling
97	numbers: Integer, Float, Double *	
98	IO *	
99	the rest for full Haskell *	

Fig. 3. EH language variants (work in progress is marked by an asterisk '*')

4 Coping with design complexity: grow stepwise

To cope with the many features of Haskell, EHC is constructed as a sequence of compilers, each of which adds new features. This enables us to experiment with non-standard features. Fig. 3 shows the standard and experimental features currently introduced in each language variant. The sequence is a didactical choice of increasingly complex features; it is not the development history. Every compiler in the sequence can actually be built out of the repository.

Each language variant in the sequence is described as a delta with respect to the previous language. Usually this delta is a pure addition, but other combinations are possible when:

- language features interact
- the overall implementation and individual increments interact: an increment is described in the context of the implementation of preceding variants, whereas such a context must anticipate later changes.

Conventional compiler building tools are neither aware of partitioning into increments nor aware of their interaction. We use a separate tool, called *Shuffle*, to take care of such issues. We describe *Shuffle* in the next section.

For each language variant in the sequence, various artefacts are created, such as example programs, a definition of the semantics, an implementation, and documentation. Fig. 4 shows some of these artefacts for some language variants. The first row shows an example program for each language variant. The second row shows a description of part of the semantics of the language variants (the type rule for functional application), by way of the LaTeX rendering of the type rule generated by *Ruler*. The third row shows the implementation of this type rule in the compiler, by way of the AG output generated by *Ruler* (from the same source). Example language variants shown in the columns of Fig. 4 are EH1 (simply explicitly typed λ-calculus), EH3 (adding polymorphic type inference), and EH4 (adding higher-ranked types).

5 Coping with maintenance complexity: generate, generate and generate

For any large programming project the greatest challenge is not to make the first version, but to be able to make subsequent versions. In order to facilitate change, the object of change should be isolated and encapsulated. Although many programming languages support encapsulation, this is not sufficient for the construction of a compiler, because each language feature influences not only various parts of the compiler (parser, structure of abstract syntax tree, type system, code generation, runtime system) but also other artefacts such as specification, documentation, and test suites. Encapsulation of a language feature in a compiler therefore is difficult, if not impossible, to achieve.

We mitigate the above problems by using *Shuffle*, a separate preprocessor. In all source files, we annotate to which language variants the text is relevant. *Shuffle* preprocesses all source files by selecting and reordering those fragments (called *chunk*s) that are needed for a particular language variant. Source code for a particular Haskell module is stored in a single "chunked Haskell" (.chs) file, from which *Shuffle* can generate the Haskell (.hs) file for any desired variant (see Fig. 2, where the stacks of intermediate files denote various variants of a module). Source files can be chunked Haskell code, chunked AG code, but also chunked LaTeX text and code in other languages we use.

Shuffle behaves similar to literate programming tools [13] in that it generates program source code. The key difference is that with the literate programming style program source code is generated out of a file containing program text plus documentation, whereas *Shuffle* combines chunks for different variants from different files into either program source code or documentation.

Shuffle offers a different functionality than version management tools: these offer historical versions, whereas *Shuffle* offers the simultaneous handling of different variants from a single source.

For example, for language variant 2 and 3 (on top of 2) a different wrapper function *mkTyVar* for the construction of the internal representation of a type variable is required. In variant 2, *mkTyVar* is equal to the constructor *Ty_Var*:

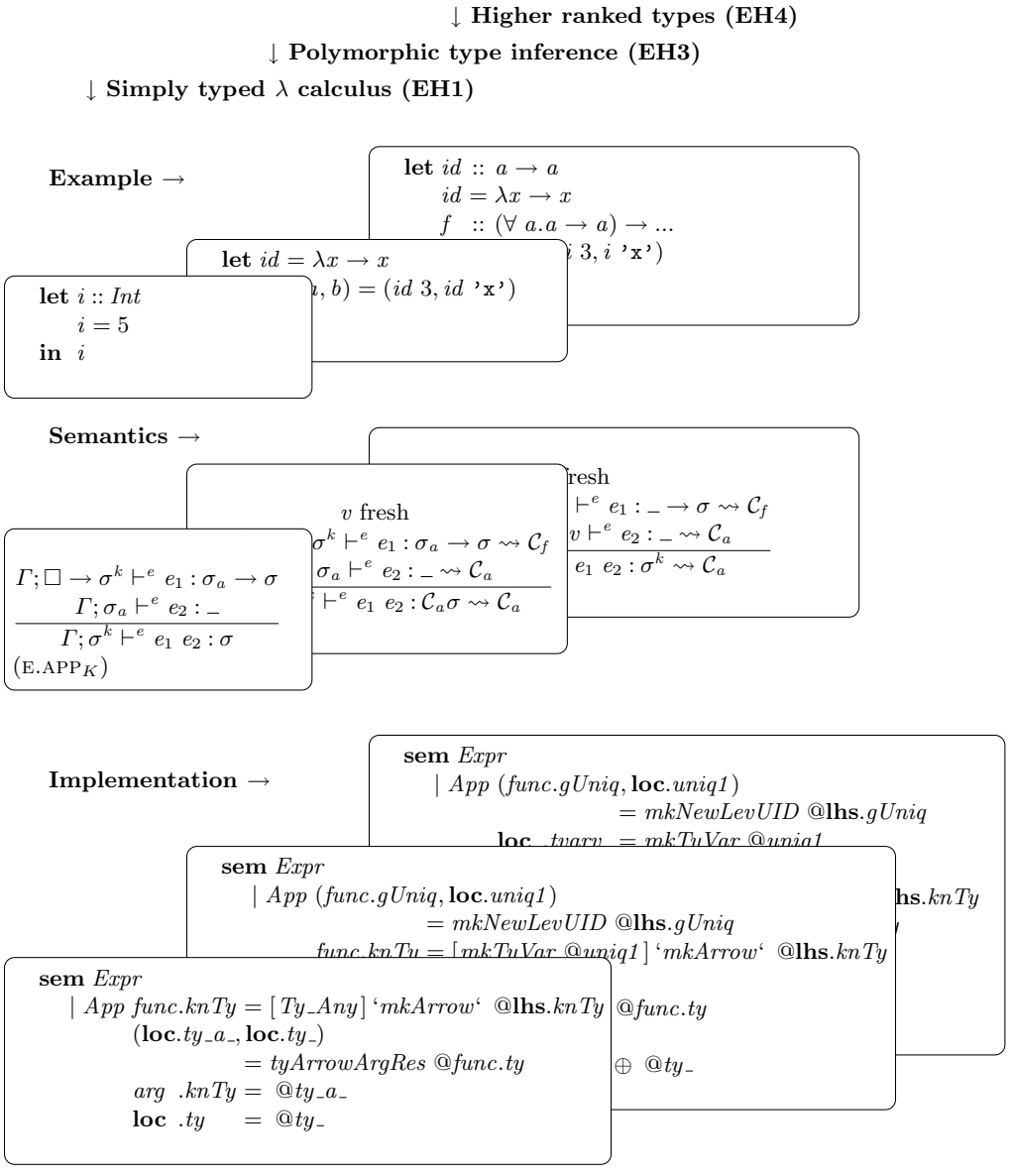


Fig. 4. Examples of created artefacts (rows) for various language variants (columns)

$mkTyVar :: TyVarId \rightarrow Ty$
$mkTyVar\ tv = Ty\_Var\ tv$

However, version 3 introduces polymorphism as a language variant, which requires additional information for a type variable, which defaults to $TyVarCateg\_Plain$ (we do not further explain this):

$mkTyVar :: TyVarId \rightarrow Ty$
$mkTyVar\ tv = Ty\_Var\ tv\ TyVarCateg\_Plain$

These two Haskell fragments are generated from the following *Shuffle* source:

```
%%[2.mkTyVar
mkTyVar :: TyVarId -> Ty
mkTyVar tv = Ty_Var tv
%%]

%%[3.mkTyVar -2.mkTyVar
mkTyVar :: TyVarId -> Ty
mkTyVar tv = Ty_Var tv TyVarCateg_Plain
%%]
```

The notation `%%[2.mkTyVar` begins a chunk for variant 2 with name *mkTyVar*, ended by `%%]`. The chunk for `3.mkTyVar` explicitly specifies to override `2.mkTyVar` for variant 3. Although the type signature can be factored out, we refrain from doing so for small definitions.

In summary, *Shuffle*:

- uses notation `%%[` ... `%%]` to delimit and name text chunks
- names chunks by a variant number and (optional) additional naming
- allows overriding of chunks based on their name
- combines chunks upto an externally specified variant, using an also externally specified variant ordering.

6 Experiences

Development and debugging. The partitioning into variants is helpful for both development and debugging. It is always clear to which variant code contributes, and if a problem arises one can use a previous variant in order to isolate the problem. Experimentation also benefits because one can pick a suitable variant to build upon, without being hindered by subsequent variants.

However, on the downside, there are builtin system wide assumptions, for example about how type checking is done. We are currently investigating this issue in the context of *Ruler*.

Use in research and education. EHC is constructed as a library and a toplevel compiler driver (see Fig. 2), facilitating the use of the implementation of EHC by other programs.

We intend to use the first three language variants (Fig. 3) in our basic course on compiler construction, thus providing students with a realistic integrated introduction to language design, compiler implementation, and software engineering. This approach is similar to that in Pierce's textbook [23], however, in contrast we focus on a realistic implementation of full Haskell instead of small independent implementations of isolated type systems.

Improvements. Although our approach to cope with complexity indeed leads to the advocated benefits, there is room for improvement:

- **Ruler and type rules** With *Ruler* we generate both AG and LaTeX. *Ruler* notation, AG, and LaTeX have a similar structure. Consequently *Ruler* does not hide as much of the implementation as we would like. We are investigating a more declarative notation for *Ruler*.
- **Loss of information while transforming** With a transformational approach to different intermediate representations, the relation of later stages to earlier available information becomes unclear. For example, by desugaring to a simpler representation, source code of the user program is reordered and the original source location has to be propagated as part of the AST. Such information flow patterns are not yet automated.
- **High level description and efficiency** Using a high level description usually also provides opportunities to optimise at a low level. For attribute grammars a large body of optimisations are available [24], some of which are finding their way into our AG system.
- **Stepwise approach vs. aspectwise approach** EH's stepwise approach imposes a fixed order in which language constructs are implemented on top of each other. Ideally one should be able to arbitrarily combine separate language constructs as aspects (independent implementation fragments), but interaction between language constructs hinders this flexibility. We are investigating the use of aspects in the context of *Ruler*.

Status and plans. We are working towards a release of EHC as a Haskell compiler: variant 99 in the sequence. At the moment, we can compile a prelude and run programs with a bytecode interpreter. We intend to work on AG optimisations, on using LLVM [14] as a backend, and on GRIN global transformations.

References

1. Hugs 98. `http://www.haskell.org/hugs/`, 2003.
2. Arthur Baars. Attribute Grammar System. `http://www.cs.uu.nl/wiki/HUT/AttributeGrammarSystem`, 2004.
3. R. Bird and O. de Moor. *The algebra of programming.* Prentice Hall, 1996.

4. Richard S. Bird. Using Circular Programs to Eliminate Multiple Traversals of Data. *Acta Informatica*, 21:239–250, 1984.
5. Urban Boquist. *Code Optimisation Techniques for Lazy Functional Languages, PhD Thesis*. Chalmers University of Technology, 1999.
6. Urban Boquist and Thomas Johnsson. The GRIN Project: A Highly Optimising Back End For Lazy Functional Languages. In *Selected papers from the 8th International Workshop on Implementation of Functional Languages*, 1996.
7. Atze Dijkstra. EHC Web. `http://www.cs.uu.nl/wiki/Ehc/WebHome`, 2004.
8. Atze Dijkstra. *Stepping through Haskell*. PhD thesis, Utrecht University, Department of Information and Computing Sciences, 2005.
9. Atze Dijkstra and S. Doaitse Swierstra. Typing Haskell with an Attribute Grammar. In *Advanced Functional Programming Summerschool*, number 3622 in LNCS. Springer-Verlag, 2004.
10. Atze Dijkstra and S. Doaitse Swierstra. Ruler: Programming Type Rules. In *Functional and Logic Programming: 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006*, number 3945 in LNCS, pages 30–46. Springer-Verlag, 2006.
11. Mark P. Jones. Typing Haskell in Haskell. In *Haskell Workshop*, 1999.
12. D.E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.
13. D.E. Knuth. Literate Programming. *Journal of the ACM*, (42):97–111, 1984.
14. Chris Lattner. The LLVM Compiler Infrastructure Project. `http://llvm.org/`, 2007.
15. Simon Marlow. The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`, 2004.
16. Simon Marlow and Simon Peyton Jones. The New GHC/Hugs Runtime System. `http://citeseer.ist.psu.edu/marlow98new.html`, 1998.
17. Erik Meijer and Johan Jeuring. Merging monads and folds for functional programming. In *First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden*, number 925 in LNCS. Springer-Verlag, May 1995.
18. Simon Peyton Jones. Compiling Haskell by program transformation: a report from the trenches. In *European Symposium On Programming*, pages 18–44, 1996.
19. Simon Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Phil Wadler. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, 1992.
20. Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, pages 393–434, 2002.
21. Simon Peyton Jones and Andre Santos. Compilation by Transformation in the Glasgow Haskell Compiler. `http://citeseer.ist.psu.edu/peytonjones94compilation.html`, 1994.
22. Simon Peyton Jones and Andre Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1-3), 3-47 1998.
23. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
24. Joao Saraiva. *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Utrecht University, 1999.
25. S. Doaitse Swierstra, P.R. Azero Alocer, and J. Saraiava. Designing and Implementing Combinator Languages. In Doaitse Swierstra, Pedro Henriques, and José Oliveira, editors, *Advanced Functional Programming, Third International School, AFP'98*, number 1608 in LNCS, pages 150–206. Springer-Verlag, 1999.

XHaskell – Adding Regular Expression Types to Haskell

Martin Sulzmann and Kenny Zhuo Ming Lu

School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
{sulzmann,luzm}@comp.nus.edu.sg

Abstract. We present an extension of Haskell, baptized XHaskell, which combines parametric polymorphism, algebraic data types and type classes found in Haskell with regular expression types, subtyping and regular expression pattern matching found in XDuce. Such an extension proves in particular useful for the type-safe processing of XML data. For example, we can express XQuery and XPath style features via XHaskell combinators. We have implemented the system which can be used in combination with the Glasgow Haskell Compiler.

1 Introduction

Functional programming and XML processing should be a good match. Higher-order functions and parametric polymorphism equip the programmer with powerful abstraction facilities while pattern matching over algebraic data types allows for a convenient notation to specify XML transformations. In the Haskell context, there are a number of tools, for example see [WR99,Sch07], which provide support for parsing, generating and transforming XML documents.

Unfortunately, XML processing in Haskell does not provide the same static guarantees compared to XML processing in domain specific languages such as XDuce [HP00] and variants such as CDuce [BCF03]. These languages natively support regular expression types and (semantic) subtype polymorphism [HVP05] and can thus give much stronger static guarantees about the well-formedness of programs. In combination with regular expression pattern matching [HP01], we can write sophisticated and concise XML transformations.

Previous work attempts to close the gap between XDuce and Haskell but some limitations remain. For example, the work in [BFS04] introduces a preprocessor to provide for regular expression pattern matching. On the down side, the approach is untyped and only supports lists. The combinator library to generate XML values introduced in [Thi02] makes use of the Haskell type class system to check for correctness of constructed values. But neither destruction (pattern matching) nor subtyping among XML values is supported. There are a number of further examples [KL05,Kis07,LS04] where Haskell's type extensions are used to encode domain-specific language extensions. While these works are impressive, they often lead to less natural programs compared to writing the same in XDuce. Another important point is that a language extension comes with

a new compiler which not only enables more optimizations but also allows for better (type) error messages compared to providing the extension via a library.

In this paper, we introduce an extension of Haskell, baptized XHaskell, which integrates XDuce features such as regular expression types, subtyping and regular expression pattern matching into Haskell. Closely related to our work is XMLambda [MS99,SM01]. However, our approach is more powerful because we can express more subtyping relations involving complex types such as $(a^* \mid b^*)$. In addition, we also support the combination of regular expression types and type classes which to the best of our knowledge has not been studied before.

Specifically, our contributions are:

- We introduce XHaskell via examples and demonstrate that the combination of regular expression types with algebraic data types (Section 2), parametric polymorphism (Section 3) and type classes (Section 4) yields a highly expressive system. For example, we can express XQuery and XPath style features via XHaskell combinators.
- We establish sufficient conditions which guarantee that type checking of XHaskell remains decidable (Section 5).
- We have fully implemented the system which can be used in combination with the Glasgow Haskell Compiler. We have taken care to provide meaningful type error messages in case the static checking of programs fails. Our system also allows to defer some static checks until run-time (Section 6.1).
- We make use of GHC-as-a-library so that the XHaskell programmer can easily integrate her programs into existing applications and take advantage of the many libraries available in GHC. We also provide a convenient interface to the HaXML parser (Section 6.2).

A complete description of XHaskell's static semantics, described in terms of a type-directed type-preserving translation from XHaskell to a System F style target language, can be found in an accompanying technical report [SL07b]. A sketch of the key ideas is given in Section 5. Further related work in the context of Java, C#, ML and XDuce is discussed in Section 7. Section 8 concludes.

2 Regular Expression and Data Types

In XHaskell we can mix algebraic data types and regular expression types. Thus, we can give a recast of the classic XDuce example also found in [HP00]. First, we provide some type definitions.

```
data Person    = Person  Name (Tel?) (Email*)
data Name      = Name String
data Tel       = Tel String
data Email     = Email String
data Entry     = Entry (Name,Tel)
```

The above extend data type definitions as found in Haskell. The novelty is the use of regular expression notation on the right-hand sides. Thus, we can for example describe that an address book consists of an arbitrary sequence of persons. Each person is described by a name, an optional telephone number and an arbitrary sequence of email addresses and so on.

Like in Haskell, we can now write functions which pattern match over the above data types. The following function (possibly) turns a single person into a phone book entry.

```
pToE :: Person -> Entry?
pToE (Person (n:: Name) (t::Tel) (es :: Email*)) = Entry (n,t)
pToE (Person (n:: Name) (t::()) (es :: Email*)) = ()
```

In the first clause we use the combination of Haskell style patterns and XDuce style type-based regular expression patterns to check whether a person has a telephone number. In the body of the second clause, we use semantic subtyping. The empty sequence value `()` of type `()` is a subtype of `(Entry?)` because the language denoted by `()` is a subset of the language denoted by `(Entry?)`. Hence, we can conclude that the above program is type correct.

XHaskell programs are translated by using a structured representation of values of regular expression types. For example, we use lists to represent sequences and sum types such as `data Or a b = L a | R b` to represent the regular expression choice operator. Thus, the source definition

```
data Person = Person  Name (Tel?) (Email*)
```

translates to the target definition

```
data Person = Person  Name (Or Tel ()) [Email]
```

Some readers may argue why not use the target definition in the first place. That is, use Haskell instead of XHaskell from the start. The problem is that we loose the convenience of having subtyping and regular expression pattern matching. Concretely, in the body of function `pToE` we must insert some explicit tags, here `L` for the first clause and `R` for the second clause, to ensure that the program type checks in Haskell. These tags effectively represent (up-cast) coercions and are automatically inserted by the XHaskell compiler. Similarly, the Haskell programmer must explicitly translate regular expression pattern matching into plain Haskell pattern matching. The XHaskell compiler will automatically insert the (down-cast) coercions, representing the regular expression pattern match, for the programmer. We believe that this is highly useful when writing more complex programs. The XHaskell programs will be more concise and readable compared to writing an equivalent program in Haskell.

To disambiguate the outcome of matching, we employ the longest match policy. For instance, the following program removes the longest sequence of spaces from the beginning of a sequence of spaces and texts.

```
data Space = Space
data Text = Text String

longestMatch :: (Space|Text)* -> (Space|Text)*
longestMatch (s :: Space*, r :: (Space|Text)*) = r
```

The sub-pattern `(s :: Space*)` is potentially ambiguous because it matches an arbitrary number of spaces. However, in XHaskell we follow the longest match policy which enforces that sub-pattern `(s :: Space*)` will consume the longest sequence of spaces. For example, application of `longestMatch` to the value `(Space, Space, Text ``Hello'', Space)` yields `(Text ``Hello'', Space)`.

XHaskell also provides support for XML-style attributes.

```
data Book = Book {{author :: Author?, year :: Year}}
type Author = String
type Year = Int

findBooks :: Year -> Book* -> Book*
findBooks yr (b@Book{{year = yr'}},bs :: Book*) =
   if (yr == yr')
      then (b, findBooks yr bs)
      else (findBooks yr bs)
findBooks yr (bs :: ()) = ()
```

The above program filters out all books published in a specified year. The advantage of attributes `author` and `year` is that we can access the fields within a data type by name rather than by position. For example, the pattern `Book{{year = yr'}}` extracts the year out of a book whereas the pattern `b@` allows us to use `b` to refer to this book.

Attributes in XHaskell resemble labeled data types in Haskell. But there are some differences, therefore, we use a different syntax. The essential difference is that attributes may be optional. For example, `Book {{year = 1997}}` defines an author-less book published in 1997. This is possible because the attribute `author` has the optional type `Author?`. In case of

```
findGoethe :: Book* -> Book*
findGoethe (b@Book{{author = "Goethe", year = _}},bs :: Book*) =
        (b, findGoethe bs)
findGoethe _ = ()
```

the first clause applies if the author is present and the author is Goethe. In all other cases, i.e. the author is not Goethe, the book does not have an author at all or the sequence of books is empty, the second clause applies. Another (minor) difference between attributes in XHaskell and labeled data types in Haskell is that in XHaskell a attribute name can be used in more than one data type.

```
data MyBook = MyBook {{author :: Author?, year :: Year, price :: Int}}
```

This is more a matter of convenience and relies on the assumption that we use the attribute in a non-polymorphic context only.

3 Regular Expression Types and Parametric Polymorphism

We can also mix parametric polymorphism with regular expressions. Thus, we can write a polymorphic traversal function for sequences similar to the `map` function in Haskell.

```
mapStar :: (a -> b) -> a* -> b*
mapStar f (x :: ()) = x
mapStar f (x :: a, xs :: a*) =  (f x, mapStar f xs)
```

In the above, we assume that type annotations are lexically scoped. For example, variable `a` in the pattern `x::a` refers to `mapStar`'s annotation.

We can now straightforwardly specify a function which turns an address into a phone book by mapping function `pToE` over the sequence of `Person`s.

```
data Book a    = Book a*
type Addrbook  = Book Person
type Phonebook = Book Entry

addrbook :: Addrbook -> Phonebook
addrbook (Book (x :: Person*)) = Book (mapStar pToE x)
```

Notice the we also support the combination of regular expressions and parametric data types.

Once we have `mapStar` it is not too difficult to define `filterStar` and thus we can express star-comprehension similar to the way list-comprehension are expressed via `map` and `filter` in Haskell. Star-comprehension provide for a handy notation to write XQuery style programs.

Here is re-formulation of the `findBooks` function using star-comprehension.

```
findBooks' :: Year -> Book* -> Book*
findBooks' yr (bs :: Book*) = [ b | b@Book{{year = yr'}} <- bs, yr == yr']
```

Like list-comprehensions, a star-comprehension consists of a sequence of statements. Concretely, the above star-comprehension has two essential statements. The first statement `b@Book{{year = yr'}} <- bs` is a generator. For each book element `b` in `bs`, we extract the year of publication attribute and bind it to `yr'`. Via the next statement, we then check whether `yr` is equal to `yr'`. If this is the case we return `b`. In XQuery, the above could be written as follows

```
declare function findbooks' ($yr, $bs) {
  for $b in $bs
  where $b/@year = $yr
  return $b
}
```

where the for-clause iterates through a sequence of books, and the where-clause filters out those books were published in year `$yr`.

Parametric polymorphism also poses some challenges. One issue is inference of type instances of polymorphic functions. For example, consider the following `foldStar` function for sequences.

```
foldStar :: (a -> b -> a)-> a -> b* -> a
foldStar f x (y::()) = x
foldStar f x (y::b, ys::b*) =  foldStar f (f x y) ys
```

We infer the missing pattern annotations, which are `f::a->b->a` and `x::a`, using well-established techniques [HP01,Hos03]. Thus, we can straightforwardly infer that `foldStar` is used at type instance `(a -> b -> a)-> a -> b* -> a` by applying standard local inference methods [PT00]. Similar methods are also applied in other languages such as GenericJava and C♯ 2.0. What makes things slightly more complicated for us is the presence of subtyping.

Let's consider an example to explain this point in more detail. Suppose we use `foldStar` to build more complex transformations. For example, we want to transform a sequence of alternate occurrences of `a`'s and `b`'s such that all `a`'s occur before the `b`'s. We can specify this transformation via `foldStar` as follows

```
transform :: (a|b)* -> (a*,b*)
```

```
transform xs = foldStar ((\x -> \y -> case y of
                           (z::a) -> (z,x)
                           (z::b) -> (x,z)
                         ) :: (a*,b*) -> (a|b) -> (a*,b*))
                () xs
```

We assume that the types of lambda-bound variables are explicitly provided. See the type annotation in the function body. The challenge here is to infer that `foldStar` is used at type instance

```
((a*,b*)->(a|b)->(a*,b*))->(a*,b*)->(a|b)*->(a*,b*)
```

From the types of the arguments and the result type of `transform`'s annotation we infer the type

```
((a*,b*)->(a|b)->(a*,b*))->()->(a|b)*->(a*,b*)
```

But this type does not exactly match the above type. The mismatch occurs at the second argument position. Our solution is to take into account subtyping when checking for type instances. We find that $\vdash$ `()` $\leq (a^*, b^*)$ and therefore the above program is accepted.

A second issue when combining parametric polymorphism and regular expressions is to guarantee that the meaning of programs remains unambiguous. The following function filters out all a's out of squence of a's or b's.

```
filter :: (a|b)* -> b*
filter (x :: b, xs :: (a|b)*) = (x, filter xs)
filter (x :: a, xs :: (a|b)*) = filter xs
filter () = ()
```

The question is what happens if we use `filter` at type instance `(C|C)* -> C*` where `C` is some arbitrary type? XHaskell functions are type-checked and translated independently from any specific use site. This is clearly important to ensure modularity. The consquence is that we unexpectedly may filter out all `C`'s if we apply `filter` to a sequence of `C`'s. On the other hand, the monomorphized version

```
filterCC :: (C|C)* -> C*
filterCC (x :: C, xs :: (C|C)*) = (x, filterCC xs)
filterCC (x :: C, xs :: (C|C)*) = filterCC xs
filterCC () = ()
```

will not filter out any `C`'s at all. To summarize. The issue is that that polymorphic function used at a monomorphic instance may behave differently compared to the monomorphized function. The solution is to reject ambiguous uses of `filter` by checking the instantiation sites. The instance `(C|C)* -> C*` is ambiguous whereas the instance `(A|B)* -> B*` is clearly fine (that is unambiguous). The exact details of the unambiguity check are beyond the scope of this paper. For a comprehensive treatment of this subject, we refer the interested to [SL07a].

4 Regular Expression Types and Type Classes

XHaskell also supports the combination of type classes and regular expression types. For example, we can define `(*)` to be an instance of the `Functor` class.

```
instance Functor (*) where
  fmap = mapStar
```

In our next example we define an instance for equality among a sequence of types.

```
instance Eq a => Eq a* where
  (==) (xs::()) (ys::()) = True
  (==) (x::a, xs::a*) (y::a, ys::a*) = (x==y)&&(xs==ys)
  (==) _ _ = False
```

In our third example, we show how to express a generic set of XPath operations in XHaskell.

```
class XPath a b where
  (//) :: a -> b -> b*

instance XPath a () where
  (//) _ _ = ()

instance XPath a t => XPath a* t where
  (//) xs t = mapStar (\x -> x // t) xs

instance (XPath a t, XPath b t) => XPath (a|b) t where
  (//) (x::a) t = x // t
  (//) (x::b) t = x // t
```

The operation `e1 // e2` extracts all "descendants" of `e1` whose type is equivalent to `e2`'s type.

In our last example, we show that it is very simple to write a pretty-printer for XML data in XHaskell using type classes and regular expression types.

```
class Pretty a where
  pretty :: a -> [Char]

instance Pretty a => Pretty a* where
  pretty xs = foldl (++) [] (mapStar pretty xs)

instance (Pretty a, Pretty b) => Pretty (a|b) where
  pretty (x :: a) = pretty x
  pretty (x :: b) = pretty x

instance (Pretty a, Pretty b) => Pretty (a,b) where
  pretty ((x :: a), (y :: b)) = (pretty x) ++ (pretty y)

instance Pretty () where
  pretty _ = ""

instance Pretty [Char] where
  pretty x = x

instance Pretty Person where
  pretty (Person (n:: Name) (t::Tel?) (es :: Email*)) =
```

```
    "<person>" ++ pretty n ++ pretty t ++ pretty es ++ "</person>"

instance Pretty Name where
  pretty (Name (s :: [Char])) = "<name>" ++ s ++ "</name>"

instance Pretty Tel where
  pretty (Tel (s :: [Char])) = "<tel>" ++ s ++ "</tel>"

instance Pretty Email where
  pretty (Email (s :: [Char])) = "<email>" ++ s ++ "</email>"
```

5 Properties

The meaning of XHaskell is explained via a type-preserving translation scheme to a System F style target language. The translation of programs is driven by the type checking process which boils down to checking subtyping among types. For each pattern we need to check that the pattern type is a subtype of the incoming type. We also need to check that the type of the function body is a subtype of the function's result type.

For concreteness, we give the translation of the earlier `filter` function. See Figure 1. We first list the subtype proof obligations which guarantee that the program is well-typed. The first function clause gives rise to $\vdash (b, (a|b)^*) \leq_{d_1} (a|b)^*$ because of the pattern match and $\vdash (b, b^*) \leq^{u_1} b^*$ because of the function body. The remaining proof obligations resulting from the second and third function clause should be clear.

The idea behind our translation scheme is to extract out of each subtype proof among a proof term (coercion).Specifically, we use up-cast coercions u for the translation of subtyping and down-cast coercions d for the translation of pattern matching among parametric regular expression types. A source expression of type a^* translates to a target expression of type $[a]$ and $(a|b)$ translates to $Or\ a\ b$. [1] Thus, down-cast coercion d_1 emulates the regular expression pattern match in the first clause and up-cast coercion u_3 injects the empty sequence (represented via the unit type in the target program) into the source type b^*. The full details of the translation process are described in [SL07b].

To obtain decidable type checking, we must impose the following two restrictions:

- We only support non-nested data types.
- Subtyping does not extend to type classes.

We explain both points in more detail below.

We say that a data type (definition) is *non-nested* iff

[1] In fact, we use our "own" list type for the translation of the Kleene star. Otherwise, we may possibly encounter overlapping instances in the translated program (though there were none in the source program). For example, the target instance `Pretty [a]` resulting from the source instance `Pretty a*` overlaps with the instance `Pretty [Char]`. We can easily avoid such issues by declaring `newtype XhsList a = XhsList [a]` and use `XhsList a` instead of `[a]`. For convenience, we will stick to standard Haskell lists in the main text.

Source program:

```
filter :: (a|b)* -> b*
filter (x :: b, xs :: (a|b)*) = (x, filter xs)
filter (x :: a, xs :: (a|b)*) = filter xs
filter () = ()
```

Proof obligations resulting from type checking:

1. $\vdash (b, (a|b)^*) \leq_{d_1} (a|b)^*, \vdash (b, b^*) \leq^{u_1} b^*$
2. $\vdash (a, (a|b)^*) \leq_{d_2} (a|b)^*, \vdash b^* \leq^{u_2} b^*$
3. $\vdash$ `()` $\leq_{d_3} (a|b)^*, \vdash$ `()` $\leq^{u_3} b^*$

Target program:

```
filter :: [Or a b] -> [b]
filter v =
  case (d1 v) of
    Just (x,xs) = u1 (x, filter xs)
    Nothing ->
      case (d2 v) of
        Just (x,xs) = u2 (filter xs)
        Nothing ->
          case (d3 v) of
            Just () -> u3 ()
            Nothing -> error "non-exhaustive
                              pattern"
```

Up-/Down-cast coercions:

```
d1 :: [Or a b] -> Maybe (b,[Or a b])
d1 [] = Nothing
d1 (x:xs) = case x of
              (R y) -> Just (y,xs)
              _ -> Nothing
```

...

```
u3 :: () -> [b]
u3 () = []
```

Fig. 1. Translation of `filter`

```
data T a1 ... an = K t1 ... tm | ...
```

and each occurrence of some data type T' in t_i, whose associated declaration `T' a1' ... ak' = ...` is in a strongly connected component with the above declaration, is of the form $T'\ b_1 ... b_k$ where $\{b_1, ..., b_k\} \subseteq \{a_1, ..., a_n\}$. We say a type t is *non-nested* if it is not composed of any nested data types. For example, the non-nested definition

```
data T a = Leaf (Maybe [a]) |  Internal (T a) (Maybe Int) (T a)
```

is accepted but we reject the nested definition

```
data T2 a = K (T2 [a])
```

Nested definitions are problematic because they may lead to non-termination when checking for subtyping. For example, the subtype proof obligation $\vdash T2\ a \leq T2\ b$ reduces to $\vdash T2\ [a] \leq T2\ [b]$ and so on.

For similar reasons, we impose the restriction that subtyping does not extend to type classes. Recall the declarations

```
class Eq a where
  (==) :: a -> a -> Bool
instance Eq a => Eq a* where ...
```

Suppose some program text gives rise to `Eq (a,a)`. In our subtype proof system, we find that

$$\vdash a^* \to a^* \to Bool \leq^u (a, a) \to (a, a) \to Bool$$

We apply here the co-/contra-variant subtyping rule for functions which leads to $\vdash (a, a) \leq a^*$. The last statement holds. Hence, we can argue that the dictionary E for $Eq\ (a, a)$ can be expressed in terms of the dictionary E' for $Eq\ a^*$ where $E = u\ E'$.

This suggests to refine the type class resolution (also known as context reduction) strategy. Instead of looking for exact matches when resolving type classes with respect to instances, we look for subtype matches. Then, resolution of `Eq (a,a)` with respect to the above instance yields `Eq a`. The trouble is that type class resolution becomes easily non-terminating. For example, `Eq a` resolves to `Eq a` and so on because of $\vdash a \leq a^*$. We have not found (yet) any simple conditions which guarantees termination under a "subtype match" type class resolution strategy. Therefore, we employ a "exact match" type class resolution strategy which in our experience is sufficient. Thus, we can guarantee decidability of type checking.

XHaskell supports type inference in the sense that we exploit local type information, for example provided in the form of user annotations, to infer the type bindings for pattern variables and the type instance at the use site of a polymorphic function. We briefly touched on this issue in Section 3. In XHaskell, we use standard local inference methods [HP01,Hos03,PT00] A complete description of our methods is beyond the scope of this paper and therefore described elsewhere [SL07b].

6 Implementation

We have fully implemented the system as described so far. The XHaskell compiler consists of a type checker and translator. We apply the type-directed translation scheme (sketched in the previous section) and generate Haskell code which compiles under GHC. In the future, we may want to directly compile to GHC's internal GHC's Core language which is a variant of System F. In the following, we discuss a number of topics which concern the practicality of our system.

6.1 Type Error Support

A challenge for any compiler system is to provide meaningful type error messages. This is in particular important in case the expressiveness of the type system increases. The XHaskell compiler is built on top of the Chameleon system [SW] and thus we can take advantage of Chameleon's type debugging infrastructure [SSW03,SSW06] to provide concise location and explanation information in case of a type error.

The following program has a type error in the function body because the value `x` of type `(B|A)*` is not a subtype of the return type `(B|C)*`.

```
data A = A
data B = B
data C = C

f :: (B|A)* -> (B|C)*
f (x :: (B|A)*) = x
```

The compiler reports the following error.

```
ERROR: XHaskell Type Error
Expression at:
f (x :: (B|A)*) = x
has an inferred type (B|A)* which is not a subtype of (B|C)*.
Trivial inconsistencies probably arise at:
f :: (B|A)* -> (B|C)*
f (x :: (B|A)*) = x
```

The error report contains two parts. The first part says that a subtyping error is arising from the body of function `f`, namely the expression `x`. The second part points out the cause of the type error. We found literal `A` in `x`'s inferred type, which is not part of the expected type. This is a very simple example but shows that we can provide fairly detailed information about the possible cause of a type error. Instead of highlighting the entire expression we only highlight subexpressions which are involved in the error.

As an extra feature we allow to post-pone certain type checks till run-time. Let's consider the above program again. The program contains a static type error because the value `x` of type `(B|A)*` is not a subtype of `(B|C)*`. In terms of our translation scheme, we cannot derive the up-cast coercion among the target expression because the subtype proof obligation $\vdash A \leq C$ cannot be satisfied. But if `x` only carries values of type `B*` the subtype relation holds. Hence, there is the option not to immediately issue a static type error here. For each failed subtype proof obligation such $\vdash A \leq C$ we simply generate an "error" case which then yields for our example the following up-cast coercion.

```
u :: [Or B A] -> [Or B C]
u (L b:xs) = (L b):(u xs)
u (R a:xs) = error "run-time failure: A found where B or C is expected"
```

The program type checks now but the translated program will raise a run-time error if the sequence of values passed to function `f` consists of an `A`.

The option of mixing static with dynamic type checking by "fixing" coercions is quite useful in case the programmer provides imprecise type information. In

case of imprecise pattern annotations we can apply pattern inference to infer a more precise type. The trouble is that the standard pattern inference strategy [HP01] may fail to infer a more precise type as shown by the following contrived example.

```
g :: (A,B)|(B,A) -> (A,B)|(B,A)
g (x :: (A|B), y :: (A|B)) = (x,y)
```

It is clear that either (1) `x` holds a value of type `A` and `y` holds a value of type `B`, or (2) `x` holds a `B` and `y` an `A`. Therefore, the above program ought to type check. The problem is that pattern inference computes a type binding for each pattern variable. The best we can do here is to infer the pattern binding $\{(x : (A|B)), (y : (A|B))\}$. But then `(x,y)` in the function body has type `(A|B,A|B)` which is not a subtype of `(A,B)|(B,A)`. Therefore, the above programs fails to type check.

The problem of imprecise pattern inference is well-known [HP01]. We can offer a solution by mixing static with dynamic type checking. Like in the example above, we generate an up-cast coercion u_2 out of the subtype proof obligation $\vdash (A|B, A|B) \leq^{u_2} (A, B)|(B, A)$ where we use "error" cases to fix failed subtype proofs. This means that application of coercion u_2 potentially leads to a run-time failure. In fact, for our example we know there will not be any run-time failure because either case (1) or (2) applies.

For the above example, we additionally need to fix the subtype proof $\vdash (A|B, A|B) \leq (A, B)|(B, A)$ resulting from the pattern match check. This check guarantees that the pattern type is a subtype of the incoming type. Out of each such subtype proof we compute a down-cast coercion to perfrom the pattern match. In case of $\vdash A \leq B$ the pattern match should clearly fail. We can apply the same method for fixing up-cast coercions to also fix down-cast coercions. Each failed subtype proof is simply replaced by an "error" case. The pattern match belonging to the failed subtype proof $\vdash A \leq B$ is fixed by generating

```
\x -> error "run-time failure: we can't pattern match A against B"
```

In our case, we fix $\vdash (A|B, A|B) \leq (A, B)|(B, A)$ by generating

```
d2 :: Or (A,B) (B,A) -> Maybe (Or A B, Or A B)
d2 (L (a,b)) = Just (L a, R b)
d2 (R (b,a)) = Just (R b, L a)
```

Notice that there are no "error" and not even any "Nothing" cases because each of the two components of the incoming type $(A, B)|(B, A)$ fits into the pattern type $(A|B, A|B)$.

6.2 Integration of XHaskell with GHC and HaXML

One of the critical factor for the acceptance of any language extension is the availability of library support and how much of the existing code base can be re-used. XHaskell supports a module system and makes use of GHC-as-a-library to process Haskell modules which are imported by a XHaskell program. We make use of these features in the application below.

```
module RSStoXHTML where

import IO          -- Haskell IO module
import RSS         -- RSS XHaskell module generated by dtdToxhs rss.dtd
import XHTML       -- XHTML module genereated by dtdToxhs xhtml.dtd
import XConversion -- XHaskell module defining parseXml and writeXml etc

filepath1 = "rss1.xml"
filepath2 = "rss2.xml"

row :: (Link, Title) -> Div
row (Link link, Title title) =
    Div ("RSS Item", B title, "is located at", B link)

filter_rss :: Rss -> Div*
filter_rss rss = [ (row (l,t)) | (Item ( (t :: Title)
                                       , (ts :: (Title|Description)*)
                                       , (l :: Link)
                                       , rs )) <- rss/Channel/Item ]

main :: IO ()
main =  do (rss1 :: Rss) <- parseXml filepath1
           (rss2 :: Rss) <- parseXml filepath2
           let filter_rss1 = filter_rss rss1
               filter_rss2 = filter_rss rss2
           html = Html (Body
                  (I ("This document is generated by RSStoXHTML convertor, \
                       a program written in XHaskell.")
                  , Hr, filter_rss1, filter_rss2))
           writeXml "myrss.xhtml" html
```

Our implementation comes with a tool called `dtdToxhs` which we use here to automatically generate XHaskell datatypes from the RSS and XHTML DTD specifications, for example `RSS`, `Link`, `Title`, `Div` etc. We can then import these data types into our main application. Another XHaskell module `XConversion` provides two functions `parseXml :: String -> IO Rss` to read and validate the RSS (XML) document and `writeXml :: Xhtml -> IO ()` to store the XHTML values into a (XML) file. We read and print from standard I/O. Therefore, we import the Haskell module `IO`. We make use of GHC-as-a-library to extract type information out of the imported Haskell module `IO`. We use this information to type check and translate the XHaskell program parts.

Function `filter_rss` extracts all `Item` elements out of the RSS document. For each `Item` element we call function `row` to generate an XHTML `Div` element which has the title and the link of this item. We make use of XQuery and XPath-style combinators to extract the immediate child elements of type `t` in expression `e`. As discussed earlier, we can de-sugar these combinators in terms of plain XHaskell. The main function finally generates an XHTML document in which part of the body content is generated using function `filter_rss`. For instance, given the input file `rss1.xml` as follows,

```
<rss>
 <channel>
```

```
   <item>
     <title>XHaskell</title>
     <link>http://www.comp.nus.edu.sg/~luzm/xhaskell</link>
   </item>
 </channel>
</rss>
```

and `rss2.xml` as follows,

```
<rss>
 <channel>
   <item>
     <title>Haskell</title>
     <link>http://www.haskell.org/</link>
   </item>
 </channel>
</rss>
```

executing the program RSStoXHTML yields the following XHTML document,

```
<html>
 <body>
  <i>This document is generated by RSStoXHTML convertor,
     a program written in XHaskell.</i>
  <hr/>
  <div> RSS Item
    <b>XHaskell</b> is located at <b>http://www.comp.nus.edu.sg/~luzm/xhaskell</b>
  </div>
  <div> RSS Item
   <b>Haskell</b> is located at <b>http://www.haskell.org</b>
  </div>
 </body>
</html>
```

To allow for easier integration of XHaskell with HaXML legacy code, we provide two XHaskell library functions `toHaXml` and `fromHaXml` to convert data from its XHaskell type representation to HaXml type representation and vice versa. Suppose that `haxml_row` is HaXml legacy function which generates a `Div` element out of a `Link` element and a `Title` element. Then we can redefine the function `row` from above as follows.

```
import MyHaXmlLib (haxml_row)
row' :: (Link, Title) -> Div
row' x = fromHaXml (haxml_row (toHaXml x))
```

7 Related Work

In the introduction we have already discussed related work in the context of Haskell. In the context of ML, the work in [Fri06] introduces OCamlDuce which is a merger of OCaml and XDuce. The focus of OCamlDuce is to develop a type inference algorithm to infer types for the OCaml components and most of the

XDuce components in a global flow analysis style. The system does not support the combination of parametric polymorphism and regular expression types.

There are a number of works [GP03,KL03,KMS04] which extend Java and C# to guarantee type-safety of XML transformations. One of the main aspects of these works is the integration of regular expressions types with the object model in Java and C#. Close to our work is C_ω [BMS05], a language extension of C# to provide first-class support for the manipulation of semi-structured data. C_ω is defined in terms of a type-preserving translation scheme to C# and only allows for more limited subtyping relation among semi-structured data compared to our system.

A novel feature of our work is the integration of parametric polymorphism and regular expression. The only prior work we are aware of are in the context of XDuce [HFC05,Vou06]. Our system can support a richer set of parametric polymorphic types involving regular expressions. See the examples in Section 3. A detailed study of the issues involved in combining parametric polymorphism and regular expressions is beyond the scope of this paper. We refer the interested to [SL07a] where we also discuss in the detail the above mentioned works.

The study of improved type error support in the context of regular expression types has only attracted little attention. We are only aware of the work in [GCF05] which proposes a static analysis to check for unused regular expression patterns. This appears to be orthogonal to our type error diagnosis methods. It would be interesting to extend the work in [GCF05] to the combination of regular expressions and data types.

8 Conclusion

We have presented an extension of Haskell which combines parametric polymoprhism, algebraic datatype, type class, regular expression types, semantic subtyping and regular expression pattern matching. We have fully implemented the system which can be used in combination of GHC. Our experience so far shows that the system is highly useful in practice. We also provide for an interface to GHC and HaXml to make use of existing libraries and legacy code.

References

[BCF03] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *Proc. of ICFP '03*, pages 51–63. ACM Press, 2003.

[BFS04] N. Broberg, A. Farre, and J. Svenningsson. Regular expression patterns. In *Proc. of ICFP'04*, pages 67–78. ACM Press, 2004.

[BMS05] G. Bierman, E. Meijer, and W. Schulte. The essence of data access in c_ω. In *Proc. of ECOOP 05*, pages 287–311. Spring-Verlag, 2005.

[Fri06] A. Frisch. OCaml + XDuce. In *Proc. of ICFP'06*, pages 192–200. ACM Press, 2006.

[GCF05] D. Colazzo G. Castagna and A. Frisch. Error mining for regular expression patterns. In *the 9th Italian Conference On Theoretical Computer Science*, pages 160–172. Springer-Verlag, 2005.

[GP03] V. Gapeyev and B. C. Pierce. Regular object types. In *ECOOP '03*, volume 2743 of *LNCS*, pages 151–175. Springer, 2003. A preliminary version was presented at FOOL '03.

[HFC05] H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for XML. In *Proc. of POPL'05*, pages 50–62. ACM Press, 2005.

[Hos03] H. Hosoya. Regular expressions pattern matching: a simpler design, 2003.

[HP00] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language (preliminary report). In *Proc. of Third International Workshop on the Web and Databases (WebDB2000)*, volume 1997, pages 226–244, 2000.

[HP01] H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. In *Proc. of POPL'01*, pages 67–80. ACM Press, 2001.

[HVP05] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.

[Kis07] O. Kiselyov. HSXML: Typed SXML. http://okmij.org/ftp/Scheme/xml.html#typed-SXML, 2007.

[KL03] M. Kempa and V. Linnemann. Type checking in XOBE. In *Proc. Datenbanksysteme fur Business, Technologie und Web, BTW '03*, LNI, pages 227–246. GI, 2003.

[KL05] O. Kiselyov and R. Lämmel. Haskell's overlooked object system. Draft; Submitted for journal publication; online since 30 Sep. 2004; Full version released 10 September 2005, 2005.

[KMS04] C. Kirkegaard, A. Møller, and M. I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transaction on Software Engineering*, 30(3):181–0_6, 2004.

[LS04] K. Z. M. Lu and M. Sulzmann. An implementation of subtyping among regular expression types. In *Proc. of APLAS'04*, volume 3302 of *LNCS*, pages 57–73. Springer-Verlag, 2004.

[MS99] E. Meijer and M. Shields. XMλ: A functional language for constructing and manipulating XML documents. (Draft), 1999.

[PT00] B. C. Pierce and D. N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, 2000.

[Sch07] U. Schmidt. Haskell XML Toolbox. `http://www.fh-wedel.de/~si/HXmlToolbox/`, 2007.

[SL07a] M. Sulzmann and K.Z.M. Lu. A faithful semantics for Hindley/Milner with regular expression types. Manuscript, July 2007.

[SL07b] M. Sulzmann and K.Z.M. Lu. XHaskell – adding regular expression types to Haskell. Manuscript, June 2007.

[SM01] M. Shields and E. Meijer. Type-indexed rows. In *Proc. of POPL'01*, pages 261–275. ACM Press, 2001.

[SSW03] P. J. Stuckey, M. Sulzmann, and J. Wazny. The Chameleon type debugger. In *Proc. of Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, pages 247–258. Computer Research Repository (http://www.acm.org/corr/), 2003.

[SSW06] P. J. Stuckey, M. Sulzmann, and J. Wazny. Type processing by constraint reasoning. In *Proc. of APLAS'06*, volume 4279 of *LNCS*, pages 1–25. Springer-Verlag, 2006.

[SW] M. Sulzmann and J. Wazny. Chameleon. http://www.comp.nus.edu.sg/~sulzmann/chameleon.

[Thi02] P. Thiemann. A typed representation for HTML and XML documents in Haskell. *Journal of Functional Programming*, 12(4 and 5):435–468, July 2002.

[Vou06] J. Vouillon. Polymorphic regular tree types and patterns. In *Proc. of POPL'06*, pages 103–114. ACM Press, 2006.

[WR99] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *ICFP '99*, pages 148–159. ACM Press, 1999.

Evaluating and Using a Grid-Enabled Parallel Haskell

Phil Trinder[1], Abyd Al Zain[1], and Kevin Hammond[2]

[1] Heriot-Watt University, UK
[2] St Andrews University, UK

Abstract. This paper reports both the performance evaluation of a Grid Enabled parallel Haskell (GpH) and ongoing work aimed at using GpH to parallelise a range of Computational Algebra systems as part of the EU FP6 SCIEnce project.
Computational Grids potentially offer low cost, readily available, and large-scale high-performance platforms. However such Grids pose a number of problems when targeting high-performance parallel program execution: in particular, they may be heterogeneous in terms both of the underlying execution platform and the communicaton structure, typically possessing hierarchical, and often shared, interconnects, with high and variable latencies between clusters.
We investigate whether Glasgow parallel Haskell (GpH) with high-level parallel coordination and a Distributed Shared Memory model (DSM) can deliver good, and scalable, performance on a range of Computational Grid configurations. GpH abstracts over the architectural complexities of both parallel systems in general, and Computational Grids in particular. We have previously developed the GridGUM2 implementation of GpH. GridGUM2 is a sophisticated grid-specific implementation and the first high-level DSM parallel language implementation for computational grids. In this paper, we report a systematic performance evaluation of GridGUM2 on combinations of high-/low-latency and homo-/hetero-geneous computational grids. We measure the performance of a small set of kernel parallel programs representing a variety of application areas, two parallel paradigms, and ranges of communication degree and parallel irregularity. We investigate GridGUM2's performance scalability on medium-scale heterogeneous and high-latency computational grids, and analyse the performance with respect to the program characteristics of communication frequency and degree of irregular parallelism.
We report ongoing work developing SymGrid-Par, a new framework for executing large computer algebra problems on computational Grids. We present the design of SymGrid-Par, which supports multiple computer algebra packages, and hence provides the novel possibility of composing a system using components from different packages. Orchestration of the components on the Grid is provided by GpH. We present a prototype implementation of a core component of SymGrid-Par, together with promising measurements of two programs on a modest Grid to demonstrate the feasibility of our approach.

Partial parsing: combining choice with commitment

Malcolm Wallace

University of York

Abstract. Monadic parser combinators are a venerable and widely-used solution to read data from some external format. However, the capability to return a partial parse has, until now, been missing. When only a small portion of the entire data is desired, it has always been necessary either to parse the entire input in any case, or to break up the grammar into smaller pieces and move some work outside the world of combinators.
This paper presents a technique for lazy, demand-driven, parsing with combinators, where the grammar specification may remain complete, and yet only sufficient input is consumed to satisfy the result demanded. It is built on the observation that a commitment to a grammar alternative precludes the choice of other alternatives, but that making a choice does not imply a commitment. Rather, the two notions are distinct, and can be captured in different combinators.
Performance results demonstrate that partial parsing is often faster and more space-efficient than strict parsing, but never worse. The trade-off is that partiality has consequences when dealing with ill-formed input.

1 Introduction

Monadic parser combinators have been with us for a long time. The original tutorial paper by Hutton and Meijer[4] illustrated a sequence of ever-more sophisticated parsers, gradually adding state, error-reporting and other facilities. Rojemo[6] demonstrated space efficient parsers, whilst Leijen's Parsec[5] aimed for good error messages with both space and time efficiency by reducing the need for backtracking except where explicitly annotated. Packrat parsing[3] eliminates backtracking altogether by memoising results (a technique that is highly space-intensive). Laarhoven's ParseP[8] also eliminates backtracking, by parsing alternative choices in parallel. Swierstra[7, 1] has shown us how to do sophisticated error-correction, and permutation parsing.

But there are several other niches still unexplored. One such niche is *partial parsing*. All current parser combinator libraries effectively require to see the entire input, before they can return even a portion of the result. Why is it not possible to be non-strict, demand-driven, partial? Because of the possibility of parse errors. If the document is syntactically incorrect, the usual policy is to report the error and do no onward processing of the parsed data, and in order to prevent onward processing, we must wait until all possible errors could have arisen.

Sometimes this is not what you want. Imagine processing a large XML document that you already know to be well-formed. Why should you wait until the final close-tag has been verified to match its opener, before beginning to produce output? There is also often an enormous memory cost to store the entire representation of the document internally, where lazy processing could in many cases reduce the needed live heap space to a small constant. Even if you do not know for certain that a document is well-formed, it can be useful to process an initial part of it.

Of course, there is a flip-side to partial processing – the parsed value may itself be partial, in the sense of containing bottom (undefinedness, or parse errors). One must be prepared to accept the possibility of notification of a parse-failure when it would be too late to undo the processing already completed.

Our presentation of partial parsing relies on a two-level failure representation, which is useful in its own right, whether or not one takes the extra step towards partiality. For good performance and good error messages, it is necessary to distinguish between recoverable and unrecoverable errors. Essentially, recoverable errors permit choice between alternatives, whilst unrecoverable errors give good positional information about the cause of the parse failure. A variant of this approach has already been used by Parsec[5], which requires explicit annotations of the points at which a choice may require arbitrary lookahead. Elsewhere, only one token of lookahead is available – that is, grammars must be *LL(1)* except where marked. We examine the dual approach to Parsec, where arbitrary choice is always available, and instead the user annotates the points at which backtracking can no longer validly occur. We call this the *commit* tactic, in opposition to parsec's *try*, and believe it leads to a much clearer specification of the grammar.

This paper will first outline some ordinary (strict) parser combinators, then add the *commit*-based implementation of two-level failure to them. Finally we show how the same *commit* tactic easily facilitates partial parsing. The combinators described here are freely available in the *polyparse* library[9].

2 Simple polymorphic parsers

We assume the reader is familiar with the basic concept and implementation of monadic parsing as described in [4]. An outline of the basic mechanism follows:

$$\textbf{newtype}\ \mathit{Parser}\ t\ a = P\ ([t] \to (\mathit{Either}\ \mathit{String}\ a, [t]))$$

The *Parser* type is parameterised on the type of input tokens, t, and the type of the result of any given parse, a. A parser is a function from a stream of input tokens, to the desired result paired with the remaining unused tokens. If a parse fails, we report the failed result in the String alternative of the *Either* type.

Parsers are sequenced together using monadic notation. The instances of *Functor* and *Monad* are:

$$\begin{aligned}&\textbf{instance}\ \mathit{Functor}\ (\mathit{Parser}\ t)\ \textbf{where}\\ &\quad \mathit{fmap}\ f\ (P\ p) = P\ (\lambda ts \to \textbf{case}\ p\ ts\ \textbf{of}\end{aligned}$$

```
                                  (Right val, ts') -> (Right (f val), ts')
                                  (Left msg, ts')  -> (Left msg, ts'))
instance Monad (Parser t) where
  return x     = P (\ts ->   (Right x, ts))
  fail e       = P (\ts ->   (Left e, ts))
  (P p) >>= q = P (\ts ->   case p ts of
                                  (Right x,  ts') -> let (P q') = q x in q' ts'
                                  (Left msg, ts') -> (Left msg, ts'))
```

A parser can be 'run' by applying it to some input token list:

```
runParser :: Parser t a -> [t] -> (Either String a, [t])
runParser (P p) = p
```

Choice between different parses is expressed by *onFail*, which tries its second argument parser only if the first one fails. Note that information may be lost, since any error message from the first parser is thrown away. We return to this point later.

```
onFail :: Parser t a -> Parser t a -> Parser t a
(P p) `onFail` (P q) = P (\ts -> case p ts of
  (Left _, _) -> q ts
  right       -> right)
```

Finally, we need a single primitive parser called *next*, that returns the next token in the stream.

```
next :: Parser t t
next = P (\ts -> case ts of
  [] -> (Left "Ran out of input (EOF)", [])
  (t : ts') -> (Right t, ts'))
```

Higher-level combinators can be defined using the primitives above. For instance:

```
  -- One token satisfying a predicate.
satisfy :: (t -> Bool) -> Parser t t
satisfy p = do x <- next
               if p x then return x else fail "Parse.satisfy: failed"
  -- Use 'Maybe' type to indicate optionality.
optional :: Parser t a -> Parser t (Maybe a)
optional p = fmap Just p `onFail` return Nothing
  -- 'exactly n p' parses precisely n items, using the parser p.
exactly :: Int -> Parser t a -> Parser t [a]
exactly 0 p = return []
exactly n p = do x <- p
                 xs <- exactly (n - 1) p
```

```
                    return (x : xs)
  -- Take the first alternative in the list that succeeds.
oneOf :: [Parser t a] → Parser t a
oneOf (p : ps) = p `onFail` oneOf ps
```

A parser for some application is then built from these combinators, and looks rather like a recursive-descent grammar. The example in Figure 1 illustrates a grammar for a very simplified form of XML, assuming the input tokens have already been lexed according to XML-like rules, with positional information calculated by the lexer.

```
data XML = Elem String [XML]
  | Text String
xml :: Parser (Posn, String) XML
xml = do satisfy (token "<")
         name ← satisfy identifier `report` "bad tagname"
         satisfy (token ">")       `report` "bad open tag"
         content ← many xml
         satisfy (token "</")      `report` "missing close tag"
         endname ← satisfy identifier
         satisfy (token ">")       `report` "bad end tag"
         when (name ≢ endname) (fail "open/close tags do not match")
         return (Elem name content)
       `onFail`
       do fmap Text string
       `report` "unrecognisable as XML"
report :: Parser (Posn, t) a → String → Parser (Posn, t) a
p `report` s = p `onFail` do posn ← next
                             fail (s ++ " at " ++ show posn)
```

Fig. 1. Example combinator grammar for a simplified XML.

3 Problems and Limitations

Error messages are often poor. Due to backtracking over choice points, they rarely point close to the location where the input fails to match the grammar. Indeed, in the worst case, errors are often reported at the topmost outer-most layer of the value's structure, i.e. column 1 of the input.

In the XML example (Figure 1), the error message from attempting to parse the incorrect input `<a><b>hello<b/></a>` is not, as one might hope, `"bad open tag at char 14"`, but rather `"unrecognisable as XML at char 1"`. Why? Because failure anywhere in the first do-block is thrown away by the enclosing *onFail*.

Backtracking over choices sometimes leads to inefficiency. Again for the example input `<a><b>hello<b/></a>` despite the fact that we have already found a valid open tag `<a>` for the element branch of the grammar, nevertheless because something further inside the element is incorrect, this parser necessarily backtracks to the top-level and attempts to match the non-element case, on which it is bound to fail.

The XML example only allows for two choices of outer construct – element or text, corresponding to the two branches of the resultant Haskell sum type – but imagine a type and its grammar having a hundred possible different constructors. A parse failure deep within the first branch could lead to the evaluation of all of the remaining 99 constructor choices, failing on all of them, before giving up. Not only is the error message imprecise, but it took much longer than necessary to deliver it!

Complete consumption of input. If you only want a small part of the parsed data, you must still parse the whole thing first. For instance, given the XML input `<a><b>hello</b><c>world</c></a>` one may wish to extract only the contents of the `<b>` tag, yet one is forced to read the `<c>` tag as well! The input could be arbitrarily large, with the fragment of sole interest close to the beginning. Not only that, but the uninteresting part of the input must be fully well-formed, which may be too restrictive for some applications.

One way to avoid complete parsing is to resort to other coding techniques outside the parsing monad. An example of such a technique is repeatedly calling *runParser* on smaller units of the input, tracking unused tokens between calls. Yet manipulation of the parse state is exactly the tedious boilerplate that the monad is supposed to hide for you! Moving outside the monad also leads to a highly non-modular grammar, requiring much special-case code to deal with the specific fragments of interest.

Ideally, one would like to keep the original grammar, and just interpret it lazily in order to return a partial result.

In the following sections, we shall first present a way to overcome the poor and inefficient error-reporting, then build on that technique to introduce lazy partial parsing.

4 Choice and commitment

We have seen how backtracking over choice points leads to poor error messages.

The solution is to divide parse failures into two separate classes: recoverable and unrecoverable. Recoverable errors allow backtracking to any enclosing choice point. By contrast, unrecoverable errors should always be reported to the user – no choice point should ignore them.

We must refine the parser type to codify the different error classes. Instead of the plain *Either* type, we introduce *Result*, which gives a three-valued logic, with two different kinds of error case.

$$\textbf{data}\ \mathit{Result}\ a = \mathit{Success}\ a \mid \mathit{FailRecover}\ \mathit{String} \mid \mathit{FailReport}\ \mathit{String}$$
$$\textbf{newtype}\ \mathit{Parser}\ t\ a = P\ ([t] \to (\mathit{Result}\ a, [t]))$$

The basic monadic definitions are modified accordingly:

```
instance Monad (Parser t) where
  return x     = P (λts → (Success x, ts))
  fail e       = P (λts → (FailRecover e, ts))
  (P p) >>= q = P (λts → case p ts of
                           (Success x,     ts') → let (P q') = q x in q' ts'
                           (FailRecover e, ts') → (FailRecover e, ts')
                           (FailReport e,  ts') → (FailReport e,  ts'))
```

We also add a new combinator for unrecoverable errors, which we call *failBad*:

```
failBad :: String → Parser t a
failBad e = P (λts → (FailReport e, ts))
```

The choice combinator must be modified to try alternatives only when errors are recoverable. Unrecoverable failures are propagated outwards.

```
onFail :: Parser t a → Parser t a → Parser t a
(P p) `onFail` (P q) = P (λts →
  case p ts of
    r@(Success _,     _) → r
    r@(FailReport _, _) → r
    (FailRecover _,   _) → q ts)
```

Finally, another new combinator allows the user to indicate commitment more conveniently than with *failBad*. It raises the severity of *any* failure discovered in its argument parser to become unrecoverable:

```
commit :: Parser t a → Parser t a
commit (P p) = P (λts →
  case p ts of
    (FailRecover e, ts') → (FailReport e, ts')
    otherwise            → otherwise)
```

Commit is similar to the *cut* operator used by Rojemo[6] in his parser combinators to achieve space efficiency. It also bears a strong similarity to the extra-logical ! operator in Prolog, which also serves to prevent backtracking.

Commit is a kind of dual of the *try* combinator in Parsec[5]. In Parsec, no backtracking is allowed normally – it must be explicitly permitted with *try*. But in our framework, backtracking is normally the default, except where explicitly disallowed by *commit*. Ultimately, they have a similar effect however: the calling context of *try* or *commit* will never be returned to; in both cases, we have committed to any particular branch that led to the current call, yet are still willing to try different alternative branches within the call.

Figure 2 refines the example grammar of Figure 1, re-expressing it in terms of *commit*. Note the careful placement of commitment after sufficient tokens have

```
xml :: Parser (Posn, String) XML
xml = do satisfy (token "<")
         commit (do
           name ← satisfy identifier `report` "bad tagname"
           satisfy (token ">")        `report` "bad open tag"
           content ← many xml
           satisfy (token "</")       `report` "missing close tag"
           endname ← satisfy identifier
           satisfy (token ">")        `report` "bad end tag"
           when (name ≢ endname) (fail "open/close tags do not match")
           return (Elem name content))
       `onFail`
       do fmap Text string
       `report` "unrecognisable as XML"
```

Fig. 2. The XML grammar re-expressed using *commit*.

been read to disambiguate the cases. Now, when given the badly-formed input string `<a><b>hello<b/></a>` in contrast to the previous attempt, we receive the error message `"bad open tag at char 14"`, as hoped.

It is worth noting that one of the commonest sources of bugs in Parsec grammars is that users do not know where to place the *try* combinator. Parsec grammars are *LL(1)* by default, but *try* is used to permit extra lookahead for disambiguation. It can be difficult to look at a grammar and count the required lookahead. If a user's grammar turns out not to work as expected, often they resort to simply sprinkling *try* into various locations to discover a fix.

By contrast, we believe that the *commit* approach is superior, because the lack of a *commit* will not cause the grammar to fail unexpectedly, merely to be inefficient or to give unhelpful error messages. In addition, the intuition needed to place a *commit* combinator correctly within the grammar is a much lower barrier. It indicates a simple certainty that no alternative parse is possible once this marked point has been reached. This is easier to verify by inspection than deciding how many tokens of lookahead are required to disambiguate alternatives.

5 How to be lazy

It turns out that two-level errors are exactly the insight that is needed to support partial (lazy) parsing.

First let us consider the type of *runParser*, when we want a partial result.

$$runParser :: Parser\ t\ a \to [t] \to (a, [t])$$

We follows immediately that the value to be returned must not be wrapped in an *Either* or *Result* type. A wrapper type would convey success/failure information,

but we want to ignore parse failures and just pretend the value is available directly, unfolded lazily on demand as we consume it.

The obvious difficulty, that has prevented partial parsing from being explored previously, is how is it possible to implement the combinator for choice between alternatives? We have already seen that choice operates by detecting failure in one branch, in order to try another. But with no representation of failure in the final result, choice looks impossible.

However, when a distinction between recoverable and unrecoverable errors is made, we can distinguish the notions of choice and commitment. If a backtracking (recoverable) choice is still available, no partial value can be returned. If an explicit commitment has been made to a particular alternative, the partial value can be returned immediately.

Our parser type changes once again:

$$\textbf{newtype}\ \mathit{Parser}\ t\ a = P\ ([t] \to (\mathit{Either}\ \mathit{String}\ a, [t]))$$

Yes this is exactly the parser type we started with before introducing two kinds of failure! What is going on? The answer is that we no longer need to report unrecoverable failure within the *type* being returned. Instead, an unrecoverable error is semantically bottom, the undefined value, or more prosaically thrown as an exception containing an explanatory message.

All of the supporting monadic machinery, including the choice combinator 'onFail', reverts to its original form, but we keep the idea of *failBad* and *commit*:

$$\begin{aligned}
&\mathit{throwE} = \mathit{Control.Exception.throw} \circ \mathit{ErrorCall}\\
&\mathit{failBad} \quad\quad :: \mathit{String} \to \mathit{Parser}\ t\ a\\
&\mathit{failBad}\ \mathit{msg} \quad = P\ (\lambda ts \to (\mathit{throwE}\ \mathit{msg}, ts))\\
&\mathit{commit} \quad\quad :: \mathit{Parser}\ t\ a \to \mathit{Parser}\ t\ a\\
&\mathit{commit}\ (P\ p) = P\ (\lambda ts \to \textbf{case}\ p\ ts\ \textbf{of}\\
&\qquad\qquad (\mathit{Left}\ e, ts') \to (\mathit{throwE}\ e, ts')\\
&\qquad\qquad \mathit{right} \quad\quad\ \to \mathit{right})
\end{aligned}$$

The implementation of runParser is different however. It strips away the enclosing *Either* type at the outer level, leaving just the value itself or an exception.

$$\begin{aligned}
&\mathit{runParser} :: \mathit{Parser}\ t\ a \to [t] \to (a, [t])\\
&\mathit{runParser}\ (P\ p) = (\lambda(e, ts) \to\\
&\quad (\textbf{case}\ e\ \textbf{of}\ \mathit{Left}\ e \quad \to \mathit{throwE}\ e\\
&\qquad\qquad\quad \mathit{Right}\ x \to x))\\
&\quad \circ p
\end{aligned}$$

This covers partiality with respect to errors, but one new combinator is needed to express partiality with respect to return values. How do we ensure that a value is returned to the consumer before parsing is complete? Should the monadic bind operator be lazy? No. As we wish to allow multiple-token lookahead to fail in a recoverable fashion, it would be highly inconvenient for monadic bind to be lazy, since it would prevent exactly that. So we need to introduce a

second kind of sequencing operator, which differs from monadic sequence only by being lazy. We borrow the notion of applicative functors, as used previously by e.g. Rojemo[6].

```
infixl 3 `apply`
apply :: Parser t (a → b) → Parser t a → Parser t b
(P pf) `apply` (P px) = P (λts →
  case pf ts of
    (Left msg, ts') → (Left msg, ts')
    (Right f, ts')  →
      let (ex, ts'') = px ts'
          x = case ex of { Right x → x; Left e → throwE e }
      in  (Right (f x), ts''))
```

The key point in this definition is that if the first parser succeeds, then the whole combined parse succeeds (returns a *Right* value). Both failures and successes within the second parser are stripped of their enclosing *Left* or *Right*, and used 'naked'.

We now have two ways to express sequence with combinators. The user must develop their grammar to make careful use of lazy or strict sequence as appropriate.

```
discard :: Parser t a → Parser t b → Parser t a
px `discard` py = do { x ← px; (return (λ_ → x)) `apply` py }
xml :: Parser (Posn, String) XML
xml = do satisfy (token "<")
         return Elem
           `apply`   satisfy identifier     `report` "bad tagname"
           `discard` satisfy (token ">")    `report` "bad open tag"
           `apply`   many xml
           `discard` satisfy (token "</")   `report` "missing close tag"
           `discard` satisfy identifier
           `discard` satisfy (token ">")    `report` "bad end tag"
       `onFail`
       do fmap Text string
       `report` "unrecognisable as XML"
```

Fig. 3. The XML grammar in lazy form.

For illustration, Figure 3 one again re-expresses the simplified XML grammar, this time in a lazy fashion. A mixture of strict monadic sequence and lazy application is used. Note also that application is of course curried, so chaining many parsers together as straightforward in the applicative case as in the monadic case.

Sometimes in the applicative case, it is unnecessary to build one of the result values into a larger structure. Whereas in the monadic sequence one would simply avoid binding the value to a name, it is more tedious to build a function that ignores one of its arguments. For exactly this situation, the extra combinator *discard* was introduced.

It is also worth making the point that this revised grammar no longer checks that XML end tags match their opening tags.

6 Evaluation

To give a flavour of the performance of lazy partial parsing, we designed a small number of (slightly artificial) tests using the Xtract tool from the HaXml suite. Xtract is a grep-like utility which searches for and returns fragments of an XML document, given an XPath-like query string. Because the intention is to find small parts of a larger document, it is an ideal test case for partial parsing. The XML parser used by Xtract is switchable between the strict and lazy variations.

We created a number of well-formed XML documents of different sizes n (ranging from 10 to 1,000,000) with interesting characteristics:

- linear: the document is a flat sequence of n identical elements enclosed in a single wrapper element.
- nested: the document contains n elements of different types, with element type i containing a single element of type $i + 1$ nested inside it, except for the nth element, which is empty.
- follow: the nested document, followed by a single trivial element, together enclosed in a wrapper element.

The queries of interest are:

- `Xtract "/file/element[0]" linear`
  Find the first element in the flat sequence of elements.
- `Xtract "/file/element[$]" linear`
  Find the last element in the flat sequence of elements.
- `Xtract "//element`n`" nested`
  Find the most deeply nested element(s) in the nesting hierarchy. The difference between this test and the following one is that this test continues searching after finding the first result.
- `Xtract "//element`n`[0]" nested`
  Find only the first most deeply nested element in the nesting hierarchy.
- `Xtract "/file/follow" follow`
  Find the single top-level element that follows the large deeply-nested element.

The time and memory taken to satisfy each query is given in Table 1, using both the strict and lazy parser variations. In all cases, the lazy parser is better (both faster, and more space efficient) than the strict parser. For extremely large documents, where the strict parser often crashes due to stack overflow, the lazy

Strict: time(s)

query	n=10	n=100	n=1000	n=10000	n=100000	n=1000000
linear first	0.018	0.034	0.144	1.165	37.958	>1200
linear last	0.020	0.036	0.176	1.260	38.976	>1200
nested	0.019	0.037	0.149	2.002	104.596	>3600
nested first	0.018	0.038	0.187	1.993	104.304	>3600
follow	0.020	0.036	0.175	1.934	103.239	–

Lazy: time(s)

query	n=10	n=100	n=1000	n=10000	n=100000	n=1000000
linear first	0.015	0.015	0.016	0.015	0.015	0.016
linear last	0.017	0.035	0.150	0.819	6.879	70.736
nested	0.019	0.035	0.092	0.996	19.081	1088.59
nested first	0.018	0.026	0.086	0.556	8.917	504.60
follow	0.017	0.033	0.134	1.256	51.976	–

Strict: peak live memory(b)

query	n=10	n=100	n=1000	n=10000	n=100000	n=1000000
linear first	3.6k	91k	905k	10.1M	96.5M	–
linear last	3.5k	112k	1.18M	4.53M	124M	–
nested	3.6k	122k	1.65M	14.7M	159M	–
nested first	3.6k	129k	1.66M	14.5M	158M	–
follow	3.7k	118k	1.51M	12.7M	140M	–

Lazy: peak live memory(b)

query	n=10	n=100	n=1000	n=10000	n=100000	n=1000000
linear first	7.7k	7.7k	7.8k	7.8k	7.8k	7.8k
linear last	7.7k	72k	844k	2.49M	498k	2.49M
nested	7.7k	63k	628k	5.9M	58.4M	521M
nested first	7.7k	49k	634k	4.23k	51.9M	486M
follow	7.8k	61k	1.06M	6.92M	56.3M	–

Table 1. Time and memory performance results, measured on a twin-core 2.3GHz PowerPC G5, with 2Gb physical RAM. All timings are best-of-three.

parser continues to work smoothly. For the cases where the only result is a small, early, fragment of the full document, laziness reduces the complexity of the task from linear to constant, that is, it depends on the required distance into the document, not on the size of the document. Even when the searched element is at the end of the document, laziness eliminates a space-leak by allowing the early portion of the parsed document to be garbage-collected before the remainder has yet been read, leading to roughly constant heap usage in contrast to the linear heap usage seen in the strict version.

None of this is very surprising of course. Lazy streaming is well-known to improve the complexity of many algorithms operating over large datasets, often allowing them to scale to extreme sizes without exhausting memory resources, where a more strict approach hits physical limitations. One such demonstration is given in the field of isosurface extraction for visualisation[2], where the pure lazy solution in Haskell is slower than a rival C++ implementation, only until very large inputs are considered, beyond which the Haskell overtakes the C++.

7 Conclusion

The contribution of this paper is a demonstration that partial parsing is both possible, and convenient, using the framework of monadic parser combinators. As expected, the resources needed to partially parse a document depend on how much of the input document is consumed, not on the total size of the document. However, partial parsing also means that the ability to report parse errors is shifted from within the parsing framework out to the world of exception handling.

References

1. A. Baars, A. Löh, and D. Swierstra. Parsing permutation phrases. In R. Hinze, editor, *Haskell Workshop*, volume 59 of *ENTCS*, Firenze, Sept 2001.
2. D. Duke, M. Wallace, R. Borgo, and C. Runciman. Fine-grained visualization pipelines and lazy functional languages. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):973–980, Sept 2006.
3. B. Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *International Conference on Functional Programming*, Pittsburgh, October 2002. ACM SIGPLAN.
4. G. Hutton and E. Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, University of Nottingham, 1996.
5. D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-35, University of Utrecht, 2001.
6. N. Röjemo. *Garbage collection and memory efficiency in lazy functional languages.* PhD thesis, Chalmers University of Technology, 1995.
7. D. Swierstra. *Combinator Parsers: from toys to tools*, volume 41 of *ENTCS*. Elsevier, 2001.
8. T. van Laarhoven. Parsep. parsing software: http://twan.home.fmf.nl/parsep/.
9. M. Wallace. Polyparse combinators. http://www.cs.york.ac.uk/fp/polyparse, 2007.

Functional Master-Worker Skeletons
– Work in Progress –

Jost Berthold, Mischa Dieterle, Rita Loogen, and Steffen Priebe

Philipps-Universität Marburg, Fachbereich Mathematik und Informatik
Hans Meerwein Straße, D-35032 Marburg, Germany
{berthold,dieterle,loogen,priebe}@informatik.uni-marburg.de

Abstract. Master-worker systems are a well-known and often applicable scheme for the parallel evaluation of a pool of tasks, a *work pool*. The system consists of a master process managing a set of worker processes. After an initial phase with a fixed amount of tasks ("prefetch"), further tasks are distributed dynamically in reply to results sent back from the workers. As this setup quickly leads to a bottleneck in the master process, the paper investigates alternative implementations of master-worker schemes. We present declarative techniques for *hierarchically nesting* master-worker instances, discuss common pitfalls and identify performance-critical characteristics of different implementations.
Nesting master-worker systems is nontrivial especially in cases where new tasks are dynamically created from previous results (typically breadth- or depth-first tree search algorithms). We discuss how to handle dynamically growing pools in a hierarchy and present a functional implementation for nested master-worker systems with dynamic task creation.
Furthermore, we compare hierarchies to an alternative non-hierarchical implementation, where the work pool is managed in a *distributed* manner. In this implementation concept, realised with a work-stealing mechanism for load-balancing, dynamically growing task pools are easier to handle. On the other hand, a distributed work pool is more sensible against irregularity, it needs responsive load-balancing mechanisms, and a sophisticated termination detection.
All implementations are carried out in the parallel functional language Eden, which allows abstract skeleton implementations as higher-order functions, and to easily change load-balancing policies and other implementation details. The presented work is intended as a conceptual study of different master-worker implementations, and the abstraction offered by the functional implementation language is therefore essential for a clear view on the implemented mechanisms.

Part of the work we present on the following pages is currently under consideration for presentation and publication at another conference. Meanwhile, we have continued our research in master-worker implementations, and obtained new results, which we will present at the IFL 2007 workshop, and possibly submit later for the final proceedings. The new material, mentioned as "future work" in the remainder, is an extended analysis and optimisation of the hierarchical variant, and a non-hierarchical master-worker skeleton implementation with a distributed task pool. The final paper will include discussions for all optimisations and a performance comparison for selected test programs.

1 Introduction

Parallelising an algorithm implemented as a functional program starts by identifying a set of largely independent evaluations. These *tasks* have to be assigned to nodes of a parallel computer, to gain high speedups by simultaneous evaluation. If the *number* of tasks and their individual *runtimes* are statically known, mapping them to the parallel nodes is trivial. The everyday situation, however, faces us with *irregular* tasks of varying and unknown complexity. The *static* task distribution has to be replaced by a *dynamic* one.

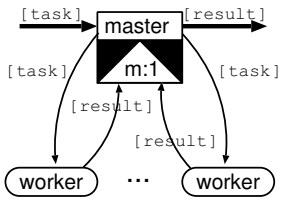


Fig. 1. Master-worker scheme

The *master-worker* scheme is a parallel skeleton for a task pool with dynamic task distribution. A master process distributes tasks to a set of subordinate worker processes, and collects the results. Many-to-one communication enables the master to evenly supply a new task to each worker every time it sends back a result. Apparently, workers are idle in the period between sending a result and receiving a new task. Idle-time can be avoided by pre-assigning a configurable amount (*prefetch*) of initial tasks to all workers. The prefetch parameter decides the behaviour of the skeleton, between completely dynamic (prefetch 1) and completely static distribution (less tasks than totalised prefetches).

So far, we have assumed a statically fixed task pool, which, in essence, results in a parallelised map function with dynamic assignment. Again, more realistic are *dynamic* settings where results might imply additional new tasks at runtime. This changes the scene completely: Tasks are not only irregular and of unknown number, but also carry an unknown 'task productivity'. This weakens the significance of the prefetch parameter, as dynamically evolving task sets often start out with a comparatively small number of initial tasks (often just one task).

A master-worker scheme essentially relies on a double functionality of the master process: it is responsible for collecting (possibly large) results, and it emits new tasks to idle workers. When a large number of workers is used, the single master process quickly becomes a bottleneck which paralyses the whole scheme. As a remedy, we investigate possibilities to nest the basic master-worker skeleton in a *master-worker hierarchy*. The master process at the top distributes tasks to several lower submasters, each of which manages a (smaller) worker set of its own, or possibly another level of submasters in a deeper hierarchy.

The hierarchical master-worker system as a whole is tree-shaped, with worker processes at the leaves and submasters as the inner nodes. Arbitrary tree shapes can be created, from a broad tree with many workers per submaster to a deep narrow (binary) tree. The optimal hierarchy layout depends on the nature of the tasks, and on the number and performance of processing elements (PEs). The basic skeleton mechanism of tasks and requests remains the same at all tree levels, but at higher levels of the tree, skeleton parameters and distribution policies have to be adjusted to achieve good performance. In the case of a dynamic task

pool, another question we investigate is whether submasters at one level should forward new tasks to upper levels, or keep them for their own worker set.

The paper is organised as follows: Section 2 presents non-hierarchical and hierarchical master-worker skeletons for a static task pool; the essential mechanism for nesting the basic skeleton, and how to automatically compute suitable skeleton parameters. In Section 3, we extend the skeleton for the case of dynamic task sets, and show the more complex nesting mechanisms needed for this skeleton variant. Each section includes experiments with an example application, discussing the behaviour for different hierarchy layouts and prefetch values. Section 4 discusses related work, and Section 5 concludes.

2 Static Task Pools

In this section, we consider master-worker systems with a *static* task pool, i.e. no tasks are created during processing. The task pool is a list of tasks which can also be provided as a stream, the total number of tasks does not have to be known in advance. System termination depends, however, on closing this task stream.

2.1 The Basic Master-Worker Skeleton

We perform our experiments in the parallel Haskell extension Eden [1] which allows to specify many different variants of the general master-worker schemes in an elegant and concise way. Figure 2 shows the Eden implementation of the basic master-worker skeleton. The task pool `tasks` is distributed to `n` worker processes, which, for each task, apply the worker function `wf` and return a pair consisting of the worker number and the result of the task evaluation to the master process, i.e. the process evaluating `mw`. The worker numbers are interpreted as requests for new tasks. The master uses a function `distribute` to send tasks to the workers according to the (`n*prefetch`) requests initially created and the ones received from the workers. Care must be taken that `distribute` is *incremental*, i.e. it can deliver partial result lists without the need to evaluate requests not yet available. The skeleton uses the following Eden functions:

- `process ::(Trans a, Trans b) => (a -> b) -> Process a b`
  wraps a function into a *process abstraction* which shifts function evaluation to a remote processing element. The `Trans` context ensures the existence of internal communication functions.
- `spawn :: [Process a b] -> [a] -> [b]`
  starts *processes* on remote machines eagerly.
- `merge :: [[r]] -> [r]`
  nondeterministically merges a set of streams into a single one.

An additional merge phase would be necessary to restore the initial task order for the results. This can be accomplished by adding tags to the task list, and passing results through an additional function `mergeByTags` (not shown) which

```
mw :: (Trans t, Trans r) => Int -> Int -> (t -> r) -> [t] -> [r]
mw n prefetch wf tasks = ress
  where
   (reqs, ress) =  (unzip . merge) (spawn workers inputs)
   -- workers   :: [Process [t] [(Int,r)]]
   workers      =  [process (zip [i,i..] . map wf) | i <- [0..n-1]]
   inputs       =  distribute n tasks (initReqs ++ reqs)
   initReqs     =  concat (replicate prefetch [0..n-1])

-- task distribution according to worker requests
distribute :: Int -> [t] -> [Int] -> [[t]]
distribute np tasks reqs = [taskList reqs tasks n | n<-[0..np-1]]
  where taskList (r:rs) (t:ts) pe | pe == r   = t:(taskList rs ts pe)
                                  | otherwise =    taskList rs ts pe
        taskList _      _      _  = []
```

Fig. 2. Eden master-worker skeleton with a static task pool

merges the result streams from all workers (each sorted by tags, thus less complex than a proper sorting algorithm). We will not go into further details.

In the following, we will investigate the properties and implementation issues of hierarchical master-worker skeletons. As proclaimed in the introduction, this should enable us to overcome the bottleneck in the master when too many workers must be served.

2.2 Nesting the Basic Master-Worker Skeleton

To simplify the nesting, the basic skeleton `mw` is modified in such a way that it has the same type as its worker function. We therefore assume a worker function `wf :: [t] -> [r]`, and replace the expression `(map wf)` in the worker process definition with `wf`. This leads to a slightly modified version of `mw`, denoted by `mw'` in the following. An elegant nesting scheme (taken from [2]) is defined in Figure 3. The parameters specify the branching degrees and prefetch values per level, starting with the root parameters. The length of the parameter lists determines the depth of the generated hierarchical system.

The nesting is achieved by folding the zipped branching degree and prefetches lists. The proper worker function, of type `[t] -> [r]`, is used as the starting value of the folding process. The folding function `fld` corresponds to the `mw'` skeleton applied to the branching degree and prefetch value parameters taken from the folded list and the worker function produced by folding up to this point.

The parameters in the nesting scheme above allow to freely define tree shape and prefetch values for all levels. As the `mw` skeleton assumes the same worker function for all workers in a group, it generates a regular hierarchy, one cannot define different branching or prefetch within the same level. It is possible to define a version of the static nestable workpool which allows for even more control (not considered here), yet more simple skeleton interfaces are desirable, to provide

```
mwNested :: (Trans t, Trans r) =>
            [Int] -> [Int] ->  -- branching degrees/prefetches per level
            ([t] -> [r]) ->    -- worker function
            [t] -> [r]         -- tasks, results
mwNested ns pfs wf = foldr fld wf (zip ns pfs)
  where fld :: (Trans t, Trans r) =>
               (Int,Int) -> ([t] -> [r]) -> ([t] -> [r])
        fld (n,pf) wf = mw' n pf wf
```

Fig. 3. Static nesting with equal level-wise branching

access to the hierarchical master-worker at different levels of abstraction. We can define an interface that automatically creates a regular hierarchy with "good" parameters for a given number of available processing elements.

```
mwNest :: (Trans t, Trans r) =>
          Int -> Int -> Int -> Int -> (t -> r) -> [t] -> [r]
mwNest depth level1 np basepf f tasks
    = let nesting = mkNesting np depth level1
      in mwNested nesting (mkPFs basepf nesting) (map f) tasks
```

In this version, the parameter lists are computed from a given base prefetch, nesting depth and top-level branching degree by auxiliary functions. These fewer parameters provide reasonable control of the tree size and shape, and prefetch adjusted to the task granularity.

Auxiliary function `mkNesting` computes a regular nesting scheme from the top-level branching degree `level1` and the nesting `depth`, which appropriately maps to `np`, the number of processing elements (PEs) to use. It calculates the branching list for a hierarchy, where all intermediate levels are binary. The number of workers per group depends on the number of remaining PEs, rounded up to make sure that all PEs are used. Please note that this possibly places several worker processes on the same PE. Workers sharing the same PE will appear as slow workers in the system, but this should be compensated by the dynamic task distribution unless the prefetch is too high.

$$l_d \;=\; \left\lceil \frac{np - \overbrace{l_1 \cdot (2^{d-1}-1)}^{\text{total \# subm.s}}}{\underbrace{l_1 \cdot 2^{d-2}}_{\text{\# lowest subm.s}}} \right\rceil \Rightarrow \text{Branching list: } \underbrace{l_1 : 2 : 2 : \ldots : l_d}_{d \text{ levels}}$$

A central problem for the usage of the nested scheme is the choice of appropriate prefetch values per level, specified by the second parameter. Suitable prefetch values for submasters at each level must be chosen carefully: a submaster with m workers requiring prefetch p should receive a prefetch of at least $m \cdot p$ tasks to be able to supply p initial tasks to its child processes. Given a worker (leaf) prefetch of `pf` and a branching list $[l_1, ..., l_{d-1}, l_d]$, this leads to the following minimum prefetch at the different levels:

$$\left[\prod_{j=k}^{d-1} l_j * pf \mid k \in [1 \ldots d-1] \right] = [(l_2 \cdot l_3 \cdot l_4 \cdot pf), (l_3 \cdot l_4 \cdot pf), (l_4 \cdot pf), pf]$$

A reserve of one task per child process is added to this minimum, to avoid the submaster running out of tasks, since it directly passes on the computed prefetch amount to its children. The list of prefetch values is computed by a `scanr1`.

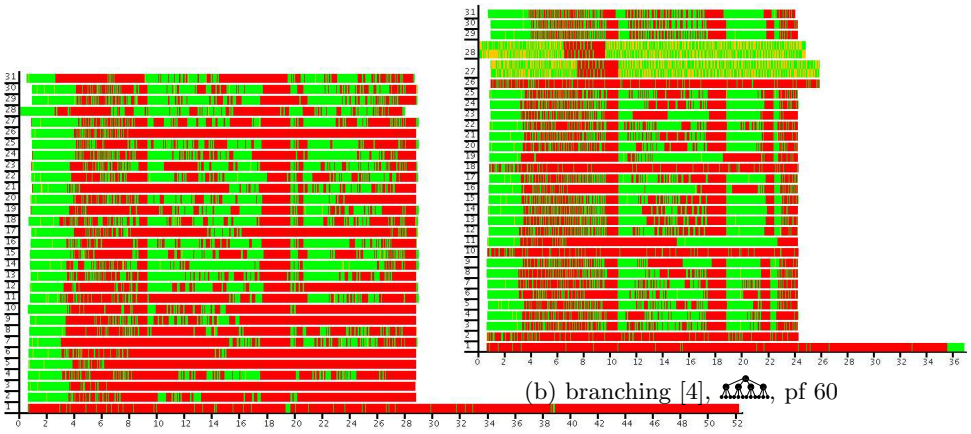
2.3 Experimental Results

We have tested the presented nesting scheme with different branching and prefetch parameters, with an application that calculates a Mandelbrot set visualisation of 5000×5000 pixels. All experiments use a Beowulf cluster of the Heriot-Watt University Edinburgh, 32 Intel P4-SMP nodes at 3 GHz with 512 MB RAM and Fast Ethernet. The *timeline diagrams* in Figure 4 visualise the process activity over time for program runs with different nesting and prefetch. Blocked processes are red (dark), and active/runnable processes green/yellow (light).

Flat vs. Hierarchical Master-worker System The hierarchical system shows better runtime behaviour than the flat, i.e. non-hierarchical version. Although fewer PEs are available for worker processes, the total runtimes decrease substantially. Figure 4(a) shows a trace of the non-hierarchical master-worker scheme. Many worker processes are blocked most of the time. In a hierarchical version with a single additional level comprising four submasters, shown in (b), workers finish faster. Due to the regular structure of the hierarchy, some of the workers in the last branch share the same PE. Nevertheless, the system is well-balanced, but not completely busy. The dynamic task distribution of the master-worker inherently compensates load imbalance due to slower workers or irregular tasks.

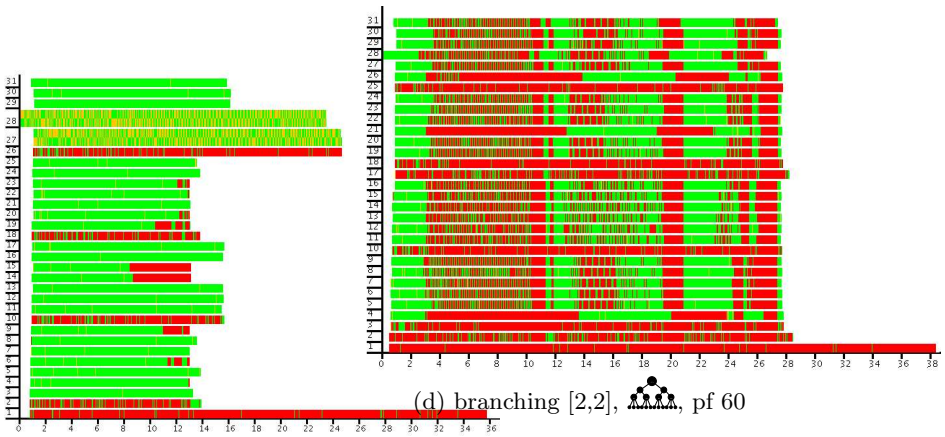
Load Balance and Prefetch Values In Figure 4(c), we have applied the same nesting as in (b), but we increased the prefetch value to 120. Small prefetch values lead to very good load balancing, especially PEs occupied by several (and therefore slow) workers do not slow down the whole system. On the other hand, low prefetch lets the workers run out of work sooner or later. Consequently, it is better to correlate prefetch values with the worker speed. Higher prefetch values (like 120) reduce the worker idle time, at the price of a worse load balance, due to the almost static task distribution.

Depth vs. Breadth Figure 4 (d) shows the behaviour of a nested master-worker scheme with *two* levels of submasters. It uses 2 submasters at the higher level, each serving two submasters. From our experiments, we cannot yet identify clear pros and cons of employing *deeper* hierarchies. Comparing runs with one and two additional submaster-levels, runtime and load balancing behaviour are almost the same, the advantage of the one-level hierarchy in Figure 4 (b) remains rather vague. However, it is clear that deeper hierarchies will be advantageous on bigger clusters with more machines. As shown in Figure 5, a broad flat hierarchy reveals the best total runtimes.



(a) Non-hierarchical, , pf 60

(b) branching [4], , pf 60



(c) branching [4], , pf 120

(d) branching [2,2], , pf 60

Fig. 4. Mandelbrot traces, with different nesting and varying prefetch

Garbage Collection Another phenomenon can be observed in traces (a), (b) and (d): If the prefetch is small, relatively short inactivity at the tree root can make the whole system run out of work and lead to global inactivity. In this case, the reason are garbage collections in the master process, which make all the submasters and workers run out of tasks. The effect is slightly intensified by higher top-level branching, and compensated by higher prefetch (c).

Post-Processing Our experiments have shown that the bottleneck in the master process is mainly caused by the huge amount of results that the master collects and stores in its local heap. As new requests are processed together with the result values, request processing is slowed down in the master processes. This is also the reason for the long post-processing phases that can be observed in our traces. When the master is freed from result processing, it has no prob-

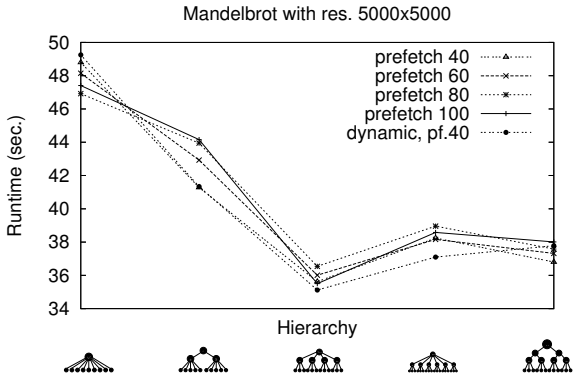


Fig. 5. Runtimes for various hierarchies and prefetch values

lems to keep all workers busy, even with small prefetch values and no hierarchy. The hierarchy throttles the flow of results and thus helps to shorten the post-processing phases. This is the main reason why the hierachical master-worker skeletons show much better total runtimes, as shown in Figure 5.

3 Dynamic Creation of New Tasks

Except for some problems consisting of independent tasks which are trivial to parallelize, e.g. mandelbrot, ray tracing and other graphical problems, many problems deliver tasks containing inherent data dependencies. Thus, the task pool is not completely known initially, or it depends on other calculation results to be fully defined. This is the case when the problem space is built hierarchically, as a tree structure or following other, more complex, patterns.

3.1 The Dynamic Master-Worker Skeleton

The elementary master-worker skeleton can easily be extended to enable the dynamic creation of additional tasks within the worker processes. In the version shown in Figure 6, the worker processes deliver a list of new tasks with each result, and the master simply adds the new tasks to its task queue. A straightforward extension would be to let the master filter or transform the task queue, considering global information [2].

The static task pool version terminates as soon as all the tasks have been processed. With dynamic task creation, explicit termination detection becomes necessary, because the task list contains a reference to potential new tasks. In the skeleton shown in Figure 6, a function `tdetect` keeps track of the current number of tasks in process. It is essential that the result list is extracted via `tdetect` and that the evaluation of this function is driven by the result stream. As long as new tasks are generated, the function is recursively called with an

```
mwDyn :: (Trans t, Trans r) => Int -> Int -> (t -> (r,[t])) -> [t] -> [r]
mwDyn n prefetch wf initTasks = finalResults
 where -- identical to static task pool except for the type of workers
       (reqs, ress) =  (unzip . merge) (spawn workers inputs)
       workers      =  [process (zip [i,i..] . map wf) | i <- [0..n-1]]
       inputs       =  distribute n tasks (initReqs ++ reqs)
       initReqs     =  concat (replicate prefetch [0..n-1])
       -- additions for task queue management and termination detection
       tasks        =  initTasks ++ newTasks
       initNumTasks =  length initTasks
       (finalResults, newTasks) = tdetect ress initNumTasks

-- task queue control for termination detection
tdetect :: [(r,[t])] -> Int -> ([r], [t])
tdetect ((r,[]):ress) 1     = ([r], []) -- final result
tdetect ((r,ts):ress) numTs = (r:moreRes, ts ++ moreTs)
  where (moreRes, moreTs) = tdetect ress (numTs-1+length ts)
```

Fig. 6. Eden master-worker skeleton with a dynamic task pool

updated task counter, initialised to the length of the skeleton input.[1] As soon as the result of the last task arrives, the system terminates by closing the tasks list and, via `distribute`, the input streams of the worker processes.

3.2 Nesting the Dynamic Task Pool Version

It would be possible to apply the simple nesting scheme from Section 2 to the dynamic master-worker skeleton `mwDyn`. However, newly created tasks would always remain in the lower submaster-worker level because the interface of `mwDyn` only passes results, but not tasks, to upper levels. For nesting, the dynamic master-worker scheme `mwDyn` has to be generalised to enable a more sophisticated task management within the submaster nodes.

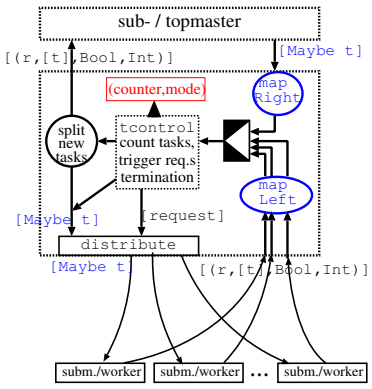


Fig. 7. Submaster functionality in the dynamic master-worker hierarchy

Each submaster receives a task stream from its master and a result stream including new tasks from its workers. It has to produce task streams for its workers and a result stream including new tasks for its master (see Figure 7).

[1] The reader might notice that the initial task list must have fixed length. The skeleton presented here cannot be used in a context where the input tasks arrive as a stream.

Sending back all dynamically generated tasks is a waste of bandwith, when they might be needed in the local subtree. A portion of the generated new tasks can be kept locally, but surplus tasks must be passed up to the next level. Furthermore, sending a result should *not* automatically be interpreted as a request for a new task, since tasks kept locally can compensate for solved task. Finally, global information about tasks in process is needed at the top-level, to decide when to terminate the system. The Eden code for the submaster is shown in Figure 8, and shows the necessary changes.

- The input stream for submasters and workers has type `Maybe t`, where the value `Nothing` serves as a termination signal, propagated downwards from the top level.
- The output stream of submasters (and workers) now includes information about the number of tasks kept locally, and a `Bool` flag, indicating the request for a new task, leading to a complex type `[(r,[t],Bool,Int)]`.
- The incoming list `initTasks` for submasters is no longer finite, but a stream. It has to be *merged* with the stream of worker answers, and processed by a central control function `tcontrol`. The origin of each input to `tcontrol` is indicated by tags `Left` (worker answers) and `Right` (parent input), using the Haskell sum type `Either (Int,(r,[t],Bool,Int)) (Maybe t)`
- All synchronisation is concentrated in the central task control function `tcontrol`. It both controls the local task queue, passes new requests to `distribute`, and propagates results (and a portion of generated tasks) to the upper levels.

The heart of the dynamic master-worker hierarchy is the function `tcontrol`, shown in Figure 9. It maintains two counters: one for the amount of tasks that have been generated and passed up to this submaster, to decide whether a request must be sent up, and the overall task count in the subtree below.

Tasks sent by the parent are simply enqueued in the local task queue. Tasks generated by workers are `split` into a part that is kept local, and a part that is passed upwards. The nested task pools can be seen as a system of interdependent *buffers*, and both buffer-underruns and buffer-overruns will spoil the skeleton performance. This is relatively easy for a static task list: the exchange of tasks and results between different buffers is limited, and the `prefetch` parameter defines the maximum buffer size. In the extension for dynamically growing task pools, more sophisticated policies are needed instead of mechanically forwarding new tasks and requests.

The `split` function decides how many tasks to hold in the subtree below a submaster. If a sufficient amount of self-generated tasks fills the subtree below the node (overall task count `numTs`), all generated tasks are forwarded to the upper level. Likewise, requests for new tasks are only emitted if a solved task cannot be compensated by self-generated ones. In our version, `tcontrol` emits requests only when all self-generated tasks have been assigned, thereby trying to maintain its initial `prefetch` value. The `split` function we use (not shown) splits generated tasks one half each, until the total task count exceeds the double `prefetch` for the whole subtree below. Different heuristics can be configured by exchanging the `split` function, and minor changes in `tcontrol`.

```
mwDynSub :: (Trans t, Trans r) =>
            Int -> Int -> ([Maybe t] -> [(r,[t],Bool,Int)])
            -> [Maybe t] -> [(r,[t],Bool, Int)]
mwDynSub n prefetch wf initTasks = finalResults where
  fromWorkers  = spawn workers inputs
  -- worker    :: [Process [Maybe t] [(Int,(r,[t],Bool,Int))]]
  workers      =  [process (zip [i,i..] . wf) | i <- [0..n-1]]
  inputs       =  distribute n tasks (initReqs ++ reqs)
  initReqs     =  concat (replicate prefetch [0..n-1])
  -- task queue management
  controlInput =  merge (map Right initTasks :
                         map (map Left) fromWorkers)
  (finalResults, tasks, reqs)
               = tcontrol (n*prefetch+n) (False,0,0) controlInput
```

Fig. 8. Eden submaster for nested dynamic master-worker skeleton

```
tcontrol _ (_,_,0) ((Right Nothing):_)   -- from above, final termination
  = ([],repeat Nothing,[])
tcontrol pf (even,local,numTs) ((Right (Just t)):ress) -- task from above
  = let (moreRes, moreTs, reqs) = tcontrol pf (even, local ,numTs+1) ress
    in (moreRes, (Just t):moreTs, reqs)
-- from i below, result plus new tasks plus flag
tcontrol pf (even,local,numTs) ((Left (i,(r,ts,wantNew, heldBelow))):ress)
  = let (localTs,fatherTs,evenAct) = split numTs pf ts even
        -- part of tasks to parent
        lenlocalTs = length localTs
        -- counts only "self-generated" tasks
        newLocal = lenlocalTs + local
                   - if wantNew && not newTasksForMe --minimum 0
                     then 1 else 0
        --  local counter: decrease, add new tasks held (local & below)
        newNumTs = numTs-1 + lenlocalTs + heldBelow
        --  new numTs: -1 came back, + new ones, local or below
        (moreRes, moreTs, reqs) -- recursion
          = tcontrol pf (evenAct, newLocal, newNumTs) ress
        -- node i below wants new tasks?
        newreqs = if wantNew then i:reqs else reqs
        -- order new tasks? only if "no compensation" in subtree
        newTasksForMe = local + lenlocalTs == 0 && wantNew
    in ((r, fatherTs, newTasksForMe, heldBelow + lenlocalTs):moreRes,
           (map Just localTs) ++ moreTs, newreqs)
```

Fig. 9. Control function for submaster of Figure 8

The top-level master in the nesting scheme for a dynamic task pool works similar to the submasters we have described, but of course cannot forward tasks to the outside. A separate top-level master has to be defined.

```
topMaster :: (Trans t, Trans r) =>
         Int -> Int -> ([Maybe t] -> [(r,[t],Bool,Int)]) -> [t] -> [r]
```

Besides termination detection, the former `tdetect` function now takes the role of `tcontrol` in the submaster processes, also incorporating the more sophisticated request handling we have introduced in the `tcontrol` function. Further changes are the adaption of the worker function to the `Maybe` type interface and the termination signal `Nothing` for all submasters upon termination.

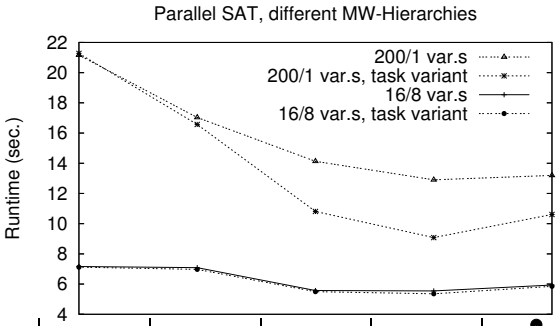
3.3 Experimental Results

The skeletons that support dynamic task creation have been tested with a special test application: a program which computes all satisfying variable assignments for a particular logical formula (i.e. it is a specialised SAT problem solver). Tasks are incomplete variable assignments, and the workers successively assign a value to a new variable and partially evaluate the result. An assignment that already yields false is immediately discarded, true results are counted, yet unspecified results are new tasks returned to the level above.

The program runs the satisfiability check for a formula which disjoins *all* conjunctions of n logical variables where k variables are negated (yielding $\binom{n}{k}$ disjoint monoms). In the underlying *search tree* of the SAT problem, each node has at most two child nodes, and the tree becomes deeper for bigger n, and more dense for bigger k. Thereby we have good control of the tasks for testing our skeleton variants. It is easy to create a basic problem where most of the subproblem nodes in the search tree ony have one successor. Using a formula of 200 variables and 1 negation tests the skeleton behaviour for a broad sparse tree. Especially with sparse search trees, it is challenging for the load balancing strategy to serve all subtrees evenly, avoiding idle times. Many tasks can be discarded early, but the test for 200 variables is complex. In contrast, a test with 8 variables out of 16 negated shows the performance for a dense tree with very small tasks. Runtimes have been compared for the basic version, for hierarchies with one level of two, four and six submasters, and for a binary hierarchy with two levels.

Flat vs. Hierarchical Skeleton In general, variants of hierarchical master-worker schemes perform better than the non-hierarchical skeleton in our test runs. However, when testing a formula with only 16 variables, tasks are numerous, but very simple. For this variant, hierarchical skeletons yield smaller runtime gains.

Depth vs. Breadth The runtime comparison in Figure 10 shows that, in our setup of 31 machines, broader hierarchies with only one level generally perform better than the binary two-level hierarchy. The variant with 6 submasters yields



Runtime (*sec*)					
16 var.s, dense tree	7.16	7.09	5.58	5.55	5.94
200 var.s, sparse tree	21.16	17.03	14.13	12.90	13.19
16 var.s, task variant	7.12	6.97	5.50	5.36	5.85
200 var.s, task variant	21.28	16.57	10.81	9.08	10.61

(Heriot-Watt Beowulf, 31 nodes)

Fig. 10. Experiments using skeletons for dynamic task pool

the best results for all variants, whether sparse or dense decision trees. Measurements with a simplified test procedure, where tasks are checked very fast using aditional knowledge about the tested formula, confirm this result: The performance of the skeleton with two-level nesting is slightly worse than for the one-level nestings. Of course, this result again has to be qualified for a bigger cluster, as our experiments use only 31 PEs.

Prefetch and Forwarding Policy: Prefetch values have little influence on performance (or trace appearance) for this test program, since there are relatively few tasks in the beginning anyway and many of the generated tasks are held close to the processing units. Contrary to the expectation, higher prefetch values only lead to "bundled" working and idle phases instead of a more steady workload. Using higher prefetches, we also observed periods of global inactivity, maybe again caused by garbages collections of the top-level master.

A crucial feature of the hierarchical skeleton is to use a good partition policy for tasks returned by workers, in order to avoid unnecessary idle times and uneven balance between submasters. The list of tasks from workers is normally split in half, but an even level of tasks in every submaster must be achieved, by keeping the task pool size between two thresholds (sometimes called low and high watermark [3]). Our minimum threshold, the prefetch parameter, is self-suggesting: requests are emitted when locally generated tasks cannot compensate for a parameterised property of the skeleton. The very existence of a maximum threshold, applied by the split function, has principal impact on the load balance, especially important in our setup, where only few new tasks are created. Our experiments have, however, confirmed that increasing the high watermark for the split policy hardly produces perfomance gains. This is why we hard coded the value as `2*prefetch`, instead of adding another parameter to the skeleton.

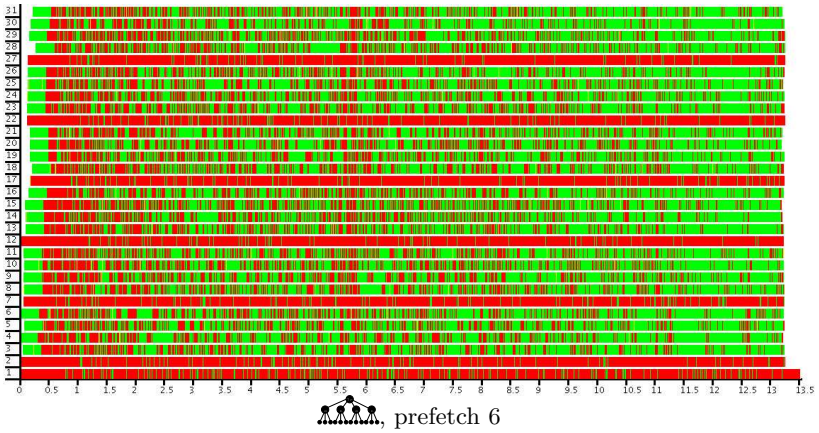


Fig. 11. Trace for SAT solver (200/1 var.)

Figure 11 shows a trace for a program run using the best skeleton in our experiment, with six submasters above the leaves, on a sparse decision tree. The workers expose a slow startup phase, since the (relatively few) tasks must first be propagated in all branches. Tasks are well distributed among the different submaster branches, leading to an even workload among the worker processes. Even though some PEs are reserved as submasters, the remaining workers outperform the non hierachic skeleton close to factor 2.

4 Related Work

The commonly used master-worker scheme with a single master managing a set of workers is a well-known scheme which has been used in many different languages [4]. Modifications of this scheme are however more rare, and we are not aware of other hierarchical master-worker schemes like ours.
Two notable, yet imperative, approaches are:

Driven by the insight that the propagation of messages is expensive in a master-worker scheme, Poldner and Kuchen [5] present a variant of the scheme where the master is divided into a sending *dispatcher* and a receiving *collector*. In the spirit of saving communication, the dispatcher of Poldner and Kuchen applies a *static* instead of our dynamic task distribution, and they argue that for a large number of tasks, a roughly even load balance can be expected. However, this contradicts one of the basic ideas of dynamic master-worker skeletons: the intention to balance not only task irregularity, but also potential differences in worker performance. An undisputed advantage is that the dispatcher can serve more workers, as the collecting is done by another process. The scheme is nevertheless limited in scalability, critical work load in either dispatcher or collector will affect the overall runtime. As own ongoing work, we investigate a *nested* master-worker skeleton with separate collector and dispatcher, especially considering a variant with a hierarchy of collectors, but only one dispatcher.

Hippold and Rünger describe *task pool teams* [6], a programming environment for SMP clusters that is explicitly tailored towards irregular problems with strong

inter-task dependences. The scheme comprises a set of task pools, each running on its own SMP node, and interacting via explicit message passing. Dynamic task creation by workers, task migration, and distributed task pools with a task stealing mechanism are possible. Locality can be exploited to hold global data on the SMP nodes, while communication between nodes is used for task migration, remote data access, and global synchronisation.

Various load balancing strategies for divide-and-conquer algorithms are discussed by Nieuwpoort et al., in [7]. The authors experiment with different techniques to exchange tasks between autonomous worker processes, in the context of WAN-connected clusters (hierachical wide-area systems). Aside from special optimisations to handle different network properties, a basic distinction is made between task *pushing* and *stealing* approaches. Demand-driven work stealing strategies are generally considered advantageous, but must take into account the high latency connections in question. The work pushing strategy speculatively (and blindly) forwards tasks to *random* peers when the amount of local tasks exceeds a prefetch threshold. Contrary to the randomised, or purely demand-driven, task distribution in this work, our skeletons are always based on task-request cycles, and concentrate surplus tasks at higher levels.

5 Conclusions

We have given a series of functional implementations of the parallel master-worker scheme. The declarative approach enables a clear conceptual view of the skeleton nesting we have developed.

Starting off with a very compact version of the standard scheme, we have given implementations for skeleton nesting, to shift the administrative load to a whole hierarchy of (sub-)masters. The hierarchies have been elegantly expressed as foldings over the modified basic scheme. In the case of a dynamically growing task pool, a termination detection mechanism is needed. Nesting this skeleton is far more complex and needs special code for submasters, especially an appropriate task forwarding policy in the submaster processes.

As our tests show, master-worker hierarchies generally speed up runtime and keep workers busier, avoiding the bottleneck of a flat skeleton. Hierarchy layout and suitable prefetch values, however, have to be chosen carefully, depending on the target architecture and problem characteristics. Our experiments show the importance of suitable task distribution and task forwarding policies, which we have described and discussed in detail.

We have presented implementations and experiments with a range of hierarchical master-worker variants, and we will continue investigations on some open topics. One interesting extension, already mentioned, is to investigate variants with separated dispatcher and collector. Furthermore, we are implementing non-hierarchical peer-to-peer solutions, extending the one proposed by Hippold and Rünger in [6], and compare them to our master-worker hierarchies.

Acknowledgements: We greatly appreciate the opportunity to conduct runtime experiments on the Beowulf cluster of the Heriot-Watt University in Edinburgh.

References

1. Loogen, R., Ortega-Mallén, Y., Peña-Marí, R.: Parallel Functional Programming in Eden. Journal of Functional Programming **15**(3) (2005) 431–475
2. Priebe, S.: Dynamic Task Generation and Transformation within a Nestable Workpool Skeleton. In Nagel, W.E., Walter, W.V., Lehner, W., eds.: European Conference on Parallel Computing (Euro-Par) 2006. LNCS 4128, Dresden, Germany (2006)
3. Loidl, H.W.: Load Balancing in a Parallel Graph Reducer. In Hammond, K., Curtis, S., eds.: SFP'01 — Scottish Functional Programming Workshop. Volume 3 of Trends in Functional Programming., Bristol, UK, Intellect (2001) 63–74
4. Danelutto, M., Pasqualetti, F., Pelagatti, S.: Skeletons for Data Parallelism in P^3L. In Lengauer, C., Griebl, M., Gorlatch, S., eds.: Euro-Par'97. LNCS 1300, Springer (1997) 619–628
5. Poldner, M., Kuchen, H.: Scalable farms. In Joubert, G.R., Nagel, W.E., Peters, F.J., Plata, O.G., Tirado, P., Zapata, E.L., eds.: Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005, 13-16 September 2005, Department of Computer Architecture, University of Malaga, Spain. Volume 33 of John von Neumann Institute for Computing Series., Central Institute for Applied Mathematics, Jülich, Germany (2005) 795–802
6. Hippold, J., Rünger, G.: Task Pool Teams: A Hybrid Programming Environment for Irregular Algorithms on SMP Clusters. Concurrency and Computation: Practice and Experience **18** (2006) 1575–1594
7. van Nieuwpoort, R.V., Kielmann, T., Bal, H.E.: Efficient load balancing for wide-area divide-and-conquer applications. In: PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming, New York, NY, USA, ACM Press (2001) 34–43

Towards an Implementation of a Computer Algebra System in a Functional Programming Langauge

Oleg Lobachev

Fachbereich Mathematik und Informatik der Philipps–Universität Marburg

Abstract. This article discusses briefly the possibility of using modern purely functional programming language like `Haskell` for an implementation of a computer algebra system. The concept of "language unity" is suggested and discussed. Some examples are discussed.

1 Background and Motivation

With the flow of the history of computing, exact methods gained more and more importance. It was clear since almost the beginning, that inexact, *numerical* operations may and will fail. The Wilkinson Monster

$$\prod_{j=1}^{20}(x-j)$$

is a nice – and old! [Wil63, Wil84] – example for the thesis "the way we compute it matters". The wish to perform exact, *algebraic* operations lead to the invention and implementation of computer algebra systems (CAS) [Dav94, Fat72]. Such systems include not only arbitrary precision arithmetic, but feature algebraic (and hence: exact) manipulation with abstract objects. To name the most popular computational algebra concept: the gröbner bases gain more and more importance in computer algebra implementations.

There are some nice ideas, already implemented in some systems, but still not mainstream. Asymptotically fast multiplication algorithm by Schönhage and Strassen [SS71, Sch82] gives significant performance boost for computations involving *really* large values (be it numbers or polynomials), but it is rather hard to be implemented, compared with older methods by Karatsuba or in general by Toom and Cook [Knu98]. Fast multiplication algorithms as well as many other aspects of computer algebra are discussed in [vzGG03, Knu98]. An example of a modern CAS, utilising this fast method is `GiNaC` [HK05, GiN07], written in `C++`. We will return to `GiNaC` in the further sections of this text.

Another interesting example is `PARI/GP`, a CAS made by Henri Cohen's team [BBB+06]. `PARI/GP` divides an implementation of the computational CAS functions (the library PARI) and a high–level language front-end (the `GP` language and the associated gp interpreter), which uses the PARI library to perform

the computations itself. It is also possible to compile a `GP` program to `C` whilst continuing to use the PARI as program library. Many of the ideas and algorithms of `PARI/GP` can be found in Henri Cohen's book [Coh95].

Writing a computer algebra system in a functional programming languages is not really new idea. The first generation CAS named `Macsyma` was written in a `LISP 1.5` dialect called `MACLISP`, and `LISP` is considered to be the first functional language ever. Some interesting aspects has `Axiom` CAS. It features an embedded (although detachable) functional programming language [BDea03]. `Axiom` uses hierarchical structure of mathematical objects (like: monoid – group – ring – integrity domain – field) to specify and perform operations on them. Taking a brief view from another point: some features of modern functional languages, like lazy evaluation indeed improve numerical computations [BMP94, BJDM97]. There are also other features, which could be useful for a CAS [Mil95]. As a side note, both functional programming language [LOP05,TBD+98,RSN95, Eri07,Roy04] and computer algebra systems [GMa07b,GMa07a,HAC+07,SCI07] are present in the field of parallel and distributed computing. But can *some* languages of a CAS be functional and which consequences will it have?

2 What a Functional Language Can Give a CAS?

Let's take a bit conciser look on what a functional language can offer a modern CAS implementation. We are basing our thoughts on `Haskell` [Pey03], a pure functional language with many interesting constructs and features (such as curring, lazy evaluation, infinite data structures, garbage collection, monads), big community, fully documented structure and – not really a feature of the language itself, but also hard to underestimate – a full–fledged compiler and more than one interpreter.

The following features are interdependent and *all* of them can be found in a modern pure functional language like `Haskell`.

- *Lazy evaluation* means that no expression is evaluated if it is not required, but also that no expression is evaluated more than once. We should think of lazy evaluation as of a double–edged sword. Indeed it reduces the amount of required computations and lets the end user of the CAS the freedom of writing his own programs in a more mathematical way. We shall state a nice short example for it later. The reason why lazy evaluation is rather unused in the industry is certain performance loss in comparison with eager languages on the same hardware platform. Hardware support is much more solid for eager and imperative languages [Pey93], the cases of of better performance of a lazy (and functional) language in some tasks are still possible. Roughly: they occur due to better optimisation and "skipping" unneeded computations. The cases of worse performance are one where lazy evaluation fails to "skip" more operations, than the overhead needed to perform the lazy evaluation itself. Detailed comparison is beyond the scope of this paper, but it is impossible to designate some particular language or even some particular language type as all round superior one.

– A data structure which can exist only in non–eager languages are the *infinite data structures*, notably: lists. Such lists cannot be implemented without lazy evaluation. But infinite data structures give us an opportunity of much "more mathematical" definition of e. g. sequences and series, than any other eager language. The following code is in `Haskell`[1].

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

This example is in linear time, which is quite bad for such sequence. Faster, but more elaborate implementations are possible.

– Pure functional languages have *no side effects*. Consider an example in `C`.

```
int i = 5;
i = ++i + i++;
```

This example is rather unnatural, but the correct value is not defined and depends on implementation – try it in other imperative languages of your choice. In a pure functional language such situation is not possible, due to the nature of "variable" declaration itself. The most feasible benefit is again a "more mathematical" semantics: is `f` a function, so has `f(5)` the same value, whenever it is called, pretty much as $f(5)$ in a mathematical notation has. This is also known as *referential transparency*.

– Some functional languages feature *strict typing*, a type system more resistant to a particular kind of programming errors. In the contest of a CAS strict typing brings some benefits. If we have, e. g. some ring A and its invertible elements A^*, then an advanced type system can report an error in the processing of the expression "a^{-1}" for some $a \in A$, if $a \notin A^*$.

But what does a language, in which one is willing to write his/her CAS has in common with language the end user of that CAS will be dealing with? The answer is in the next section.

3 The two languages of a CAS

Computer algebra systems possess two different languages, we shall call them in this paper as follows. The *external* language of a CAS is the language the system is written in, the implementation language. Since the end user of the CAS is to perform some kind of *programming*, there is also a second language. The *internal* language of CAS is the language with which the CAS interfaces with its user, the embedded language.

It is desirable to write as much as possible of the CAS itself in its internal language. Such decision gives the user opportunity to inspect and (if needed) to modify some internal functions of the CAS, giving him/her as much freedom, as he/she can take. However this is impossible in most cases. Firstly the internal language of most CASes is weaker, than the external. It is still Turing–complete but some technical things may be hard or even impossible. On the other side the internal language is better laid of for the typical computer algebra operations:

[1] Cf. `http://haskell.org/haskellwiki/The_Fibonacci_sequence`

we may expect, e. g. polynomials and matrices as native objects or an interesting handling of lists, non–existent in the external imperative language of the CAS.

Secondly, unfortunately, internal languages of most CASes are not as fast, as the external one. The cause may be the interpreted origin of these languages or it very high level nature. The heavy inheritance of an interpreter is fought in a nowadays common way in practical software development. E. g. `Matlab`, despite being rather a scientific calculation system, and not a CAS in its narrow sense[2], compiles the input files in his own internal language into "pseudo-code" [Mat99]. Another possibility of immense speedup pays the price of shutting the user out of the program internals. The implementation of `S` programming language for statistical computations, `GNU R`, utilises a `Scheme` dialect as its internal language. The whole R system can be implemented in in `Scheme`. But because of performance lack in core operations, the later ones are replaced with function calls from the bundled `C` library. These functions can still be overloaded and replaced by user's own version, but one cannot simply look inside of the routines, which are sped up this way. There is also a third possibility: to use a functional language and to perform typical for a functional language optimisations in the language compiler. This way our internal language could be feature–rich and reasonably fast, but it will have the price of writing a, say, `LISP` interpreter in an imperative language, thus not as a primary task, but just as a mere implementation of an internal CAS language. However a decent implementation of the all computer algebra algorithms, we wish to see in a proper computer algebra system, should also be not forgotten.

An interesting approach in this field was taken by Christian Bauer, Alexander Frink, Richard Kreckel et al., the developers of `GiNaC` [BFK02, GiN07]. This computer algebra system was written in `C++` and it maintains `C++` as its main interface. It is made in a very simple way: `GiNaC` is rather a computer algebra library, than a complete system with 3D plotter, audio file importer and colourful movie maker. So the primary use of `GiNaC` is to write your own `C++` program, while using arbitrary precision numbers, polynomials, matrices, expression evaluation and other nice (and lightening–fast!) computer algebra functions, offered by the `GiNaC` library. As the authors of GiNaC state:

> Its design is revolutionary in a sense that contrary to other CAS it does not try to provide extensive algebraic capabilities and a simple programming language but instead accepts a given language (`C++`) and extends it by a set of algebraic capabilities.

This approach is very interesting and powerful, but the interactive front end program of `GiNaC`, the `ginsh`, is much less powerful due to a rather weak language. It was, however, never intended to be a complete `GiNaC` interface. The possibility to use all the `GiNaC`'s features at an interactive prompt requires a `C++` *interpreter*.

[2] Matlab is still capable of computer algebra system stylish arbitrary exact computations with a symbolic computation "toolbox".

While interpreting `C++` is not very nice (although possible: [cin07]), it is much easier with `Haskell`: aside from the Glasgow Haskell Compiler [ghc07], we have Hugs, the `Haskell` interpreter. Also, GHC itself offers an interactive version, GHCi. The later is also capable of loading precompiled object files into the interpreted environment. With this achievement one has the possibility to write a computer algebra system, which *internal*, interface language equals its *external*, implementation language, and this language is a functional one.

The idea of `GiNaC` was not born in vain: most *long* CAS–supported computations are run in "batch mode", with no user interaction. It seems plausible not to wait in front of a command prompt for the result for hours, days or even months.[3] On the other hand, most of CAS–based *development* is done in an interactive environment, in a "shell". If one could use the same language both for developing and for lengthy computations, this would be a major success in sparing developers' work time [PH94] and gaining stability of computations.

Now why not just make both: a compiler *and* an interpreter of CAS' internal language? The problem is, that despite many efforts, the internal languages of computer algebra systems are slow. On the other hand, we already have a fast language in our (currently fictitious) CAS–developing project. This is the language, the CAS itself is written in, the *external* language. But, one may oppose, the whole game with computer algebra system's *internal* language was started, because the external language was not high–level enough for vectors, matrices, polynomials and all the other expressions, which are eager wanted in a full–fledged CAS. Now we come back to the beginning of this article. Functional languages *are* complicated and high–level enough to have all the aforementioned objects [Pey03, TT07, Mec07, BDea03]. Functional languages have very compact code size and rapid development times [PH94]. Most functional languages have very interesting data structures [Oka98] and language design features, which benefit both featuring them as an external or as an internal language. And some modern functional languages already have an efficient compiler and an interpreter implemented, which leads us to the future goal of external and internal language fusion. Languages like `Haskell`.

4 Getting more practical

Now as we have seen some *theoretical* reasons for a CAS to be implemented in `Haskell`, let's take a look at some examples. At first we shall examine the univariate polynomials. One can hardly imagine a computer algebra system without them, polynomials are used in thousands of higher–level algorithms and the operations with polynomials should undoubtedly be fast. In this article we do not develop fast polynomial multiplication – it is a large topic beyond the scope of

[3] In this case one might think of porting his/her CAS–based program to some low–level language and let, say, `FORTRAN` run the number–crunching mills. However this is an highly interactive and bug–ridden process. And the `FORTRAN` program is to be tested for errors *again*, before the real computations may begin: the thoroughly tested CAS–routines are not enough!

this text. We rather look at the existent implementations and note some design patterns, making the code faster. Unfortunately as of today neither of proposed `Haskell` implementations of univariate polynomials uses sub–quadratic algorithms like Karatsuba[4], Toom–Cook or Schönhage–Strassen algorithms.

We have tested four different implementations:

1. Our own naive implementation with lists of integers,
2. Implementation from *Haskell for Math* [Amo07],
3. Implementation from *Numeric Prelude* [TT07]
4. Our naive implementation, modified à la Numeric Prelude

All tests were run on the same machine[5] with the same compiler – GHC 6.6.1, the latest stable version currently available. For the same length n the test was run ten times, the mean value of measured execution time is further used. We multiply two dense univariate $(n-1)$–grade polynomials with random coefficients. The coefficients are normally distributed integers and lie in the range $[-400, 400]$.

What we are testing are not the algorithms – they all represent pretty the same "school" multiplication, and not the compiler options, but the layout of classes and data structures. Naive implementation uses "dumb" lists of `Int`s, other implementations build chain of types similar to the algebraic objects. E. g. one can define addition and subtraction in a *group* object, multiplication of some group members in a *ring* object and the reciprocal value of an arbitrary ring element in a *field*. Two consequences are to be stated.

– The simplest implementation is *not* the fastest.
– the type hierarchy matters more than the design of a single function.

We have also tried a naive implementation with floating point arithmetic, but it produces roughly twice worse result, as naive integer implementation: it seems that the integer values are too small for floating point arithmetic to beat the hardware integer arithmetic implementation: for larger integer values the performance gap is narrower, for "sufficient large" integers the floating point implementation should outperform the integer one, while still remaining exact, cf. [vzGG03].

There was also a remarkable side effect. We have to state that *really* large lists in the source files kill the compiler. Simply defining a module consisting of two "functions" `f` and `g` as lists of 10000 elements each led to the compilation time of several minutes for this module. An overview of test results is provided in Figure 1.

We would like to discuss briefly another example. We take a well–known and very quickly growing function on integers: the factorial. We have tested the famous `Haskell` one–liner `fac n = product [1..n]`, and two `C++` implementations. Both base on the CLN – the arbitrary precision library used in GiNaC.

[4] Although an implementation of Karatsuba's algorithm in `Haskell` was suggested in [HL00, Sch98].
[5] AMD Athlon XP 2200+ with 512 Mb of RAM, running Debian GNU/Linux 4.0

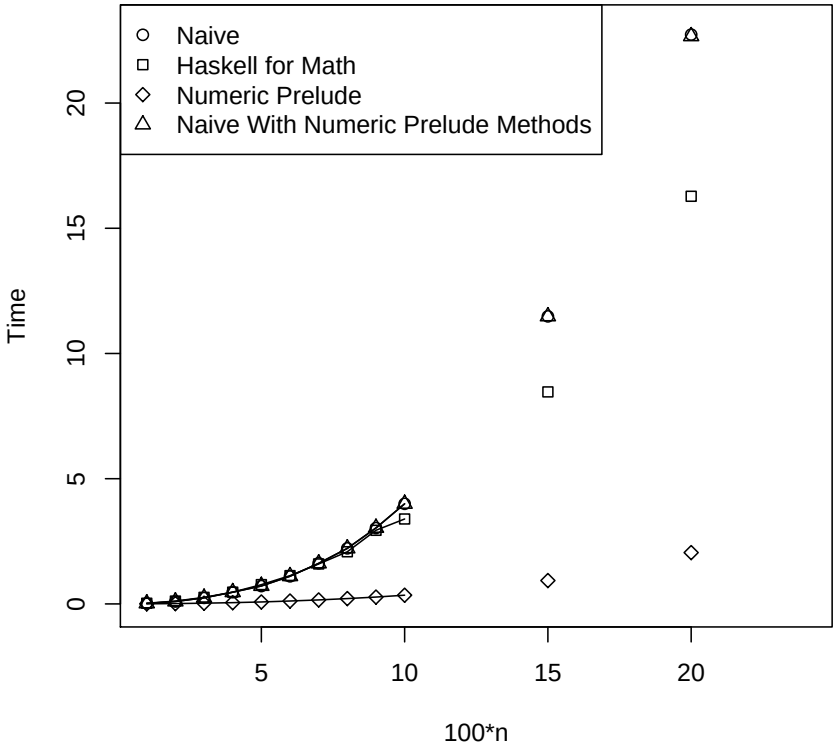


Fig. 1. Multiplication of two univariate polynomials of degree $n-1$. The coefficients are integers in $[-400, 400]$. Time is measured in seconds(!). We conclude the strong need in sub–quadratic implementations. Note the overlapping values for both naive methods.

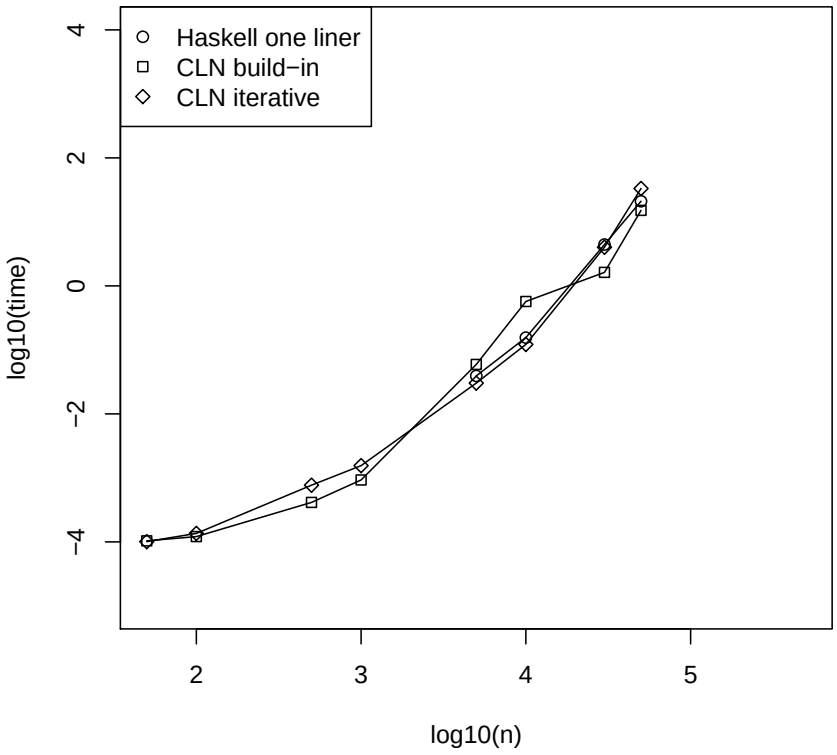


Fig. 2. Computing the factorial. Both execution time and the argument are stated as $\log_{10}$ of the real value, otherwise the difference in execution time for different arguments is too big to be plotted precisely. Note that the difference e. g. between computing $10^4!$ with naive Haskell method and the build–in CLN method is tenfold. The CLN library utilises sub–quadratic multiplication.

One implementation uses build–in factorial function from the CLN. It makes use of table look–ups and computes some parts of the factorial value in divide and conquer fashion. The other `C++` implementation is an unoptimised one, but it still uses CLN build–in multiplication and large integers. We find this implementation comparable with the naive `Haskell` implementation. It should be stated that the graphical representation of the obtained results (see Figure 2) presents the logarithms of the original values, so the difference between two pretty narrow values is larger, as it may appear. The first values for `Haskell` implementation are not available – the results were too biased. For example the timings for $n = 50$ and $n = 100$ were equal.

5 Conclusions and Future Work

Functional languages and computer algebra are two rapid developing research areas, an intersection of these two areas is highly interesting and rather unexplored. In the aspect of practical implementations: modern algorithms of computer algebra should be implemented in relevant `Haskell` software packages, as the naive implementation typically leads to asymptotically bad complexity. One should carefully design his/her data structures in such implementations, as they control a significant factor in the execution time for the same complexity class. The aforementioned algorithms should provide

- fast integer multiplication,
- fast polynomial multiplication,
- efficient Euclid's algorithm for polynomials,
- efficient vector and matrix computations,
- framework for symbolic computation and object manipulation.

Such foundation will be a solid base for more complex research areas, including

- algorithms of numerical number theory,
- implementation of public key cryptography,
- algorithms of computational algebraic geometry, based on gröbner bases,
- symbolic integration,
- automated theorem proofs.

References

Amo07. David Amos. Haskell for Math program. `http://www.polyomino.f2s.com/david/haskell/codeindex.html`, August 2007.

BBB+06. C. Batut, K. Belabas, D. Bernardi, H. Cohen, and M. Olivier. *User's Guide to PARI/GP.* Universite Bordeaux I, version 2.3.2 edition, 2006. `http://pari.math.u-bordeaux.fr/pub/pari/manuals/2.3.2/users.ps.gz`, retrieved in August 2007.

BDea03. Manuel Bronstein, James Davenport, and Albrecht Fortenbacher et al. *AXIOM – the 30 year horizon.* 2003. `http://portal.axiom-developer.org/public/book2.pdf`, retrieved in August 2007.

BFK02. Christian Bauer, Alexander Frink, and Richard Kreckel. Introduction to the GiNaC Framework for Symbolic Computation within the C++ Programming Language. *J. of Symbolic Computation*, 33:1–12, 2002.

BJDM97. Richard S. Bird, Geraint Jones, and Oege De Moor. More haste, less speed: lazy versus eager evaluation. *Journal of Functional Programming*, 7(5):541–547, 1997.

BMP94. M. O. Benouamer, D. Michelucci, and B. Peroche. Error-free boundary evaluation based on a lazy rational arithmetic: a detailed implementation. *Computer Aided Design*, 1994.

cin07. Cint, the c/c++ interpreter, version 5.16.19. `http://root.cern.ch/root/Cint.html`, March 2007. retrieved in August 2007.

Coh95. Henri Cohen. *A Course in Computational Algebraic Number Theory*. Springer, 1995.

Dav94. J. H. Davenport. Computer algebra – past, present and future. In *Euromath Bulletin*, volume 1, pages 25–44, 1994.

Eri07. Ericsson AB. *Erlang Reference Manual*, 2007.

Fat72. Richard J. Fateman. *Essays in Algebraic Simplification*. PhD thesis, Massachusetts Institute of Technology, April 1972. A Revision of A Thesis.

ghc07. The Glorious Glasgow Haskell Compilation System User's Guide. `http://www.haskell.org/ghc/docs/latest/users_guide.pdf`, April 2007. retrieved in August 2007.

GiN07. GiNaC program. `http://www.ginac.de`, August 2007.

GMa07a. Hpc-grid for maple program. `http://www.maplesoft.com/products/toolboxes/HPCgrid/index.aspx`, August 2007.

GMa07b. gridmathematica2 program. `http://www.wolfram.com/products/gridmathematica/`, August 2007.

HAC+07. K. Hammond, A. Al Zain, G. Cooperman, D. Petcu, and P. Trinder. Symgrid: a framework for symbolic computation on the grid. LNCS 4703. EuroPar'07 – European Conf. on Parallel Processing, Rennes, France, Spinger-Verlag, August 2007.

HK05. Bruno Haible and Richard Kreckel. *CLN, a class library for numbers manual*, 2005.

HL00. Christoph A. Herrmann and Chrisitan Lengauer. HDC: A Higher–Order Language for Divide–and-Conquer. *Parallel Processing Letters*, 2000.

Knu98. Donald E. Knuth. *The Art of Computer Programming*, volume 2. Addison–Wesley, third edition, 1998.

LOP05. Rita Loogen, Yolanda Ortega–Mallén, and Ricardo Peña–Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, (15):431–475, 2005. Special Issue on Functional Approaches to High-Performance Parallel Programming.

Mat99. The Mathworks Inc. *Using MATLAB*, 5.3 edition, 1999. Manual.

Mec07. Serge Mechveliani. DoCon the algebraic domain constructor program. `http://www.haskell.org/docon/`, 2007. retrieved in August 2007.

Mil95. Gérard Milmeister. Functional kernels with modules. Master's thesis, ETH Zürich, 1995.

Oka98. Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

Pey93. Simon L Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G–machine. Department of Computing Science, University of Glasgow, 1993. Version 2.5.

Pey03. Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, December 2003.

PH94. M. P. Jones P. Hudak. Haskell vs. Ada vs. C++ vs. awk vs.... An experiment in software prototyping productivity. Yale University, Department of Computer Science, July 1994.

Roy04. Peter Van Roy, editor. *Multiparadigm Programming in Mozart/Oz.* Second International Conference, MOZ, 2004.

RSN95. J. Hicks S. Aditya L. Augustsson J. Maessen Y. Zhou R. S. Nikhil, L. A. Arvind. *pH Language Reference Manual, Version 1.0.* Massachusetts Institute of Technology, 1995. Computation Structures Group Memo No. 396.

Sch82. Arnold Schönhage. Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coefficients. volume 144 of *Lect. Notes Comp. Sci.*, pages 3–15. EUROCAM '82: European Computer Algebra Conference (Marseille, France), April 1982.

Sch98. Christian Schaller. Elimination von Funktionen höherer Ordnung in Haskell–Programmen. Master's thesis, Universität Passau, September 1998.

SCI07. Symbolic Computation Infrastructure for Europe project. `http://www.symbolic-computation.org/`, 2007. retrieved in August 2007.

SS71. Arnold Schönhage and Volker Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3–4):281–292, 1971.

TBD+98. Philip W. Trinder, Ed. Barry Jr., M. Kei Davis, Kevin Hammond, Sahalu B. Junaidu, Ulrike Klusik, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. GpH: An Architecture–Independent Functional Language. In *Glasgow Functional Programming Workshop*, Pitlochry, Scotland, 1998.

TT07. Dylan Thurston and Henning Thielemann. Haskell Numeric Prelude program. `http://darcs.haskell.org/numericprelude/`, August 2007.

vzGG03. Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra.* Cambridge University Press, second edition, 2003.

Wil63. J. H. Wilkinson. *Rounding Errors in Algebraic Processes.* Prentice Hall, 1963.

Wil84. J. H. Wilkinson. The perfidious polynomial. In G. H. Golub, editor, *Studies in Numerical Analysis*, volume 24, pages 1–28. Mathematical Association of America, Washington, D. C., 1984.

Lazy Contract Checking for Immutable Data Structures

Robert Bruce Findler, Shu-yu Guo, and Anne Rogers
{robby,arc,amr}@cs.uchicago.edu

University of Chicago

Summary

Existing contract checkers for data structures force programmers to choose between poor alternatives. Contracts are either built into the functions that construct the data structure, meaning that each object can only be used with a single contract and that a data structure with the invariant cannot be viewed as a subtype of the data structure without the invariant (thus inhibiting abstraction) or contracts being checked eagerly when an operation on the data structure is invoked, meaning that many redundant checks are performed, potentially even changing the program's asymptotic complexity.

We explore the idea of adding a small, controlled amount of laziness to contract checkers so that the contracts on a data structure are only checked as the program inspects the data structure. Unlike contracts on the constructors, our lazy contracts allow subtyping and thus preserve the potential for abstraction. Unlike eagerly-checked contracts on the inspection operations, our contracts do not affect the asymptotic behavior of the program.

1 Introduction

Assertion-based contracts play an important role in the construction of robust software. They give programmers a technique to express program invariants in a familiar notation with familiar semantics. Contracts are expressed as program expressions of type boolean. When the expression's value is true, the contract holds and the program continues. When the expression's value is false, the contract fails, causing the contract checker to abort the program and identify the violation and the violator. Identifying the faulty part of the system helps programmers narrow down the cause of the violation and, in a component-oriented programming setting, exposes culpable component producers.

Contracts enjoy widespread popularity. For example, contracts are currently the most requested addition to Java.[1] In C code, assert statements are particularly popular, even though they do not have enough information to assign blame properly and thus are a degenerate form of contracts. In fact, 60% of the C and C++ entries to the 2005 ICFP programming contest [7] used assertions, even though the software was produced for only a single run.

Despite the popularity of contracts, the state of the art in contract checking for data structure contracts is poor. In order to use contracts on data structures, programmers are forced to choose between copied code (and thus doubled maintenance costs) and very poor performance (often infeasible, as we will show). Our contract checker provides a new alternative. It is designed to strike a balance between performance and the amount

[1] http://bugs.sun.com/bugdatabase/top25_rfes.do, June 27, 2007

```
(module bt mzscheme
  (define-struct node (n left right))
  ...
  (provide (struct node (n left right)) marshal-bt unmarshal-bt))

(module bst
  (require bt)
  ;; a Binary Search Tree (bst) is either null or
  ;; (make-node number[n] bst[left] bst[right])
  ;; where the numbers in left are less than (or equal to) n
  ;; and the numbers in right are greater (or equal to)

  (provide find-bst)
  ;; find-bst : bst number → boolean
  (define (find-bst t n)
    (and (node? t)
         (or (= n (node-n t))
             (and (< n (node-n t))
                  (find-bst (node-left t) n))
             (and (> n (node-n t))
                  (find-bst (node-right t) n))))))
```

Fig. 1. Binary search trees, without contracts

of checking, motivated by the desire to avoid changing the asymptotic complexity of operations that have contracts.

The next section uses binary search trees to make the programmer's existing poor choices plain. Section 3 explains the design of our contract checker and how it limits the amount of checking to recoup tractable performance. Since our design is partially motivated by performance, we spend Section 4 explaining our implementation and Section 5 presenting some performance measurements that validate our design. Section 6 discusses an extension to our contract checker that relaxes the strict asymptotic complexity requirements; by giving the contract checker the freedom to preserve only the amortized complexity, we gain the ability to write more expressive contracts. Section 7 discusses related work and Section 8 concludes.

2 A Rock and a Hard Place

To see how existing techniques for data structure contracts fail programmers, consider a binary search tree library (shown in Figure 1) that is built on top of a binary tree library. The binary tree library is left mostly to the reader's imagination, but a skeleton is shown in the `bt` module, written in PLT Scheme [11]. It exports basic operations on binary trees (marshaling them to and from disk) and a node struct (record) for building and querying the nodes in a binary tree. The **`define-struct`** introduces a new struct that consists of three fields. The **`define-struct`** also introduces five function definitions: `make-node` used to build new nodes, `node?` used to recognize node structs, and

```
(module bst mzscheme
  (require (lib "contract.ss") bt)

  ;; bst? : any → boolean
  (define (bst? t) (bst-between? t −∞ +∞))
  (define (bst-between? t low high)
    (or (null? t)
        (and (node? t)
             (number? (node-n t))
             (≤ low (node-n t) high)
             (bst-between? (node-left t) low (node-n t))
             (bst-between? (node-right t) (node-n t) high))))
  (define (find-bst t n) ...) ;; as in figure 1

  (provide/contract
   [find-bst (→ bst? number? boolean?)]
   [bst? (→ any/c boolean?)]))
```

Fig. 2. Binary search trees, with contracts

`node-n`, `node-left`, and `node-right` used to extract the fields from a node struct. In general, a struct definition introduces a single maker, a single predicate, and one selector per field. The **`provide`** clause exports the struct and the marshaling functions.

The `bst` module requires the `bt` module and defines a binary search tree data structure in a comment, according to the discipline of *How to Design Programs* [5]. The comment specifies that binary search trees have the same shape and use the same node struct as binary trees, but also have the binary search tree invariant. The programmer carefully uses the same basic data structure so that the existing library for binary trees (marshaling and unmarshaling functions in this case) can also be used with binary search trees. Beyond the data definition, the `bst` module also provides `find-bst` a function for finding numbers in binary search trees that takes advantage of the binary search tree invariant to avoid the recursive calls when it is safe to do so.

As the program grows from a little script to a part of a robust application, its author decides to improve the reliability of the program by writing a checkable contract on the data structure as shown in Figure 2. The `bst?` function uses the `bst-between?` helper function to test whether its input is a binary search tree. The function `bst-between?` enforces the binary search tree invariant using two accumulators, a lower and upper bound on the values in the tree. The accumulators are initially negative and positive infinity respectively, and as the traversal passes each interior node, the bounds tighten in the recursive calls.

Finally, the `bst?` predicate is used in the contracts for the provided functions.[2] The contract on `find-bst` is an → contract and is written using prefix notation. The last argument to → is a predicate on the result of `find-bst`, ensuring that it always produces

[2] We use the PLT Scheme contract library's notation [21] throughout.

booleans. The other two arguments are predicates on the inputs to `find-bst`, ensuring that the first argument is a binary search tree and that the second argument is a number. Similarly, the contract on `bst?` ensures that it is a predicate function.

Although it may not be obvious at first glance, the binary search tree portion of the revised library is now completely useless. In particular checking `find-bst`'s contract means that the `bst?` predicate is called on each argument supplied to `find-bst` in order to enforce the pre-condition (domain) contract. Since `bst?` traverses the entire tree, it ruins the optimization built into the `find-bst` function, changing the asymptotic complexity from logarithmic to linear, an exponential slowdown.

This state of affairs leaves the programmer in a bind; both the loss of performance and the loss of reliability are unacceptable. The conventional solution to this problem is to hide the raw struct operations behind an opaque module boundary and only export operations that guarantee the binary search tree invariants (*e.g.*, self-balancing insert plus an empty binary search tree). Of course, this solution has the problem that a client of `bst` module cannot reuse the `bt` operations on `bst`s.

A programmer may attempt to work around this by providing new versions of each of the `bt` operations that simply unwrap a `bst` struct, apply the operation, and then rewrap it. This approach is not desirable for two reasons. The `bst` programmer has to be able to anticipate all future extensions to the `bt` library but, even more troublesome, is that the binary search tree author must now verify that none of the `bt` operations violate the binary search tree invariant, rather than letting the system itself ensure the binary search tree invariant holds.

Another solution is to provide injection and projection functions that convert binary search trees to and from binary trees and, along the way, verify the invariant. This solution amounts to changing the pre-condition on the `find` operation to a simple check, but requiring that programmers rewrite their programs to explicitly decide where to do the real checks. Worse, it is not always possible to avoid an asymptotic slowdown when binary search tree operations are interleaved with binary tree operations.

In general code reuse is enabled by the ability to view a data structure with an invariant (like the binary search tree) as the same data structure but without the invariant. Or, put another way, code reuse is hindered by taking that ability away or allowing it only when accompanied by expensive invariant checks. Thus, the goal of this work is to provide a new form of contract checking that allows programmers to view binary search trees as binary trees, without any special action on the part of the programmer.

Throughout the remainder of this paper, we continue to use binary search trees as a motivating example. Nevertheless, our technique applies to many data structures that have invariants: heaps, self-balancing trees, sorted lists, etc. It is also useful whenever one wishes to use refinement types (but when a refinement type checker is not strong enough) such as even-length or non-empty lists, or viewing the result of Scheme's `read` as having a particular shape. Another particularly fertile ground is a compiler's intermediate representation. Well-known intermediate representations like CPS and A-normal form [10] are easily expressed as contracts over the general expression type, and compiler authors who take advantage of them can determine which pass of a compiler has failed when bad output is produced.

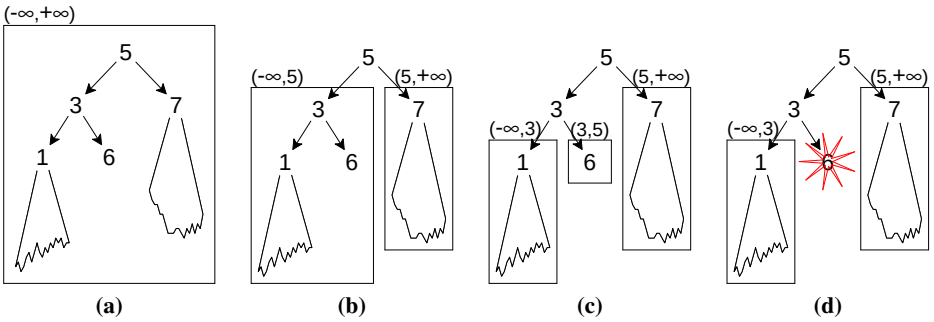


Fig. 3. Evolution of attributes during traversal of tree

3 Lazy Contract Checking

Our solution to this problem is to introduce a new kind of contract for data structures. These contracts have the benefit of the contracts in Figure 2, namely they permit the programmer to use a single value with multiple, different contracts, but instead of eagerly checking the entire data structure when checking a contract, our contracts lazily check the portions of the data structure that the function inspects, and only when it inspects them.

Our contracts extend MzScheme's **`define-struct`** to **`define-contract-struct`**. It has the same syntactic shape as **`define-struct`**, but in addition to introducing a maker, predicate, and selectors, it also introduces a contract constructor. For example, the declaration

```
(define-contract-struct node (n left right))
```

introduces `node/dc` (read as "node dependent contract"). Its shape is

```
(node/dc [n contract-expr]
         [left (n) contract-expr]
         [right (n) contract-expr])
```

where each clause indicates the contract on the respective field. The `(n)` in the `left` and `right` contract specifications indicates that the contracts for the `left` and `right` fields depend on the value of the `n` field (the variables in the parenthesis are ordinary bound variables, but their names must match the names of other fields of the struct; that is, they may not be alpha-renamed). In general, the contract on any field may depend on any of the fields before it, but the dependencies must be specified explicitly. Of course, `node/dc` is just an instance of a contract constructor; each **`define-contract-struct`** declaration introduces its own dependent contract constructor that expects as many fields as there are in the struct.

Using `node/dc`, the contract for a binary search tree is written as:

```
;; bounded-bst : number number -> contract
(define (bounded-bst lo hi)
  (or/c null?
        (node/dc [n (between/c lo hi)]
                 [left (n) (bounded-bst lo n)]
                 [right (n) (bounded-bst n hi)])))
(define bst (bounded-bst −∞ +∞))
```

The `or/c` contract combinator accepts any number of contracts (or simple predicates) and checks that at least one of them holds. The `between/c` contract combinator accepts two numbers and returns a contract that matches numbers in those bounds. The contract on the left and right sub-trees of an interior node are built by recursively calling `bounded-bst` with different bounds on the values in the tree. The initial contract on a binary search tree is built by calling `bounded-bst` with negative and positive infinity.

The remainder of this section explains how dependent struct contracts behave and continues to use binary search trees as the motivating example.

3.1 Checking During Traversal

The contract checker only checks struct contracts as the program itself inspects the data structure. To see how this plays out, consider this binary search tree and call to `find-bst`.

```
(define a-bst (make-node 5
                (make-node 3
                 (make-node 1 ...)
                 (make-node 6 null null))
                (make-node 7 ...)))
(find-bst a-bst 4)
```

The series of diagrams in Figure 3 shows the evolution of the contracts as `find-bst` traverses `a-bst` seaching for `4`. To represent the contract on the tree, we draw a box around the tree and annotate the box with the contract. So, when the tree is first passed to `find-bst`, it picks up the binary search tree contract and is labeled "(−∞,+∞)", meaning that the elements in the tree must be between −∞ and +∞, corresponding to the contract obtained by calling `(bounded-bst −∞ +∞)`. The first step `find-bst` takes is to examine the top node in the tree. At the point when `find-bst` first extracts a field of the top `node` struct, the contract checker steps in and verifies that the values of the fields of the node match the contract. Verifying that the number in the tree is in the appropriate range is a simple check, but to ensure that the subtrees match their contracts, the contract checker must create new boxes to avoid exploring more of the tree than the program does, as shown in Figure 3 (b).

The labels on the new boxes indicate the new contracts, derived from the binary search tree invariant (as implemented by `bounded-bst`). The left sub-tree's elements must be smaller than `5` and the right sub-tree's elements must be larger than `5`. Figure 3 (c) shows the state of the tree after `find-bst` inspects the left child of the root. Again, the contract checker verifies that the node's value is appropriate and creates new boxes for the sub-trees. At this point in the program, no contract violation is signaled, because the program has not yet discovered the contract violation lurking one level

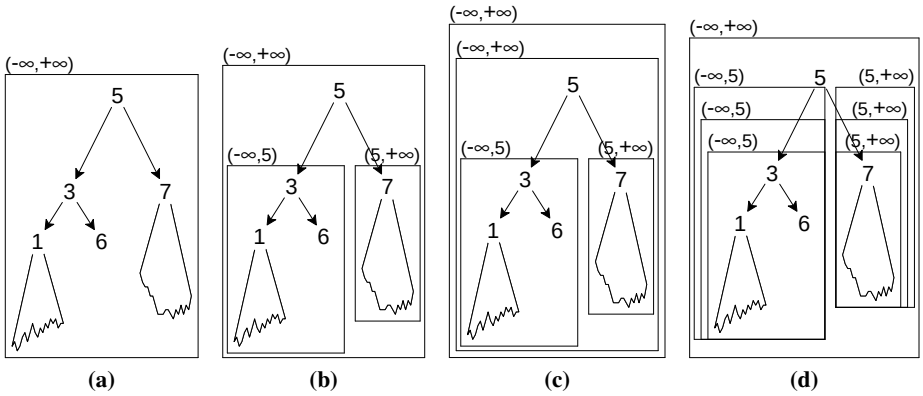


Fig. 4. Evolution of contracts during tree traversal without stronger check

down in the tree. Indeed, if the program never explores that part of the tree, a contract violation will never be signaled. But, since `find-bst` is searching for a `4`, it does inspect that node, and a contract violation is signaled blaming the caller of `find-bst`.

3.2 Redundant Contracts

Although the boxes help eliminate much of the redundant work that eager contract checking would incur, it is still possible to do too much work. In particular, we must be careful to avoid accumulating multiple, redundant boxes on the same tree. To see how this happens, imagine that a tree is built up via an `insert : bst number` $\rightarrow$ `bst` operation that first calls `find-bst` to see if the value is in the tree and, if so, just returns the original tree. Consider the effect of these two calls happens during the evaluation of `(insert (insert a-bst 5) 5)`. Even though the two calls do not change the tree, a naive strategy for putting boxes on contracts accumulates surprisingly many new boxes.

Figure 4 shows what would happen for those two calls. Initially, the tree has no contracts, but as soon as it is passed to `insert`, the binary search tree contract is wrapped around it, as shown in Figure 4 (a). The first thing `insert` does is pass the tree to `find-bst`, along with `5`. Since `5` is in the root node of the tree, `find-bst` triggers the checking of only the first layer of the contracts, pushing contracts down to the left and right sub-children, and removing the outer layer of contracts. After that, the first call to `insert` returns and its post-condition adds another box around the entire tree and we are left with the tree shown in Figure 4 (b).

As the second call to `insert` happens, the pre-condition adds another wrapper to the tree, leaving us with Figure 4 (c). When `insert` calls `find-bst`, it inspects the top portion of the tree, pushing both of the contracts to its subtrees, and then the post-condition of `insert` adds yet another contract outside the tree, leaving us with Figure 4 (d).

To avoid this accumulation, we must be able to detect redundant contracts. In the case of a binary search tree, we can simply compare the bounds. If the box around a

tree has the same (or tighter) bounds than the new box would, then we can just leave the tree alone, relying on the existing contract to guarantee that the new contract holds.

To detect redundant contracts in general, our contract system supports a partial ordering on contracts that is used to compare two contracts to determine if one is stronger than or equal to the other. The ordering is tied to the particular contracts that our system supports. Each contract knows how to compare itself to certain other contracts in our system; if the contract does not recognize the other one, we avoid unsoundness and assume that neither contract is stronger than the other.

As a design principle for our system, we decided that programmers who merely use contracts should not have the responsibility of specifying the stronger relationships. Instead, that responsibility should lie with the programmers that implement the contract combinators (such as `between/c` or $\rightarrow$ or the struct contracts). Accordingly, the stronger relationships are set in stone once a particular contract combinator has been defined. So far, this method has worked well enough for us, but we may also eventually investigate separating the stronger relation definition from the contract combinators and allowing programmers to extend it.

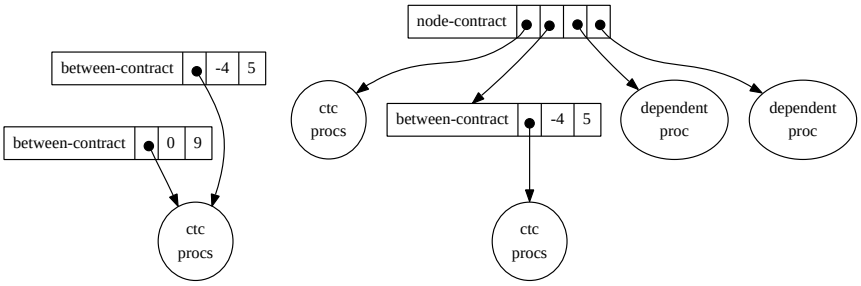
For `between/c` contracts, our system treats the one that accepts the same or fewer numbers to be the stronger contract. One contract on a struct is stronger than another if the contracts on the fields of the first are stronger than the contracts on the fields of the second. Comparing function contracts uses the usual contra-variant ordering. To date, simple structural equality of contracts, combined with the bounds checking of `between/c` has been sufficient for all of the data structure invariants we have encountered (including all those in Okasaki's book [16] and in Cormen, Leiserson and Rivest [4]).

To exploit our new relation on contracts, we simply avoid adding a new contract if the contract already on the data structure is stronger than or equal to the new contract. Note that we do not need to consider blame here; indeed, if two contracts surround a single data structure, the inner contract is always checked before the outer one, since the inner contract was placed on the object first. If the contract already on the data structure is stronger than the new contract, it does not matter who might be blamed if the new contract were to be violated; the existing contract guarantees it never fails.

Once we avoid adding redundant contracts, calling `insert` as above (or even arbitrarily many more times) would result in the wrappers shown in Figure 4 (b). That is, each sub-tree would only have a single wrapper, no matter how many times `insert` is called.

4 Implementation and an Optimization

In our implementation, each contract is represented as a struct that has at least one field. That field contains a reference to a group of functions specific to that kind of contract that interpret the values in the other fields. The representation is similar to the way objects are represented in class-based object-oriented languages: the record of functions is like the vtable of methods and is shared among every contract of a particular kind. As an example, Figure 5 (a) shows a box-and-line diagram for the result of `(between/c −4 5)` and `(between/c 0 9)`. Each points to the same record of functions and has two numbers indicating the range it accepts.



(a) Two between contracts **(b) A node contract**

Fig. 5. Contract layouts

A contract on a struct also has a shared record of contract procedures, but it has one additional field per struct field. Each of those fields is either a contract that the contents of the field must satisfy directly, or it is a function that accepts the values in the other fields and returns such a contract. As an example, the contract

```
(node/dc [n (between/c −4 5)]
         [left (n) (bounded-bst −4 n)]
         [right (n) (bounded-bst n 5)])
```

is shown in Figure 5 (b). The first field is the record of functions. Since the contract on the `n` field does not depend on other contracts, the second field of the contract record is the `between/c` contract. But, the `left` and `right` fields depend on the value of the `n` field, so they are functions that consume the `n` field's value and produce contracts.

Each contract's record of functions includes three functions. The first accepts the contract record and a value and enforces the contract. The second accepts two contracts and returns a boolean indicating whether the first is stronger than the second or not. This function inspects the structure of the contracts and behaves essentially as described in Section 3, except for dependent contracts when the fields are functions, as in Figure 5 (b). In that case, it inspects the closure, comparing the code pointer and the pointers to the elements of the closure (but does not recursively traverse the closure). Since this comparison may fail when standard compiler optimizations are performed, our implementation communicates with the compiler, telling it not to optimize these particular closures. Finally, the third function in the contract accepts a contract record and builds a name for the function to be used in error reporting.

Of course, to support lazy structure contracts, we must not examine the struct's fields right away. Accordingly, the checking function for structs merely verifies that the struct's type matches, and then pairs the contract with the struct. Later, when a selector is applied to the struct, the contract is checked. Figure 6 contains a series of box and pointer diagrams that illustrate this process. The first diagram shows an example binary search tree, where the `null`s representing the empty tree are written `mt` to clarify the figure.

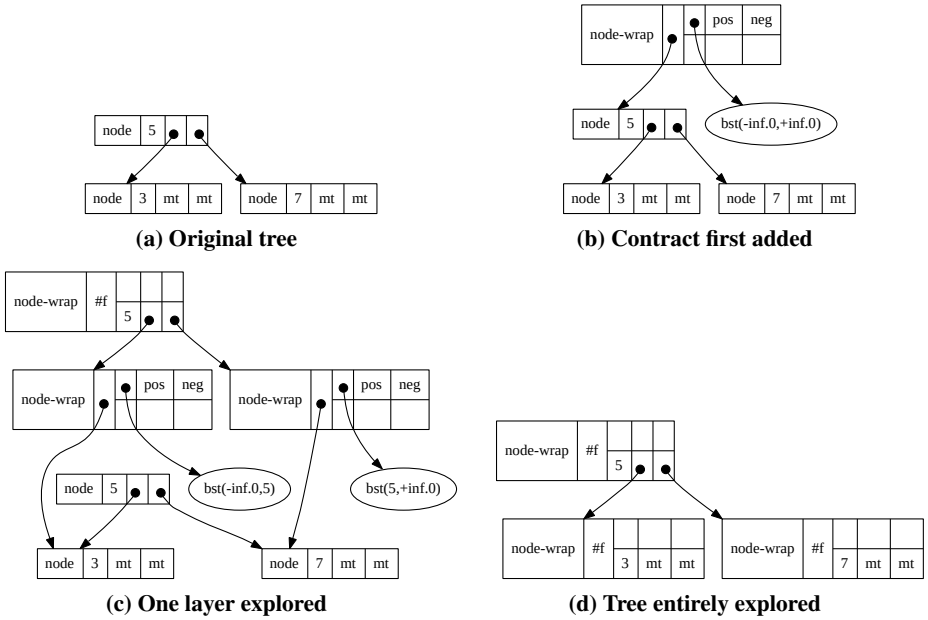


Fig. 6. Evolution of objects during contract checking

Figure 6 (b) shows the tree paired with a between contract in a node-wrap struct. The node-wrap struct that holds the pair has a number of extra fields. The first field refers to the original object, but is also used as flag to indicate if the top row or the bottom row of fields are active. In the case shown, since that first field contains a reference to a struct, the top fields are active. Those fields contain a pointer to the contract, and two names that indicate who is to blame for contract violations. The first name indicates who is to be blamed if this contract fails to hold. The second name is only used to support contract checking of functions that may appear inside this structure. It indicates the name of the party responsible for inputs to those functions (as described by Findler & Blume [8] and Findler & Felleisen [9] in their work on higher order function contract checking).

The other fields are used to implement the removal of the boxes described in Section 3. In particular, once the contract has been checked we know that it will continue to hold for all time, because the data structure is immutable. Accordingly, we place the contracted versions of the fields of the original struct into the bottom row of the node-wrap, to avoid recomputing them. When that happens, we also change the first field to `#f` in order to indicate that the bottom row is active.

Figure 6 (c) shows the same tree, but after a selector has been applied to the struct with the contract, causing the contracts on the fields to be checked. The top node-wrap struct in this diagram is the same node-wrap struct in the top of diagram (b), but now the lower fields are active. The second field (in the bottom row) in that structure is `5`, the contracted version of the first field in the original struct. The final two fields are

the contracted versions of the left and right sub-trees. The left sub-tree now has the contract `(bounded-bst` $-\infty$ `5)`, so it is a node-wrap struct whose first field is not `#f`. This node-wrap's top row is active, since its contract has not yet been checked. Similarly, the right sub-tree now has the unchecked `(bounded-bst 5` $+\infty$`)` contract. Finally, the fourth diagram shows the tree after all of the contracts have been checked. At this point, the tree is very similar to the original tree.

Of course, since the top row and the bottom row are never simultaneously active for any given node-wrap struct, our implementation only has a single set of fields and uses the second field to indicate how to interpret the remaining fields.

After some experimentation with our implementation, we discovered that a significant amount of time is spent in redundant allocation. In particular, the implementation allocates a record for each contract combinator. This becomes expensive when combined with dependent contract checking, since the allocation of the contracts themselves happens during the traversal of the data structure. To compensate, we built an optimization for lazy contracts that folds nested contracts together, in order to cut down on the allocation.

5 Performance

This section presents the results of three experiments we performed on our implementation. Although performing three experiments is not conclusive, the experiments do provide some validation of our contract checker. The first experiment merely validates the claims from Section 2 by showing that eagerly checking the contracts can be arbitrarily slower than lazily checking them. The second experiment is designed to measure the cost of laziness, in the case when laziness is superfluous. The third demonstrates how our lazy contract checker behaves for more realistic applications and provides empirical evidence that it does indeed preserve the asymptotic complexity of the underlying operations.

We ran all of our experiments using MzScheme [11] v370.2 on a dual core 1.66 GHz Mac mini with 2 gigabytes of memory (although each test ran sequentially and only a test that disabled the stronger check allocated a significant amount of memory, discussed in Section 5.3).

5.1 The Cost of Eagerness

As we discussed earlier in this paper, the cost of eagerly checking data structure contracts can be arbitrarily bad. To verify this claim, we ran a simple test with our implementation. We built a toy program that constructs increasingly larger complete binary trees, numbers them via an inorder traversal (to satisfy the binary search tree invariant), and then measure the time it takes to search for each number.

We added the contracts from Figure 2 and our lazy contracts to the `find-bst` function and timed the calls to the two versions of `find-bst`. Figure 7 shows the results. The x-axis ranges over the number of elements in the binary search trees, and the y-axis shows the slowdown as the ratio of the time required to call `find-bst` with the the eager contracts to the time required to call `find-bst` with the lazy contracts. Each point on the graph represents a single run of each program. Even at the relatively modest size of a 10,000 element binary search tree, eager checking incurs an overhead of more than

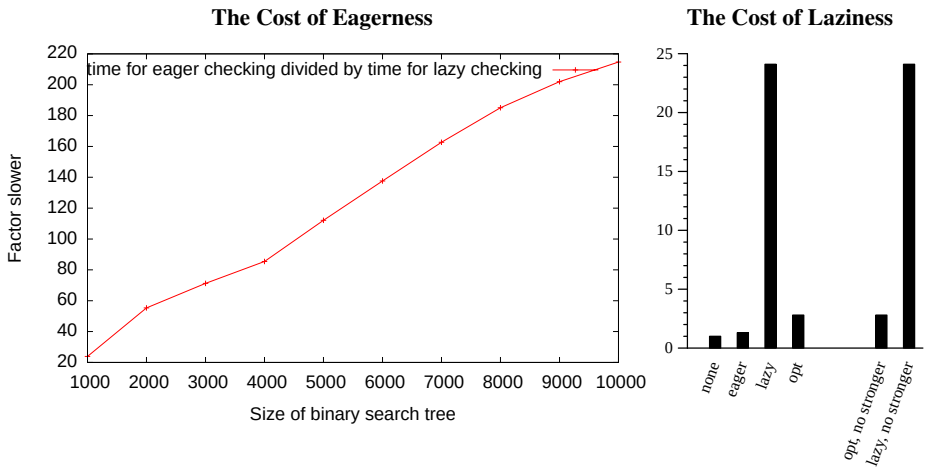


Fig. 7. Synthetic benchmark results

200 times that of lazy checking. More worryingly, however, is the shape of the graph; as the size of the binary search tree increases, so does the slowdown.

5.2 The Cost of Laziness

To measure the cost of laziness, we wrote a program that constructs a list of the numbers from 1 to 100,000. We did not use MzScheme's built-in `cons` function, since our contracts only support user-defined structs. Instead, we made a two field struct and used that for the pairs in the list. Once the list is built, the program applies different implementations of a contract that specifies that the list is sorted in ascending order, and then iterates over the list. Since the function always iterates over the entire list, delaying the contract does not improve the running time. Accordingly, this test helps us understand the cost of our implementation's bookkeeping. The right-hand side of Figure 7 shows those measurements. The height of each bar in the figure is the ratio of the performance of a particular contract to the performance of the code without any contracts.

The first four bars show the slowdown of the running time as compared to the version without contracts. The first bar (none) just give a sense of scale; the slowdown for the version without contracts as compared to itself is 1. The second bar (eager) shows the slowdown for the eager contract that iterates down the entire list during the pre-condition checking, the third for the lazy contracts (lazy) and the fourth for lazy contracts with our optimization (opt). Each bar corresponds to the average result of five runs. We see that the cost of the lazy contract bookkeeping is about a factor of 24 for this program, compared to a factor of 1.2 for the eager contract. Our optimization brings this down to a more reasonable factor of 2.8.

For a final experiment to measure the cost of laziness, we also set out to determine the cost of evaluating the stronger relation. For the program in this section, we know that no contract is ever going to be applied twice to the same object, so the stronger

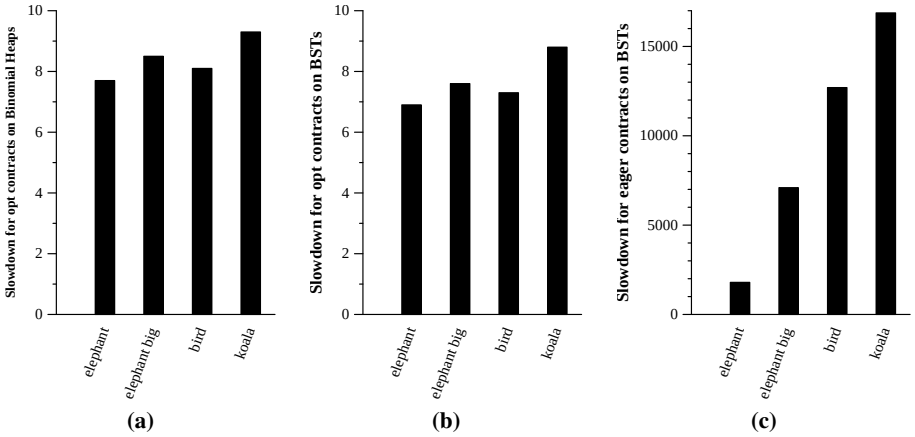


Fig. 8. Binomial heap and binary search tree experiments

relation has no positive effect on the running time. So, we disabled the code that does that check and re-ran the tests. The results are shown as the final two bars in Figure 7. They show that the stronger check does not have a significant cost, when compared to the cost of the contract checking itself.

5.3 A Realistic Benchmark

For this experiment, we extracted traces of calls to a heap data structure from a colleague's vision algorithm [6]. We used four traces that are named after the images we used when extracting each trace: elephant, elephant-big, bird, and koala. The traces vary in size: elephant has roughly 22,000 inserts and 5,700 remove mins, whereas koala has more than 300,000 inserts and nearly 150,000 remove mins. We then coded up a binomial heap, as described in Okasaki's book [16] and ran the traces with three variations of the contracts on the heap operations: no contracts, optimized lazy contracts, and eager contracts. Accordingly, these results represent the times for only the data structure operations, not the original program that used the heap.

Figure 8 (a) shows the slowdown for running the optimized lazy contract checking; note that this chart's scale is not the same as that in Figure 7. As you can see, even though the traces vary in size, the overhead is relatively constant, encouraging us to believe that our contract checker only adds a constant overhead.

We also synthesized traces for binary search trees from the heap traces. We replaced each heap insertion with a naive binary search tree insertion and replaced each extract minimum with a lookup of a random element in the tree. Figure 8 (b) shows the slowdown for those when using the optimized contract checker and, as before, the overheads are relatively constant.

Figure 8 (c) shows the slowdown for using the eagerly checked contracts. These runs take a long time — running the koala trace requires about four days of cpu time, for example. There are two important features of this chart. First, the scale is significantly different from that of the first two charts. The overheads are at least 2,000 and can be as

bad as 20,000. Second, the overheads vary considerably between traces, demonstrating that the eager checking does not preserve the asymptotic complexity of the program.

Finally, we performed one more experiment. We disabled the stronger check and then re-ran program. Partway through the smallest trace, MzScheme had 1.5 gigabytes of resident storage (according to top) and then the machine proceeded to swap, making very little additional progress. This behavior indicates that a well-designed stronger relation is a crucial part of making the implementation practical.

6 Preserving Amortized Complexity

Implicit in the strategy of our lazy contract checker is a limitation its expressiveness. In particular, the contract for the unexplored portion of the data structure must be expressible using only information in the explored portion of the data structure. For example, we cannot express a contract that says that a binary tree is full using only the contracts described so far. Indeed, our strategy is based on exploiting this limitation to preserve the asymptotic complexity.

In order to gain additional expressiveness, we allow programmers to write contracts that only preserve the asymptotic complexity in an amortized sense. This weakening allows the contract system to propagate information back up the tree and gives us enough expressiveness to be able to support the full binary tree contract. Roughly, when the program inspects a particular path from the root of a tree to a leaf, we propagate the height along that path back to the root, allowing the contract system to signal an error when two different heights are encountered. In the jargon of attribute grammars, this gives our contracts the power of synthesized attributes (inherited attributes were all they could use without this extension). For details of this extension, see the MzLib manual [21], specifically the details of **`where`** clauses.

7 Related work

The idea of software contracts dates back to the 1970s [17]. In the 1980s, Meyer developed an entire philosophy of software design based on contracts, embodied in his object-oriented programming language Eiffel [14]. Nowadays, contracts are available in one form or another for many programming languages (e.g., C [22], C++ [19], Haskell [12], Java [13], Perl [3], Python [18], Scheme [20], and Smalltalk [1]).

Although the authors did make the connection until much of this work had been done, this work is a direct intellectual descendent of Okasaki's dissertation [15], where Okasaki demonstrates that a controlled amount of laziness, in an otherwise strict language, makes achieving desired asymptotic bounds tractable. We cannot, however, use Okasaki's `$` operator directly, since we need fine grained control over the laziness to exploit the stronger relation.

From a contracts perspective, our work is anticipated by Chitil, McNeill, and Runciman's Lazy Assertions [2]. They observe that eagerly checking assertions in a lazy setting can introduce non-termination where none should rightly be. In particular, a strict assertion on an infinite list should not explore the entire list unless the program itself explores the entire list. Our work is motivated by a similar concern, but in a strict setting, but we believe some of the ideas here (like the stronger relation and possibly the ideas in Section 6) carry back to thier setting.

Hinze, Jeuring, and Löh's contract checker [12] is also a contract checker for Haskell (that correctly handles blame), but their checker explores parts of the data structure that the program does not. For example, the `is(sort)` example contract in Section 6 of their paper explores the entire list; a similar contract in our system would not.

Beyond that, there is little other work on checking data structure contracts, except when using naive strategies. Eiffel, the language most focused on contract checking, provides no native support for lazy contract checking. Tremblay and Cheston [23] wrote an algorithms and data structures textbook using Eiffel, but the contracts in their text either only partially check the data structure invariants or check them as the data structure is constructed.

8 What Have We Gained?

In some sense, this work puts data structure contract checking on an even footing with function-based contract checking. Specifically, when checking a contract on a function, violations can go undetected if the function is never called with an input that would trigger an error. Similarly, consider this (supposed) binary search tree:

```
(make-node 5 (make-node 7 #f #f) (make-node 207 #f #f))
```

If `find-bst` is called with that tree and with, say, `6`, the contract checker will not discover the violation. Even worse, if it is called with `7`, `find-bst` with indicate that `7` is not in the binary search tree, and the contract checker will still fail to detect the violation. Of course, similar behavior can happen with functions (in fact, this binary search tree could be encoded as a function to achieve precisely the same behavior) and yet function contracts enjoy wide-spread use.

We believe that our data structure contracts have the potential to enjoy similar wide-spread use, for two reasons. First, it is rare for a data structure to be built that will not eventually be completely explored in a long-running application. Even though the two calls to `find-bst` above do not detect the violation, it seems likely that some later call to `find-bst` will ask for a number smaller than `5`, resulting in a contract violation.

Second, our checker makes checking data structure contracts feasible. As discussed in Section 5.3, using either the naive strategy of eagerly checking the contracts, or even avoiding the stronger check makes checking the contracts infeasible, for at least one realistic program. Intuitively, we expect the naive strategy to fail in general, simply because the change to the asymptotic complexity incurred by the naive checker is a tremendous expense.

Fundamentally, the question we ask is how much contract checking can we expect a program to be able to afford? Our contract checker represents one answer to this question that does not take into account any a priori knowledge about the program's behavior; it provides a maximal amount of contract checking that we can reasonably expect the program to be able to afford, namely a constant factor.

Acknowledgements. Thanks to Ryan Culpepper and Matthew Flatt for PLT Scheme infrastructure support and to Matthew for comments on drafts of the paper. Thanks to Pedro Felzenszwalb for supplying us with the heap traces we use in our experiments. Also, thanks to Matthias Felleisen and Jacob Matthews for several enlightening discussions and comments on this paper. This work is supported in part by the NSF.

References

1. Carrillo-Castellon, M., J. Garcia-Molina, E. Pimentel and I. Repiso. Design by contract in Smalltalk. *Journal of Object-Oriented Programming*, 7(9):23–28, 1996.
2. Chitil, O., D. McNeill and C. Runciman. Lazy assertions. In Trinder, P., G. Michaelson and R. Pena, editors, *Implementation of Functional Languages: 15th International Workshop, IFL 2003*, LNCS 3145. Springer, November 2004.
3. Conway, D. and C. G. Goebel. Class::Contract – design-by-contract OO in Perl. `http://search.cpan.org/~ggoebel/Class-Contract-1.14/`.
4. Cormen, T. H., C. E. Leiserson and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
5. Felleisen, M., R. B. Findler, M. Flatt and S. Krishnamurthi. *How to Design Programs*. MIT Press, 2001. `http://www.htdp.org/`.
6. Felzenszwalb, P. and D. McAllester. A min-cover approach for finding salient curves. In *IEEE Workshop on Perceptual Organization in Computer Vision*, 2006. `http://people.cs.uchicago.edu/~pff/papers/`.
7. Findler, Barzilay, Blume, Codik, Felleisen, Flatt, Huang, Matthews, McCarthy, Scott, Press, Rainey, Reppy, Riehl, Spiro, Tucker and Wick. The eighth annual ICFP programming contest. `http://icfpc.plt-scheme.org/`.
8. Findler, R. B. and M. Blume. Contracts as pairs of projections. In *International Symposium on Functional and Logic Programming*, 2006.
9. Findler, R. B. and M. Felleisen. Contracts for higher-order functions. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, 2002.
10. Flanagan, C., A. Sabry, B. Duba and M. Felleisen. The essence of compiling with continuations. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1993.
11. Flatt, M. PLT MzScheme: Language manual. Technical Report PLT-TR05-1-v300, PLT Scheme Inc., 2005. `http://www.plt-scheme.org/techreports/`.
12. Hinze, R., J. Jeuring and A. Löh. Typed contracts for functional programming. In *International Symposium on Functional and Logic Programming*, 2006.
13. Karaorman, M., U. Hölzle and J. Bruno. jContractor: A reflective Java library to support design by contract. In *Proceedings of Meta-Level Architectures and Reflection*, volume 1616 of *lncs*, July 1999.
14. Meyer, B. *Eiffel: The Language*. Prentice Hall, 1992.
15. Okasaki, C. *Purely Functional Data Structures*. PhD thesis, Carnegie Mellon University, September 1996. Technical Report CMU-CS-96-177.
16. Okasaki, C. *Purely Functional Data Structures*. Cambridge University Press, 1999.
17. Parnas, D. L. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.
18. Plösch, R. Design by contract for Python. In *IEEE Proceedings of the Joint Asia Pacific Software Engineering Conference*, 1997. `http://citeseer.nj.nec.com/257710.html`.
19. Plösch, R. and J. Pichler. Contracts: From analysis to C++ implementation. In *Technology of Object-Oriented Languages and Systems*, pages 248–257, 1999.
20. PLT. PLT MzLib: Libraries manual. Technical Report PLT-TR05-4-v300, PLT Scheme Inc., 2005. `http://www.plt-scheme.org/techreports/`.
21. PLT. PLT MzLib: Libraries manual. Technical Report PLT-TR2006-4-v352, PLT Scheme Inc., 2006. `http://www.plt-scheme.org/techreports/`.
22. Rosenblum, D. S. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, Janurary 1995.
23. Tremblay, J.-P. and G. A. Chesterton. *Data Structures and Software Development in an Object-Oriented Domain: Eiffel Edition*. Prentice Hall, 2001.

Haskell − Join − Rules

Martin Sulzmann and Edmund S. L. Lam

School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
{sulzmann,lamsoonl}@comp.nus.edu.sg

Abstract. We report some preliminary results on integrating Constraint Handling Rules (CHR) into Haskell to support concurrent programming. CHR is a concurrent committed-choice constraint logic programming language to describe transformations (rewritings) among multi-sets of constraints (atomic formulae). The CHR model is related to the join calculus which also supports a form of multi-set rewriting. CHR additionally support guards and propagation rules whereas the join calculus supports besides asynchronous also synchronized method calls. We integrate all those features in an extension of Haskell, referred to as HaskellJoinRules. We compare HaskellJoinRules against programming languages based on the join calculus such as JoCaml and Polyphonic C# and demonstrate that HaskellJoinRules offers some highly useful concurrent programming patterns. We have built a prototype which can be used in combination with the Glasgow Haskell Compiler.

1 Introduction

There are numerous calculi and programming languages to support concurrent programming. A particular fruitful model appears to be the join calculus [4] which provides the basis for the concurrency abstractions found in JoCaml [11] and Polyphonic C# [2]. Here is a simple example in Polyphonic C#

```
public class Buffer {
  public async Put(string s);
  public string Get() & Put(string s) { return s; }
}
```

We implement a (communication) buffer via a join pattern (also known as synchronization pattern or chord) which declares a synchronous method `Get()`. The body of the join pattern will return `s` if both the `Get()` and `Put(s)` methods have been called. Method `Put(s)` is asynchronous, that is, the method call will not block and immediately returns to the caller. On the other hand, a call to `Get()` will block and only returns to the caller if we find a partner `Put(s)`.

The advantage of join patterns is that they provide for a more declarative and higher-level model of concurrency compared to the traditional model based on threads and locks. Interestingly, a similar model of concurrency has been developed independently in the context of logic and constraint programming.

Constraint Handling Rules (CHR) [5, 6] is a concurrent committed-choice constraint logic programming language to exhaustively transform (rewrite) multi-sets of (asynchronous) constraints into simpler ones. In CHR, we can write the

above join pattern as follows.

$$Get(x), Put(y) \Longleftrightarrow x = y$$

For example, the above CHR will rewrite $\{Get(x), Put(1)\}$ to $\{x = 1\}$. Initially, CHR were developed to specify incremental constraint solvers, CHR are now used in a multitude of other applications such as planning etc [7]. In our own work we have made extensive use of CHR to specify type inference systems[18, 17].

In this paper, we report some preliminary results on employing CHR for the coordination of concurrent computations where the underlying computations are carried out in Haskell [14]. From the join calculus, we adopt the concept of synchronized method calls. We refer to the resulting system as HaskellJoinRules. We choose Haskell as our host language because Haskell is well-suited to develop domain-specific language extensions. Furthermore, we can easily take advantage of our concurrent implementation of CHR in Haskell which we have built using the Glasgow Haskell Compiler (GHC) [8].

Here is a recast of the above Polyphonic C# example in HaskellJoinRules:

```
Chord Get(x) & Put(y) = x `is` y
```

We introduce chords where the left-hand effectively corresponds to the left-hand side of a CHR. The right-hand side, which will be executed if we find a match of the left-hand, yields a join computation. The interfaces are

```
Get :: Chan a -> UCons
Put :: Chan a -> UCons
is :: Chan a -> Chan a -> Join ()
```

Constraint symbols are treated like (data) constructors and their arguments are channels. The join computation `x `is` y` instantiates `x` by the value of `y`. This computation is only successful if `x` has not yet been instantiated. It is straightforward to compile chords to CHR rules.

To trigger the above chord, we provide interfaces to manipulate the CHR store (i.e. multi-set of CHR constraints).

```
chan :: a -> Chan a                    -- Lift value into channel.
newChan :: Join (Chan a)               -- Create a new empty channel.
addToCHR :: UCons -> Join ConsId       -- Add Constraint to store.
waitUntilDeleted :: ConsId -> Join () -- Wait until constraint deleted.
```

`chan` simply lifts a value into a channel type, `newChan` creates a new channel. `addToCHR` adds a constraint to the CHR store and returns a constraint id. `waitUntilDeleted` checks whether the constraint has been removed from the store and will block otherwise.

Thus, we can implement a asynchronous put operation and synchronous get operation.

```
get :: Join a
get = do { x <- newChan
         ; cid <- addToCHR (Get(x))
         ; waitUntilDeleted cid }
```

```
put :: a -> Join ()
put x = do { addToCHR (Put(chan x))
           ; return () }
```

Notice how we turn method calls into constraints.

Join style patterns in the context of Haskell have been explored earlier [16]. To the best of our knowledge, we are the first to introduce the idea of compiling join patterns into CHR. Most importantly, CHR has additional features such as guards (Section 3.2) and propagation (Section 3.3) which add functionality and programming patterns not available in standard join pattern implementations.

In summary, we make the following contributions:

- We review CHR and draw a comparison to the join calculus (Section 2).
- We consider a series of examples to demonstrate HaskellJoinRules. We show how CHR features (guards and propagation) add useful functionalities to join patterns. We also draw comparisons to JoCaml and Polyphonic C# (Section 3).
- We give an overview of the current implementation (Section 4).

We conclude in Section 5 where we also discuss some related work. We assume that the reader is familiar with Haskell and has a basic understanding of the join calculus.

2 Constraint Handling Rules

We first review the basics of CHR and then draw a comparison to the join calculus.

2.1 Syntax and Semantics

A CHR *simpagation* rule r is of the form

$$r \ @ \ H_1 \backslash H_2 \iff g \mid C$$

where rule *heads* H_1 and H_2 consist of CHR constraints, the *guard* g is a built-in constraint from some built-in theory $\mathcal{CT}$ (for example Herbrand) and the rule *body* C consists of CHR and built-in constraints. In case H_2 is empty we call the CHR a *propagation* rule and in case H_1 is empty we call the CHR a *simplification* rule. The idea is that H_1 will be removed from the store and replaced by C whereas H_2 will be propagated. That is, remains in the store.

A set of CHR operates on a constraint store by exhaustively applying each CHR until no further CHR is applicable. A constraint *store* is a multi-set of CHR and built-in constraints. To distinguish between CHR and built-in constraints, we denote the store as the pair $\langle S, B\rangle$, abbreviated by Str, where S are the CHR constraints and B the built-in constraints. We write $\uplus$ to denote multi-set union and $\wedge$ to denote conjunction among built-in constraints. Depending on the context we treat substitutions θ, ϕ as conjunctions of equations.

We can apply the above CHR on a store $\langle S, B\rangle$ if we find matching copies H_1' and H_2' in S, that is, $S \equiv S' \uplus H_1' \uplus H_2'$, such that $\phi(H_1) \equiv H_1'$, $\phi(H_2) \equiv H_2'$ and $\mathcal{CT} \models B \supset \phi(g)$ for some substitution ϕ. We assume that variables in the CHR are fresh. We write $\equiv$ to denote syntactic equivalence and $\supset$ to denote

Communication channel:

$$get \ @ \ Get(x), Put(y) \Longleftrightarrow x = y$$

$$\frac{\langle \{Get(m), Put(1)\}, true\rangle \rightarrowtail_{get} \langle \emptyset, m = 1\rangle \parallel \langle \{Get(n), Put(8)\}, true\rangle \rightarrowtail_{get} \langle \emptyset, n = 8\rangle}{\langle \{Get(m), Put(1), Get(n), Put(8)\}, true\rangle \rightarrowtail^* \langle \emptyset, m = 1 \wedge n = 8\rangle}$$

Greatest common divisor:

$$gcd1 \ @ \ Gcd(0) \Longleftrightarrow true$$
$$gcd2 \ @ \ Gcd(n) \backslash Gcd(m) \Longleftrightarrow m >= n \& \& n > 0 \mid Gcd(m - n)$$

$$\frac{\begin{array}{c}\{Gcd(3), Gcd(9)\} \rightarrowtail_{gcd2} \{Gcd(3), Gcd(6)\} \\ \rightarrowtail_{gcd2} \{Gcd(3), Gcd(3)\} \rightarrowtail_{gcd2} \{Gcd(3), Gcd(0)\} \parallel \{Gcd(4), Gcd(8)\} \rightarrowtail^* \{Gcd(4)\} \\ \rightarrowtail_{gcd1} \{Gcd(3)\}\end{array}}{\{Gcd(3), Gcd(9), Gcd(4), Gcd(8)\} \rightarrowtail \ldots \rightarrowtail \{Gcd(3), Gcd(4)\} \rightarrowtail^* \{Gcd(1)\}}$$

Fig. 1. CHR Examples

Boolean implication. Then, we can transform S to $S' \uplus H_1' \uplus \phi(C')$, where C' is the set of all CHR constraints in C, and B to $B \wedge \phi \wedge C''$ where C'' are all the built-in constraints in C. We refer to this as a CHR derivation step, written $\langle S, B\rangle \rightarrowtail_r \langle S' \uplus H_1' \uplus \phi(C'), B \wedge \phi \wedge C''\rangle$. Notice that H_2' is simplified by the instantiated rule body $\phi(C')$ whereas H_1' is propagated, that is, remains in the resulting constraint store. Notice that in contrast to Prolog we never instantiate variables in the store to fire a CHR.

Let P be a finite set of CHR. Then, we write $Str \rightarrow_P^* Str'$ for the exhaustive application of P on some initial store Str yielding in a finite number of CHR derivation steps the final store Str'. We often drop P and r and write $Str \rightarrowtail Str'$ and $Str \rightarrowtail^* Str'$ as shorthand for $Str \rightarrowtail_r Str'$ and $Str \rightarrowtail_P^* Str'$ if its clear which CHRs are involved in the derivation step.

Figure 1 contains two examples. The first CHR program represents the communication channel which we have seen earlier. The $Get(x)$ constraint represents the action of writing a value from the communication channel into the variable x, while the $Put(y)$ constraint represents the action of putting the value y into the channel. In contrast to the standard CHR syntax which follows Prolog syntax, we follow Haskell syntax where variables start with a lower-case letter. CHR satisfy a monotonicity property. Hence, the two sample derivations can be executed concurrently, indicated by the symbol $\parallel$, and we can straightforwardly combine both derivations as long as they do not interfere.

The second CHR program computes the greatest common divisor among a set of numbers. For illustration purposes, we write out some of the intermediate steps for the left derivation. We also simplify the store because there are no built-in constraints for this example.

2.2 Comparing CHR and Join Calculus

In CHR, conjunction of constraints can be regarded as interacting collections of multiple asynchronous agents or processes. That is, a CHR constraint will never

```
data UCons                                -- User CHR Constraint
data ConsId                               -- Constraint Identifier
addToCHR :: UCons -> Join ConsId          -- Add a constraint to CHR solver
waitUntilDeleted :: ConsId -> Join ()     -- Block until constraint is deleted

data Chan a                               -- Communication Channel
newChan :: Join (Chan a)                  -- Create a new empty channel
chan :: a -> Chan a                       -- 'Lift' an item into a channel
readChan :: Chan a -> Join a              -- Read and block until channel is full
is :: Chan a -> Chan a -> Join ()         -- Assign a value to a channel, block if
                                          -- source is empty.

async :: UCons -> Join () -- an asynchronous call
async uc = do { addToCHR uc
              ; return () } -- just proceed

sync :: UCons -> Join () -- a synchronous call
sync uc = do { cid <- addToCHR uc
             ; waitUntilDeleted cid } -- block until cid is deleted
```

Fig. 2. HaskellJoinRules Interfaces

block whereas a synchronized method call in Polyphonic C# such as `Get()` will block if there is no partner `Put(s)`.

In Join, a join pattern effectively resembles a CHR simplification rule. Additionally, CHR may have guard and propagation heads. For example, see the above CHR *gcd2*.

We conclude that CHR and the join calculus have quite a bit in common and can gain from each other. For instance, CHR can benefit from synchronized constraints and the join calculus can benefit from guarded join patterns which contain propagated parts. We integrate these concepts into an extension of Haskell, referred to as HaskellJoinRules.

3 HaskellJoinRules

In this section, we give an informal overview of HaskellJoinRules and its main features. Implementation details, such as how to compile HaskellJoinRules chords to CHR and how to implement synchronized constraints efficiently are discussed in the next section.

The main feature of HaskellJoinRules is the specification of chords such as the one we saw earlier in the introduction.

```
Chord Get(x) & Put(y) = x `is` y
```

In general, a chord in HaskellJoinRules can be of the form

```
Chord P1 & ... & Pi \ S1 & ... & Sj | g = body
```

where each `Pi` refers to a propagated constraint and each `Sj` refers to a simplified constraint. A n-ary constraint `U` comes with the interface

```
U :: (Chan a1,...,Chan an) -> UCons
```

The guard `g` must be of type `Join Bool` whereas the chord `body` must be of type `Join ()`. Like CHR, a chord fires if we find a match for the (chord) head and the guard yields true. Then, we execute the body. Figure 2 summarizes the HaskellJoinRules interfaces which allow the programmer to communicate with the underlying CHR solver. Most are straight-forward except for the blocking interfaces which we will discuss in detail in Section 4.2.

Next, we consider a series of examples to explore the various features of HaskellJoinRules.

3.1 HaskellJoinRules Basics : Standard Buffer

Recall in Section 1 we introduce the buffer example in Polyphonic C# The buffer example is implemented by the following chord and constraint interfaces:

```
Chord Get(x) & Put(y) = x `is` y

Get(x) :: Chan a -> UCons
Put(y) :: Chan a -> UCons
```

Yet the above definition does not specify what type of constraints (synchronous or asynchronous) `Get` and `Put` are meant to be. We do this by defining the `get` and `put` monadic operations (method calls) of the `Join` monad.

```
put :: a -> Join ()
put y = async (Put(chan y))
```

```
get :: Join a
get = do { x <- newChan
         ; sync (Get(x))
         ; readChan x }
```

The `put` operation is meant to be asynchronous, hence it simply involves introducing a `Put` constraint into the CHR solver. The synchronous `get` operation is more complex: a `Get` constraint is introduced and the operation blocks until it is removed from the constraint store. A local channel is also created to observe the results of the `Get` constraint.

It may seem strange that we define our method calls separately from chords. In section 3.4 we shall illustrate the advantages of having the flexibility of such explicit synchronization.

3.2 Exploiting Guards: Conditional Buffer

In this section, we highlight a simple example which exploits guard conditions. The conditional buffer example is an extension of the buffer example: `condGet` method calls consists of an addition higher order filter Join operation so we can select only items that passes this filter. The conditional buffer HaskellJoinRules program is illustrated by the following:

```
Chord CondGet(f,x) & Put(y) | f y = x `is` y

CondGet :: (Chan (a -> Join Bool),Chan a) -> UCons
```

```
condGet :: (a -> Join Bool) -> Join ()
condGet f = do { x  <- newChan
               ; sync (CondGet(chan f,x))
               ; readChan x }
```

Note we omit details of the `put` asynchronous call, since it is the same as the previous example. Guard conditions may seem to be a trivial extension, however that is entirely not the case: activating the above chord involves a search for matching pairs of `CondGet(f,x)` and `Put(y)` constraints. For instance, a call to `CondGet(prime,x)` where `prime` filters away non-prime numbers, involves searching through the collection of `Put` constraints for one with a prime number, rather than simply dequeue the first.

In general this search procedure is an expensive task, but efficient optimization techniques in the context of CHR have already been widely and still actively studied. [10, 3, 15]. Execution of Standard join pattern (JoCaml and Polyphonic C#) however does not include such a built-in search mechanism (call instances are simply dequeued), hence the programmer is left on her/his own to explicitly program such search mechanism. For instance, the prime integer instance of the conditional buffer example can be implemented in polyphonic C#:

```
public class Buffer2 {
  public async Put(int n);
  public int PrimeGet() & Put(int n) {
     if(IsPrime(n)) return n;
     else { Put(n) ; PrimeGet(); }
  }
}
```

We assume that `IsPrime(n)` is simple procedure that tests if `n` is a prime number. Note that the 'search' is implemented by replacing a `Put` asynchronous call if it is not a prime number and calling `PrimeGet` again. This however is terribly inefficient: If there are at least one `Put` call, but none with a prime number argument, a call to `PrimeGet` blocks by executing a non-terminating cycle of `Put` and `PrimeGet` calls until a prime integer is introduced.

3.3 Exploiting Propagation: Authorized Buffer

Propagation adds the convenience of defining chords with method calls that act as 'catalysts': These method calls are necessary for activating the chord but are not consumed upon successful activation. The authorized buffer is an extension of the buffer example in which `get` calls (in this case `idGet`) will only be successful if they are called by authorized processes. The following highlights a simple implementation of the authorized buffer:

```
Chord Authorize(id) \ IdGet(id,x) & Put(y) = x `is` y

Authorize :: Chan String -> UCons
IdGet     :: (Chan String,Chan a) -> UCons
```

```
authorize :: String -> Join ()
authorize id = async (Authorize(chan id))

idGet :: String -> Join a
idGet id = do { x <- newChan
              ; sync (IdGet(chan id,x))
              ; readChan x }
```

Note that this chord contains a non-linear pattern (`id` appears in two locations), but we can always compile this into a guard (eg. `id1==id2`). Propagation does not offer greater expressiveness, but it provides not only convenience, most importantly it provides better a concurrency behavior. To illustrate this, we consider an implementation of Authorized buffers in Polyphonic C#:

```
public int IdGet(string id1) & Put(int y) & Authorize(string id2) {
    Authorize(id2);
    if(id1==id2) return y;
    else { Put(y); IdGet(id1); }
}
```

The `Authorize(id2)` asynchronous calls are consumed and reintroduced, causing unnecessary writes into the shared memory and inevitably interleaving all `IdGet` calls using the same `Authorize` call. With propagation however, `Authorize` calls are never removed but simply checked for presence in the store, hence multiple `IdGet` calls with the same id can share the same `Authorize` call.

3.4 Exploiting Customized Synchronization: N-way Synchronization

In this section, we demonstrate how we can customize synchronization protocols. We consider the N-way synchronization problem. The following illustrates a naive implementation of n-way in HaskellJoinRules which is meant for `n` less than 4.

```
-- n way for n < 4
Chord Accept(1) & Entry() = return ()
Chord Accept(2) & Entry() & Entry() = return ()
Chord Accept(3) & Entry() & Entry() & Entry() = return ()

Accept :: Chan Int -> UCons            Entry :: UCons

accept :: Int -> Join ()               entry :: Join ()
accept n = sync (Accept(chan n))       entry = sync (Entry())
```

A synchronous call to `accept n` will block until `n` `Entry` calls are found, during which all `n+1` synchronous calls will unblock together. Consider the following attempt to generalized n-way synchronization:

```
-- Liberal version for generalized n way
Chord Accept() & Entry() = return ()

Accept :: UCons                        Entry :: UCons
```

```
accept :: Int -> Join ()
accept 0 = return ()
accept n = do { sync (Accept())
              ; accept (n-1) }

entry :: Join ()
entry = async (Entry())
```

Note that this version of n-way synchronization does not work like the previous one. Each `entry` call is unblocked immediately once paired with a `accept` call and does not need to wait for the other `n-1 entry` calls. Interestingly, this is the more liberal interpretation of n-way synchronization adopted by [1]. Fortunately, in HaskellJoinRules 'strict' n-way synchronization is just a small step away:

```
-- Generalized n way
Chord Accept(ch) & Entry(ch') = ch' is (chan ch)

Accept :: Chan () -> UCons
Entry  :: Chan (Chan ()) -> UCons

accept :: Int -> Join ()
accept n = do { ch <- newChan   -- create synchronization channel
              ; mapM (\_ -> sync (Accept(ch))) [1..n]
              ; ch `is` () }    -- n-way complete, let's go!

entry :: Join ()
entry = do { ch' <- newChan
           ; sync (Entry(ch'))
           ; ch <- readChan ch' -- block and wait for Accept
           ; readChan ch }      -- block and wait for n-way complete
```

This implements n-way synchronization by using a single channel to synchronize with all `n entry` calls. This synchronizing channel is created by an `accept` call and distributed to `entry` calls via the chord show above. Upon receiving this synchronization channel, each `entry` call blocks until the n-way synchronization is completed. `accept` completes the n-way after finding n `entry` calls, and writes the synchronization channel with a dummy value. (unit '()' in this case).

Programmers are also free to define their own variants of synchronous constraints, for instance:

```
-- Append a given Channel as last argument of constraint
appendChan :: UCons -> Chan a -> UCons

mysync :: UCons -> Join a
mysync uc = do { x <- newChan
               ; addToCHR (appendChan uc [x])
               ; readChan x }
```

The `mysync` synchronous operation blocks until the last argument of a constraint is instantiated. This variant of synchronous calls is especially useful for modeling concurrent programs like n-way synchronization.

3.5 Santa Claus Example

Originally highlighted in [19], the Santa claus problem is an interesting challenge for concurrent programming. In this section, we briefly illustrate a solution in HaskellJoinRules . The santa claus problem is summarized by the following:

Santa sleeps until wakened by either all of his nine reindeer, or by a group of three of his ten elves. If awakened by the reindeer, he delivers toys with all of them. If awakened by a group of elves, he consults with them on toy R&D. At the end of either task, all helpers (elves or reindeer) are released from duties. Santa should give priority to the reindeer in the case that there is both a group of elves and a group of reindeer waiting.

We first define the chord and join operations that models the elf waiting routine:

```
-- ElfWait CHR chord and synchronous call.
Chord ElfWait(ch1) & ElfWait(ch2) & ElfWait(ch3) = elfGroup [ch1,ch2,ch3]

ElfGroup :: Chan [Chan ()] -> UCons
ElfWait  :: Chan () -> UCons

elfGroup :: [Chan ()] -> Join ()
elfGroup chs = async (ElfGroup(chan chs))

elfWait :: Join ()
elfWait = do { x <- newChan
             ; sync (ElfWait(x))
             ; readChan x }
```

This program essentially synchronizes 3 waiting elves and creates an elf group by making a asynchronous call `elfGroup`. We do the same for the 9 reindeers, only difference is we assume the presence of a asynchronous call `NoReindeersGroup` that will be removed once a reindeer group is successfully formed.

```
-- ReindeerWait CHR chord and synchronous call.
Chord ReindeerWait(ch1) & ReindeerWait(ch2) & ReindeerWait(ch3) &
      ReindeerWait(ch4) & ReindeerWait(ch5) & ReindeerWait(ch6) &
      ReindeerWait(ch7) & ReindeerWait(ch8) & ReindeerWait(ch9) &
      NoReindeerGroup() = reindeerGroup [ch1,ch2,ch3,ch4,ch5,ch6,ch7,ch8,ch9]

ReindeerGroup :: Chan [Chan ()] -> UCons
NoReindeerGroup :: UCons
ReindeerWait :: Chan () -> UCons

reindeerGroup :: [Chan ()] -> Join ()
reindeerGroup chs = async (ReindeerGroup(chan chs))

noReindeerGroup :: Join ()
noReindeerGroup = async (NoReindeerGroup())

reindeerWait :: Join ()
reindeerWait = do { x <- newChan
```

```
                ; sync (ReindeerWait(x))
                ; readChan x }
```

Next, we introduce a variant of n-way synchronization that provides 2 partitions of synchronization (`RoomIn` and `RoomOut`). This is what santa is going to be using to control the movement of his helpers.

```
-- N-Way Synchronization
data Partition = RoomIn | RoomOut

Chord Accept(s,ch) & Entry(s,ch') = ch' `is` (chan ch)

Entry  :: (Chan Partition,Chan (Chan ())) -> UCons
Accept :: (Chan Partition,Chan ()) -> UCons

entry :: Partition -> Join ()
entry s = do { ch' <- newChan
             ; sync (Entry(chan s,ch'))
             ; ch <- readChan ch'
             ; readChan ch }

accept :: Partition -> Int -> Join ()
accept s n = do { x <- newChan
                ; mapM (\_ -> sync (Accept(chan s,ch))) [1..n]
                ; ch `is` () }
```

Now we can implement santa by defining the chords which allows him to synchronize with reindeer/elf groups:

```
-- SantaWait CHR chord and synchronous call.
Chord SantaWait(j) & ReindeerGroup(gch) = do { chs <- readChan gch
                                             ; mapM (\ch -> ch `is` ()) chs
                                             ; noReindeerGroup
                                             ; j `is` "Reindeer" }

Chord NoReindeerGroup() \ SantaWait(j)
                  & ElfGroup(gch) = do { chs <- readChan gch
                                       ; mapM (\ch -> ch `is` ()) chs
                                       ; j `is` "Elf" }

SantaWait :: Chan String -> UCons

santaWait :: Join String
santaWait = do { j <- newChan
               ; sync (SantaWait(j))
               ; readChan j }
```

We enforce priority by making elf group synchronization explicitly requiring `NoReindeerGroup` as a catalyse. In section 4 we will discuss some issues of rule priorities. Upon a successful synchronization with a group (reindeers or elves), santa will 'wake' each helper in the group via instantiating their synchronization channels. The final piece of the puzzle is the top level routines of santa and his helpers:

```
-- Top Level Santa / Helper Routines --
elfRoutine :: Join ()
elfRoutine = do { elfWait
                ; entry RoomIn
                  -- consult santa
                ; entry RoomOut }

reindeerRoutine :: Join ()
reindeerRoutine = do { reindeerWait
                     ; entry RoomIn
                       -- deliver toys
                     ; entry RoomOut }

santaRoutine :: Join ()
santaRoutine = do { grp <- santaWait
                  ; case grp of
                     "Elf"      -> do { accept RoomIn 3
                                        -- consult elves
                                      ; accept RoomOut 3 }
                     "Reindeer" -> do { accept RoomIn 9
                                        -- deliver toys
                                      ; accept RoomOut 9 }
                  }
```

We assume that each of these routines are repeated indefinitely by execution threads, represent the 10 elves, 9 reindeers and santa himself. Elves and reindeers simply run their respective wait operations and blocks until santa has choosen them. They are re-synchronized again by the `entry RoomIn` and finally waits until santa has released them (`entry RoomOut`). Santa, on the other hand, calls his own waiting operation, which returns the type of his helper once synchronization is successful. Depending on the helper type, he will usher in and on the respective number of helpers via the `accept` operations, while intermediately executing the required task.

4 Implementation Highlights

This section highlights some of the important aspects of the implementation of HaskellJoinRules. The underlying CHR solver is our very own concurrent CHR implementation [12] in Haskell with Software Transactional Memory [9].

4.1 Compiling and Executing HaskellJoinRules Chords

Recall the general form of HaskellJoinRules chords:

```
Chord P1 & ... & Pi \ S1 & ... & Sj | G = body
```

We use a straight forward scheme to compile the above into CHR rules. Basically, heads of the chord are mapped as CHR rule heads and guard condition of the chord is the CHR rule guard. The chord body, which is a join `computation` is compiled as a CHR body. This is a minor extension to the CHR semantics:

Rather than just a collection of constraints, rule bodies are actual join computations that are executed by the CHR solver upon triggering of the rule. Hence we have the following CHR rule:

$$P1\ ,\ \ldots\ ,\ Pi \setminus S1\ ,\ \ldots\ ,\ Sj \Longleftrightarrow G\ \mid\ body$$

Compiling our chords into CHR rules is highly beneficial: we now have a well established incremental search technique for activating HaskellJoinRules chords. Hence, unique CHR features like propagation and guards are available to our disposal. Since the body computation can potentially block as it may contain synchronous join operations, a new execution thread which we shall refer to as the surrogate thread, is spawned by the CHR solver to execute this computation.

In polyphonic C#, chord bodies are executed by the program thread that executed the synchronous method of the chord, rather than using surrogate threads. This means that the original thread will block if a sub-operation of the chord body blocks. In HaskellJoinRules we provide the flexibility for the programmer to decide what happens, for instance consider the following variant of buffers:

```
Chord Get(x) & Put(y) = do { incGetCount -- increment 'Get' count
                             -- do some other admin operations ...
                           ; x `is` y }

Get :: Chan a -> UCons            Put :: Chan a -> UCons

get :: Join a                     put :: a -> Join ()
get = do { x <- newChan           put y = async (Put(chan y))
         ; sync (Get(x))
         ; readChan x }
```

The chord of this buffer variant contains a body that increments a global counter and possibly do other adminstrative operation before executing `x `is` y` which unblocks the `get` call. Polyphonic C# we are force to write this buffer variant similar to the above way, hence executing these admin operations before unblocking the `get` call. In HaskellJoinRules however, we can redefine this chord in a unique and intuitive way:

```
Chord Get(x) & Put(y) = do { x `is` y
                           ; incGetCount
                             -- do some other admin operations ... }
```

The `get` and `put` operations remain the same while we change the chord body to execute `x `is` y` first. This immediately unblocks the thread which has called `get` while the surrogate thread executes the adminstrative operation.

4.2 Implementing Efficient Blocking Mechanism

In this section, we review the implementation of the blocking interfaces in figure 2. Similar to our concurrent CHR implementation, we use Haskell software

```
data TVar a                           -- Transactional Variable
newTVar :: STM (TVar a)               -- Create a new TVar
readTVar :: TVar a -> STM a           -- Read value of a TVar
writeTVar :: TVar a -> a -> STM ()    -- Write value into a TVar
retry :: STM ()                       -- Block and retry STM operation
```

Fig. 3. STM Transactional Memory Interfaces

transactional memory concurrency primitives. Figure 3 shows the interfaces of STM transactional memory.

A transactional variable (`TVar`) is a shared mutable memory location. It can be created, read and written via `newTVar`, `readTVar` and `writeTVar`. The `retry` operation is an explicit blocking procedure that can be called when certain user defined conditions are not favorable for continued execution. For instance, the following procedure blocks until the given boolean transactional variable contains `False`

```
blockUntilFalse :: TVar Bool -> STM ()
blockUntilFalse bt = do { b <- readTVar bt
                        ; if b then retry
                               else return () }
```

GHC's implementation of the retry procedure ensures that the operation is actually rescheduled only if certain transactional variables have been changed. This gives us the means of implementing efficient blocking join pattern interfaces almost for free. In the above case, `blockUntilFalse` is only ever rescheduled if `bt` has been changed. The following reviews the implementation of the data types `ConsId` and `Chan` seen earlier in Figure 2.

```
data ConsId = ConsId (TVar Bool)    -- A Constraint identifier
data Chan a = Chan (TVar (Maybe a)) -- A Channel (constraint argument)

doSTM :: STM a -> Join a -- Do an STM computation from Join computation

waitUntilDeleted :: ConsId -> Join ()
waitUntilDeleted (ConsId b) = doSTM (blockUntilFalse b)

readChan :: Chan a -> Join a
readChan (Chan t) = doSTM (do { mb <- readTVar t
                              ; case mb of
                                  Just a  -> return a -- Not empty, return
                                  Nothing -> retry }) -- Empty, block

is :: Chan a -> Chan a -> Join ()
is c1 c2 = do { v2 <- readChan c2               -- Blocks if c2 is empty
              ; doSTM (writeTVar c1 (Just v2)) }
```

A constraint identifier is nothing more than a shared boolean flag uniquely assigned to each stored constraint while our CHR solver is programmed to set this variable to False once the constraint is deleted. Thus, the `waitUntilDeleted` operation simply waits until the flag has been set to False by running `blockUntilFalse`

on the flag. Channels (`Chan`) are also transactional variables, which can be empty (containing `Nothing`) or contain some value v (`Just v`). `readChan` uses the same STM retry trick, while `is` inherits the blocking condition from `readChan`.

4.3 Implementing Rule Priorities

Recall in Section3.5 we enforce rule priorities by explicitly modeling the absence of reindeer groups (`NoReindeerGroup`). This essence of technique is illustrated by the following:

```
Chord A() & A() & NoAPair = makeAPair -- Chord 1: Add an APair
Chord B() & B() = makeBPair           -- Chord 2: Add a BPair

Chord APair & C() = do {..}           -- Chord 3: High Priority
Chord NoAPair \ BPair & C() = do {..} -- Chord 4: Low Priority
```

Chords 1 and 2 simply creates A and B pairs. Note that we model the absence of A pairs via `NoAPair`. Priority is enforced by specifying that chord 4 require `NoAPair` to activate. However, this is a flawed solution: consider the program state {`NoAPair, A, A, B, B`} we have no control over which chord (1 or 2) is schedule first, thus it is possible for chord 2 to activate and be immediately followed by chord 4, hence breaking the priority.

Fortunately, we can write a better solution in HaskellJoinRules. Recall that chord guard conditions can be any boolean join computations (`Join Bool`). We can rewrite the above program as:

```
notExists :: [UCons] -> Join Bool

Chord A() & A() & C() = do {..}
Chord B() & B() & C() | (notExists [A(),A()]) = do {..}
```

`notExists` is a join computation that explicitly check if the constraint store contains a given set of constraints. With this primitive, we can specify the negation of chord heads in guard conditions of chords, hence modeling priority in a more direct manner.

5 Conclusion

We have provided an overview of our proposed language extension to Haskell, HaskellJoinRules. In HaskellJoinRules, CHRs are employed as join pattern style concurrency coordination where underlying computations are executed in Haskell. From join calculus, we adopt the concept of synchronized method calls.

There is lots of related work, some which we will briefly discuss below. Extending Haskell with join style patterns have been explored earlier in [16], which introduces Join patterns to Haskell as a higher order combinator library. In [13], an extension of JoCaml with pattern matching is introduced. This extension allows ML style patterns in chord heads to be compiled into standard join patterns. Yet, in HaskellJoinRules the marriage of rule based constraint programming and process calculi has resulted in a concurrent abstraction with unique features: Propagation (Section 3.2), user defined guards (Section 3.3) and customizable synchronization patterns (Section 3.4).

References

1. N. Benton. Jingle bells: Solving the santa claus problem in polyphonic c#. Technical report, Microsoft Research, 2003.
2. N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.
3. G. J. Duck. *Compilation of Constraint Handling Rules*. PhD thesis, The University of Melbourne, 2005.
4. C. Fournet and G. Gonthier. The join calculus: A language for distributed mobile programming. In *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, pages 268–332. Springer-Verlag, 2002.
5. T. Frühwirth. Constraint handling rules. In *Constraint Programming: Basics and Trends*, LNCS. Springer-Verlag, 1995.
6. T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138, 1998.
7. T. Frühwirth. Constraint handling rules: the story so far. In *Proc. of PPDP '06*, pages 13–14. ACM Press, 2006.
8. Glasgow haskell compiler home page. http://www.haskell.org/ghc/.
9. T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *Proc. of PPoPP'05*, pages 48–60. ACM Press, 2005.
10. C. Holzbaur, M. J. García de la Banda, P. J. Stuckey, and G. J. Duck. Optimizing compilation of Constraint Handling Rules in HAL. *TPLP*, 5(4-5):503–531, 2005.
11. Jocaml. http://jocaml.inria.fr/.
12. E. S. L. Lam and M. Sulzmann. A concurrent Constraint Handling Rules implementation in Haskell with software transactional memory. In *Proc. of ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming (DAMP'07)*, pages 19–24, 2007.
13. Q. Ma and L. Maranget. Compiling pattern matching in join-patterns. In *CONCUR*, pages 417–431, 2004.
14. S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
15. T. Schrijvers. Analyses, optimizations and extensions of Constraint Handling Rules: Ph.D. summary. In *Proc. of ICLP'05*, volume 3668 of *LNCS*, pages 435–436. Springer-Verlag, 2005.
16. S. Singh. Higher-order combinators for join patterns using stm, 2006. Proc. of TRANSACT'06: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing.
17. P. J. Stuckey and M. Sulzmann. A theory of overloading. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1–54, 2005.
18. M. Sulzmann, G. J. Duck, S. Peyton Jones, and P. J. Stuckey. Understanding functional dependencies via constraint handling rules. *J. Funct. Program.*, 17(1):83–129, 2007.
19. J. A. Trono. A new exercise in concurrency. *SIGCSE Bulletin*, 26(3):8–10, 1994.

Splitting and Merging Program Refactorings

Christopher Brown[1] and Simon Thompson[2]

[1] University of Kent, Canterbury, Kent, UK.
cmb21@kent.ac.uk
[2] University of Kent, Canterbury, Kent, UK.
S.J.Thompson@kent.ac.uk

Abstract. Program slicing is a well understood concept in the imperative paradigm, but so far there has been little work on program slicing in the context of functional languages. This paper describes a program slicing technique for Haskell that takes tuple returning functions apart (called splitting); the converse of this is also described (called merging). The slicer is implemented as a transformation for the Haskell Refactorer, HaRe. Splitting functions is a useful transformation to allow the programmer to extract a particular subset of the functionality of a tuple returning function into a new definition. Merging is a useful transformation because it allows many definitions to be merged together, thus eliminating duplicate code and encouraging code reuse. Splitting and merging can help to reduce dead code and increase program productivity and can be also used for debugging purposes.

1 Introduction

Refactoring was first introduced by Opdyke in his PhD. thesis in 1992 [10]. Refactoring is the process of changing the internal structure and organization of a program, while preserving its semantics. The key aspect of refactoring —in contrast to general program transformations, such as genetic programming [4] — is the focus on purely structural changes rather than changes in program functionality. Refactoring also contrasts with other meaning-preserving transformations which emphasize a change in efficiency or other non-functional aspects. Refactoring is aimed at improving code quality, increasing programming productivity and increasing the ability for code to be reused. This functionality-preservation is crucial so that refactorings do not introduce, or remove, any bugs.

This paper is concerned with the investigation of a number of refactorings for Haskell. We start with dead code elimination and based upon that the process is extended to introduce a notion of function splitting and merging. As an example of merging and splitting, consider the following Haskell library functions, `take` and `drop`:

```
take :: Int -> [a] -> [a]
take 0 _ = []
take _ []= []
```

```
take n (x:xs)
  | n > 0 = x : take (n-1) xs
take _ _ = error "PreludeList.take: negative argument"

drop :: Int -> [a] -> [a]
drop 0 xs            = xs
drop _ []            = []
drop n (x:xs)
  | n>0  = drop (n-1) xs
drop _ _ = error "PreludeList.drop: negative argument"
```

As a concrete example of the usage of `take` and `drop`, consider:

```
> (take 10 "hello world", drop 10 "hello world")
> ("hello worl","d")
```

A merge refactoring allows the creation of a function which provides both results using only one list traversal rather than one traversal for each of `take` and `drop`. In the following example, `splitAt` is the result of merging `take` and `drop`:

```
splitAt 0 xs = ([],xs)
splitAt _ [] = ([],[])
splitAt n (x:xs)
  | n > 0 = (x:ys,zs)
    where
    (ys,zs) = splitAt (n-1) xs
splitAt _ _ = (error "PreludeList.take: negative argument",
               error "PreludeList.drop: negative argument")
```

The following is an example of the usage of `splitAt`:

```
> splitAt 10 "hello world"
> ("hello worl","d")
```

Splitting is the converse of merging, for example, the extraction of the definitions of `take` and `drop` from `splitAt`. Although `splitAt` is a predefined Haskell function, it serves as a useful example to illustrate splitting and merging. The process of these refactorings is described in section 3.

The refactorings are implemented in the Haskell Refactorer, HaRe. HaRe is the result of the combined effort of the *Refactoring Functional Programs* project at the University of Kent [7]. HaRe provides refactorings for the full Haskell 98 standard, and is integrated with the two most popular development environments for Haskell programs: Vim and (X)Emacs. HaRe refactorings can be applied to both single- and multi- module programs; HaRe is itself implemented in Haskell, and is built upon the Programatica [11] compiler front-end, and the Strafunski [6] library for generic tree traversal. The HaRe programmers' API provides the user with an abstract syntax tree (AST) together with utility functions (for example, tree traversal and tree transforming functions) to assist with the implementation of refactorings.

Pure functional programs are referentially transparent [14], therefore the opportunities for refactoring are much greater than for imperative programs. The classical example of this, of course, is that in a functional language it is always possible to transform `f x + g x` into `g x + f x` (assuming, of course, that `+` is a binary commutative operator). This is not possible for an imperative language because either `f` or `g` may change the value of the parameter `x` and therefore the result will depend on execution order. This is not, of course, the case in a functional language.

The following sections first introduce the idea of eliminating code within a function that is not needed; this has two flavours: unused code elimination and irrelevant code elimination. A notion of program slicing in Haskell is then outlined. A backwards, static program slicing technique that takes apart tuple returning functions is defined. Following this the converse of the splitting operation, namely merging is also defined. The paper concludes by looking at possible future developments in program slicing for Haskell.

2 Code Elimination

Dead code elimination [3] is a compiler optimization used to reduce a program size by removing the parts of the program which are not needed. In Haskell dead code contains code that is unreachable by evaluating the `main` function.

This section introduces two flavours of code elimination: dead code elimination and irrelevant code elimination. Dead code elimination is concerned with taking a particular top-level function of interest, and removing any nested declarations within that function that are not needed. Irrelevant code elimination is a generalization of the former. Irrelevant code elimination focuses upon removing nested declarations that are not needed to compute a particular sub-expression within the top-level declaration. Irrelevant code elimination is used to aid the programmer in debugging code.

2.1 Dead Code Elimination

Programmers in general tend to produce programs rapidly and badly [5]. Often, having written a program, the programmer will realise that a different approach would have been much better. Refactoring provides software support for modifying the original program into a better written program thus avoiding the expense of starting from scratch.

Programming rapidly and badly can cause lots of unnecessary declarations in the program; declarations that are never called and are hence "dead".

For example, the code below takes a list of type variables and produces a data structure containing the list of type variables:

```
createApp [var]
 = (Typ (HsTyVar (nameToTypePNT var)))
createApp (v:vars)
```

```
 = Typ (HsTyVar (nameToTypePNT (v ++
                                    (concatMap (" -> "++) vars))))
    where
     myConcat :: [ String ] -> String
     myConcat [] = []
     myConcat (x:xs) = (x ++ " -> ") ++ (myConcat xs)
```

The function `createApp` contains one declaration that is not needed: `myConcat` can be removed, as it is not needed to evaluate the result. Dead code can make a function look particularly messy and depending on the particular compiler used may actually consume memory and slow execution. Dead Code Elimination is a refactoring implemented in HaRe that searches the AST for the definition of a particular function (supplied as a location in the original code), removing any declarations that are not needed. The examples provided in this section present only a `where` clause; a similar technique is used to cope with lambda definitions and `let` clauses.

There are two stages to the refactoring. An analysis stage that collects all the information required to do the modification, and a modification stage that actually performs the modification on the AST with the information provided by the analysis stage. Currently the refactoring only removes dead code from within one function. However, it could easily be expanded to take a whole module of functions (or, indeed, a set of modules) into consideration.

The main algorithm is as follows:

1. The particular function clause in question is extracted from the AST.
2. The right-hand-side is traversed until the result is found.
3. A list of free identifiers (identifiers that are declared at another point in the program) are calculated from the expression. Identifiers that are declared outside of the scope of the function clause are removed from the list.
4. There are three main stages to the removal of dead code:
   (a) declarations that are used on the right-hand-side are then also traversed to determine free variables contained therein.
   (b) steps 3 - 4 are repeated for each declaration appearing in the list of free variables (this checks to see whether some declarations depend on other declarations in scope). Nested declarations are also considered.
   (c) the `where` clause is then traversed. Steps 3 - 4 is repeated for each declaration in the `where` clause. This takes into account nested `where` clauses.
5. For mutually referential declarations the algorithm traverses the AST for all declarations to ensure that all free variables in those declarations are retained.
6. Any declarations remaining are removed from the AST.

In order to use this tool, firstly the user selects a function from the editor window. The user then selects *dead code elimination* from the HaRe drop-down menu. To capture the entire namespace, the whole program is parsed into an AST and token stream. The AST is then traversed using Strafunski to find the particular function clause in question. This traversal is performed by using the

location information in the AST; it is possible to traverse into any functions corresponding to the selection region in the editor. Once the function is found the function's `where` clause and right-hand-side is retrieved. The function's right-hand-side is then traversed until the result is reached. For example the result of `createApp` is:

```
Typ (HsTyVar (nameToTypePNT (v ++ (concatMap (" -> "++) vars))))
```

Declarations declared in the scope of `createApp` must be analysed to check whether they are dead; these declarations can appear within `let` clauses and lamdba expressions within the right-hand-side of the function. The refactoring takes into consideration nested declarations; for example: `where`/`let` clauses. Once the returning subexpression is reached, the free and bound names within the subexpression are calculated. The list of free names is then used to remove those declarations residing on the right-hand-side that do not appear within the list of free names. For example:

```
f x =  z + res
       where
       res = f (x-1)
       res2 = f (x+1)
       y = x + 1
       z = 46
```

The result expression is `z + res` and the only free variables are `z` and `res`; therefore, `y` can be removed from the right-hand-side of `f` as it is not used within `z + res`.

Any mutually referential declarations must be taken care of by ensuring that all free variables in those declarations are retained. For example in the code below it can clearly be observed that the declaration `z` depends on the declaration `y`.

Once the right-hand-side has been modified to remove the declarations that are not used, the `where` clause of the function in question is then analysed. This time the free variables are calculated for each sub-expression within the modified right-hand-side; each member of the `where` clause that appears in the list of free variables is then analysed for its free variables. All the declarations in the `where` clause that are not needed by the right-hand-side of the function in question, and do not appear in the dependancy graph of any needed declaration, are removed:

```
f x = z + res
      where
      res = f (x-1)
      y = x + 1
      z = 46 + y
```

After the AST has been modified, the source code is also modified to mirror the changes of the refactoring.

A popular technique in abstract interpretation [2] is strictness analysis [8]. Dead code elimination is related to strictness analysis in that strictness analysis

searches for parts of a program that will always be used. Dead code elimination searches for parts of the program that will never be used. However, strictness analysis commonly works dynamically: inferring that the boolean conditional in an `if` expression is `False`, say, and therefore calculating that the consequence of the `if` is never evaluated. This paper takes a static approach to determining dead code.

2.2 Irrelevant Code Elimination

Irrelevant code elimination is useful for debugging purposes and to some extent is used in algorithmic debugging [13]. In algorithmic debugging the debugging tool asks the user a series of questions about whether a particular sub-expression in the code is generating the correct result or not. As the questions proceed, the particular parts of the program that the debugging tool is asking questions about becomes more clearly focused. It is often the case, however, that the programmer will have some intuition where the bug will lie within the code. Extracting a particular sub-expression that is suspected to cause a bug increases the chances of fixing the error. Parts of a function that are known (or at least assumed) to be correct are temporarily removed so the programmer can concentrate effort on fixing the incorrect sub-expression.

As described below, the Dead Code Elimination technique may be generalized to facilitate debugging. It is possible to select a particular sub-expression of interest and to have the function pruned of declarations that are not needed by that particular sub-expression. The expression on the right-hand-side is replaced with the selected sub-expression. This particular generalization of dead code elimination is not a refactoring, it is in fact a transformation since it changes the semantics. For example, consider:

```
count :: [ [a] ] -> Int
count (l:list) = maximum ( map length list )

pad :: [ [a] ] -> [ [a] ]
pad lists = map (pad' (count lists)) lists
     where
        pad' count entry = entry ++ (replicate count (head entry))
```

Suppose the programmer suspects there is a bug in `pad`, specifically, the programmer believes that the bug is in the call `count lists`. Isolating out only the call to `count lists` into a new function would allow the programmer to test that call explicitly, eliminating the parts of `pad` that the programmer believes to be correct:

```
count :: [ [a] ] -> Int
count (l:list) = maximum ( map length list )

pad2 :: [ [a] ] -> Int
pad2 lists = count lists
```

The programmer can then place a call to `pad2` in the code, test the program, discover that the formal parameter to `count` is in fact incorrect and undo the previous transformation and correct the error:

```
count :: [ [a] ] -> Int
count list = maximum ( map length list )
```

The definition of `pad` remains unchanged.

3 Slicing Based Refactorings

3.1 Program Slicing

Weiser, in [16], introduces a program slice S as a *reduced executable program obtained from a program P by removing statements, such that S replicates part of the behavior of P.* This process is driven from a *slicing criterion*, usually a variable representing the line number and expression of interest, which is used to represent the point in the code whose impact is to be observed with respect to the entire program. Weiser introduced the concept that is now known as a *backwards, static* slicing method. A backwards slice consists of all parts of a program that have an effect on the criterion in question. Another form of program slicing is a forwards slice [15]. Starting with the slicing criterion, or the program point of interest, a forwards slice is all parts of the program that the criterion will affect. Orthogonal to forwards and backwards slicing there is also *static* and *dynamic* slicing: static program slicing means that all possible computations of a program are considered and dynamic program slicing means inferring names with particular values, giving a very specific computation of a program. Often, the slicing criterion is a sub-set of program variables —the program slice becoming the parts of the program related to the variables. The obvious analogue to this is a subset of the components of a structured result, for example, the fields of a tuple in Haskell.

Hitherto, there has been little work on program slicing for functional languages. Ochoa et al.[9] introduced a dynamic slicing technique for a lazy logic language. The Haskell debugger, Hat [1], also includes a form of program slicer. However, at this time there is no standalone program slicing tool available for Haskell and therefore we attempt to define a backwards static slicer for Haskell.

This section introduces the notion of program splitting and merging. A number of known issues with performing splitting and merging is also given.

3.2 Splitting

A function may return a structured value, for example, a tuple. The particular examples presented in this section use only pairs, however the technique can easily work over tuples of any order.

Splitting works by getting an element of the tuple and then working out everything needed to calculate that element. The calculated dependencies are then simply extracted and isolated from the rest of the function.

Splitting is mostly used for debugging purposes. However splitting may also be used to extract functionality out of the function so that it can be extended or re-used. The user passes a parameter to the splitter the elements of the result of the function in question which are to be extracted.

Consider the function `parseMessage` below. `parseMessage` takes a `String` of messages each separated by the `&` character. `parseMessage` removes the initial message and returns the next message as the first element of the result and the remainder of the message as the second element:

```
type MessageList = String
type Message = String
parseMessage :: MessageList -> (Message, MessageList)
parseMessage [] = ([], [])
parseMessage xs = (takeWhile (/= '&') (tail ys),
                   dropWhile (/= '&') (tail ys) )
        where
          ys = dropWhile (/= '&') xs
```

As an example of the usage of `parseMessage` consider:

```
> parseMessage "goodbye&hello&world"
> ("hello","&world")
```

The splitter works through each function clause in turn, extracting the elements of the function clauses' result into separate definitions. The first pattern clause of `parseMessage` is essentially trivial. Therefore the value `[]` is extracted for both elements of the result and two new functions are then created:

```
parseMessage1 :: MessageList -> Message
parseMessage1 [] = []

parseMessage2 :: MessageList -> MessageList
parseMessage2 [] = []
```

The splitter appends an index to the end of the names of the new functions, if the new names conflict with any other identifier in scope then the splitter chooses a new distinct name. This is simply done by incrementing the index until the name no longer conflicts with another identifier in scope.

The next function clause's result is then analysed. Irrelevant code elimination is then performed for each element in the resulting tuple and the result of the code elimination is placed into new function clauses for `parseMessage1` and `parseMessage2`. `ys` is required by both elements of the result of `parseMessage` so it is retained; the new function clauses are then added to the definitions of `parseMessage1` and `parseMessage2` within the AST.

```
parseMessage1 xs = takeWhile (/= '&') (tail ys)
        where
            ys = dropWhile (/= '&') xs
```

```
parseMessage2 xs = dropWhile (/= '&') (tail ys)
        where
            ys = dropWhile (/= '&') xs
```

This gives the new definitions of `parseMessage1` and `parseMessage2`. The source code is then modified to reflect the changes within the AST.

3.3 Merging

Merging is the process of taking a number of functions and unifying them together into one tuple returning function. The process described here only merges two functions; obviously the functionality can be easily extended however, to merge together any number of functions.

Merging works by unifying all the code from the selected functions into a new tuple returning function. Duplicate parts of the function are also removed. Unlike when doing splitting, where names are generated automatically, the user must specify a name for the merged function.

Merging is mostly used to reuse code and improve code efficiency. For example, merging `take` and `drop` into `splitAt` results in only one recursive call instead of two. Merging functions together has the possibility to introduce further code sharing.

The remainder of this section will focus on merging `parseMessage1` and `parseMessage2`.

```
parseMessage1 :: MessageList -> Message
parseMessage1 [] = []
parseMessage1 xs = takeWhile (/= '&') (tail ys)
        where
            ys = dropWhile (/= '&') xs

parseMessage2 :: MessageList -> MessageList
parseMessage2 [] = []
parseMessage2 xs = dropWhile (/= '&') (tail ys)
        where
            ys = dropWhile (/= '&') xs
```

Firstly, the merger checks to see whether the pattern sets of `parseMessage1` and `parseMessage2` are the same. If they are not then the merger terminates with an error to the user. The merger also checks to see whether the types of the arguments to `parseMessage1` and `parseMessage2` are the same. If the pattern sets are not the same then the merger cannot correctly unify the patterns; if they cannot be unified the merger terminates with an error message to the user.

Each function clause in question is merged together. The first clause of `parseMessage1` and `parseMessage2` both have the same result value. Merging those clauses together is trivial:

```
parseMessage :: MessageList -> (Message, MessageList)
parseMessage [] = ([], [])
```

The second clauses of `parseMessage1` and `parseMessage2` are now considered. The results are unified together and placed into a new function clause:

```
parseMessage xs = (takeWhile (/= '&') (tail ys),
                   dropWhile (/= '&') (tail ys) )
        where
          ys = dropWhile (/= '&') xs
          ys = dropWhile (/= '&') xs
```

The duplicate declaration of `ys` within the `where` clause is removed.

3.4 Related Issues

Sometimes when splitting is used patterns that are the result of recursive calls become redundant. All patterns that are not needed in the result are replaced with wildcards so that the namespace is kept clean.

One element of a tuple may depend upon another element. For instance, it is possible for one element to come from the first part of a recursive call, but returned in the second element in the main body. In very unusual circumstances the following can occur:

```
f :: Int -> (Char, (Char, Int))
f 10 = (‘a’, (g ‘f’ 1, 2))
f n = (x, (g x 1, 2))
  where
  (_, (x, 2)) = f (n+1)

g :: Char -> Int -> Char
g x y = chr (ord x + 1)
```

The first element of the result of `f` comes from the second element in its recursive call. If one element depends on another, both of these elements must be extracted from the function, otherwise an error occurs.

Before merging, it is necessary that both functions have the same sets of patterns. If the first function has more pattern clauses than the second for example, then the merger cannot determine what to place on the right-hand-side when patterns from the first function are not matched by patterns in the second. An error message is presented to the user if the pattern sets are not the same. An additional refactoring was introduced to allow the user to build particular pattern clauses by instantiating general patterns with values of the same type. Consider, for example:

```
f1 0 l = take 42 l
f1 n l = take n l

f2 n l = drop n l
```

Both `f1` and `f2` have general cases defined, however, `f1` has an additional base case defined. This is problematic because the merger cannot infer what the intention is of the user during the merge:

```
merged 0 l = (take 42 l, undefined)
merged n l = (take n l, drop n l)
```

When merging functions, it is also necessary that the arguments to the functions have the same type so that they can be successfully merged. If the functions have different argument types then the merger returns an error. It is possible that the functions have different return types as this will be captured in a tuple.

Merging and splitting monadic functions is a difficult area, especially if the monad in question is a state monad, for example, the `IO` monad. Merging two `IO` monads is problematic because the merger cannot infer the correct order of sequencing. Side effects also affect splitting in a similar way, for example, an element of the tuple return value may depend upon some data written to a file. It is very difficult for the splitter to determine which parts of the monad can have a potential affect as all expressions have the potential to alter the state. Merging and splitting refactorings on monads will be the subject of a future paper.

4 Conclusions and Future Work

This paper presented a number of refactorings for HaRe. Firstly, a technique was defined to eliminate dead code from Haskell functions and then generalized further to remove irrelevant code. A backwards, static program slicer for Haskell was then described as splitting tuple returning functions; its converse, namely merging, was also described.

In the future it is planned that dead code elimination will be expanded to take a whole program into account and not just a selected function. Dead code elimination could remove the parts of the program that are not directly related to the `main` function, which would allow the process to be extended to the whole of a large Haskell project.

It is also planned to modify the splitter to analyze the whole scope of a program rather than, simply, tuple-returning functions in the scope of a particular function of interest. It is also planned to modify the splitter so that tuple-returning functions that are called from outside a function's scope to be extracted.

A further paper will detail work on refactoring with monads.

It is intended for the work on slicing to be extended further to investigate backwards, dynamic slicing and forwards slicing (both static and dynamic).

There are many more exciting possibilities to analyze for the refactoring of Haskell code; and it is hoped to continue work in these directions.

5 Acknowledgments

Thanks to Dave Harrison of Northumbria University for numerous editorial consultations.

A Source Code

All the code that has been described in this paper can be downloaded using *darcs* [12] with the following command:

```
darcs get http://www.cs.kent.ac.uk/projects/refactor-fp/HaRe_Project/
```

References

1. Olaf Chitil. Source-based trace exploration. In *Draft Proceedings of the 16th International Workshop on Implementation of Functional Languages, IFL 2004*, pages 239–244. Technical Report 0408, University of Kiel, September 2004.
2. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
3. Matthew S. Hecht. *Flow Analysis of Computer Programs.* Elsevier Science Inc., New York, NY, USA, 1977.
4. John H. Holland. *Adaptation in natural and artificial systems.* MIT Press, Cambridge, MA, USA, 1992.
5. Fred P. Brooks Jr. The Mythical Man-Month. In *Proceedings of the international conference on Reliable software*, page 193, New York, NY, USA, 1975. ACM Press.
6. Ralf Lammel and Joost Visser. A strafunski application letter. In *Proc. of PADL'03*, January 2003.
7. Huiqing Li, Simon Thompson, and Claus Reinke. The Haskell Refactorer: HaRe, and its API. In John Boyland and Grel Hedin, editors, *Proceedings of the 5th workshop on Language Descriptions, Tools and Applications (LDTA 2005)*, April 2005. Published as Volume 141, Number 4 of Electronic Notes in Theoretical Computer Science, http://www.sciencedirect.com/science/journal/15710661.
8. Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings of the Fourth 'Colloque International sur la Programmation' on International Symposium on Programming*, pages 269–281, London, UK, 1980. Springer-Verlag.
9. Claudio Ochoa, Josep Silva, and Germán Vidal. Dynamic slicing based on redex trails. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 123–134, New York, NY, USA, 2004. ACM Press.
10. William F. Opdyke. *Refactoring object-oriented frameworks.* PhD thesis, Champaign, IL, USA, 1992.
11. PacSoft. Programatica: Integrating programming, properties and validation. www.cse.ogi.edu/PacSoft/projects//, 2005.

12. David Roundy. Darcs: distributed version management in Haskell. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 1–4, New York, NY, USA, 2005. ACM Press.
13. Josep Silva and Olaf Chitil. Combining algorithmic debugging and program slicing. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 157–166, New York, NY, USA, 2006. ACM Press.
14. Harald Sondergaard and Peter Sestoft. Referential transparency, definiteness and unfoldability. *Acta Inf.*, 27(6):505–517, 1990.
15. F. Tip. A survey of program slicing techniques. Report CS-R9438, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1994.
16. Mark David Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, 1979.

An Interpretation of Temporal Properties in Functional Programs — Extended Abstract*

Máté Tejfel, Tamás Kozsik, and Zoltán Horváth

Department of Programming Languages and Compilers
Eötvös Loránd University, Budapest, Hungary
{matej,kto,hz}@inf.elte.hu

Due to referential transparency, it is straightforward and common practice to express properties of functional programs in some first-order logic, and to prove these properties with an equational reasoning using proof techniques such as induction and co-induction. However, as explained in e.g. [6, 7, 11, 12], sometimes it is useful to formulate *temporal logical properties*, like invariants, of the values computed during the execution of a functional program. Temporal properties can conveniently and compactly describe the expected behaviour of – or more precisely, the expected relations among the expressions appearing in – a functional program, hence they support reasoning about the correctness of functional programs at a high level of abstraction. This facility is especially important when interactive, concurrent, parallel or distributed functional programs are concerned.

Our goal is to extend Sparkle [2], the dedicated theorem prover for Clean [10] with support for temporal logical reasoning. This would make it possible, for instance, to prove the correctness of interactive and concurrent Clean programs, written in (the spirit of) the Object IO library [1]. Earlier papers, such as [11, 12] explained the basic mechanisms for defining and proving temporal properties of Clean programs. As the axiomatic semantics of such properties, they introduced tactics to rewrite temporal logical theorems into first-order logical subgoals. This paper presents the semantic background for verifying the soundness of the introduced tactics and the induced axiomatic semantics.

The operational semantics of functional programs can be defined with rewrite systems. For example, [9] defines the semantics of Clean programs with Term Graph Rewrite Systems, and Sparkle is based on Term Rewrite Systems. In these semantics, a syntactically desugared Clean program is represented as an initial set of graphs (or set of terms), together with rewrite rules. The semantic value of the program is a graph (or term) in some normal form, which is computed by reduction, namely by the iterative application of the rewrite rules. According to these semantics, the state of the rewrite system during reduction is irrelevant; only the final result matters, which, if it exists, is unique. Clearly, this approach is not enough for the investigation of temporal properties. For example, when reasoning about the local states of Clean Object IO processes, like in [4, 5], it should be possible to refer to those local states as they change during reduction,

* Supported by GVOP-3.2.2.-2004-07-0005/3.0 ELTE IKKK and OMAA 66öu2: Programm-Verification mit Hilfe algebraischen Methoden, 2007.

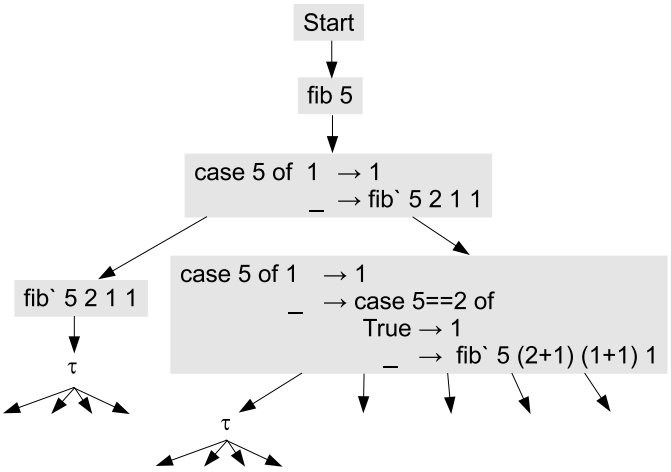


Fig. 1. A program computing the fifth Fibonacci number.

and it should be possible to analyse the relations among the changing local states. For this reason, a semantic function providing more details is required. The alternative semantics will record the inner states of the rewrite system as obtained by the reduction steps. The semantic function maps a program to a tree, where the nodes of the tree are labelled with the possible states of the corresponding rewrite system, and the edges of the tree are labelled with the reduction steps that are applicable in the parent node and produce the child nodes. The number of children equals to the number of the redexes found in the parent, and each branch in the tree represents a possible reduction sequence. Consider a program that computes the fifth Fibonacci number.

```
module fibonacci
import StdInt

fib 1 = 1
fib n = fib‘ n 2 1 1

fib‘ n m u v
  | n == m
    = u
    = fib‘ n (m+1) (u+v) u

Start = fib 5
```

A fragment of the tree that is the semantic value of this program is illustrated in Figure 1. The grey boxes are the nodes of the tree; each box contains a graph under reduction. The graph denoted by τ is as follows.

```
case 5==2 of True -> 1
             _    -> fib‘ 5 (2+1) (1+1) 1
```

For every `a`, `b`, `c` and `d` such that the expression `fib‘ a b c d` occurs in the tree the following holds:

$$\texttt{a} = 5 \ \wedge \ 1 < \texttt{b} \leq \texttt{a} \ \wedge \ \texttt{c} = \texttt{fib b} \ \wedge \ \texttt{d} = \texttt{fib} \ (\texttt{b} - 1).$$

This property is supposed to be expressible as an invariant.

The above technique lets the programmer examine (and reason about) the possible ways in which the state of the rewrite system can change during reduction, for example, the ways in which the local states of Object IO processes may change during program execution. However, there is one remaining problem: how to identify those subgraphs which the temporal properties refer to. Such subgraphs are parts of the graphs appearing in the tree nodes – in the Fibonacci example these were the arguments of the `fib‘` function. In this simple case we propose two ways to describe the invariant property explicitly. The first way is to use subtype marks [8], and the second, more general approach is *object abstraction* [11]. This paper focuses on the latter approach.

Object abstraction is a technique for declaring that certain expressions occurring in the program text represent different states of the same “abstract object”. This technique introduces the necessary syntax for identifying objects and state transitions. In an appropriate Integrated Development Environment this (rather obscure) syntax can be made part of an intermediate language, hidden from the programmers, and the selection of objects can be performed through a GUI. The Fibonacci example with object abstraction can be written in the following way.

```
fib‘ (.|. a n) (.|. b m) (.|. c u) (.|. d v)
  | n == m
     = u
  | otherwise
    .#. ((.|. b m),(.|. c u),(.|. d v))
           = ((.|. b m)+1, (.|. c u)+(.|. d v), (.|. c u))
     =  fib‘ (.|. a n) (.|. b m) (.|. c u) (.|. d v)
```

The objects `a`, `b`, `c` and `d` are introduced by the `.|.` token, and the state transitions are introduced by the `.#.` token. State transitions are a variant of the “let-before” (i.e. `#`) construct of Clean – even their scoping rules are identical. The states of object `b`, for example, are the variables identified with the variable name `m`; there are two such variables in the function body. One of them is visible before the state transition and to the right of the equality sign of the state transition, and the other is visible after the state transition and to the left of the equality sign of the state transition.

Object abstraction does not influence the way computations are performed. By simply removing the syntax for identifying objects and state transitions, a semantically equivalent program is obtained. Furthermore, due to the scoping rules for “let-before”, this definition is equivalent (in the original Clean semantics) to the first version of the `fib‘` function.

```
fib‘ n m u v
  | n == m
     = u
  | otherwise
     # (m,u,v) = (m+1, u+v, u)
     = fib‘ n m u v
```

The next task is to extend the tree-semantics and adapt it to the syntactical changes. This is achieved by annotating the graphs found in the tree nodes by object identifiers, and inserting an extra edge for each state transition on every branch of the tree. In each node of the tree every object is attached to at most one subgraph. The inserted extra edges are orthogonal to reduction; they merely alter the object annotations. Hence the tree-semantics can be regained from the "annotated tree-semantics" by projection.

Now it is possible to define the meaning of temporal properties. For instance, a formula P (defined in terms of some objects $a, b \ldots$) is "always true" in a program *if and only if* it holds for every node in the annotated tree, which is the semantic value of the program. Invariants form a special case of always true formulas: they impose further restrictions on the states that are not reachable during the reduction. Basically, given a reachable or unreachable annotated program graph that satisfies P, all edges starting from such a tree node should result in an annotated program graph that also satisfies P. If a formula is proven to be an invariant, then it is also an always true formula. Therefore invariants are more strict requirements than alway true properties, but they are preferable for being compositional (cf. [3]), which is very important when concurrency is a concern.

Thanks to referential transparency, it turns out that during the proof of temporal properties it is sufficient to consider only those edges of the annotated tree, which are responsible for object redirections. The full paper will provide the details on the annotated tree-semantics, the interpretation of some temporal logical operators, the subgoals to be generated from temporal logical theorems, and the soundness of the resulting proof technique with respect to the annotated tree-semantics.

References

1. Achten, P., Plasmeijer, R.: Interactive Objects in Clean. In: *Proceedings of Implementation of Functional Languages, 9th International Workshop, IFL'97*, St. Andrews, Scotland, UK, September 1997, LNCS 1467, Springer-Verlag, pp. 304–321.
2. de Mol, M., van Eekelen, M., Plasmeijer, R.: Theorem Proving for Functional Programmers, Sparkle: A Functional Theorem Prover, In: LNCS 2312, Springer-Verlag, 2001, p. 55 ff.
3. Horváth, Z.: The Formal Specification of a Problem Solved by a Parallel Program—a Relational Model. In: *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, Tomus XVII. (1998) pp. 173–191.

4. Horváth, Z., Achten, P., Kozsik, T., Plasmeijer, R.: Proving the Temporal Properties of the Unique World. In: *Proceedings of the Sixth Symposium on Programming Languages and Software Tools*, Tallin, Estonia, August 1999. pp. 113–125.
5. Horváth, Z., Achten, P., Kozsik, T., Plasmeijer, R.: Verification of the Temporal Properties of Dynamic Clean Processes. In: *Proceedings of Implementation of Functional Languages, IFL'99*, Lochem, The Netherlands, Sept. 7–10, 1999. pp. 203–218.
6. Horváth, Z., Kozsik, T., Tejfel, M.: Proving Invariants of Functional Programs. In: *Proceedings of Eighth Symposium on Programming Languages and Software Tools*, Kuopio, Finland, June 17–18, 2003., pp. 115–126.
7. Horváth, Z., Kozsik, T., Tejfel, M.: Verifying invariants of abstract functional objects – a case study. In: *6th International Conference on Applied Informatics*, Eger, Hungary, January 27–31, 2004.
8. Kozsik, T.: Proving Properties Specified with Subtype Marks. In: *Implementation and Application of Functional Languages, 18th International Symposium, IFL 2006, Budapest, Hungary, September 4–6, 2006, Revised Selected Papers*, LNCS 4449, Springer-Verlag, 2007. pp. 163–180.
9. Plasmeijer, R., van Eekelen, M.: *Functional Programming and Parallel Graph Rewriting*, Addison-Wesley Longman Publishing Co., Inc., ISBN 0201416638, Boston, MA, USA, 1993.
10. Plasmeijer, R., van Eekelen, M.: *Concurrent Clean Version 2.1 Language Report*, 2002. Available at `http://www.cs.kun.nl/~clean/Manuals/manuals.html`
11. Tejfel, M., Horváth, Z., Kozsik, T.: Extending the Sparkle Core language with object abstraction. In: *Acta Cybernetica* Vol. 17 (2005). pp. 419–445.
12. Tejfel, M., Horváth, Z., Kozsik, T.: Temporal Properties of Clean Programs Proven in Sparkle-T, In: *Proceedings of Central-European Functional Programming School, CEFP 2005*, Budapest, Hungary, July 4–16, 2005, LNCS 4164, Springer-Verlag (2006), pp. 168–190.

Approaches to Subtyping in Functional Languages

Glenn Strong

Trinity College, Dublin, Ireland

Abstract. We present and discuss two closely related type systems for functional programming languages which extend the normal model by offering a notion of subtyping relations between some values in the program.
Subtyping may be treated in a specialized functional programming language which allows declarations of data types which contain sets of overlapping labels. The extension to the type system required to permit the typing of terms which make use of such ambiguous labels can be done in several ways. Two such approaches that are commonly used are (broadly) known as Structural and Nominative subtyping. The selection of approach leads to a very different set of software engineering pressures in the resulting language.
Two type systems are presented which use subtyping to give types to terms which are otherwise ambiguous or incorrect. The type rules are explained and the particular differences that give either structural or nominative subtyping are discussed, as are the necessary inference algorithms, and some observations on the similarities between the two systems are made.
Plans for future work, including experiments involving a complete programming language to determine what software engineering patterns are best supported by each approach are outlined.

On the Validation of Specifications used in Model-Based Testing

Pieter Koopman, Peter Achten, and Rinus Plasmeijer

Software Technology, Nijmegen Institute for Computing and Information Sciences,
Radboud University Nijmegen, The Netherlands
{pieter, p.achten, rinus}@cs.ru.nl

Abstract. In model-based testing the behavior of a system under test, sut, is compared automatically with the behavior of the specification. A significant fraction of issues found in testing appear to be caused by problems with the specification. In order to ensure that the specification prescribes the desired behavior, it has to be *validated* by a human. In this work we introduce a tool to support this validation. In addition to an interactive simulator of the specification, the tool is able to generate transition tables and diagrams of the observed behavior. In order to make simulation and the displaying of the observed behavior finite, we introduce equivalence of states, inputs and outputs.

Extended Abstract

In model-based testing the behavior of a system under test, sut, is compared automatically with the behavior of the specification. The specification is a state transition system that can be nondeterministic. Usually the number of states, inputs and outputs possible is infinite. The sut is also assumed to be a state transition system, but its state is hidden. One can only apply input to the system and observe the corresponding output. We have used model-based testing successfully to improve controllers, protocols, javacard applets and more.

For this comparison of behavior, the test system takes a specification and executes a user defined number of traces. For each trace the sut and the specification starts in their initial states. The test system selects an input that is covered by the specification, applies this input to the sut, and computes the allowed states of the specification. If no states are possible for the specification the sut has shown behavior that is not covered by the specification. The testers say that an *issue* is found.

Ideally, each issue indicates an error in the sut. However, in practice a significant fraction of issues appear to be caused by problems with the specification: the specification does not correctly capture the intentions of the users and the sut does something different. Although the fraction of issues caused by the specification differs with the kind of system and the amount of effort put in the correctness of the specification, we estimate that on average in about 25% of the issues found in model-based testing one has to blame the specification.

Incorrect specifications are a problem for several reasons. First, if an issue is found it is not clear whether we have to blame the specification or the sut. Finding and correcting errors in the specification takes time during the test phase of the project. This is not the right moment to create a correct specification. In many projects there is a significant time pressure during the testing phase of a system. Second, only behavior that is implemented differently by the sut can cause issues. All other errors in the specification are not found at all during model-based testing. Third, any change in the specification during the testing phase can cause significant implementation changes to the sut. Finally, any change in the specification invalidates in principle all previous test results (just like any change in the sut). This implies that errors in the specification can be very expensive and it is worthwhile to invest effort to ensure the quality of the specification.

In our model-based test system G∀st we use the functional language Clean as specification language. Due to the high abstraction level of this language it is possible to write concise specifications which contributes to their quality. The Clean compiler will check quality aspects like type correctness and consistent definition of identifiers used. We have shown that quality aspects such as the reachability of states, determinism and completeness of the specification, and the preservation of constraints can be checked by systematic testing.

However, this does not rule out the possibility that the specification prescribes the wrong behavior in a consistent way. In order to ensure that the specification prescribes the desired behavior, it has to be *validated* by a human. In this work we introduce tools to support this validation. First, a simulator enables the user to execute the specification. Such an interactive execution can be much more illustrative than looking at the code of the specification. Second, it is possible to record the traces of the specification executed in the simulator. The states visited and their transitions can be visualized in a table or a state transition diagram. Since the number of states, inputs and outputs can be infinite and different in each and every specification, this is not straightforward. The key to the solution is an operator to define equivalence of states, inputs and outputs. For instance, values that are handled by the same symbolic transition in the specification (function alternative in the specifying function) are usually considered to be equivalent. All states that are considered equivalent can be mapped to the same entry of the table or the same place in the transition diagram. Since the equivalence of values is problem dependent, some human input is required to define equivalence.

Car Damage Subrogation Workflow
- an iTask exercise -

Erik Zuurbier and Rinus Plasmeijer

ABZ / Solera, Zeist {Erik.Zuurbier@Abz.nl}
Software Technology, Nijmegen Institute for Computing and Information Sciences,
Radboud University Nijmegen {rinus@cs.ru.nl}

Abstract. In this paper we report our experiences and findings while implementing a workflow in the iTask framework [1]. Traditional workflow frameworks model the process and control flow of a tasks: which task has to be carried out when and by whom? They typically model data as global variables: case data reside in a database and the workflow tasks read, write and update the database.
Control flow patterns [2] specify when and how control branches split and merge. Recent developments recognize the structured variants of control flow patterns, which means that the patterns occur in related pairs. For instance a structured discriminator pattern is "The convergence of two or more branches into a single subsequent branch following a corresponding divergence earlier in the process model". The iTask framework follows the reverse route: by default it makes use of structured patterns. Unstructured patterns are only introduced when needed.
The workflow we have implemented using the iTask framework is a reimplementation of an existing workflow application which is daily used in the Dutch insurance industry. The original workflow application is implemented in a traditional programming environment without a workflow framework. The workflow is carried out between employees of two different insurance companies. Both cover against liability claims of their insured and at least one also covers car damage repair costs. In this context the term subrogation refers to the right of a car damage insurer to recover a paid claim from the liability insurer of the party who caused the damage. The Car Damage Subrogation Workflow (CDSW) controls the claim negotiations that take place between the insurance companies. It turned out that the CDSW can be specified very elegantly in the iTask framework.
To give the user the freedom to fill in information in advance, new workflow patterns (iTask combinators) for speculative tasks have been added. Compared to standard patterns the new patterns are quite advanced, yet it was very easy to add them to the iTask system.

1. Rinus Plasmeijer, Peter Achten, Pieter Koopman. iTasks: Executable Specifications of Interactive Work Flow Systems for the Web. Accepted for publication in 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007), Freiburg, Germany, October 1 3, 2007.
2. http://www.workflowpatterns.com

Towards Open Type Functions for Haskell

Tom Schrijvers*[1], Martin Sulzmann[2], Simon Peyton-Jones[3], and Manuel Chakravarty[4]

[1] K.U.Leuven, Belgium (tom.schrijvers@cs.kuleuven.be)
[2] National University of Singapore (sulzmann@comp.nus.edu.sg)
[3] Microsoft Research Cambridge, UK (simonpj@microsoft.com)
[4] University of New South Wales (chak@cse.unsw.edu.au)

Abstract. We report on an extension of Haskell with type(-level) functions and equality constraints. We illustrate their usefulness in the context of phantom types, GADTs and type classes. Problems in the context of type checking are identified and we sketch our solution: a decidable type checking algorithm for a restricted class of type functions. Moreover, functional dependencies are now obsolete: we show how they can be encoded as type functions.

1 Introduction

Experimental languages such as ATS [6], Cayenne [1], Chameleon [25], Epigram [15] and Omega [21] equip the programmer with various forms of "type functions" to write entire programs on the level of types. In the context of Haskell, there are two distinct languages extensions that that support such type-level computation: *functional dependencies* which are well established [12], and *associated types* which are a more recent experiment [5]. In this paper, we make the following contributions:

- We generalise the so-called "associated type synonyms" [5] by decoupling them from `class` declarations, thereby allowing us to define stand-alone type functions (Section 2). We give examples which show the usefulness of stand-alone type functions in combination with GADTs and phantom types.
- It turns out that pure type *inference* for our extended language is very easy. However, in the presence of user-supplied type signatures (which are ubiquitous in Haskell) and GADTs, the type *checking* problem becomes unexpectedly hard. We identify the problem and sketch our solution (Section 3). This is the main technical contribution of the paper.
- We show that type functions are enough to express all programs involving functional dependencies, although the reverse is problematic (Section 4). Other related work is discussed in Section 5.

For space reasons, and because it reports work in progress, this paper is entirely informal. We have much formal material in an accompanying draft technical report [20].

* Post-doctoral researcher of the Fund for Scientific Research - Flanders.

2 Informal overview

We begin informally, by giving several examples that motivate type functions, and show what can be done with them. Notably, we have found three uses of type functions: in combination with type classes, in combination with GADTs and even in the basic Hindley/Milner type system.

2.1 Type classes and type functions

The original paper on functional dependencies [12] presented the following class of collections:

```
class Collects c e | c -> e where
  empty  :: c
  insert :: e -> c -> c
  toList :: c -> [e]
instance Collects BitSet Char where ...
instance Eq e => Collects c [c] where ...
```

The notation "`| c -> e`" means "the collection type `c` determines the element type `e`". The two instance declarations explain that the collection of type `BitSet` has elements of `Char` elements; and a collection of type `[e]` has elements of type `e`. The "..." parts give the implementations of the methods `empty`, `insert`, etc.

Using our proposed type-function extension we would re-express the example as follows:

```
type family Elem c
class Collects c where
  empty  :: c
  insert :: Elem c -> c -> c
  toList :: c -> [Elem c]

type instance Elem BitSet = Char
instance Collects BitSet where ...

type instance Elem [e] = e
instance Eq e => Collects c [c] where ...
```

The type class now has only one parameter, `c`. A new *type family* `Elem` is defined, using a `type family` declaration. We think of `Elem` as a function from the collection type to the element type, and indeed often refer to it as a "type function", although the term "function" is so heavily used that we use "type family" when we want to be precise. The types of the class methods should now be self-explanatory; indeed, they are more perspicuous than before.

Each instance declaration now has two related parts. First, we add an equation to the definition of `Elem`, using a `type instance` declaration. Second, we have a perfectly ordinary Haskell `instance` declaration.

One might wonder whether functional dependencies are more expressive than type functions, or vice versa, and we discuss that in Section 4.

2.2 Plain type functions

The main benefit of decoupling the type functions from type classes is that the former can now be used independently. (We still offer the syntax for associated type synonyms proposed in [5], but it is purely syntactic sugar.) For example, here is how we might write a library that manipulates lengths, areas, volumes, and so on:

```
data Z     -- Peano numbers
data S a   -- at the level of types

newtype Val u = V Float

type Scalar = Val Z
type Length = Val (S Z)
type Area   = Val (S (S Z))
type Volume = Val (S (S (S Z)))

addVal :: Val u -> Val u -> Val u
addVal (V v1) (V v2) = V (v1+v2)

mulVal :: Val u1 -> Val u2 -> Val (Sum u1 u2)
mulVal (V v1) (V v2) = V (v1*v2)
```

The phantom-type parameter `u` keeps track (statically) of the units of the value [11]. The idea is that `u` will be instantiated by a Peano-number representation of the dimension of the number, as suggested by the (ordinary, Haskell) type synonym declarations of `Scalar` etc. The signature of `addVal` specifies that it can only add two values of the same units.

The signature of `mulVal` is more interesting, because the dimension of the result is the sum of the dimensions of its argument — for example, multiplying a `Length` by an `Area` gives a `Volume`. So we need a type-level computation, expressed using the type function `Sum`:

```
type family Sum n m
type instance Sum Zero x     = x
type instance Sum (Succ x) y = Succ (Sum x y)
```

Notice that no type classes are involved here. The same program can be written using functional dependencies, but only by bringing in a type class (with no methods), and only by using a more relational notation:

```
mulVal :: Sum u1 u2 r => Val u1 -> Val u2 -> Val r
```

The example can readily be extended to handle multiple units (e.g. time as well as length).

This encoding does not express the fact that `Val` should *only* be applied to compositions of `S` and `Z`. It would be better to express this idea in the kinds thus:

```
datakind Nat = Z | S Nat

newtype Val (u::Nat) = V Float
```

Not only is this more explicit, but it also allows us to check that we have provided all the equations for `Sum`, and permits induction over `Nat`. In effect, `Sum` is a *closed* type function whereas `Elem` was an *open* one. In this paper we concentrate on open functions, and leave the exploitation of closed-ness for future work.

2.3 GADTs and type functions

Generalised Algebraic Data Types (GADTs) are extremely useful for expressing rich data structure invariants at the type level. A well-known example is that of length-indexed lists, or *vectors* for short:

```
data Vector el len where
  Nil  :: Vector el Z
  Cons :: el  -> Vector el len -> Vector el (S len)
```

where we use the type encoding of the natural numbers from the previous section.

With vectors we can easily avoid some of the pitfalls of ordinary lists. Consider the well-known Haskell function `zip :: [a] -> [b] -> [(a,b)]` for pairing up the corresponding elements in two lists. It has an annoying corner case: when the lengths of the two lists are not matched, then the trailing elements of the longer list are simply and silently discarded. With vectors we can easily rule out this corner case at compile time. Consider the definition of `vzip`, a zip for vectors:

```
vzip :: Vector a len -> Vector b len -> Vector (a,b) len
vzip Nil Nil = Nil
vzip (Cons x xs) (Cons y ys) = Cons (x,y) (vzip xs ys)
```

Observe that the length type parameter of both input lists is identical. This means that the type checker verifies for every call `vzip as bs` whether the vectors `as` and `bs` have the same length. If not, the program is rejected.

For length-indexing to be useful, we should be able to express the impact of list transformations on the length. Unfortunately, without resorting to overly complicated[5] type classes with functional dependencies, Haskell's type system does not allow us to express even the most basic of transformations.

Concatenation of vectors is a good example. While this implementation is easy enough to write:

```
vconcat Nil         l   = l
vconcat (Cons x xs) ys  = Cons x (vconcat xs ys)
```

and, like `mulVal`, its signature involves a type-level computation;

```
vconcat :: Vector e n -> e Vector e m -> Vector e (Sum n m)
```

This function `Sum` is a type-level function, defined in the previous section.

[5] and rather ill-understood

2.4 Equality constraints

Suppose that we want to write a function `merge` that adds all the elements of one collection to another collection. It cannot have type

```
merge :: (Collects c1,Collects c2) => c1 -> c2 -> c2
```

because not all collections have the same element types. On the the other hand, it is over-restrictive to write

```
merge :: (Collects c) => c -> c -> c
```

because it is perfectly OK to merge collections of different types *provided their element types are the same*. For example one could merge a `BitSet` with a `[Char]` because they both have element type `Char`.

The way to achieve this is to use an *equality constraint*:

```
merge :: (Collects c1,Collects c2,Elem c1 ~ Elem c2) =>
              c1 -> c2 -> c2
```

where constraint "`Elem c1 ~ Elem c2`" says that `c1` and `c2` must only be instantiated to types for which `Elem c1` and `Elem c2` are equal. These equality constraints are, in fact, quite familiar from GADTs. Recall the definition of `Vector` from the previous section:

```
data Vector el len where
  Nil  :: Vector el Z
  Cons :: el  -> Vector el len -> Vector el (S len)
```

One way to think of `Cons` is that it has type

```
Cons :: (slen ~ S len) => el -> Vector el len -> Vector el slen
```

Our new design allows arbitrary type equalities to be specified in a type signature, with GADTs as a useful special case. In [5] a number of restrictions are imposed on the form of equality constraints. We do not impose any restrictions; even constraints that do not involve any type functions are allowed, e.g. `Int ~ Bool`.

2.5 Summary

In general, a type function is introduced by a top-level `type family` declaration. An optional kind signature may be used for both the argument types and the result type; for example:

```
type family MonadRef (monad :: * -> *) :: (* -> *)
```

Otherwise, these kinds are assumed to be `*`.

Like a regular Haskell 98 type synonym, a type function has an *arity*, given by the number of named arguments to the left of the "`::`". For example `MonadRef`

has arity 1, even though it has kind "`*->*->*`". Like regular type synonyms, type-function applications must be *saturated*: they must be supplied with at least as many type as prescribed by their arity. Again like type synonyms, over-application is of course allowed, e.g. `MonadRef IO Int`.

Unlike regular type synonyms, however, type functions are open functions, whose definition is extended by `type instance` declarations; for example:

```
type instance F [a] k = (a,k)
```

The part to the left of the "=" is called the *definition head* and the part to the right the *definition body*. The head must have exactly as many type parameters as the arity of the type function.

In order to ensure modularity, consistency of the type function definition and termination of type inference, a number of conditions must be imposed on the instances:

1. Instance heads must not overlap.
2. Type function applications in the body must be smaller than the head.
3. Type function applications in the body must not occur inside other type function applications.

We return in more detail on these conditions when we discuss type checking.

Just as Haskell has data types as well as type synonyms, we also support *data type families* as well as type functions. For example:

```
data family GMap k v
data instance GMap Int    v = GI (Map.Map Int v)
data instance GMap (a, b) v = GP (GMap a (GMap b v))
```

Like a `type instance`, there may be many `data instance` declarations for each data family, each having a different type pattern to the left of the "`=`". The rest of the declaration is just like a regular Haskell data type declaration: it defines one or more constructors. (They can even be GADTs!)

Data type families are really only useful in association with type classes; we refer the reader to [4] for details. In contrast to type functions, it is extremely straightforward to add data type families to the type inference engine, and we do not discuss them further here.

3 Technical challenges

To add type functions to Haskell we must explain how to adapt the type inference engine to accommodate them. Parts of this turned out to be very easy but, somewhat to our surprise, other parts were much harder than we anticipated. One of the main contributions of this paper is to identify just what is hard, although we have space only to sketch our solution.

3.1 The easy part: type inference

Consider the problem of doing pure type inference (i.e. with no types declared by the programmer) in the presence of type functions.

This is an easy problem. Recall that the `type instance` declarations are restricted (Section 2.5) so that they can be regarded as a left-to-right rewrite system that is (a) confluent and (b) terminating. Type inference is conventionally done using unification (see [18] for a tutorial). When type functions are added, we modify the unifier so that when it tries to unify two types, it first *normalises* them using the rewrite rules.

If performed too early, this normalisation may get "stuck". For example, consider inferring the type for

```
\c -> (insert 'x' c, length c)
```

where `insert` was defined in Section 2.1, and `length` has its usual type:

```
insert :: Collects c => Elem c -> c -> c
length :: [a] -> Int
```

Initially, type inference assigns an unknown type α to `c`. The `insert` call requires us to unify `Char` (the type of `'x'`) with `Elem` α (the argument type of `insert`). Since we do not know what α is, normalisation gets stuck. But all is well, because "later", the call `length c` forces α to be unified with `[`β`]`; and now the stuck normalisation can proceed, rewriting `Elem [`β`]` to β.

So all we need is a way to suspend stuck unifications, and try them again later. That is, we must gather as-yet-unsatisfied equality constraints from the term, and attempt to solve them later. Happily, Haskell already requires us to gather type-class constraints from the term, so all the plumbing is already in place.

All that remains is to consider generalisation. Consider the definition

```
f = \c -> insert 'x' c
```

When we come to generalise `f`, the stuck unification is still stuck! But that is easy: just as we abstract over *type class* constraints in this situation, so we abstract over *equality* constraints, to give the type

$$\texttt{f} :: \forall a.(\texttt{Collects}\ a, \texttt{Elem}\ a \sim \texttt{Char}) \implies a \to a$$

3.2 The hard part: type checking

Alas, we cannot live with type inference alone. Type checking is necessary as well, for a number of reasons:

- Programmers want to write signatures, as a form of specification or documentation of their program.
- Full type inference is infeasible for a number of type system features, notably for GADTs [24].

Consider again the `vconcat` function:

```
vconcat :: Vector e n -> Vector e m -> Vector e (Sum n m)
vconcat Nil         l   = l
vconcat (Cons x xs) ys  = Cons x (vconcat xs ys)
```

Let us focus on the first equation alone, the case for `Nil`. We know that `l` is of type `Vector e m`. The program type checks if we can show that `l` is also of type `Vector e (Sum n m)`. If we drop the identical parts, this boils down to showing that `m` equals `Sum n m`. How can we establish this equality? The pattern match `Nil` makes available the (local) assumption $\texttt{n} \sim \texttt{Z}$. So we want to deduce that

$$\texttt{n} \sim \texttt{Z} \implies \texttt{m} \sim \texttt{Sum n m}$$

And this holds, of course, because we can make use of the top-level type-function equations for `Sum`:

$$\begin{array}{c}(\forall ys.\texttt{Sum Z}\ ys \sim ys),\\ (\forall xs, ys.\texttt{Sum (S}\ xs)\ ys \sim \texttt{S (Sum}\ xs\ ys))\end{array} \models \texttt{n} \sim \texttt{Z} \implies \texttt{m} \sim \texttt{Sum n m}$$

Similar reasoning applies to the `Cons` case.

In general, type checking is reduced to an entailment check among type equations with respect to an equational theory:

$$E_t \models E_g \implies E_w$$

where

- E_t (top-level equations) refers to the type function theory, i.e. the top-level type function definitions. These equations may involve universal quantification; e.g. $\forall ys.\texttt{Sum Z}\ ys \sim ys$.
- E_g are the given equations arising from type annotations and GADT pattern matchings, for example $\texttt{n} \sim \texttt{Z}$. These equations are over monotypes, with no universal quantification.
- E_w are the wanted equations arising out of expressions, for example $\texttt{m} \sim \texttt{Sum n m}$. Again, the equations are over monotypes.

3.3 The type checking strategy

The type inference strategy was to use the top-level equations E_t to normalise the wanted constraints E_w. But we cannot do this for type checking, *because the additional given constraints E_g do not necessarily form a terminating, confluent rewrite system*, particularly when combined with E_t:

1. They are not properly oriented to ensure termination. E.g. the TRS formed by top-level equation `F Bool = Int` and given equation $\texttt{Int} \sim \texttt{F Bool}$ is clearly looping.
2. They may well be inconsistent (i.e. non-confluent) with respect to each other or the top-level equations. E.g. the TRS formed by top-level equation `F Bool = Int` and given equation $\texttt{F Bool} \sim \texttt{Char}$ is not consistent.

The solution of these issues is to transform the given equations E_g into an equivalent set of equations E'_g that *does* satisfy all the necessary properties. In TRS-terminology, the problem of finding an such an E'_g is known as the *completion* problem.

Unfortunately, there is no off-the-shelf completion algorithm that suits our needs. Existing completion procedures are either undecidable [3] or restricted to systems of ground equations [19]. What we require is a completion algorithm that is (1) decidable and (2) takes into account the non-ground top-level equations. In addition, we want to exploit the injectivity property of Haskell type constructors (usually not considered in TRS). We have therefore devised a novel completion algorithm that satisfies all our requirements; this algorithm is our main technical contribution.

Our completion algorithm comprises the following steps:

- TOP: E_g is normalised with respect to E_t, e.g. `Int` $\sim$ `F Bool` is normalised to `Int` $\sim$ `Char` with respect to `F Bool = Int`, exposing the inconsistency.
- TRIVIAL: Trivial equations are dropped, e.g. `F a` $\sim$ `F a`, avoiding trivial non-termination.
- DECOMP: Non-essential type constructors are dropped , e.g. `(F a,F b)` $\sim$ `(Int,Bool)` becomes `F a` $\sim$ `Int` and `F b` $\sim$ `Bool`.
- SWAP: Equations are oriented properly, e.g. `Int` $\sim$ `F Bool` becomes `F Bool` $\sim$ `Int`.
- SUBST: of E_g are substituted in each other, exposing inconsistencies. E.g `F a` $\sim$ `Int` is substituted in `F a` $\sim$ `Char`,resulting in `Int` $\sim$ `Char`.

Moreover, our completion algorithm successfully deals with particularly difficult given equations like `F Int` $\sim$ `[G (F Int)]`. From left-to-right, the equation is non-terminating (the left-hand side occurs in the right-hand side), while the SWAP rule rejects the right-to-left orientation. Our solution is to break the equation into two new equations: `F Int` $\sim$ `a` and `[G (F Int)]` $\sim$ `a` where `a` is a skolem constant. After further completion we end up with `F Int` $\sim$ `a` and `[G a]` $\sim$ `a` which is a proper strongly-normalising TRS.

An inconsistency discovered during completion, e.g. `Int` $\sim$ `Char`, means that no evidence can be provided to support the given equations. While not ill-typed, the code under consideration is effectively unreachable.[6]

3.4 Restrictions on type function definitions

We already mentioned that a number of conditions must be imposed on the top-level type function definitions for reasons of soundness and completeness of our type checking strategy. The ground rules are these:

Modularity `type instance` declarations may be added one at a time, and must be individually accepted or rejected. It is not acceptable to require a global analysis of all the `type instance`, followed by a "yes" or "no" answer.

[6] Our implementation raises an error to alert the programmer.

Arbitrary given constraints We may place restrictions on the `type instance` definitions, but we should place no restrictions on the additional given constraints E_g, because pattern matching on a GADT can give rise to arbitrary constraints.

Simplicity The simpler the rules, the better.

As an example of the need for arbitrary given constraints E_g consider the following program:

```
data Eq a b where
  EQ :: EQ a a

f :: Eq (F a) (G a) -> Int
f EQ = ...
```

where `F` and `G` are type functions In the right hand side of `f`, we have the given equation `F a` $\sim$ `G a`, and clearly we could have given rise to an arbitrary such equation simply by choosing a different type signature for `f`.

Confluence The TRS-based type checking strategy requires that the top-level equations are confluent. For terminating rewrite systems, confluence is a decidable property: the test is based on the normalisation of critical pairs [13]. For reasons of modularity and simplicity[7], we propose more restrictive properties:

1a. The heads of type function definitions do not contain (nested) type functions.
1b. The heads of type function definitions may not overlap.

The first of these is analogous to requiring that the patterns in an ordinary function definition use only variables and constructors, but not functions.

The second ensures that only one equation can match, and hence their order does not matter. Remember that, unlike Haskell function definitions, but like instance declarations, the `type instance` declarations for a type function are not required to occur all together, and hence are un-ordered. For example, the rule excludes this non-confluent overlap:

```
type instance F Int  = Bool
type instance F Int  = Char
```

but also excludes this set of confluent definitions:

```
type instance F Int  = G Bool
type instance F Int  = G Char

type instance G Bool = ()
type instance G Char = ()
```

[7] from the points of view of programmers and compiler writers

Termination Next to confluence, termination of the TRS is essential for the completeness of our type checking strategy. The main principle for establishing termination in rule-based languages (to which type definitions belong) is that of decreasing calls [2]. A level-mapping assigns a value to all function calls; and all rules must satisfy the property that the level mappings of all calls in the right-hand side are smaller than the level-mapping of the rule-head. State-of-the-art termination analysers, e.g. [10], are capable of automatically inferring level-mappings in terms of various well-founded orders [7] .

For reasons of modularity and simplicity, we propose not implement a state-of-the-art termination analysis, but rather to impose two simple conditions on all individual type definition clauses:

2a. The number of symbols (type constructors and schema variables) in each type function call in the body, is smaller than the number of the head.
2b. The number of occurrences of any schema variable in each type function call in the body, is smaller than the number of the head.

Completion Perhaps surprisingly, confluence and termination of the top-level equations is not enough. We must still account the completion of the given equations. Let's consider a single `type instance` that respects all the above conditions:

```
type instance H [[a]] = H (G a)
```

and the single given equation `G Int` $\sim$ `[[Int]]`. The completion algorithm preserves this given equation, and yet the union of the two equations is not terminating: `H [[Int]]` $\rightarrowtail$ `H (G Int)` $\rightarrowtail$ `H [[Int]]` $\rightarrowtail \ldots$.

It turns out that the problem is caused by the nested function call `H (G a)`. We have gained much insight by expressing our problem as a set of Constraint Handling Rules (CHRs); in that setting, a nested function call corresponds to a "non-range restricted simplification rule", which is known to be symptomatic of termination problems [22].

Our current solution is simple, if brutal; we add one further restriction:

3. No type function call may occur inside another type function call in a type definition clause.

Sadly, this restriction renders illegal a class of useful (usually closed) functions, e.g.:

```
type instance Mult Z     m = Z
type instance Mult (S n) m = Sum (Mult n m) m
```

Perhaps a more relaxed rule would suffice, a question we leave for future work.

3.5 Type-directed compilation

In a type-directed compiler, the type checker's task goes beyond providing a simple yes (the program is well-typed) or no (it's not). It must also generate the

necessary type information to enable the desugaring of the source language into the strongly-typed intermediate language. Hence, we adapt our type checking algorithm to generate type information for System F_C [23]. This is an extension of System F, which has been specifically designed as a practical compiler backed for Haskell, and is in actual use in GHC.

Encoding System F_C already has the essential ingredients, type functions and equality coercions, which have already proven their usefulness for encoding GADTs and associated type synonyms.

System F_C's type functions and their definition are essentially identical to those in the source language, but the equality coercions deserve a little explanation. Thanks to its syntax-directedness, type checking in System F_C is much cheaper than in Haskell: declared and inferred types are checked for syntactic equivalence, e.g. `Int` $\equiv$ `Int`.

However, the inference of non-syntactical equivalence proofs, like `F Int` $\sim$ `Bool`, is problematic in System F_C. The reason is that the set of equational axioms in System F_C may be inconsistent. In particular, the internally consistent set of type function clauses may be at odds with the `newtype` axioms.

Example 1. The `newtype X = Int` is encoded in System F_C as an axiom `X`$\sim$`Int`, which conflicts with the type function:

```
type instance F X    = Char
type instance F Int  = Bool
```

We can show that `F X` is both equal to `Char` and `Bool`, the former via the first clause of `F` and the latter via the `newtype` axiom and the second clause.

Fortunately, it is not necessary to repeat a proof that was already made by the Haskell type checker. The Haskell type checker can create a witness γ for the proof. Now the System F_C type checker can simply check the proof, based on its witness, rather than to infer it anew. This avoids the unsoundness pitfall and, as a bonus, checking a proof is also much cheaper than inferring it.

In System F_C, evidence is represented by a *coercion* γ, a special form of types whose kind is an equation, e.g. γ : `F Int`$\sim$`Bool` means that γ is evidence for the equation `F Int`$\sim$`Bool`. Coercion constants are denoted C and coercion variables co.

In System F_C, a unique coercion constant is associated with every type function clause, e.g. `C : type instance F Int = Bool`. Similarly, a given equational constraint, translates to a coercion variable, e.g.

```
id :: forall a b . a ~ b => a -> b
id = \x -> x
```

is encoded in System F_C as:

```
id :: forall a b . a ∼ b => a -> b
id x = id = Λ(a:*).Λ(b:*).Λ(co:a ∼ b).λ(x:a).(...)
```

In order to type check an expression `x` whose inferred and expected types are `a` and `b` respectively, it has to be *cast* with the appropriate the evidence: `e` ▶ γ where γ has kind $a \sim b$. Because types are eventually erased, these casts do not incur any runtime overhead.

Example 2. The full encoding of the above `id` function in System F_C is:

```
id = Λ(a:*).Λ(b:*).Λ(co:a ∼ b).λ(x:a).(x ▶ co)
```

Complex coercions can be constructed from primitive coercions with coercion constructors:

- `sym` γ has kind $a \sim b$ if γ has kind $a \sim b$.
- $\gamma_1 \circ \gamma_2$ has kind $a \sim c$ if γ_1 has kind $a \sim b$ and γ_2 has kind $b \sim c$.
- $T\ \gamma$ has kind $T\ a \sim T\ b$ if γ has kind $a \sim b$, where T is a type constructor.
- $T\ \gamma$ has kind $T\ a \sim T\ b$ if γ has kind $a \sim b$, where T is a type constructor.
- $\mathtt{decomp}_{T,i}\ \gamma$ has kind $a_i \sim b_i$ if γ has kind $T\ \bar{a} \sim T\ \bar{b}$, where T is a type constructor.

Example 3. Using the previously defined type function `F`, the program:

```
main :: F Int
main = id True
```

is encoded in System F_C as:

```
main :: F Int
main = id @ Bool @ (F Int) @ (sym C) True
```

where type applications are denoted by @.

Coercion generation Given the appropriate coercions, the System F_C encoding of a Haskell program is a pretty straightforward matter. The hard part is of course the generation of these appropriate coercions, a task of the Haskell type checker.

Whenever the Haskell type checker constructs a wanted equation $\tau_1 \sim \tau_2$, i.e. to equate the inferred and expected types τ_1 and τ_2 of an expression `e`:

1. it creates a fresh *unknown* coercion γ of kind $\tau_1 \sim \tau_2$,
2. it inserts a cast in the code: `e` ▶ γ, and
3. it associates the coercion with the wanted equation, denoted $\gamma : \tau_1 \sim \tau_2$.

Whenever the type checker discharges a wanted equation, it fills in the unknown coercion, e.g. $\gamma := \gamma'$ where γ' has the same kind as γ. This is the *hard* part: how do we track the coercions of the top-level and given equations through the rewriting process of our type checking algorithm?

Firstly, it's not a simple matter of matching up some given equation with a *whole* wanted equation. Our algorithm is based on rewriting *individually* the left- and right-hand sides of a wanted equation, i.e. $\tau_1 \rightarrowtail^* \tau$ and $\tau_2 \rightarrowtail^* \tau$, to obtain a trivial equation of the form $\tau \sim \tau$.

Hence, we must construct two coercions γ_1 and γ_2, one to justify each rewriting, i.e. $\gamma_1 : \tau_1 \sim \tau$ and $\gamma_2 : \tau_2 \sim \tau$. From these two coercions we can then determine the unknown coercion: $\gamma := \gamma_1 \circ$ `sym` γ_2.

Example 4. Given these two clauses:

```
C1 : type instance F Int  = Bool
C2 : type instance F Char = Bool
```

the type checker rewrites the left- and right-hand sides of the wanted equation γ : `F Int` $\sim$ `F Char` to `Bool` with coercions `C1` and `C2` respectively. Hence, the unknown coercion γ is determined as `C1` $\circ$ `sym C2`.

Secondly, we have to account for the completion phase. In the completion phase, the given equations are transformed. Hence, it's corresponding evidence has to be transformed accordingly. For that purpose, all the steps in the completion algorithm of Section 3.3 have to be augmented with coercion transformations. For example:

- γ : `Bool` $\sim$ `F Int` transformed with the SWAP step, becomes `sym` γ : `F Int` $\sim$ `Bool`,
- γ : `(F a,F b)`$\sim$`(Int,Bool)` decomposed with `Decomp`, becomes $\texttt{decomp}_{\texttt{(,)},1}\ \gamma$: `F a` $\sim$ `Int` and $\texttt{decomp}_{\texttt{(,)},2}\ \gamma$: `F b` $\sim$ `Bool`.

4 Type functions versus functional dependencies

One of the most hotly debated questions in the latest standardisation process of the Haskell language (Haskell Prime [17]) is:

> Should Haskell Prime adopt either functional dependencies or associated type synonyms?

There is little sense in providing two features for expressing functional relations. The above question now subsumed by a new one:

> Should Haskell Prime adopt either functional dependencies or *type functions*?

We see three possible reasons for preferring type functions:

1. Type functions are inherently more familiar to functional programmers: it is a small step from functions at the value level to functions at the type level.
2. Type functions have their uses outside of type classes. Similar encodings with functional dependencies are rather bloated.
3. While functional dependencies have been around for quite a while now, it seems type checking for them is still rather ill-understood. In contrast, our prototype implementation of type functions type checks has no problems with GHC's open bugs related to functional dependencies.

However, before we consider the question from the point of view of language and compiler design (as the above arguments do), we first have to study a more pressing matter:

Is either of functional dependencies or type functions more expressive than the other?

While we do not yet have a formal result, we claim that both language features are indeed equally expressive. In the remainder of this section we justify our claim constructively and present translations both ways. Hence, language designers and compiler writers can happily disagree: the first group gets to choose what language feature to program in and the second group what language feature to implement.

4.1 From functional dependencies to type functions

We claim that every program involving functional dependencies can be re-expressed to one involving only type functions. This can often by done in an idiomatic way; for example, consider the way in which we re-expressed the two-parameter `Collects` class using a single-parameter class together with a type function (Section 2.1). But it is less clear how to translate classes with multiple or bi-directional functional dependencies, such as

```
class C a b | a -> b, b -> a where ..
```

Furthermore, if one starts with an existing program, the idiomatic translation is somewhat invasive because every occurrence of `Collects` must be changed to remove a type parameter, and new equality constraints must sometimes be added. For example,

```
merge :: (Collects c1 e, Collects c2 e) => c1 -> c2 -> c2
```

must become that of Section 2.4:

```
merge :: (Collects c1, Collects c2, Elem c1 ~ Elem c2) => c1 -> c2 -> c2
```

Thus motivated, we have developed an alternative, minimally invasive translation scheme from functional dependencies to type functions. The scheme is minimally invasive because it only affects `class` and `instance` declarations, and leaves all else untouched.

It works as follows. In the `class` declaration each functional dependency $\bar{a} \rightarrow b$ is replaced by: (1) a new (associated) type function $F\ \bar{a}$ and (2) a context constraint $F\ \bar{a} \sim b$. In every `instance` the proper type function instance is added.

Example 5. The transformed type class for collections is:

```
class Elem c ~ e => Collects c e where
  type Elem c

instance Collects [e] e where
  type Elem [e] = e

instance Collects BitSet Char where
  type Elem BitSet = Char
```

4.2 From type functions to functional dependencies

At first sight, the second part of the question should be answered negatively: as type functions do not have to be associated with type classes they are strictly more expressive. However, we can of course consider a fresh type class with functional dependencies (but no methods) to replace a stand-alone type family.

Example 6. The stand-alone type function `Sum` could be replaced by:

```
class Sum a b c | a b -> c

instance Sum Zero b b
instance Sum a b c => Sum (Succ a) b (Succ c)
```

In general, we can replace an n-ary type function with an $(n + 1)$-ary type class, with a functional dependency from the n first arguments to the last one. Every function instance becomes a class instance where the n arguments of the LHS make up the n first arguments and the RHS becomes the $(n+1)$th argument. Any function calls in the RHS have to be flattened into relational form in the instance context.

Matters become more complicated when data types are involved. For example, the following is perfectly legal in our system:

```
data T c = MkT [Elem c]
```

That is, a value of type `T c` is a `MkT` constructor wrapping a list of elements of collection type `c`. Notice that no type-class constraints are involved here. It is unclear how to translate this to functional dependencies. Certainly, we must add a new type parameter to the data type `T`, but then we need a way to express the connection between the two parameters. Something like this, perhaps?

```
data Collects c e => T c e = MkT [e]
```

But it is not clear that the `Collects c e` context on this declaration has the "right" effect. Similar complications arise with type synonyms and `newtype`s; see [8, Chapter 5]. A substantial advantage of our approach is that these complications go away.

5 Related work

Existing languages with type functions differ on various accounts from Haskell type functions. They only offer a fixed set of predefined functions (e.g. ATS [6]), type checking is incomplete (e.g. Cayenne [1], Epigram [15], Omega [21]) or the programmer has to construct the proofs himself (e.g. LH [14]). Moreover, all these languages assume that type functions are closed. More closely related to our work is the Chameleon system described in [25]. Chameleon makes use of the Constraint Handling Rules (CHR) [9] formalism for the specification of type class and type improvement relations. CHR is a committed-choice language

consisting of constraint rewrite rules. We expect to model open type functions via CHR rewrite rules which hopefully allows us to transfer some of the existing CHR type inference results [22] to the type function setting. The hard part so far has been modelling the treatment of evidence in CHR, which is reasonably straightforward in our current TRS formalism.

Both Neubauer et al. [16] and Diatchki [8] propose a functional notation for type classes with a functional dependencies. However, this notation is essentially syntactical sugar for the conventional relational notation of type classes. So these approaches gain the convenience of a functional notation, but miss the other advantages of our approach, especially concerning the use of type functions in the definition of data types.

6 Conclusion & future work

We have presented type functions, open functions at the type level. While they're equally expressive as functional dependencies when used with type classes, type functions can also be put to good use with GADTs and phantom types. We sketched a type checking strategy based on completion and term rewriting. Our implementation of type functions is available in the GHC HEAD branch, and is documented at `http://haskell.org/haskellwiki/GHC/Type_families`.

In future work we would like to extend the decidable class of type functions. It seems that closed type functions would allow us to relax a number of current modularity restrictions. Moreover, they should allow for additional proof strength. For example, we cannot currently show that `Sum a Zero` $\sim$ `a` because `Sum` is an open function. Valid extensions of the function, like `Sum Int Zero = Bool`, do not satisfy this property.

The further comparison of functional dependencies and type functions is of great interest. We believe that type functions are a better choice, from the point of view of both language design and type checking.

Finally, the performance of the our type checking algorithm deserves further attention. In particular we should establish its worst and average time complexities, and the impact on programs that do not involve type functions.

Acknowledgments

We would like to thank Roman Leshchinskiy for his comments and for uprooting bugs in our preliminary implementation, and James Chapman for explaining the current state of Epigram.

Part of this work was conducted during an internship of Tom Schrijvers at Microsoft Research Cambridge.

References

1. L. Augustsson. Cayenne - a language with dependent types. In *Proc. of ICFP'98*, pages 239–250. ACM Press, 1998.

2. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
3. W. Boone. The word problem. *Annals of Mathematics*, 70:207–265, 1959.
4. M. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. In *ACM Symposium on Principles of Programming Languages (POPL'05)*. ACM, 2005.
5. M. M. T. Chakravarty, G. Keller, and S. P. Jones. Associated type synonyms. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 241–253, New York, NY, USA, 2005. ACM Press.
6. C. Chen and H. Xi. Combining programming with theorem proving. In *Proc. of ICFP'05*, pages 66–77. ACM Press, 2005.
7. S. Decorte, D. D. Schreye, and M. Fabris. Automatic inference of norms: a missing link in automatic termination analysis. In *ILPS '93: Proceedings of the 1993 international symposium on Logic programming*, pages 420–436, Cambridge, MA, USA, 1993. MIT Press.
8. I. S. Diatchki. *High-level abstractions for low-level programming*. PhD thesis, OGI School of Science & Engineering, May 2007.
9. T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, 1998.
10. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated Termination Proofs with AProVE. In *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA-04)*, volume 3091 of *Lecture Notes in Computer Science*, pages 210–220, Aachen, Germany, 2004.
11. R. Hinze. Fun with phantom types. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, pages 245–262. Palgrave Macmillan, 2003. ISBN 1-4039-0772-2 hardback, ISBN 0-333-99285-7 paperback.
12. M. P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming (ESOP 2000)*, number 1782 in Lecture Notes in Computer Science. Springer-Verlag, 2000.
13. D. Knuth and P. Bendix. Simple word problems in universal algebras. *Computational Problems in Abstract Algebra*, pages 263–297, 1970.
14. D. R. Licata and R. Harper. A formulation of Dependent ML with explicit equality proofs. Technical Report CMU-CS-05-178, Carnegie Mellon University Department of Computer Science, 2005.
15. C. McBride. Epigram: A dependently typed functional programming language. http://www.dur.ac.uk/CARG/epigram/.
16. M. Neubauer, P. Thiemann, M. Gasbichler, and M. Sperber. A functional notation for functional dependencies. In *Proceedings of the 2001 Haskell Workshop*, 2001.
17. S. Peyton-Jones et al. The Haskell Prime Report, 2007. Working draft.
18. S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17:1–82, Jan. 2007.
19. D. A. Plaisted and A. Sattler-Klein. Proof lengths for equational completion. *Inf. Comput.*, 125(2):154–170, 1996.
20. T. Schrijvers, M. Sulzmann, S. Peyton-Jones, and M. Chakravarty. Type checking for type functions. Draft report available from the authors, July 2007.
21. T. Sheard. Type-level computation using narrowing in Omega. In *Proceedings of the Programming Languages meets Program Verification (PLPV 2006)*, volume 174 of *Electronic Notes in Computer Science*, pages 105–128, 2006.
22. P. J. Stuckey and M. Sulzmann. A theory of overloading. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1–54, 2005.

23. M. Sulzmann, M. M. T. Chakravarty, S. P. Jones, and K. Donnelly. System F with type equality coercions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66, New York, NY, USA, 2007. ACM Press.
24. M. Sulzmann, T. Schrijvers, and P. J. Stuckey. Type inference for GADTs via Herbrand constraint abduction. Manuscript, July 2006.
25. M. Sulzmann, J. Wazny, and P.J.Stuckey. A framework for extended algebraic data types. In *Proc. of FLOPS'06*, volume 3945 of *LNCS*, pages 47–64. Springer-Verlag, 2006.

Transparant Ajax and Client-Site Evaluation of iTasks

– Draft Version –

Rinus Plasmeijer, Jan Martin Jansen, Pieter Koopman, and Peter Achten

Radboud University Nijmegen, Netherlands
rinus@cs.ru.nl, jm.jansen.04@nlda.nl, pieter@cs.ru.nl, p.achten@cs.ru.nl

Abstract

The iTask system is a combinator library written in Clean which allows the specification of multi-user workflow systems for the web. The iTask system generates forms that have to be filled in and submitted by the user. Each user has a set of tasks that can be processed in any order. The submission of a form might terminate existing tasks or create new tasks for the user herself as well as for other users. As a consequence a single event can cause a very complex state change on the server and can effect the work of many other users.

The advantages of using a browser as interface to a workflow system created with the iTask library is that no software has to be installed at the client site and that the look and feel of the GUI is familiar to every user. A drawback of this architecture is that the response might become rather slow when there are many users and many tasks. For each and every event on the client a message is sent to the server over the world wide web. The server processes the event and generates a new web-page for the user containing all her new tasks. For the calculation of the set of new tasks, the state of all other tasks has to be examined. Due to the delay of the world wide web and the creation, transportation and rendering of the complete new page by the browser, the response of a workflow system can become relatively slow.

In this paper we present two solutions for dealing with this performance problem.

First we introduce a combinator 'UseAjax' that cause the workflow system to use Ajax technology for handling a (sub)task. This has as consequence that only a part of the web page is updated instead of the creation, sending and rendering of an entire new page. The advantage of this extension is not only a smoother reaction in the browser on changes being made. Also the efficiency for large workflow systems is commonly improved in this way because most of the time only a smaller, for this (sub)task relevant, part of the current state needs to be recalculated.

For the definition of the workflow system a single annotation 'UseAjax' is sufficient. The implementation of this feature in the iTask library requires a Java script that runs on the client as well as a call-back function that handles the event. For the implementation this requires the possibility to store Clean functions temporarily in a web page as well as the possibility to store them in a persistent store at the server site such as in a file or in a database.

The second extension is another annotation, 'OnTheClient', which allows client site evaluation of tasks.

Since no call at all has to be made to the server when such a task is evaluated, there is no web communication overhead anymore as is the case when Ajax technology is being used. For the implementation of this feature one needs to be able to execute the tasks specified in Clean in the browser at the client site. We realize this with an interpreter for Clean code running in the browser. Therefore Clean is compiled to Sapl and this code is loaded into the browser together with the compact and efficient Sapl interpreter. Of course, code interpreted by the Sapl interpreter running in the browser is not as efficient as the execution of compiled Clean code at the server. So, there is also an efficiency penalty when 'OnTheClient' is chosen instead of 'UseAjax'.

By choosing one of the annotations, the programmer can define which evaluation method preferably should be used for a certain (set of) tasks.

Whenever evaluation 'OnTheClient' is not possible for some reason (e.g. when a database needs to be inspected on the server) the system can automatically decide to 'UseAjax' instead.

Static inference of non-monotonic polynomial sized types

Olha Shkaravska, Marko van Eekelen

Institute for Computing and Information Sciences
Radboud University Nijmegen
{shkarav, marko}@cs.ru.nl

Extended Abstract

This work is done in the context of the NWO-funded[1] AHA-project [5] which studies heap space analysis. In this paper we consider first-order function definitions over lists such that the sizes of outputs polynomially depend on the sizes of inputs. An example of such program is a standard definition for ++, appending two lists of the same type. Size dependencies we study are not necessarily linear, as in Pareto's approach [2], or monotonic as in [7] and for polynomial interpretations [1].

In [3] we introduced a type system with static type checking and with dynamic type inference based on the fact that a polynomial is defined by a finite number of points on its graph. Running tests of a given program on an appropriate set of inputs (see [6] for the principles of choice of inputs) one obtains a finite collection of test data that defines a system of linear equations. Its solution is the vector of coefficients of the polynomial expressing the hypothetical size dependency. There are several size polynomials if the output type is a nested list. The polynomials annotate corresponding underlying first-order types and these sized types are *checked* by a type-checker. If the type-checker rejects the first-order type, one continues the procedure for a polynomial of a higher degree.

The procedure non-terminates in two cases: either a program under consideration does not terminate on the proposed test inputs, or it is not well-typed.

In this paper we "lift" the non-termination issue from the level of a code to the level of types. This makes the approach uniform and we can control termination. Given a first-order function definition, a standard type-inference procedure ends up with a set of (recurrent) relations for size dependencies. In general, recurrences derived by a standard type-inference procedure may be non-linear and multivariate (see [4] for the example). It is very unlikely that there exists a solver that solves all types of them. However, once one assumes that the solution is a polynomial, one may statically "unfold" the recurrences to obtain values of the size functions(s), compute the coefficients and type-check the resulting polynomials.

[1] This project is sponsored by the Netherlands Organization for Scientific Research (NWO) under grantnr. 612.063.511.

The recurrence relations fully define the output size dependencies: if the function definition has a sized type, then its size annotations solves the recurrences, and, vice verse, if the recurrence has a solution then it annotates the output type. This implies, that *if the function definition does have a sized type with unique polynomial size annotations of given degrees, then the recurrence defines it uniquely, i.e. with this recurrence one is able to evaluate some points on the graph of the polynomial that define this polynomial uniquely.* Indeed, if the recurrences does not give enough points to define a polynomial of a given degree, then either it is of a lower degree, or the size annotations are not unique, or they are not polynomial.

To illustrate our idea, consider as an example the function `diff(x, y)` with the size polynomial $p(n, m) = n - m$ that non-terminates if $|y| > |x|$:

```
diff [] [] =[]
diff [] xs:ys = diff [] xs:ys (*non-terminating branch!*)
diff xs:ys [] = xs:ys
diff xs:ys xxs:yys = diff ys yys
```

To continue, we need to recall some ML-style typing rules. A typing judgment is a relation of the form $D;\ \Gamma\ \vdash_\Sigma e : \tau$ which means that if the free program variables of the expression e have the types defined by Γ, and the functions called have the types defined by Σ, and the size constraints D are satisfied, then e will be evaluated to a value of type τ, if it terminates. For example:

$$\frac{D \vdash p = p' + 1}{D;\ \Gamma,\ hd : \tau,\ tl : [\tau]^{p'}\ \vdash_\Sigma \mathsf{cons}(hd, tl) : [\tau]^{p}}\ \textsc{Cons}$$

$$\frac{\Gamma(x) = \mathtt{Int} \qquad D;\ \Gamma\ \vdash_\Sigma e_t : \tau \qquad D;\ \Gamma\ \vdash_\Sigma e_f : \tau}{D;\ \Gamma\ \vdash_\Sigma \mathsf{if}\ x\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f : \tau}\ \textsc{If}$$

$$\frac{\begin{array}{c} p = 0,\ D;\ \Gamma,\ x : [\tau']^p\ \vdash_\Sigma e_{\mathsf{nil}} : \tau \\ hd, tl \notin dom(\Gamma) \qquad D;\ \Gamma, hd : \tau',\ x : [\tau']^p,\ tl : [\tau']^{p-1}\ \vdash_\Sigma e_{\mathsf{cons}} : \tau \end{array}}{\begin{array}{l} D; \Gamma, x : [\tau']^p\ \vdash_\Sigma \mathsf{match}\ x\ \mathsf{with} \mid \mathsf{nil} \Rightarrow e_{\mathsf{nil}} \qquad\qquad : \tau \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{cons}(hd, tl) \Rightarrow e_{\mathsf{cons}} \end{array}}\ \textsc{Match}$$

Sized type checking eventually amounts to checking entailments of the form $D \vdash p = p'$, which means that $p = p'$ is derivable from D in the axiomatics of the ring of integers. Because p and p' are known polynomials of universally quantified size variables, comparing them by a *type-checker* is straightforward. A syntactical condition that prohibits let-bindings before pattern matching was shown to be necessary and sufficient to make type checking decidable for this system [3].

We want to infer `diff`'s sized type $[a]^n \times [a]^m \longrightarrow [a]^{(n-m)}$. Recurrence equations for its size function are

$$
\begin{aligned}
&p(0,0) = 0 && (1)\\
&p(0,m) = p(0,m) && (2)\\
&p(n,0) = n && (3)\\
&p(n,m) = p(n-1,m-1) && (4)
\end{aligned}
$$

Using, for instance, diagonal parsing of the space $\mathbf{N}^2$, we see that we can compute $p(0,0) = 0$, $p(1,\,0) = 1$ $p(1,\,1) = 0$. This is suffice to obtain $a,\ b,\ c$ for $p(x,y) = ax + by + c$. Solving the corresponding system

$$
\begin{aligned}
0a + 0b + c &= p(0,0) = 0\\
a + 0b + c &= p(1,0) = 1\\
a + b + c &= p(1,1) = 0
\end{aligned}
$$

gives $a = 1,\ b = -1,\ c = 0$. Note, that due to diagonal search the nodes satisfy the configuration that assures the uniqueness of the solution of the system of linear equations for $a,\ b,\ c$.

In the first version of the polynomial type-inference procedure we just use the knowledge that with careful use of the recurrent rules one can eventually compute enough points to define the size polynomial, if it exists. Using, for instance, diagonal ordering of points on $\mathbf{N}^2$, one adds more and more points to the set of terminating tests, and stops once the condition of the uniqueness of the solution of the system of linear equations is fulfilled [6]. This is checkable by finite search, but it is very exhaustive. We study possible optimisations of the procedure.

References

1. J-Y Marion and R. Péchoux. Resource analysis by sup-interpretation. In *FLOPS*, volume 3945 of *Lecture Notes in Computer Science*, pages 163–176. Springer, 2006.
2. L. Pareto. *Sized Types.* Chalmers University of Technology, 1998. Dissertation for the Licentiate Degree in Computing Science.
3. O. Shkaravska, R. van Kesteren, and M. van Eekelen. Polynomial size analysis for first-order functions. In S. Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications (TLCA'2007), Paris, France*, volume 4583 of *LNCS*, pages 351 – 366. Springer, 2007.
4. O. Shkaravska, R. van Kesteren, and M. van Eekelen. Polynomial size analysis of first-order functions. Technical Report ICIS-R07004, Radboud University Nijmegen, December 2007. `http://www.cs.ru.nl/icis/Research`.
5. M. van Eekelen, O. Shkaravska, R. van Kesteren, B. Jacobs, E. Poll, and S. Smetsers. AHA: Amortized Heap Space Usage Analysis. In Marco Morazán, editor, *Trends in Functional Programming 8: Selected Papers of the* 8^{th} *International Symposium on Trends in Functional Programming (TFP07), New York, USA*. Intellect Publishers, UK, 2007. to appear.
6. R. van Kesteren, O. Shkaravska, and M. van Eekelen. Inferring static non-monotonically sized types through testing. In Rachid Echahed, editor, 16^{th} *International Workshop on Functional and (Constraint) Logic Programming (WFLP07), Paris, France*, pages 123 – 139. CNAM, France, 2007.

7. P. B. Vasconcelos and K. Hammond. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In P. Trinder, G. Michaelson, and R. Peña, editors, *Implementation of Functional Languages: 15th International Workshop, IFL 2003, Edinburgh, UK, September 8–11, 2003. Revised Papers*, volume 3145 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag, Berlin, 2004.

Efficient, Modular Tries

Sebastian Fischer and Frank Huch
University of Kiel, Germany
{sebf,fhu}@informatik.uni-kiel.de

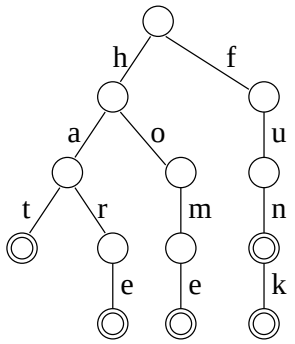
Abstract. Tries are known to be an efficient data structure for storing key value maps. Operations like insert, lookup, and delete can be implemented as linear algorithms with respect to the size of the key and independently of the size of the map.
However, all implementations of tries presented so far have drawbacks in practical applications. The paper analyzes the existing proposals of trie implementations, develops optimizations, such that the resulting implementation can compete with balanced search trees and in many cases is even faster. Finally, the paper presents a modularized implementation of tries, which makes use of multi-parameter classes with functional dependencies.

1 Introduction

Tries were originally invented by Axel Thue in 1912 and used to efficiently store strings, conf. [5]. The idea is to collect strings in a tree data structure, such that common prefixes are shared. The degree of branching within the tree is restricted to the size of the alphabet. For instance, the following set of strings

```
{hat, home, hare, fun, funk}
```

can be represented by the following trie, where the elements within the set are marked by a double circle:



The access to a string within the trie is linear with respect to the length (size) of the string.

In Haskell such a trie data structure can be implemented as follows[1]:

[1] For simplicity we restrict to an alphabet (`Sigma`) with only three letters.

```
data Sigma = A | B | C
type Str = [Sigma]

data SigmaT a = SigmaT (Maybe a) (Maybe a) (Maybe a)
                    -- for A     for B     for C

data StrT a = StrT (Maybe a) (Maybe (SigmaT (StrT a)))
              --   for []    for (:)
```

We define a trie data structure (`SigmaT` and `StrT`) for every involved data structure. Each trie data structure has a single constructor. The alternative constructors of the underlying data structure (e.g., `A`, `B`, `C`) are represented as arguments within the trie structure, since each of them may be present within the represented key set. Since a key can be present or not within a trie we wrap every argument of the trie structure with a `Maybe` type. If a constructor of the underlying data structure takes arguments, like `(:)` for `Str`, corresponding tries are nested, such that the argument values can be inserted into the trie successively.

For instance, a trie for the set of words {`[A]`,`[A,B]`,`[B]`} is represented by the following Haskell value:

```
StrT Nothing
     (Just (SigmaT (Just (StrT (Just ())
                               (Just (SigmaT Nothing
                                             (Just (StrT (Just ())
                                                         Nothing))
                                             Nothing))))
                   (Just (StrT (Just ())
                               Nothing))
                   Nothing))
```

Operations for manipulating tries will be given later in this paper.

A generalized variant of this trie structure was proposed in [1]. Similarly to strings, it allows arbitrary data structures as key within a trie. In this paper, we use a very simple tree structure to demonstrate the idea of generalized tries:

```
data Tree = Leaf | Node Tree Tree

data TreeT a = TreeT (Maybe a) (Maybe (TreeT        (TreeT a)))
               -- for Leaf      Node   left subtree  right subtree
```

The implementation uses non-uniform recursive types (a nested data type) which requires polymorphic recursion in functions over this type.

2 Tries by Okasaki

In his famous book "Purely Functional Data Structures" [6] Okasaki presents an implementation of string tries and generalized tries. The algorithms are presented as ML implementation using functors as well as a Haskell implementation using multi-parameter classes. In this paper we simplify his Haskell implementation such that special access functions are defined for every trie type and we do not

provide tries for polymorphic data types. This simplification will make it easier to analyze and optimize the implementation. We will later generalize our results with a class framework, although simpler than his framework, as well.

The idea behind Okasakis implementation is slightly different than presented in Section 1. He avoids the `Maybe` constructor for constructors with arity larger than zero within the trie data structure, i.e.:

```
data StrT a = StrT (Maybe a) (SigmaT (StrT a))
              --   for []    for (:)
```

In his framework, a trie implementation provides three functions:

- `empty` for the construction of an empty trie,
- `lup` for looking up the value a key is bound to, and
- `bind` to bind a key to a value.

For `SigmaT` the implementation of these functions is straight forward:

```
emptySigmaT :: SigmaT a
emptySigmaT = SigmaT Nothing Nothing Nothing

lupSigma :: Sigma -> SigmaT a -> Maybe a
lupSigma A (SigmaT ma _ _) = ma
lupSigma B (SigmaT _ mb _) = mb
lupSigma C (SigmaT _ _ mc) = mc

bindSigma :: Sigma -> a -> SigmaT a -> SigmaT a
bindSigma A v (SigmaT _ mb mc) = SigmaT (Just v) mb mc
bindSigma B v (SigmaT ma _ mc) = SigmaT ma (Just v) mc
bindSigma C v (SigmaT ma mb _) = SigmaT ma mb (Just v)
```

For `StrT` the implementation becomes a bit trickier.

```
emptyStrT :: StrT a
emptyStrT = StrT Nothing emptySigmaT

lupStr :: Str -> StrT a -> Maybe a
lupStr []     (StrT ma _) = ma
lupStr (a:as) (StrT _ sigmaT) = lupSigma a sigmaT >>= lupStr as
```

Looking up a nonempty `Str` successively looks up the two arguments of `(:)` in the corresponding sub-trie. For the first argument (of type `Sigma`), we use the corresponding `lup` function for `SigmaT` which returns a value of type `Maybe StrT`. For a successful lookup we continue looking up the remaining `Str`. This can be elegantly expressed by the bind operator `(>>=)` of the `Maybe` monad.

Binding a `Str` works similarly.

```
bindStr :: Str -> a -> StrT a -> StrT a
bindStr []     v (StrT _  sigmaT) = StrT (Just v) sigmaT
bindStr (a:as) v (StrT ma sigmaT) =
    let t  = maybe emptyStrT id (lupSigma a sigmaT)
        t' = bindStr as v t in
      StrT ma (bindSigma a t' sigmaT)
```

For a nonempty `Str`, we first look up the trie for the first letter (`t`), bind the remaining `Str` within this trie (`t"`), and finally bind the first letter to the modified trie. If the trie does not contain an entry for the first letter yet, then we create a new empty trie. This successively extends the trie for new keys.

Unfortunately, this implementation has some drawbacks, which become clear for the generalization of the presented technique to arbitrary key types. We consider a simple tree type to be used as keys within a trie:

```
data Tree = Empty | Node Tree Tree
```

Similar to the definitions for `Str` [6] proposes the following trie implementation for this data structure:

```
data TreeT a = TreeT (Maybe a) (TreeT (TreeT a))

emptyTreeT :: TreeT a
emptyTreeT = TreeT Nothing emptyTreeT

lupTree :: Tree -> TreeT a -> Maybe a
lupTree Empty      (TreeT lT nT) = lT
lupTree (Node l r) (TreeT lT nT) = lupTree l nT >>= lupTree r

bindTree :: Tree -> a -> TreeT a -> TreeT a
bindTree Empty      x (TreeT lT nT) = TreeT (Just x) nT
bindTree (Node l r) x (TreeT lT nT) =
    let t  = maybe emptyTreeT id (lupTree l nT)
        t' = bindTree r x t in
     TreeT lT (bindTree l t' nT)
```

The functions `lupTree` and `bindTree` recursively access the nested data type `TreeT` which requires polymorphic recursion, for which in general type inference is undecidable. Haskell 98 provides polymorphic recursion if a type signature is given, such that type checking is sufficient.

Unfortunately, with this code several disadvantages of the proposed trie implementation can be identified:

- The trie structure is infinite (conf. the definition of `emptyTreeT`). Hence, the implementation cannot be used in a strict language like ML. This problem occurs whenever a constructor of the key data type (here `Node`) is recursive in its first argument. The ML example in [6] avoids this problem by an additional value for `Node`s within the tree as a first argument.
  Although the infinite trie structure works fine in Haskell, it has another disadvantage, if one extends the trie signature with a function `unbind` to "delete" keys from the trie:
- A function for unbinding key could be defined as follows:

```
unbindTree :: Tree -> TreeT a -> TreeT a
unbindTree L       (TreeT lT nT) = TreeT Nothing nT
unbindTree (N l r) (TreeT lT nT) =
    TreeT lT (maybe nT (\t1 -> bindTree l (unbindTree r t1) nT)
                    (lupTree l nT))
```

With this function it is not possible to really "delete" keys from a trie. The trie structure does not shrink. We can only associate the unbound key to `Nothing`.
This can cause memory problems, e.g., when we encode a set and successively add and remove elements from the set.

– In worst-case, the function `bindTree` and `unbindTree` are exponential with respect to the key: for every `Node` its left sub-tree is looked-up and then bound, which means traversed two times. Since this occurs in every recursive call of `bindTree`, we obtain exponential worst-case complexity. The same holds for `unbind`.

To overcome these drawbacks we keep the additional `Maybe` constructor within the definition of the trie data structure. In a second step, we will optimize this structure and avoid these additional `Maybe` values in Section 4.

3 Tidy Tries

As a first step, we extend the trie structure with a `Maybe` type again, such that the trie can be terminated everywhere. This has several advantages:

– We can define linear `bind` and `unbind` and unbind operations,
– we have a finite representation for every trie which is also preferable for a strict language, and
– as a consequence we are able to check whether complete subtrees are empty when unbinding keys and cut off these sub-trees.

3.1 Linear Operations

As a first step, we improve the efficiency of binding and unbinding keys. Instead of looking up sub-tries and inserting these trees again we pass continuations through the trie, which finally modify the trie in the relevant positions. Instead of defining functions `bind` and `unbind` we can define a more general and flexible update function:

```
upTree :: Tree -> (Maybe a -> Maybe a) -> TreeT a -> TreeT a
upTree Leaf       up (TreeT lT nT) = TreeT (up lT) nT
upTree (Node l r) up (TreeT lT nT) =
  TreeT lT ((Just . upTree l (Just . upTree r up . ensureTreeT) .
             ensureTreeT) nT)

ensureTreeT :: Maybe TreeT -> TreeT
ensureTreeT Nothing = emptyTreeT
ensureTreeT (Just t) = t
```

The key idea is that we can construct a special update function for each level with the nested data type. The update function of the outer recursive call of `upTree` modifies a `TreeT` (it has type `Maybe (TreeT a) -> Maybe (TreeT a)`), while we use the original update function `up :: Maybe a -> Maybe a` in the inner recursive call.

The function `ensureTreeT` generates an empty trie for branches terminating with `Nothing`. This extends the trie structure for new keys. Additionally, the result of `upTree` has to be extended to a `Maybe` type by applying `Just`.

With this definition, we can easily implement Okasakis interface by means of the more general function `upTree`:

```
bindTree :: Tree -> a -> TreeT a -> TreeT a
bindTree t v = upTree t (const (Just v))

unbindTree :: Tree -> TreeT a -> TreeT a
unbindTree t = upTree t (const Nothing)
```

3.2 Collapsing Tries

As a next step we can optimize the `upTree` function such that when unbinding a value, empty sub-tries are collapsed. In other words, the trie structure is guaranteed to be as small as possible. If a key is deleted from the trie and there exists no larger tree with the same prefix within the trie, then the trie should contain a value of the form `TreeT Nothing Nothing`. Hence, instead of simply applying `Just` to this trie, we can collapse it in this case and return `Nothing` instead:

```
upTree (Node l r) up (TreeT lT nT) =
  tidyTreeT (TreeT lT ((tidyTreeT . upTree l (tidyTreeT . upDTree r up .
                                                ensureTreeT) .
                        ensureTreeT) nT))

tidyTreeT :: TreeT -> Maybe TreeT
tidyTreeT (TreeT Nothing Nothing) = Nothing
tidyTreeT t                       = Just t
```

Although this implementation via a more general function `upTree` is quite elegant it has some overhead to a direct definition of `bind` and `unbind`: if the user unbinds a key for which no binding exists, then a `Nothing` value for this key is first added to the trie and deleted again afterwards. This can be avoided with the definition of a special function `upDTree` for deleting values:

```
upDTree (Node l r) up (TreeT lT nT) =
  tidyTreeT (TreeT lT ((tidyTreeT .
                        upDTree l (liftToMaybe (tidyTreeT .
                                                upTree r up)))
                       nT))

liftToMaybe :: (TreeT a -> TreeT a) -> Maybe (TreeT a)
               -> Maybe (TreeT a)
liftToMaybe f Nothing   = Nothing
liftToMaybe f (Just tr) = Just (f tr)
```

The function `liftToMaybe` applies its functional argument (the recursive call) only to existing tries. Otherwise the recursion is directly stopped and `Nothing` is returned.

Similarly, we can avoid checking whether the tree can be collapsed when inserting a new or updating an existing value with the first definition of `upTree` which uses `Just` instead of `tidyTreeT`.

4 Avoid Maybes again

In comparison to Okasakis tries, the additional `Maybe` constructors consume more memory, as the following trie for the key `Node (Node Leaf Leaf) Leaf` shows:

```
TreeT Nothing
      (Just (TreeT Nothing
                   (Just (TreeT (Just (TreeT (Just TreeT (Just value))
                                                          Nothing)
                                              Nothing)
                                Nothing))))
```

The underlines constructors would be avoided in Okasakis implementation. The `Just` constructors do not occur at all, while the `Nothing` constructors, are represented by empty infinite trees. Is it possible, to avoid these additional constructors without losing the opportunities of our implementation?

As a solution, we can push the `Maybe` constructor into the following trie data type by adding a constructor for the empty trie:

```
data TreeT a = TreeT (Maybe a) (TreeT (TreeT a))
             | NoTreeT
```

Then the superfluous `Just` constructors can be omitted and we can use `NoTreeT` instead of `Nothing`. For the key `Node (Node Leaf Leaf) Leaf` we obtain the following trie:

```
TreeT Nothing
      (TreeT Nothing
             (TreeT (Just (TreeT (Just TreeT (Just value))
                                              NoTreeT)
                                 NoTreeT)
                    NoTreeT))
```

The functions for accessing the trie can be modified as follows:

```
lookupTree :: Tree -> TreeT a -> Maybe a
lookupTree _          NoTreeT      = Nothing
lookupTree Leaf (TreeT lT _)       = lT
lookupTree (Node l r) (TreeT _ nT) = lookupTree l treeT >>= lookupTree r

upTree :: Tree -> (Maybe a -> Maybe a) -> TreeT a -> TreeT a
upTree t up NoTreeT = tidyTreeT (upTree t up (TreeT Nothing NoTreeT))
upTree Leaf       up (TreeT lT nT) = tidyTreeT (TreeT (up lT) nT)
upTree (Node l r) up (TreeT lT nT) =
  tidyTreeT (TreeT lT (tidyTreeT (upTree l (treeTToMaybe . tidyTreeT .
                                            upTree r up . ensureTreeT)
                                          nT)))
```

```
tidyTreeT ::  TreeT a -> TreeT a
tidyTreeT (TreeT Nothing NoTreeT) = NoTreeT
tidyTreeT trie = trie

treeTToMaybe ::  TreeT a -> Maybe (TreeT a)
treeTToMaybe NoTreeT = Nothing
treeTToMaybe trie = Just trie
```

5 Combining Tries

Operations combining two tries can be defined quite easily by traversing both tries in parallel. For instance the union and the intersection of two tries can be defined as follows:

```
unionTreeT :: TreeT a -> TreeT a -> TreeT a
unionTreeT NoTreeT t = t
unionTreeT t NoTreeT = t
unionTreeT (TreeT lT1 nT1) (TreeT lT2 nT2) =
  TreeT (maybe lT2 (const lT1) lT1) (unionTreeT nT1 nT2)

intersectTreeT :: TreeT a -> TreeT a -> TreeT a
intersectTreeT NoTreeT t = NoTreeT
intersectTreeT t NoTreeT = NoTreeT
intersectTreeT (TreeT lT1 nT1) (TreeT lT2 nT2) =
  tidyTreeT (TreeT (lT2 >> lT1) (intersectTreeT nT1 nT2))
```

In both algorithms the values of the first trie are preferred.

6 Comparison with Balanced Search Trees

The most popular data structure in functional programming for efficiently storing key-value pairs are balanced search trees (*bst*), like also provided by the module `Data.Map` for the Glasgow Haskell Compiler [3]. Popular implementations are red-black-trees or AVL trees. Hence, we should compare our trie implementation with bsts.

It is not so straight forward, to compare the complexity of the manipulations of the two different map implementations. For simplicity, we first concentrate on `lookup`. In worst-case looking up a key in a bst takes $O(\log n)$ time, where n is the size of the bst, i.e., the number of keys already stored. In contrast to this, looking up a value in a trie is independent of the elements within the trie and takes $O(m)$, where m is the size of the key. Hence, both algorithms seem incomparable. But taking a closer look at the bst we see that also in the bst looking up a value is linear in the size of the key, since two keys have to be compared at every layer of the bst. We obtain $O(m \cdot \log n)$ for the bst. The trie is more efficient.

Unfortunately, this only holds for keys bound in the map. If a key is not bound, then the finite map can be more efficient than the trie. In worst-case (if the key is a prefix of another key bound in the trie), it takes $O(m)$ to detect that the key is not bound. In contrast, in the bst it may be the case that we detect the missing binding in $\log n$ steps, where only small parts of the keys were compared. Our experiments show, that in practice none of the algorithms is ahead in general.

For binding keys in the map, the situation is similar. If a key (or a similar key) is already bound, then the trie implementation is more efficient ($O(m)$ vs. $O(m \cdot \log n)$). If the key is new, then the size of the key does not matter in the bst implementation and the bst implementation is faster. Unbinding a key behaves similar to lookup.

Combining maps with operations like union, intersection, or subtraction is very simple and efficient in the trie implementation. Algorithms of the same complexity ($O(n+k)$, where n and k are the sizes of the two maps to be combined) can be defined for bsts. However, these algorithms are quite complex and the constants (e.g., for re-balancing the bst) slow down these operations in practice.

Our experiments so fare show, that it highly depends on the practical application which map implementation is faster. We are still working on good non-artifical benchmarks, resulting from practical applications and will hopefully finish them soon.

Besides efficiency, there are some more aspects we should consider when comparing bsts and tries. Maps are often used to encode sets. Unfortunately, bsts do not provide a unique representation of a set. Differently balanced trees can represent the same set. This makes it difficult to define equality or a total ordering for bsts. The only appropriate way seems to be a conversion to a list and using a lexicographical ordering on this list. Unfortunately, this is inefficient since this list has to be newly constructed for every comparison of two bsts. As a consequence, it is not possible to efficiently encode sets of sets with bsts. Using tries, this is no problem. Because there is only one unique trie representation for a set, we can define a trie which uses a trie data type as key.

Another important aspect is the practical usability of the data type. If a programmer wants to use bsts, then they only has to define an ordering (instance of class `Ord`) for their key type. For using tries, they has to define a complicated trie data structure (for the data type and every data type occurring in the data type) as well as complicate algorithms for every defined trie structure. Since the definition of such data types and algorithms is quite automatic Ralf Hinze proposed to use generic programming to generate these functions [4]. As a consequence, a trie implementation can be used almost as easily as bsts in a generic setting. Unfortunately, his generic trie implementation does not provide as efficient operations as we presented them in this paper. Furthermore, it requires a generic setting and is not directly applicable in Haskell. Therefore, we also developed a general class framework for trie implementations, which allows an elegant combination/definition of trie data structures.

Our class framework makes use of modern Haskell extensions, like multi-parameter classes and functional dependencies [2] and provides special abstractions that make the definition of trie operations as simple as possible. Generically deriving instances for user defined data types should be quite simple and could for instance be implemented by means of Template Haskell [7], which we plan for future work. Unfortunately, introducing a class context for tries has some overhead with respect to efficiency. It remains to check whether this is acceptable in practice.

References

1. Richard H. Connelly and F. Lockwood Morris. A generalization of the trie data structure. *Mathematical Structures in Computer Science*, 5(3):381–418, 1995.
2. Gregory J. Duck, Simon L. Peyton Jones, Peter J. Stuckey, and Martin Sulzmann. Sound and decidable type inference for functional dependencies. In David A. Schmidt, editor, *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004*, volume 2986 of *Lecture Notes in Computer Science*, pages 49–63. Springer, April 2004.
3. The Glasgow Haskell compiler. `\\http://www.haskell.org/ghc/`.
4. Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 10(4):327–351, 2000.
5. Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley Publishing Company, Boston, MA, USA, 1998.
6. Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, UK, 1998.
7. Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In *Haskell Workshop 2002*, October 2002.

FunSETL–Functional Reporting For ERP Systems

Michael Nissen and Ken Friis Larsen

Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark
michaeln@diku.dk and ken@friislarsen.net

Abstract. We present the FunSETL language which is a small functional language. The purpose of FunSETL is twofold. First, we want to describe a language that is suitable for generating reports in ERP systems. Second, we want the language to restricted enough so that we can perform automatic incrementalisation. In this paper we describe the current status of our project, which is that we have an implementation that we can use to make experiments with and we present the encouraging results of our preliminary experiments.

1 Introduction

Todays ERP systems like Microsoft Dynamics AX and Navision use *multi purpose* programming languages and/or SQL queries to express *reports*, where reporting essentially means computing (simple) functions on large amount of data. In Dynamics AX the multi purpose language X++ (see [1][p.91-118]) is used to express reports and in Navision C/AL (see [2]) is used. Both X++ and C/AL are state based (imperative) programming languages, which makes them even more unfit to express reports on data, since a *declarative* approach seams more intuitive. Increasing competition in the business field and rapidly growing amount of data in ERP systems has dictated the need for faster computation of reporting functions and *real time* access to the results of all reports, ie. real time access to *business intelligence*.

General purpose languages are used in ERP systems because sometimes SQL is not expressive enough to support a specific report. Therefore we suggest that ERP systems should contain a *domain specific* language in which to express reports, ie. a language that has the power of SQL and furthermore it should also be able to express the reports that SQL can not.

To increase the computation speed of reports, we would like to be able to make *automatic incrementalisation* of reporting functions. That is, transform the reporting function to equivalent functions, that does not need to traverse all data every time they are computed. This transformation should be completely automatic and transparent to the user. Thus, it becomes crucial to limit the reporting language to only the necessary constructs.

We have defined a small functional language called FunSETL with the following properties:

1. It is a *functional* language.
   We have opted for a functional (declarative) language, since the experience from SQL is that it seems more productive when expressing reports. Also, we believe, it makes it easier for non-programmers to learn.
2. FunSETL is *strongly normalizing*. That is, every program always terminates. Again we have borrowed a design objective from SQL. While it is a limitation of expressiveness, it is a desirable property. Simply because all reporting programs ought terminate with a result, hence the programming language might as guarantee this property.
3. It is possible to represent and iterative over large amount of data.
   This property is central, when we need to compute functions over large collections of data.

The above properties only apply to the *pure* FunSETL language. That is, it is possible for FunSETL programs to call external methods and programs, and thereby bypass the FunSETL properties. To ensure usability of the language any to be able to conduct experiments where we integrate with real systems we have integrated FunSETL with .NET by compiling FunSETL code to C# code.

2 Data Analysis

In an ERP system we need a module that can perform computational *tasks* e.g. data analysis on the data stored by the ERP system (ie. computing *reports*).

This section will describe how our data analysis module will look like, both from a users perspective and "under the hood" (internally). A very simplified conceptual model of the architecture of our ERP system is shown in figure 1, page 3.

The following sections will describe the architecture and how the data analysis module should interact with the users and database, and how it works internally.

2.1 Architecture

In the heart of the architecture we have a database of *events*, ie. a log of *everything* that has happened. Events can be committed by users (note that in this context users also can be other automated systems) and they are filtered by a module called *Filtering & Decoration*, which will not be described here, but the module either accepts an event and commits it to the database or rejects the event and informs the user.

When an event is committed to the database it is also passed to the data analysis module, ie. every newly committed event is passed to the analysis module when it is committed by the Filtering & Decoration module.

This means that the data analysis module has access to *all information* at commit time. Later we will get back to why this is important.

A user can interact in two ways with the Data analysis module. Either the user can commit a report to the module, which is then stored in the *report*

repository, or the user can request the result of a report computed on the current event database.

In our context a report is a program written in a *report language* called *FunSETL*, which is a *declarative* programming language. Section 3, page 4 will give a description of FunSETL.

Furthermore it should be noted that conceptually current ERP systems could be considered as the event database and the filtering and decoration module, and where we have a *monotone* growing event log. This means that the data analysis module can be considered as a *plug-in* for current ERP systems.

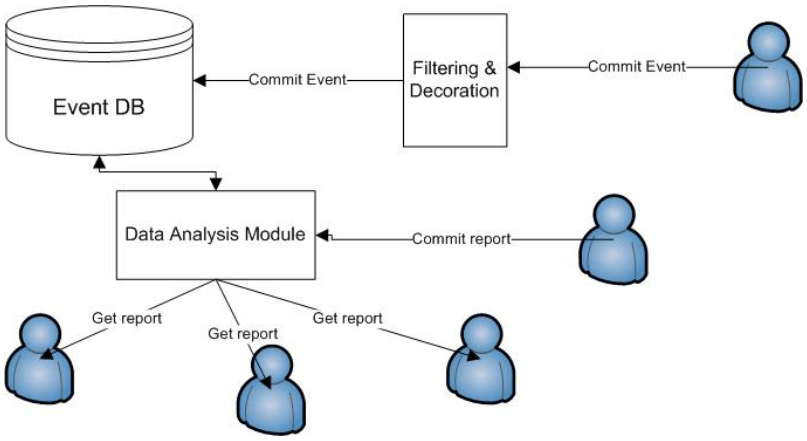


Fig. 1. Overview of Architecture

2.2 Data Analysis "under the hood"

The data analysis module contains a report repository where all reports that needs to be computed are stored.

When a report is committed it is then *automatically incrementalized*, ie. it is transformed into a new report, that hopefully will not need to traverse all the data in the event database every time it is computed. In Section 4, page 8 there will be an explanation of incrementalization and the advantages it gives us.

The analysis module then *incrementally* maintains in *real time* the results of the reports in the repository based on the events that are continuously added to the event database, and handed to the analysis module. This means that we have *real time* access to the results of all the reports.

Furthermore all the advanced incrementalization stuff is transparent to the user, since the user only sees the declarative specification of the report that he/she writes, and then the result of the report, when it is requested. The automatically incrementalized versions of the reports are only used internally in the

$$
\begin{aligned}
\tau &::= id \mid \textbf{bool} \mid \textbf{int} \mid \textbf{real} \mid \textbf{date} \mid \tau_1 + \tau_2 \mid \{lab_1 : \tau_1, \ldots, lab_k : \tau_k\} \mid \textbf{map}(\tau_1, \tau_2) \mid \textbf{mset}(\tau) \\
c &::= n \mid r \mid yyyy - mm - dd \mid \textbf{true} \mid \textbf{false} \\
binop &::= + \mid - \mid * \mid / \mid = \mid <= \mid < \mid \textbf{and} \mid \textbf{or} \mid \textbf{with} \mid \textbf{inter} \mid \textbf{union} \mid \textbf{diff} \mid \textbf{in} \mid \textbf{subset} \\
unop &::= \textbf{not} \mid \textbf{dom} \mid \textbf{toSet} \\
e &::= x \mid e_1\ binop\ e_2 \mid unop\ e \mid \textbf{inL}(e)\ \textbf{as}\ \tau \mid \textbf{inR}(e)\ \textbf{as}\ \tau \mid \\
&\quad \textbf{valL}(e) \mid \textbf{valR}(e) \mid \{lab_1 := e_1, \ldots, lab_k := e_k\} \mid \#lab(e) \mid f(e_1, \ldots, e_m) \mid \\
&\quad [\ ]\ \textbf{as}\ \tau \mid e[e'] \mid e[e_1 \rightarrow e_1'] \mid \\
&\quad \{\}\ \textbf{as}\ \tau \mid \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \mid \textbf{foreach}\ (a, b \rightarrow e_1)\ e_2\ e_3 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2\ \textbf{end} \\
fdecl &::= \textbf{fun}\ id(x_1 : t_1, \ldots, x_m : t_m) = e \\
p &::= fdecl_1 \ldots fdecl_k
\end{aligned}
$$

Where $TVar$ is the set of type names, Var is the set of identifiers, $FVar$ is the set of function identifiers, Z is the set of (syntactic) integers and R is the set of (syntactic) reals, $k \geq 1$, $m \geq 0$, $n \in Z$, $r \in R$, $x \in Var$, $f \in FVar$, and $id \in TVar$.

Fig. 2. FunSETL syntax

Data analysis module to provide real time access to the results of the reports in the repository.

3 The FunSETL Language

Here the syntax of the language is defined together with a type-system and semantics for type-correct programs. FunSETL is an explicitly typed language, such that static analysis should become easier.

3.1 Syntax

As said in Section 1, FunSETL is a declarative (functional) language, which means that the language does not contain *statements* but only *expressions*. The syntax of FunSETL is described in Figure 2.

Before we continue to the type-system and the semantics, let us make an *informal* description of the language constructs.

Simple Constants: n denotes integers, r denotes reals, $yyyy-mm-dd$ denotes dates and **true** and **false** are the boolean values.

Arithmetic operators: $e_1+e_2, e_1-e_2, e_1/e_2$ and e_1*e_2 are the usual arithmetic expressions the operators and can only be applied to integers and reals.

Logical operators: e_1 **and** e_2, e_1 **or** e_2 and **not** e has the usual semantics and these operators can only be applied to boolean expressions.

Comparison operators: $e_1 = e_2$ denotes equality of e_1 and e_2 and equality can only be applied to expressions of type integer, real, boolean or date. $e_1 <= e_2$ and $e_1 < e_2$ are the "less than equal" and "less than" operators and they can only be applied to integers and reals.

Sum-type construction/destruction: The **inL**(e) **as** τ and **inR**(e) **as** τ are the sum-type constructors. τ should have the form $\tau_1+\tau_2$ and **inL**(e) **as** $\tau_1+\tau_2$ constructs something of type $\tau_1+\tau_2$ if e has type τ_1. Symmetrically for **inR**(e). **valL**(e) returns the value v if e evaluates to **inL**(v). Symmetrically for **valR**(e).

Record construction/destruction: As usual $\{lab_1 := e_1, \ldots, lab_k := e_k\}$ denotes the construction of a record with fields $lab_1, \ldots, lab_k$ and $\#lab(e)$ returns the lab field of e if e evaluates to a record with a field named lab.

Function Application: As usual $f(e_1, \ldots, e_n)$ denotes the application of function f on arguments $e_1, \ldots, e_n$.

Multi-sets: $\{\}$ **as mset**(τ) denotes the construction of an empty multi-set, where added elements should have type τ. e_1 **with** e_2 means the resulting multi-set of adding element e_2 to the multi-set e_1. e_1 **inter** e_2, e_1 **union** e_2 and e_1 **diff** e_2 means the intersection, union and difference of multi-sets e_1 and e_2. e_1 **in** e_2 returns whether or not element e_1 is in multi-set e_2 and e_1 **subset** e_2 return whether or not multi-set e_1 is a subset of multi-set e_2.

Finite Map: [] **as map**(τ_1, τ_2) denotes the construction of an empty finite map from elements of type τ_1 to elements of type τ_2. $e[e']$ is the lookup operation on a finite map, ie. if e' evaluates to v' and e evaluates to a finite map with a binding on v then the binding of v is returned. $e[e_1 \rightarrow e_2]$ denotes the update operation on finite maps, ie. the finite map e is updated (overwritten if a binding already exists) with the binding of e_1 to e_1'. **dom**(e) returns a multi-set consisting of all elements in the domain of the finite map e and **toSet**(e) returns a multi-set of pairs, where each pair denotes a binding from argument to value in the finite map e.

Conditional: if e_1 **then** e_2 **else** e_3 denotes the usual conditional expression.

Iteration: foreach $(a, b \rightarrow e_1)$ e_2 e_3 is like the usual *fold-left* from SML (see [4][p. 145-148]). Hence $a, b \rightarrow e_1$ should be viewed as a *lambda* expression, ie. an anonymous function with arguments a and b and function body e_1. b is the accumulating parameter with starting value e_2 and the anonymous function is then folded over the multi-set e_3.

Let: let $x = e_1$ **in** e_2 **end** denotes the computation of e_1, where the result is bound to x and x can then be used in e_2.

Furthermore we define some extra constructs on finite maps, which are just syntactic sugar on some of the existing constructs:

Syntactic sugar: $[e_1 \rightarrow e_1', \ldots e_n \rightarrow e_n']$ **as map**(τ_1, τ_2) is just syntactic sugar for ([] **as map**(τ_1, τ_2))$[e_1 \rightarrow e_1'] \ldots [e_n \rightarrow e_n']$ and $e[e_1 \rightarrow e_1', \ldots, e_n \rightarrow e_n']$ is syntactic sugar for $e[e_1 \rightarrow e_1'] \ldots [e_n \rightarrow e_n']$.

A FunSETL program is a series of *function declarations*. In reality we would like a program to be a series of function declarations together with an expression, that can make use of the declared functions. So typically we will refer to a series of declarations and an expression to be a program, but in order to handle theoretical issues better the definition of a program is just a series of function declarations.

$$TInt\ \frac{}{\Gamma, \Delta \vdash n : \mathbf{int}}$$

$$TReal\ \frac{}{\Gamma, \Delta \vdash r : \mathbf{real}}$$

$$TVar\ \frac{}{\Gamma, \Delta \vdash x : \sigma(x)}(x \in \mathrm{Dom}(\sigma))$$

$$TDate\ \frac{}{\Gamma, \Delta \vdash yyyy - mm - dd : \mathbf{date}}(\mathrm{isdate}(yyyy, mm, dd))$$

$$TTrue\ \frac{}{\Gamma, \Delta \vdash \mathbf{true} : \mathbf{bool}}$$

$$TFalse\ \frac{}{\Gamma, \Delta \vdash \mathbf{false} : \mathbf{bool}}$$

$$TIf\ \frac{\Gamma, \Delta \vdash e_1 : \mathbf{bool} \quad \Gamma, \Delta \vdash e_2 : \tau \quad \Gamma, \Delta \vdash e_3 : \tau}{\Gamma, \Delta \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau}$$

$$TForeach\ \frac{\Gamma, \Delta \vdash e_2 : \tau \quad \Gamma, \Delta \vdash e_3 : \mathbf{mset}(\tau') \quad \Gamma, \Delta[a \to \tau', b \to \tau] \vdash e_1 : \tau}{\Gamma, \Delta \vdash \mathbf{foreach}\ (a, b \to e_1)\ e_2\ e_3 : \tau}$$

$$TLet\ \frac{\Gamma, \Delta \vdash e_1 : \tau' \quad \Gamma, \Delta[x \to \tau'] \vdash e_2 : \tau}{\Gamma, \Delta \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} : \tau}$$

$$TApp\ \frac{\Gamma, \Delta \vdash e_1 : \tau_1 \quad \ldots \Gamma, \Delta \vdash e_n : \tau_n}{\Gamma, \Delta \vdash f(e_1, \ldots, e_n) : \tau}(\Gamma(f) = ((\tau_1, \ldots, \tau_n), \tau)$$

$$TFunc\ \frac{\Gamma, [x_1 \to \tau_1, \ldots, x_n \to \tau_n] \vdash e : \tau}{\Gamma \vdash \mathbf{fun}\ f(x_1 : \tau_1, \ldots, x_n : \tau_n) = e : \Gamma[f \to ((\tau_1, \ldots, \tau_n), \tau)]}$$

$$TProg\ \frac{[] \vdash fdecl_1 : \Gamma_1 \quad \Gamma_1 \vdash fdecl_2 : \Gamma_2 \quad \ldots \quad \Gamma_{n-1} \vdash fdecl_n : \Gamma_n}{\vdash fdecl_1 \ldots fdecl_n : \Gamma_n}$$

Fig. 3. Type judgments for FunSETL

3.2 Type-system

Now that we have defined the syntax of the FunSETL language we would like to restrict the number of legal programs, by putting a *type system* on FunSETL programs. Figure 3 shows the type judgments for FunSETL. The judgment for typing of expressions has the form $\Gamma, \Delta \vdash e : \tau$, where Γ is an environment that describes the types of functions, Δ is an environment that described the types of variables and τ is the type of e.

There is not anything surprising in the typing judgments, except that in the $TProg$ rule only functions that are defined before a given function can be called from that function. That is, the type systems ensures that there is no recursion in the language.

3.3 Semantics

In this section we define the semantics of FunSETL programs, and furthermore we prove that all typeable FunSETL programs terminates with a *value*. The semantics is given as a *big-step operational* semantics.

$$SConst\ \frac{}{p, \delta \vdash c \Downarrow c}$$

$$SVar\ \frac{}{p, \delta \vdash x \Downarrow \delta(x)}(x \in Dom(\delta))$$

$$SIfTrue\ \frac{p, \delta \vdash e_1 \Downarrow \textbf{true} \quad p, \delta \vdash e_2 \Downarrow v}{p, \delta \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Downarrow v}$$

$$SIfFalse\ \frac{p, \delta \vdash e_1 \Downarrow \textbf{false} \quad p, \delta \vdash e_3 \Downarrow v}{p, \delta \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Downarrow v}$$

$$SLet\ \frac{p, \delta \vdash e_1 \Downarrow v' \quad p, \delta[x \rightarrow v'] \vdash e_2 \Downarrow v}{p, \delta \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} \Downarrow v}$$

$$SApp\ \frac{p, \delta \vdash e_1 \Downarrow v_1 \quad \ldots \quad p, \delta \vdash e_n \Downarrow v_n \quad p, [x_1 \rightarrow v_1, \ldots, x_n \rightarrow v_n] \vdash e_f \Downarrow v}{p, \delta \vdash f(e_1, \ldots, e_n) \Downarrow v}(\textbf{fun } f(x_1 : \tau_1, \ldots, x_n : \tau_n) = e_f \in p)$$

$$SForeach1\ \frac{}{p, \delta \vdash^{fold} \textbf{foreach } (a, b \rightarrow e_1)\ v\ \{\} \Downarrow v}$$

$$SForeach2\ \frac{p, \delta[a \rightarrow v_1, b \rightarrow v'] \vdash e_1 \Downarrow v'' \quad p, \delta \vdash^{fold} \textbf{foreach } (a, b \rightarrow e_1)\ v''\ \{v_2, \ldots, v_n\}}{p, \delta \vdash^{fold} \textbf{foreach } (a, b \rightarrow e_1)\ v'\ \{v_1, \ldots, v_n\}}$$

$$SForeach3\ \frac{p, \delta \vdash e_2 \Downarrow v' \quad p, \delta \vdash e_3 \Downarrow \{v_1, \ldots, v_n\} \quad p, \delta \vdash^{fold} \textbf{foreach } (a, b \rightarrow e_1)\ v'\ \{v_1, \ldots, v_n\} \Downarrow v}{p, \delta \vdash \textbf{foreach } (a, b \rightarrow e_1)\ e_2\ e_3 \Downarrow v}$$

Fig. 4. Operational semantics for FunSETL

First we need to define values, v. The syntax of FunSETL values are:

$$v ::= c \mid \textbf{inL}(v) \mid \textbf{inR}(v) \mid \{lab_1 := v_1, \ldots, lab_n := v_n\} \mid$$
$$[v_1 \rightarrow v'_1, \ldots, v_n \rightarrow v'_n] \mid \{v_1, \ldots, v_n\}$$

Figure 4 contains the judgments defining the operational semantics for FunSETL. The judgment for the operational semantics has the form $p, \delta \vdash e \Downarrow v$, which means that the expression e will evaluate to the value v in program context p and where δ is mapping from variables to values.

The intention in the definition of the semantics is that if $\vdash p : \Gamma$ and $\Gamma, \Delta \vdash e : \tau$ then for all mappings δ of variables to values of a certain type given by Δ (written $\delta \in Dom(\Delta)$) then there exists a unique v such that $p, \delta \vdash e \Downarrow v$ where v has type τ. This is proven formally in this section.

Now that we have defined the semantics of FunSETL programs, we need to make sure that it is a reasonable semantics, ie. that it is deterministic, typepreserving and as promised earlier that it is *strongly normalizing*.

Theorem 1 (The semantics is deterministic). *Assume* $\vdash p : \Gamma$ *and assume that* $\Gamma, \Delta \vdash e : t$ *then for all* $\delta \in Dom(\Delta)$ *we have*

$$p, \delta \vdash e \Downarrow v \wedge p, \delta \vdash e \Downarrow v' \quad \Rightarrow \quad v = v'$$

Proof: See B.1.

Before we can prove the theorem that every program terminates with a value, we need to introduce *call-graphs*.

Definition 1 (Call-graph). *Assume* $\vdash p : \Gamma$*, where* $p = fd_1 \ldots fd_n$*. Then the call-graph* $G = (V, E)$ *of* p *is defined as*

$V = \{fd_1, \ldots, fd_n\}$
$E = \{(fd_i, fd_j) | \textit{The body of } fd_i \textit{ contains a call to the function declared by } fd_j\}$

We now prove formally that FunSETL does not contain *recursion*, by showing that the call-graph of a type-correct program is a *directed acyclic graph* (DAG).

Theorem 2 (FunSETL call-graphs are DAGs). *Assume* $\vdash p : \Gamma$ *then the call-graph* G *of* p *is a DAG.*

Proof: See B.2.

We need these call-graphs to make an ordering on FunSETL expressions that will allow us to prove by induction on this ordering that every FunSETL program is strongly normalizing. To define this ordering we need to define the *call-height* of a FunSETL expression.

Definition 2 (Call-height). *Assume* $p = fd_1 \ldots fd_n$*,* $\vdash p : \Gamma$ *and* $\Gamma, \Delta \vdash e : \tau$ *then*

$\text{callheight}(e) = \max\{h \mid fd_i \textit{ is called from } e \textit{ and } h \textit{ is the height of the call-graph starting from node } fd_i\}$

Since we know that there is no recursion in FunSETL it seems reasonable to order expressions on the maximum number of function calls in any computation path and then secondary look at the size of the expression.

Definition 3 (Ordering on expressions). *Assume* $\vdash p : \Gamma$*,* $\Gamma, \Delta \vdash e : \tau$ *and* $\Delta', \Gamma \vdash e' : \tau'$*. Let as usual* $|e|$ *denote the syntactic size of the expression* e *then*

$$e \prec e' \quad \text{iff} \quad \text{callheight}(e) < \text{callheight}(e') \vee (\text{callheight}(e) = \text{callheight}(e') \wedge |e| < |e'|)$$

This enables us to prove the theorem:

Theorem 3 (Every program terminates with a value). *Assume* $\vdash p : \Gamma$ *and assume that* $\Gamma, \Delta \vdash e : \tau$*. Then for all* $\delta \in Dom(\Delta)$ *there exists* v *such that*

$$p, \delta \vdash e \Downarrow v \quad \wedge \quad \Gamma, [] \vdash v : \tau$$

Proof: See B.3.

4 Incremental functions

In this section we will see what is meant by a function's *incremental counterpart*. This will be described together with an example of an incremental program.

Furthermore we shall describe why incremental programs are interesting in relation to ERP systems.

4.1 Definition of incremental functions

Let us first define what is meant by an incremental function:

Definition 4. *Incremental function.*
Assume f is a function and $\oplus$ an update operation then a program f' that computes $f(x \oplus y)$ by making use of the value of $f(x)$ is called an incremental function (incremental version of f with respect to the update operation $\oplus$). Furthermore we also allow that the incremental version makes use of the original input. Hence we want the following implication to hold for all x, all y and all r:

$$r = f(x) \quad \Rightarrow \quad f'(x, y, r) = f(x \oplus y)$$

The definition above may be extended, because in some cases the intermediate results of the computation of $f(x)$ can be used in an incremental computation of $f(x \oplus y)$, but for now it is sufficient with the above definition.

Furthermore the definition says nothing about the running time of the functions and their incremental counterparts, but the idea is that by incrementalizing a function we should gain an *asymptotic speedup* in the computation time when we try to compute $f(x \oplus y)$ from $f(x)$.

Let us now see an example of function and its incremental counterpart.

Example 1. In Fig. 5 we see a function $SumClass1$ which as input takes a multiset of records with fields $accountnr, amount$ and $time$. $SumClass1$ computes the sum of all amounts of records satisfying that the accountnumber is between 1000 and 1999 and where time is between $2007-01-01$ and $2007-12-31$ (we name the conditions for being added to the sum the *filter* condition).

We want to incrementalize $SumClass1$ such that, if we add one element to the multi-set of integers then we can compute the new sum based on the filter condition and the old result. Ie. we incrementalize with respect to the update operation **with** on multi-sets and get the function seen in Fig 6.

```
1: type event = {accountnr : int, amount : real, time : date}
2:
3: fun filter(e : event) =
4:    let accnr = #accountnr(e) in
5:    let ti = #time(e) in
6:       1000 <= accnr and accnr <= 1999 and
7:       2007 − 01 − 01 <= ti and ti <= 2007 − 12 − 31
8:    end end
9:
10: fun SumClass1(events : mset(event)) =
11: foreach (event, sum =>
12:    if filter(event) then #amount(event) + sum else sum) 0.0 events
```

Fig. 5. Non-incremental financial function

10

```
 1: type  event = {accountnr : int, amount : real, time : date}
 2:
 3: fun  filter(e : event) =
 4:    let accnr = #accountnr(e) in
 5:    let ti = #time(e) in
 6:       1000 <= accnr and accnr <= 1999 and
 7:       2007 − 01 − 01 <= ti and ti <= 2007 − 12 − 31
 8:    end end
 9:
10: fun SumClass1′(events : mset(event), enew, sum) =
11:    if filter(enew) then #amount(enew) + sum else sum
```

Fig. 6. Incremental financial function

Let us discuss the example *informally*. Using any reasonable measure of running time of FunSETL programs, we would expect that the $SumClass1$ function has running time proportional to the number of elements in the multi-set (because of the **foreach** operation). Furthermore we would expect that the $SumClass1'$ function can be computed in constant time, since we have eliminated the **foreach** operation, ie. we have gained an asymptotic speedup in the computation time when trying to compute $SumClass1(x$ **with** $y)$ if we already know the value of $SumClass1(x)$ (otherwise we have gained nothing).

4.2 Automatic incrementalization

As we saw above, we could hand code an incremental version of a function with respect to some update operation, but it would be really interesting, if we could *automatize* the process of incrementalization of FunSETL functions.

There has been made a lot of work in the field of incrementalization of programs, where *semantics preserving transformations* are used to construct the incremental counterpart of a function. Among the most interesting work can be mentioned [5] and [6].

In [5] there is a description of an incrementalization process for a functional language with mutual recursion, ie. a language which is more general that FunSETL. The incrementalization process is split in three steps:

(i) *Caching all intermediate results.*
In the hand coded example we saw earlier, we did not use any intermediate results, but they can often be used when trying to make a more efficient version of a program, fx. when computing the average of a multi-set of integers. When adding an element to the multi-set we are not able to compute the new average from the old average and the recently added element. Hence this part of the process produces from a function f a new function $\hat{f}$, which returns the return value of f together with the intermediate results computed by f.

(ii) *Incrementalization*

This step incrementalizes the function $\hat{f}$ from the previous step, with respect to an update operation $\oplus$. This step is performed by doing semantics preserving program transformations and by using the intermediate results. The result of this stage is called $\hat{f}'$.

(iii) *Pruning*
In this stage the intermediate results, which are not used by $\hat{f}'$ will be removed producing a new function f', which is a incremental version of f returning only the necessary intermediate results to compute efficiently.

The incrementalization process incrementalizes a program with respect to a given operation.

Steps (i) and (iii) can be made fully automatic, but step ii is a bit more involved. Depending on how powerful the incrementalizer engine will be made it will vary from automatic to semi-automatic. Our hope is that because we have chosen a language without full recursion then it will be possible to make an incrementalizer which is both powerful and fully automatic.

4.3 Incremental functions in ERP systems

In this section we will discuss why incremental computations is interesting in relation to ERP systems.

ERP systems contains a lot of reports that presents the computation of some analytical/financial function. Typically these functions are computed by iterating over large collections of data (usually all the rows from a big database table), and it is suitable to make incremental versions of functions like these.

In section 5 we will see an implementation of the Microsoft Dynamics AX financial statement in FunSETL. There are many advantages when using incremental computations and automatic incrementalization. Among the most important we have:

- Efficiency improvement on programs.
- Simpler looking programs.
- Reduce the number of errors induced by humans.
- Gain reduction in programming time.

As we have seen in the small example, a program running in linear time have an incremental counterpart, which run in constant time. This seems to be the case with many computations in ERP systems, since many reports does not have internal dependencies between the data when a report is computed. This means that, if the incremental counterpart is used we can get *real time* computations of many reports in ERP systems, which today is usually computed by a nightly bachrun. Hence if the incremental programs are used, one will also be able to set *alerts* and put *triggers* on reports, since they are computed in *real time* and *on the fly*.

Furthermore if automatic incrementalization techniques are used, another side-effect will be that we get simpler looking programs, because the *incrementalizing* software can be used as a front-end to the compiler and hence we do not need to look at the incrementalized code.

Automatic incrementalization will probably also lead to fewer programming errors and a reduction in programming time, since the non-incremental programs are often easier to write.

5 Application of FunSETL

This section will show how the Financial statement from Microsoft Dynamics AX can be implemented in FunSETL.

5.1 Microsoft Dynamics AX Financial Statement

The Microsoft Dynamics AX ERP system contains a database with a lot of tables, and the Financial statement makes use of only a couple of these tables.

The Financial statement is in principle an aggregate function, which aggregates information on different financial accounts. Dynamics AX has an accounting system where accounts are numbered from 0 and up. The financial statement computes the following information

- Sumclass computations (balance of account intervals $X000 - X999$, where $X = 1, 2, 3, 4, 5, 6, 7, 8$ and from 9000 and up. These summation data are named $SumclassX$ for each X.
- Assets and liabilities
- Other summation data.

All these numbers have in common that they are aggregate information on the Microsoft Dynamics AX table, which contains all transactional data on the different accounts in the system.

5.2 Implementation of Financial Statement in FunSETL

In Section C, page 19 we see an implementation of the *financial statement function* which is computed by Microsoft Dynamics AX 4.0.

The FunSETL implementation is based on the code which can be viewed from *Microsoft Dynamics Application Object Tree* and furthermore the implementation has been made, such that it was possible for a demo version of our incrementalization engine to incrementalize the program.

The incrementalization engine we have been using so far requires, that the code is written in a specific way in order to make the *automatic incrementalization*. Therefore the code is not in the shortest and simplest form. The incrementalized version of the code is not included, since it is almost unreadable and longer than the code of the Financial statement and it is written in an earlier version of the FunSETL syntax.

Furthermore we have made a couple of simplifications, since the database table in Dynamics AX 4.0 containing all the transactional data contains more columns than necessary when computing the Financial statement. Therefore we have made a projection on the transaction table, such that each transaction only

contains $accountnr, amount$ and $date$. $accountnr$ is the number of the account where there has been made a transaction of $amount$ on $date$. Actually the example in Section 4 (a non incremental and incremental version of a function named $SumClass1$) is a part of the Financial statement, where we have hard-coded the interval dates.

5.3 The Experiment

This section will discuss an experiment on the performance of the Financial statement. To make the experiment we have retrieved a database with 200.000 events from a real company.

The experiment contains two parts

1. Computing the Financial statement on a database with 100.000 events and comparing the results of the non-incremental and incremental version of the Financial statement.
2. Adding 10 events to database containing 100.000 and recomputing the Financial statement after adding every event and then comparing the results of the non-incremental and incremental version.

ad 1: In figure 7 we see the number of different FunSETL operations used to compute the Financial statement on a database with 100.000 events and the quotient between the numbers. As we can see the incremental version of the program is slower than the non-incremental version. When incrementalizing the Financial statement we introduce some overhead, because the result of the Financial statement is available between each event in the incremental case, where as it is only available in the non-incremental case after the 100.000 events and it instantly becomes obsolete, when there is added a new event to the database.

Operation	non-incremental	incremental	inc. / non-inc
Arithmetic	13.291.113	18.991.112	1,42
Record	10.082.682	14.382.682	1,42
Map	198.677	1.198.677	6,03
Control	5.296.707	8.596.707	1,62

Fig. 7. Financial statement on 100K event database

ad 2: In figure 8 we see the number of FunSETL operations used when adding 10 events to the database with 100.000 events and then computing the result of the Financial statement between every addition of an event. The number of operations used in the non-incremental program is around 11 times as big as the number of operations used, when computing the Financial statement on the database consisting of 100.000 events. This is not surprising because the non-incremental version will run through all the data every time the Financial statement is computed, since it has running time linear in the number of events.

The number of operations used in the incremental program, when adding 10 events, is not many more than the number of operations used, when computing the Financial statement on the database with 100.000 events. This is also not surprising, because we expect that the incremental version only need constant time to recompute the Financial statement, when we add one event to the database.

From the quotient between the number of operations used in the non-incremental and incremental programs we can see, that there is a slight overhead in maintaining the Financial statement incrementally. However, if we want to query the ERP system for the Financial statement more than once, we see that the number of operations used increases dramatically, if we not use the incremental version. Hence, we are able to benefit from maintaining the result incrementally.

If we compare our implementation of the Financial statement to the implementation given in Dynamics AX, we can see that the Dynamics AX code has the same time-complexity as the non-incremental version of the Financial statement implemented in FunSETL. Hence, there is really room for improvement in the time usage! But our program is a bit bigger than the implementation in Dynamics AX. We would be able to express the Financial statement in fewer lines, but then we were not able to automatically incrementalize the code, ie. we need to make a much better automatic incrementalization engine.

Operation	non-Incremental	Incremental	Inc. / non-Inc
Arithmetic	146.209.118	18.992.932	0,13
Record	110.915.057	14.384.122	0,13
Map	2.185.557	1.198.797	0,55
Control	58.266.692	8.597.567	0,15

Fig. 8. Adding 10 events to 100K event database

6 Summary & Future work

In this paper we have given a rough sketch of the architecture of the next generation of ERP systems.

Particularly we have been interested in the *Data analysis* aspect of the ERP system, and how it should work. Therefore this paper proposes the next generation of ERP systems should have the following properties:

- There system should contain a domain specific language to express reports, and the language should be powerful enough to eliminate the need for general purpose languages.
  This paper proposes the language FunSETL described in Section 3, which is thought to have the desired properties. In section 5 we made an empirical

study and implemented an existing ERP system report using FunSETL and used the implementation of a real dataset.
- The system should support real time access to all business intelligence. This is the most important property of the system. Microsoft Navision contains some aspects of this, since some reports are maintained using the SIFT technology. We have proposed a generalization of the SIFT technology.

Currently we have an implementation of a compiler from FunSETL to C#, ie. FunSETL runs on .NET. The next steps in the development of the next generation ERP system will be

- Implement a prototype of the architecture.
- Futher develop the incrementalization algorithm.
- Improve the userbility of the language

Acknowledgements

Many of the idead presented in this article have been developed in close cooperation with Professor Fritz Henglein. Also, Daniel Brixensen's master thesis[3] paved the road for us by mking the first version of FunSETL and enabling us to perform the experiments presented in Section 5.

A Literature

References

1. [GPO+06] Inside Microsoft Dynamics AX 4.0
   Arthur Greef, Michael Fruergaard Pontoppidan, Lars Dragheim Olsen, and experts from the Microsoft Dynamics AX team
   ISBN: 0-7356-2257-4
   Microsoft Press
2. http://www.consultec.es/DocTutoriales/Introduction_to_CAL_Programming.pdf
3. [DB05] Inkrementelle Metoder til REA-Baseret Rapportering
   Daniel Brixen
   Master Thesis, Department of Computer Science, University of Copenhagen
   http://www.charme.dk/detkanendatalog/Specialer/danielbrixenspeciale.pdf
4. [HR99] Introduction to Programming Using SML
   Michael R. Hansen & Hans Rischel
   ISBN: 0-20139820-6
   Biddles Ltd., Guildford and King's Lynn
5. [LST98] Static Caching for Incremental Computation
   Yanhong A. Liu, Scott D. Stoller & Tim Teitelbaum
   ISSN: 0164-0925
   ACM Press
6. [RP81] Formal Differentiation: A Program Synthesis Technique
   Robert Paige
   ISBN: 0-83571213-3
   UNI Research Press

B Proofs

B.1 The semantics is deterministic

Theorem 4. *The semantics is deterministic.*
Assume $\vdash p : \Gamma$ *and assume that* $\Gamma, \Delta \vdash e : t$ *then for all* $\delta \in Dom(\Delta)$ *we have*

$$p, \delta \vdash e \Downarrow v \wedge p, \delta \vdash e \Downarrow v' \quad \Rightarrow \quad v = v'$$

Proof: Not done yet, and requires that the functions used i **foreach** expressions are commutative.

B.2 Call-graph is a Dag

Theorem 5. *FunSETL call-graphs are DAGs.*
Assume $\vdash p : \Gamma$ *then the call-graph* G *of* p *is a DAG.*

Proof: Induction on $|p|$ ($|p|$ is the number of function declarations in p)
$|p| = 0$: Trivial, since the typesystem does not allow programs without any function declarations. This completes the basis case of the induction.
$|p| > 0$: Write $p = fd_1 \dots fd_n$. Since $\vdash p : \Gamma$ we have a derivation

$$\frac{[] \vdash fd_1 : \Gamma_1 \quad \dots \quad \Gamma_{n-1} \vdash fd_{n-1} : \Gamma_{n-1} \quad \Gamma_{n-1} \vdash fd_n : \Gamma}{\vdash fd_1 \dots fd_n : \Gamma}$$

Hence for $p' = fd_1 \dots fd_{n-1}$ we can make the derivation

$$\frac{[] \vdash fd_1 : \Gamma_1 \quad \dots \quad \Gamma_{n-2} \vdash fd_{n-1} : \Gamma_{n-1}}{\vdash fd_1 \dots fd_{n-1} : \Gamma_{n-1}}$$

By induction we get that the call-graph of p' is a DAG.

To see that the call-graph of p is a DAG we only need to make sure that the body of fd_n only contains calls to the functions declared in $fd_1 \dots fd_{n-1}$ and not to itself.

But since we have a derivation of $\Gamma_{n-1} \vdash fd_n : \Gamma_n$ and since Γ_{n-1} can not contain a binding of the function declared in fd_n then the function declared in fd_n can not contain a call to itself. Hence the call-graph of p is a DAG. ■

B.3 The semantics is strongly normalizing

Theorem 6. *Every program terminates with a value.*
Assume $\vdash p : \Gamma$ *and assume that* $\Gamma, \Delta \vdash e : \tau$. *Then for all* $\delta \in Dom(\Delta)$ *there exists* v *such that*

$$p, \delta \vdash e \Downarrow v \quad \wedge \quad \Gamma, [] \vdash v : \tau$$

Proof: By induction on the relation $\prec$.
We split the proof depending on the structure of e.
$e = c$: The theorem trivially holds, since all constants c are values.
$e =$ **if** e_1 **then** e_2 **else** e_3: Since $\Gamma, \Delta \vdash$ **if** e_1 **then** e_2 **else** e_3 we have derivations of:

$$\Gamma, \Delta \vdash e_1 : \textbf{bool}$$
$$\Gamma, \Delta \vdash e_2 : \tau$$
$$\Gamma, \Delta \vdash e_3 : \tau$$

Because $e_1 \prec e$,$e_2 \prec e$ and $e_3 \prec e$ we get by induction that for all $\delta \in Dom(\Delta)$ there exists v_1, v_2 and v_3 such that

$$p, \delta \vdash e_1 \Downarrow v_1 \qquad \Gamma, [] \vdash v_1 : \textbf{bool}$$
$$p, \delta \vdash e_2 \Downarrow v_2 \qquad \Gamma, [] \vdash v_2 : \tau$$
$$p, \delta \vdash e_3 \Downarrow v_3 \qquad \Gamma, [] \vdash v_3 : \tau$$

We now split the proof depending on v_1.
$v_1 =$ **true**: Hence we can make the derivation

$$\frac{p, \delta \vdash e_1 \Downarrow \textbf{true} \quad p, \delta \vdash e_2 \Downarrow v_2}{p, \delta \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Downarrow v_2}$$

Since $\Gamma, [] \vdash v.\tau$ we have completed this sub-case.
$v_2 =$ **false**: Symmetric, which completes this case.
$e = f(e_1, \ldots, e_n)$: Since $\Gamma, \Delta \vdash e : \tau$ the derivation must have the form

$$\frac{\Gamma, \Delta \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma, \Delta \vdash e_n : \tau_n}{\Gamma, \Delta \vdash f(e_1, \ldots, e_n) : \tau}(\Gamma(f) = ((\tau_1, \ldots, \tau_n), \tau))$$

Since $\vdash p : \Gamma$ we must have **fun** $f(x_1 : \tau_1, \ldots, x_n : \tau_n) = e' \in P$. By Theorem 2 the call-graph of p is a DAG and hence

$$\text{callheight}(e') < \text{callheight}(f(e_1, \ldots, e_n))$$

This means that $e' \prec f(e_1, \ldots, e_n)$. Since $\vdash p : \Gamma$ we have a derivation of

$$\Gamma' \vdash \textbf{fun } f(x_1 : \tau_1, \ldots, x_n : \tau_n) = e' : \Gamma'[f \rightarrow ((\tau_1, \ldots, \tau_n), \tau)] \tag{1}$$

for some $\Gamma' \subset \Gamma$. (1) gives us a derivation of

$$\Gamma', [x_1 \rightarrow \tau_1, \ldots, x_n \rightarrow \tau_n] \vdash e' : \tau.$$

Since $\Gamma' \subset \Gamma$ then clearly

$$\Gamma, [x_1 \rightarrow \tau_1, \ldots, x_n \rightarrow \tau_n] \vdash e' : \tau \tag{2}$$

because we must be able to prove the same, when adding more function declarations to the environment (this will however not be proven formally, since it is

intuitively clear).
Clearly callheight$(e_i) \leq$ callheight(e) and since $|e_i| < |e|$ for $i = 1, \ldots, n$ we get that $e_i \prec e$ for $i = 1, \ldots, n$. Hence by induction we get that there exist $v_1, \ldots, v_n$ such that

$$p, \delta \vdash e_i \Downarrow v_i \quad \Gamma, [] \vdash v_i : \tau_i \quad \text{for } i = 1, \ldots, n$$

Using induction on (2) we get that for all $\delta' \in Dom([x_1 \rightarrow \tau_1, \ldots, x_n \rightarrow \tau_n])$ there exists v such that

$$p, \delta' \vdash e' \Downarrow v \wedge \Gamma, [] \vdash v : \tau$$

Use $\delta' = [x_1 \rightarrow v_1, \ldots, x_n \rightarrow v_n]$, ie. there exists v such that

$$p, [x_1 \rightarrow v_1, \ldots, x_n \rightarrow v_n] \vdash e' \Downarrow v \quad \Gamma, [] \vdash v : \tau$$

Hence we can make the derivation

$$\frac{p, \delta \vdash e_1 \Downarrow v_1 \quad \ldots \quad p, \delta \vdash e_n \Downarrow v_n \quad p, [x_1 \rightarrow v_1, \ldots, x_n \rightarrow v_n] \vdash e' \Downarrow v}{p, \delta \vdash f(e_1, \ldots, e_n) \Downarrow v} (\textbf{fun } f(x_1 : \tau_1, \ldots, x_n : \tau_n) = e' \in p)$$

Since we already know that $\Gamma, [] \vdash v : \tau$ we have completed this case.
$e =$ **let** $x = e_1$ **in** e_2 **end**: Since $\Gamma, \Delta \vdash$ **let** $x = e_1$ **in** e_2 **end** $: \tau$ we have derivations of

$$\Gamma, \Delta \vdash e_1 : \tau'$$
$$\Gamma, \Delta[x \rightarrow \tau'] \vdash e_2 : \tau$$

for some τ'. Since $e_1 \prec e$ and $e_2 \prec e$ we get by induction that there exist v_1 and v_2 such that

$$p, \delta \vdash e_1 \Downarrow v_1 \qquad \Gamma, [] \vdash v_1 : \tau'$$
$$p, \delta[x \rightarrow v_1] \vdash e_2 \Downarrow v_2 \qquad \Gamma, [] \vdash v_2 : \tau$$

Hence we can make the derivation

$$\frac{p, \delta \vdash e_1 \Downarrow v_1 \quad p, \delta[x \rightarrow v_1] \vdash e_2 \Downarrow v_2}{p, \delta \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} \Downarrow v_2}$$

Since we already know that $\Gamma, [] \vdash v_2 : \tau$ we have completed this case.
$e =$ **foreach** $(a, b \rightarrow e_1)$ e_2 e_3: Since $\Gamma, \Delta \vdash$ **foreach** $(a, b \rightarrow e_1)$ e_2 $e_3 : \tau$ we get that there is derivations of

$$\Gamma, \Delta \vdash e_2 : \tau$$
$$\Gamma, \Delta \vdash e_3 : \textbf{mset}(\tau')$$
$$\Gamma, \Delta[a \rightarrow \tau', b \rightarrow \tau] \vdash e_1 : \tau$$

for some τ'. Since $e_1 \prec e, e_2 \prec e$ and $e_3 \prec e$ we get that there exist v_2 and v_3 such that

$$p, \delta \vdash e_2 \Downarrow v_2 \qquad \Gamma, [] \vdash v_2 : \tau \tag{3}$$
$$p, \delta \vdash e_3 \Downarrow v_3 \qquad \Gamma, [] \vdash v_3 : \textbf{mset}(\tau') \tag{4}$$

and for all $\delta' \in Dom(\Delta[a \to \tau', b \to \tau])$ there exists v such that

$$p, \delta' \vdash e_1 \Downarrow v_1 \quad \Gamma, [] \vdash v_1 : \tau \tag{5}$$

Since $\Gamma, [] \vdash v_3 : \mathbf{mset}(\tau')$ we must have $v_3 = \{v'_1, \ldots, v'_n\}$ for some $v'_1, \ldots, v'_n$. We now show by induction on n that

$$\forall \overline{v}.\ \Gamma, [] \vdash \overline{v} : \tau\ \ \exists v.\ \Gamma, [] \vdash v : \tau \wedge p, \delta \vdash^{fold} \mathbf{foreach}\ (a, b \to e_1)\ \overline{v}\ \{v'_1, \ldots, v'_n\} \Downarrow v \tag{6}$$

$n = 0$: Assume we are given $\overline{v}$ such that $\Gamma, [] \vdash \overline{v} : \tau$. Hence we can make the derivation

$$\frac{}{p, \delta \vdash^{fold} \mathbf{foreach}\ (a, b \to e_1)\ \overline{v}\ \{\} \Downarrow \overline{v}}$$

Since we have assumed that $\Gamma, [] \vdash \overline{v} : \tau$ we have completed the basis case of the induction.
$n > 0$: Assume we are given $\overline{v}$ such that $\Gamma, [] \vdash \overline{v} : \tau$. By (5) we get for some v_1 that

$$p, \delta[a \to v'_1, b \to \overline{v}] \vdash e_1 \Downarrow v_1 \quad \Gamma, [] \vdash v_1 : \tau$$

By induction we get that there exist v such that

$$p, \delta \vdash^{fold} \mathbf{foreach}\ (a, b \to e_1)\ v_1\ \{v'_2, \ldots, v'_n\} \Downarrow v \quad \Gamma, [] \vdash v : \tau'$$

Ie. we can make the derivation

$$\frac{p, \delta[a \to v'_1, b \to \overline{v}] \vdash e_1 \Downarrow v_1 \quad p, \delta \vdash^{fold} \mathbf{foreach}\ (a, b \to e_1)\ v_1\ \{v'_2, \ldots, v'_n\} \Downarrow v}{p, \delta \vdash^{fold} \mathbf{foreach}\ (a, b \to e_1)\ \overline{v}\ \{v'_1, \ldots, v'_n\} \Downarrow v}$$

which completes the proof by induction of (6).
Combining (3),(4) and (6) we get that there exist v such that we can make the derivation

$$\frac{p, \delta \vdash e_2 \Downarrow v_2 \quad p, \delta \vdash e_3 \Downarrow \{v'_1, \ldots, v'_n\} \quad p, \delta \vdash^{fold} \mathbf{foreach}\ (a, b \to e_1)\ v_2\ \{v'_1, \ldots, v'_n\}}{p, \delta \vdash \mathbf{foreach}\ (a, b \to e_1)\ e_2\ e_3 \Downarrow v}$$

where $\Gamma, [] \vdash v : \tau$, which completes this case of proof. ■

C Financial statement

```
1  type event = {accountnr : int, amount : real, time : date}
   type eventset = mset(event)
   type interval = {startdate : date, enddate : date}
   type classmap = map(int,real)

6  fun inInterval(e : event, I : interval) =
       #startdate(I) <= #time(e) and #time(e) <= #enddate(I)

   fun aktiv(e : event, I : interval) =
       let
11         accnr = #accountnr(e)
       in
           if ((0 <= accnr and accnr <= 449) or
```

```
                    (451 <= accnr and accnr <= 539) or
                    (541 <= accnr and accnr <= 679) or
                    (681 <= accnr and accnr <= 2999)) and
                   inInterval(e,I)
                then #amount(e)
                else 0.0
        end

fun aktiver(E : eventset, I : interval) =
        foreach (a,b => b + aktiv(a,I)) 0.0 E


fun passiv(e : event, I : interval) =
        let
                accnr = #accountnr(e)
        in
                if ((3000 <= accnr and accnr <= 3999) or
                    (9000 <= accnr and accnr <= 999999)) and
                   inInterval(e,I)
                then #amount(e)
                else 0.0
        end


fun passiver(E : eventset, I : interval) =
        foreach (a,b => b + passiv(a,I)) 0.0 E

fun aufwandSub(e : event, I : interval) =
        let
                accnr = #accountnr(e)
        in
                if (accnr = 450 or
                    accnr = 540 or
                    accnr = 680 or
                    (5000 <= accnr and accnr <= 7999) or
                    (8110 <= accnr and accnr <= 8299) or
                    (8351 <= accnr and accnr <= 8357) or
                    accnr = 8500) and
                   inInterval(e,I)
                then #amount(e)
                else 0.0
        end

fun aufwand(E : eventset, I : interval) =
        foreach (a,b => b + aufwandSub(a,I)) 0.0 E

fun ertragSub(e : event, I : interval) =
        let
                accnr = #accountnr(e)
        in
                if ((4000 <= accnr and accnr <= 4999) or
                    (8010 <= accnr and accnr <= 8100) or
                     accnr = 8300 or
                     accnr = 8400 or
                     accnr = 8450 or
                    (8600 <= accnr and accnr <= 8970)) and
                   inInterval(e,I)
                then #amount(e)
                else 0.0
        end

fun ertrag(E : eventset, I : interval) =
        foreach (a,b => b + ertragSub(a,I)) 0.0 E


fun bestandsKontenSub(e : event, I : interval) =
        let
```

```
                accnr = #accountnr(e)
        in
                if ((0 <= accnr and accnr <= 449) or
86                  (451 <= accnr and accnr <= 539) or
                    (541 <= accnr and accnr <= 679) or
                    (681 <= accnr and accnr <= 3999)) and
                   inInterval(e,I)
                then #amount(e)
91              else 0.0
        end

    fun bestandsKonten(E : eventset, I : interval) =
        foreach (a,b => b + bestandsKontenSub(a,I)) 0.0 E
96
    fun erfolgsKontenSub(e : event, I : interval) =
        let
                accnr = #accountnr(e)
        in
101             if ( accnr = 450 or
                     accnr = 540 or
                     accnr = 680 or
                    (4000 <= accnr and accnr <= 8999)) and
                   inInterval(e,I)
106             then #amount(e)
                else 0.0
        end

    fun erfolgsKonten(E : eventset, I : interval) =
111     foreach (a,b => b + erfolgsKontenSub(a,I)) 0.0 E



    fun getClass(e : event) =
116     let
                accnr = #accountnr(e) / 1000
        in
                if accnr <= 9
                then accnr
121             else 9
        end

    fun SumClassesSub(e : event, I : interval, f : classmap) =
        if inInterval(e,I)
126     then let
                c = getClass(e)
            in
                f[c -> f[c] + #amount(e)]
            end
131     else f

    fun SumClasses(E : eventset,I : interval) =
        foreach (e,f => SumClassesSub(e,I,f))  ([0 -> 0.0, 1 -> 0.0, 2 -> 0.0,
                                                 3 -> 0.0, 4 -> 0.0, 5 -> 0.0,
136                                              6 -> 0.0, 7 -> 0.0, 8 -> 0.0,
                                                 9 -> 0.0] as classmap) E



141 fun sumAmountsSub(e : event, I : interval) =
        if inInterval(e,I)
        then #amount(e)
        else 0.0

146 fun sumAmounts(E : eventset, I : interval) =
        foreach (e,s => s + sumAmountsSub(e,I)) 0.0 E

    fun nrOfEventsSub(e : event, I : interval) =
        if inInterval(e,I)
151     then 1.0
```

```
        else 0.0

    fun nrOfEvents(E : eventset, I : interval ) =
        foreach (e,c => c + nrOfEventsSub(e,I)) 0.0 E
156

    fun getReturn(aa : real, ab : real) =
        if ab = 0.0
        then 0.0
161     else aa / ab

    fun averageAmount(E : eventset, I : interval) =
        let suma = sumAmounts(E,I) in
        let nre = nrOfEvents(E,I) in
166             getReturn(suma,nre)
        end end


    fun FinancialStatement(E : eventset, I : interval) =
171     let f = SumClasses(E,I) in
        let aktiver = aktiver(E,I) in
        let passiver = passiver(E,I) in
        let aufwand = aufwand(E,I) in
        let ertrag = ertrag(E,I) in
176     let bestandskonten = bestandsKonten(E,I) in
        let erfolgskonten = erfolgsKonten(E,I) in
        let average = averageAmount(E,I) in
        {SumClass0 := f[0],
         SumClass1 := f[1],
181      SumClass2 := f[2],
         SumClass3 := f[3],
         SumClass4 := f[4],
         SumClass5 := f[5],
         SumClass6 := f[6],
186      SumClass7 := f[7],
         SumClass8 := f[8],
         SumClass9 := f[9],
         Aktiver := aktiver,
         Passiver := passiver,
191      Aufwand := aufwand,
         Ertrag := ertrag,
         BestandsKonten := bestandskonten,
         ErfolgsKonten := erfolgskonten,
         Average := average}
196     end end end end
        end end end end
```

The Reduceron:
Widening the von Neumann Bottleneck for Graph Reduction using an FPGA

Matthew Naylor and Colin Runciman

University of York, UK
{mfn,colin}@cs.york.ac.uk

Abstract. For the task of graph reduction, modern PCs are limited in speed not by their fast and perhaps numerous processing units, but by the rate that data can flow to and from a single and relatively slow memory unit. This limitation is known as the *von Neumann bottleneck*. This paper argues that the von Neumann bottleneck in a graph reduction machine can be effectively widened by using an FPGA. We present a prototype of such a machine – the Reduceron – and give preliminary results for several small Haskell programs running on it. The results suggest that our prototype, when refined, will outperform a C implementation of the same machine running on a modern PC by around a factor of five.

1 Introduction

The processing power of PCs has risen astonishingly over the past few decades, and this trend looks set to continue with the introduction of multi-core CPUs. However, increased processing power does not necessarily imply faster programs! One reason for this is that the architecture of the PC remains very much *von Neumann*, where execution speed is often limited by the rate that data can flow to and from a single and relatively slow memory unit. Therefore many programs, particularly *memory intensive* ones, do not benefit significantly by simply increasing processing power.

A prime example of a memory intensive application is *graph reduction* [8], the operational basis of any lazy functional language implementation. The core operation of graph reduction is *function unfolding*, whereby a function application $f\ a_1 \cdots a_n$ is reduced to a fresh copy of f's body with its free variables replaced by the arguments $a_1 \cdots a_n$. On a PC, unfolding a single function in this way requires the sequential execution of *many* machine instructions. However, this sequentialisation is merely a consequence of the von Neumann architecture, *not* of any data dependencies in the reduction process.

In this paper, we introduce a *non-von Neumann* reduction machine, called *the Reduceron*, and investigate how much faster function unfolding, and more generally sequential graph reduction, can really be. Our machine is built using an FPGA. Modern FPGAs contain hundreds of independent memory units called *block RAMs*, each of which can be accessed in parallel. Combining this with the

ability to process large amounts of data in parallel, FPGAs can rapidly execute the block read-modify-write memory operations that lie at the heart of function unfolding.

FPGAs also have two other prime attractions. Firstly, compared to other forms of custom computing, they have negligible development time overheads. And secondly, they are an advancing yet fairly stable technology that is already widely used in several areas of computing. So our reduction machine appears to be a cheap and worthwhile investment in an existing and promising technology.

The work described here is *work in progress*. We report some initial promising results from our prototype machine, showing that it runs several small Haskell [9] programs within a factor of two of a C version of the machine running on a modern PC, despite being clocked thirty times slower than the PC. The main parallelisation opportunity in sequential graph reduction has *not yet* been exploited. We give reason to expect that this parallelisation, along with the use of a more modern FPGA, will give a factor of seven speed-up. Our intention is to implement this extension to our prototype machine for the final version of this paper.

1.1 Road-map

This paper is structured as follows. Section 2 describes how Haskell programs are compiled down to bytecode that the Reduceron executes. Section 3 describes how this bytecode can be evaluated. Section 4 describes our FPGA implementation of the Reduceron. Section 5 presents results, conclusions, and future work.

2 Compilation

This section describes the compilation of Haskell programs to Reduceron bytecode. As a running example we use the following function for computing the factorial of a given integer:

```
fact :: Int -> Int
fact n = if n == 1 then 1 else n * fact (n-1)
```

There are two main aims of our compilation scheme. First, we want to allow the bytecode interpreter to be as *simple* as possible so that we can produce an efficient and working FPGA implementation within a short period of time. To achieve simplicity, we translate input programs to continuation passing style (Section 2.2) so that the machine only needs to deal with function unfolding, not data constructors and case expressions.

The second aim is to expose the parallelism in sequential graph reduction. An earlier version of the Reduceron [6] was based on Turner's combinators [12], giving it a small set of basic instructions. Unfortunately, that version performed only a small amount of work in clock cycle. Since our aim is to do as much work as possible in each clock cycle, the Reduceron is now based on the much coarser *supercombinator* [4] approach to reduction.

2.1 Desugaring and Compilation to Supercombinators

The first stage of compilation is to translate the input program to *Yhc Core* [3] using the York Haskell Compiler [11]. The result is an equivalent but simplified program in which expressions contain only function names, function applications, data constructions, case expressions, and let expressions. All function definitions are *supercombinator* definitions. In particular, they do not contain any lambda abstractions. In our example, `fact` is already a supercombinator, but in Yhc Core it now looks as follows:

```
fact n = case (==) n 1 of
           True -> 1
           False -> (*) n (fact ((-) n 1))
```

Here, infix applications have been made prefix, and the `if` expression has been desugared to a `case`.

2.2 Church Encoding

The second stage eliminates all data constructions and case expressions from the program. First, each data type d of the form

$$\texttt{data}\ d\ \texttt{=}\ c_1\ \texttt{|}\ \cdots\ \texttt{|}\ c_n$$

is replaced by a set of function definitions, one for each data constructor c_i, of the form

$$c_i\ v_1\ \cdots\ v_{\#c_i}\ w_1\ \cdots\ w_n\ \ \texttt{=}\ \ w_i\ v_1\ \cdots\ v_{\#c_i}$$

where $\#c$ denotes the number of arguments taken by the constructor c. Each of these newly introduced functions encodes one of the original data constructors as a *continuation*.

Next, all default alternatives in case expressions are removed. Case expressions in Yhc Core already have the property that the pattern in each alternative is at most one constructor deep. So removing case defaults is simply a case of enumerating all unmentioned constructors. Now each case expression has the form

$$\texttt{case}\ e\ \texttt{of}\ \{c_1\ v_1\ \cdots\ v_{\#c_1}\ \texttt{->}\ e_1\ ;\ \cdots\ ;\ c_n\ v_1\ \cdots\ v_{\#c_n}\ \texttt{->}\ e_n\}$$

and can be straightforwardly translated to continuation passing style

$$e\ (\lambda v_1\ \cdots\ v_{\#c_1}\ \texttt{->}\ e_1)\ \cdots\ (\lambda v_1\ \cdots\ v_{\#c_n}\ \texttt{->}\ e_n)$$

Since this transformation reintroduces lambda abstractions, the lambda lifter is reapplied to make all function definitions supercombinators once again. After this stage of compilation, our factorial example looks as follows:

```
fact n = (==) n 1 1 ((*) n (fact ((-) n 1)))
```

```
fact = ⟨1, 15, body⟩
body = [Int 1, Ap a0, Int 1, Ap a1]
a0   = [Arg 0, Prim [==]]
a1   = [Fun "fact", Ap a2, Ap a3]
a2   = [Int 1, Ap a4]
a3   = [Arg 0, Prim [*]]
a4   = [Arg 0, Prim [-]]
```

Fig. 1. Result of compiling the `fact` function to Reduceron bytecode.

2.3 Primitive Values to Continuations

So far, values of primitive data types, such as `Int`, have not been affected. But in the third stage of compilation, primitive values are turned into continuations. The idea is to treat each primitive integer n as if it were the Haskell function

$$\lambda\texttt{k -> seq } n \texttt{ (k } n\texttt{)}$$

For this to make sense, primitive function applications of the form $p\ n\ m$ are translated to m `(`$n\ p$`)`. Then when it comes to evaluate the primitive p, it is known that the top two elements of the stack are the fully evaluated arguments of p. This translation smooths the way for handling strict primitives, as will become apparent in the Section 3. The factorial function now looks as follows:

```
fact n = 1 (n (==)) 1 (fact (1 (n (-))) (n (*)))
```

2.4 Translation to Bytecode

Finally, each function definition is translated to a triple $\langle n, m, b\rangle$, where n is the number of arguments taken by the function, b is an explicit graph structure representing the function's body, and m is the number of nodes in the body. A node in the graph has the following algebraic type:

```
data Node = Int Int | Prim Prim | Ap [Node] | Fun String | Arg Int

data Prim = [+] | [-] | [*] | [==] | [<=] | ···
```

The function `fact`, after this stage of compilation, is shown in Figure 1. All that remains is to encode the definitions in a serial bit-stream. This is quite straightforward, but there are two important things to point out. First, applications have a variable size, so we explicitly mark the last node in an application with a special terminator bit. Second, when application nodes are turned into pointers, they are made *relative* to the address of the beginning of the function's first node. These representations make it easier to create a fresh copy of the body of a function during evaluation.

3 Bytecode Interpreter

In this section, the meaning of the Reduceron bytecode is defined by a simple small-step state transition relation, $\Rightarrow$, between pairs of the form $\langle p,\ s\rangle$, where p represents the program and is a function that maps function names to function bodies, and s is a stack. Initially the stack contains a single node `Fun "main"`, and we assume that the main function of program has the type `main ::Int`. So, the final result of a program p is defined to be r where

$$\langle p,\ [\texttt{Fun "main"}]\rangle\ \Rightarrow^{\star}\ \langle p,\ [\texttt{Int}\ r]\rangle$$

3.1 Semantics

The following paragraphs describe how evaluation proceeds, depending on the kind of node that sits at the top of the stack.

Integers. Recall that we are treating each integer as a function that fully evaluates the integer and then passes it on to a continuation. So when an integer node appears at the top of the stack, we simply do a swap:

$$\langle p,\ \texttt{Int}\ i\ :\ a\ :\ s\rangle\ \Rightarrow\ \langle p,\ a\ :\ \texttt{Int}\ i\ :\ s\rangle$$

Primitives. When a primitive appears at the top of the stack, then we know that the next two stack elements are the fully evaluated arguments of the primitive. So we have the rule

$$\langle p,\ \texttt{Prim}\ f\ :\ \texttt{Int}\ x\ :\ \texttt{Int}\ y\ :\ s\rangle\ \Rightarrow\ \langle p,\ \texttt{Int}\ (f\ x\ y)\ :\ s\rangle$$

if f is a binary arithmetic function, and the rule

$$\langle p,\ \texttt{Prim}\ g\ :\ \texttt{Int}\ x\ :\ \texttt{Int}\ y\ :\ t\ :\ f\ :\ s\rangle\ \Rightarrow\ \langle p,\ \mathit{cond}\ (g\ x\ y)\ t\ f\ :\ s\rangle$$

if g is a binary arithmetic predicate. Here, $\mathit{cond}\ p\ t\ e$ is defined to be t if p holds, and e otherwise.

Applications. To evaluate an application, we simply place all the nodes in the application on top of the stack using the concatenation operator $+\!\!+$.

$$\langle p,\ \texttt{Ap}\ n\ :\ s\rangle\ \Rightarrow\ \langle p,\ n\ +\!\!+\ s\rangle$$

Functions. When a function sits on top of the stack, it needs to be *unfolded*. The body b of the function and the number of arguments n it takes are obtained by looking f up in program mapping p. To perform the reduction, a fresh copy of b is made whose free variables have been replaced for corresponding elements on the stack. The arguments are then popped of the stack, and the root application node of the new body is then unwound onto the stack. This gives the rule

$$
\begin{array}{lcl}
\mathit{subst}\ s\ [] & = & [] \\
\mathit{subst}\ s\ (\texttt{Arg}\ i\ :\ n) & = & (s\ !\ i)\ :\ \mathit{subst}\ s\ n \\
\mathit{subst}\ s\ (\texttt{Ap}\ a\ :\ n) & = & \texttt{Ap}\ (\mathit{subst}\ s\ a)\ :\ \mathit{subst}\ s\ n \\
\mathit{subst}\ s\ (m\ :\ n) & = & m\ :\ \mathit{subst}\ s\ n
\end{array}
$$

Fig. 2. Substitution function

$$\langle p,\ \texttt{Fun}\ f\ :\ s\rangle\ \Rightarrow\ \langle p,\ \mathit{subst}\ s\ b\ +\!\!+\ \mathit{drop}\ n\ s\rangle$$
$$\underline{\text{where}}\ \langle n, m, b\rangle\ =\ p\ f$$

The *subst* function is defined in Figure 2. Note that an expression of the form $s\ !\ i$ denotes the i^{th} element from the top of the stack s. And one of the form $\mathit{drop}\ n\ s$ denotes popping the top n elements of the stack s.

This concludes a simple small-step semantics of a reduction machine for Reduceron bytecode. However, several important details that make the machine practical are not reflected in the semantic rules. These details are discussed informally in the following sections. In future we intend to extend our semantics to include these details.

3.2 Explicit Heap

The above semantics enjoys the luxury of an implicit heap. It uses abstract list structures to encode function bodies and recursion to traverse them. A real machine must treat the heap as an *explicit* entity. The main detail introduced by an explicit heap is that when a function is unfolded, its body must be copied from one place on the heap to another. In addition to replacing free variables by arguments on the stack, the *subst* function must also translate application node pointers from relative addresses to absolute ones. This copying, substituting, and translating of blocks of memory is the core operation of graph reduction, and is precisely the operation that we are aiming to optimise. In particular, our view is that this operation need not be implemented as a sequence of smaller instructions, but can, in a non-von Neumann architecture, be performed "all in one go".

Furthermore, when using an explicit heap, it is vitally important to have a garbage collector. Otherwise, for all but the simplest programs, heap space will quickly be exhausted.

3.3 Sharing

To obtain *lazy* evaluation, rather than just *non-strict* evaluation, unfolding a function should *overwrite* the corresponding application application node. When an application node is unwound onto the stack, the final node of that application has the application node's pointer stored alongside it. Now when a function f

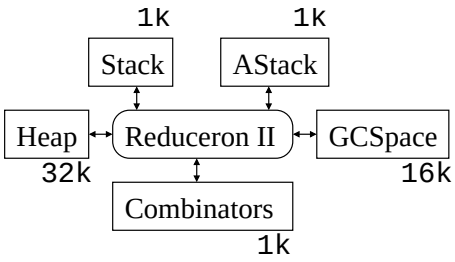


Fig. 3. Memory layout of FPGA implementation

is unfolded and the final argument to f on the stack has a pointer alongside it, the value at that pointer is overwritten with an application node that points to the freshly instantiated body of f.

4 FPGA Implementation

We use a Virtex-II FPGA chip from Xilinx for our implementation. Although this chip is over five years old, it is still quite powerful and has a very similar architecture to the latest Virtex-5 series. The structure the Reduceron's design is shown in Figure 3 and is explained in the following sections.

4.1 Heap

The Virtex-II contains 56 18kbit dual port block RAMs. Block RAMs can be configured to have a range of different address-bus widths and data-bus widths. We configure them all as 1k by 18 bit RAMs giving an 18 bit word size. We use the first 3 bits of each word as *tag bits* which encode the kind of node that the word represents. The remaining 15 bits are used to hold either an address, e.g. the address of an application or combinator, or a piece of data, e.g. an integer.

To implement a heap, we turn 32 1k by 18 bit block RAMs into a single 32k by 18 bit RAM using cascade logic (a multiplexor and a decoder). This gives quite a narrow memory, but one could just as easily cascade the block RAMs to give a much wider memory, e.g. 8k by 72 bit or 4k by 144 bit. A wider memory would allow for more parallelism in the reduction process, but we have not explored this possibility yet. Another feature that we have not yet explored is the *dual-port* nature of each block RAM, which could effectively double the bandwidth.

The cascade logic on the heap is quite complex. To obtain an efficient design, a register must be placed on the output of the data-bus's multiplexor. This means that two clock cycles are needed between placing an address on the address-bus and reading the resulting value off the data-bus. We eliminate this overhead using pipelining techniques, although this has the side-effect of making our design slightly more complicated.

4.2 Stack and Combinators

We implement the stack using 2 1k by 18-bit block RAMs. One of these RAMs corresponds to the stack used in the semantics of Section 3.1, and the other corresponds to the stack holding addresses of application nodes, as discussed in Section 3.3. The combinator memory, holding the program, is also implemented using a single 1k by 18 bit block RAM.

By having separate memories for each of these storage units, rather than a single storage unit for everything, we double the performance of the machine. This is because it is possible, for example, to read from combinator memory and the stack at the same time as writing to the heap. The fact that tripling the memory bandwidth doubles performance gives us a strong motivation for expanding the memory bandwidth further, as discussed above. Encouragingly, we also found that having separate memories increased the clocking frequency of our design by about 10%.

4.3 Garbage Collection

For any serious computations to be performed in such a small amount of memory, a garbage collector is essential. We opted to implement a simple stop-and-copy two-space garbage collector [2]. The idea is that active nodes in the heap are copied to a separate 16k by 18 bit memory called "GCSpace". Then GCSpace, which now contains a compacted version of the heap, is copied back to the heap again before reduction continues. Although not the cleverest collector, it has the advantage of simplicity. We prefer to concentrate on optimising the reduction process rather than exploring advanced garbage collectors.

4.4 Resource Usage

Our current design has the following resource usage on the Virtex-II:

Slices used:	1423/10752
Block RAMs used:	51/56
Max. frequency:	96.5MHz

The maximum clocking frequency of the Virtex-II is around 140MHz, so to produce a fairly complex design at 96.5MHz seems to be quite a good result. It is quite possible that a faster design is achievable using more buffers and deeper pipelines, but this may require sacrificing some simplicity.

4.5 Description Language

Our FPGA machine is completely implemented in Haskell using the Lava library [1]. One particularly attractive feature of Lava was the ability to mix pure functional code for describing circuit structure concisely with monadic abstractions for capturing circuit behaviour. Nevertheless, on several occasions we wished for

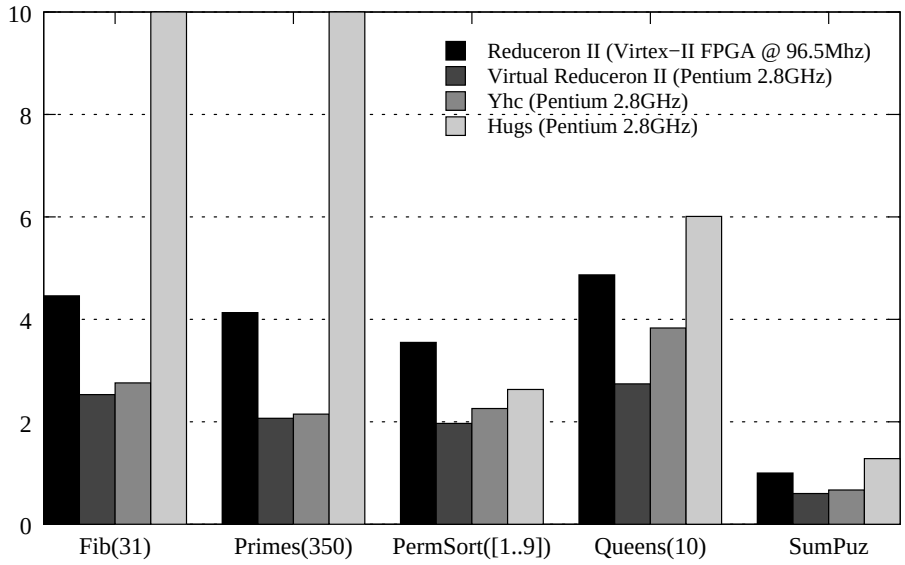


Fig. 4. Execution times (s) of programs running on FPGA and PC.

a high-level synthesis [5] tool in which we could describe our machine without concern for low-level timing issues. Unfortunately, to our knowledge, such tools are not widely available. So in future, it would be interesting to try describing our machine using high-level synthesis algorithms built on top of Lava.

5 Results and Conclusions

Figure 4 shows the execution times of a range of programs using (1) the Reduceron on the 96.5Mhz FPGA, and (2) the Reduceron, the Yhc virtual machine, and Hugs [10] all on a Pentium 2.8GHz PC. The programs used are shown along the horizontal axis and the inputs to the programs are given in parentheses. Most of the programs are well known from the nofib benchmarking suite [7], except `PermSort` which is a permutation sort and `SumPuz` which is a program that counts the number of solutions to an arbitrary cryptarithmetic problem. The problem instance given to `SumPuz` was `GRAPH + ROOT = REDEX` which has 24 solutions. Furthermore, where required, we implement division by repeated subtraction since our machine currently doesn't support division.

In all cases, despite a clock-speed on the FPGA around *thirty times slower* than on the PC, the FPGA implementation of the Reduceron runs within a factor of two of the PC version. This is a promising result as there are two main improvements that we can make to our machine. Firstly, our current implementations of function-unfolding and application-unwinding are *sequential*, requiring execution-times proportional to the number of nodes they contain. As discussed in Section 4.1, it is possible to widen the memory structures, paving the way for

a faster unfolding and unwinding. But how much of a speed-up can we expect? Let us take the `fact` supercombinator of Figure 1 as an example. It contains a total of 15 nodes and contains applications ranging from size two to four. If we were to quadruple the memory bandwidth then we can expect unfolding `fact` to be four times faster, and fetching applications of `fact` to be two to four times faster. By averaging, we obtain an estimate of a 3.5 times speed-up.

The second improvement that we can make is to move to the latest Virtex-5 series FPGAs, capable of running at 550MHz and containing ten times as many block RAMs. We can reasonably expect our machine to run twice as fast on this newer device, giving a combined improvement of a factor of seven speed-up. Our prototype FPGA-based machine would then outperform the same machine running on a modern PC by around a factor of five. It is our goal to confirm this result for the final version of this paper.

Despite the simplicity of the Reduceron, we do consider it to be a realistic machine. Indeed, our PC implementation of it outperforms the Yhc virtual machine, a G-Machine variant. But currently, we do lack support for many Haskell'98 primitives. Questions we leave for the final version of this paper are: How does the Reduceron compare with GHC-compiled programs? How should I/O be done? What applications is the Reduceron suited to?

The full source code for the Reduceron compiler, machine, and examples can be obtained from `http://www.cs.york.ac.uk/fp/darcs/reduceron2`.

Acknowledgements

The first author is supported by an award from the Engineering and Physical Sciences Research Council of the United Kingdom. We thank Jack Whitham and Ian Gray for their help in the Virtual Lab and to the Real Time Systems group for providing the Virtex-II FPGA. We also thank Emil Axelsson for comments on a draft. Most of all, we thank Neil Mitchell for many useful discussions, for implementing Yhc Core, and for providing several transformations including the Church encoder and the lambda lifter.

References

1. Koen Claessen. Embedded Languages for Describing and Verifying Hardware. PhD Thesis, Chalmers University of Technology, 2001.
2. Robert R. Fenichel and Jerome C. Yochelson. A lisp garbage-collector for virtual-memory computer systems. *Commun. ACM*, 12(11):611–612, 1969.
3. Dimitry Golubovsky, Neil Mitchell, and Matthew Naylor. Yhc.Core - from Haskell to Core. *The Monad.Reader*, (7):45–61, April 2007.
4. R. J. M. Hughes. Super Combinators—A New Implementation Method for Applicative Languages. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 1–10, Pittsburgh, 1982.
5. Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Inc., 1994.

6. Matthew Naylor. The Reduceron: An FPGA Machine for Executing Haskell. `http://www.cs.york.ac.uk/~mfn/reduceron/`.
7. Will Partain et al. The `nofib` Benchmark Suite of Haskell Programs. `http://darcs.haskell.org/nofib/`, 2007.
8. Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Computer Science. Prentice-Hall, 1987.
9. Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
10. The Hugs Team. The Hugs interpreter, may 2006. `http://www.haskell.org/hugs/`, October 2006.
11. The Yhc Team. The York Haskell Compiler - user's guide. `http://www.haskell.org/haskellwiki/Yhc`, February 2007.
12. D. A. Turner. A new implementation technique for applicative languages. *Softw., Pract. Exper.*, 9(1):31–49, 1979.

Incremental Extension of a Domain Specific Language Interpreter

Olivier Michel[1] and Jean-Louis Giavitto[1]

IBISC - FRE 2873 CNRS & Université d'Évry, Genopole
Tour Évry 2, 523 place des terrasses de l'Agora, 91000 Évry, France

Abstract. We have developed an interpreter for the domain-specific language `MGS` using `OCAML` as the implementation language. In this third implementation of `MGS`, we wanted to provide the end-user with easy incremental addition of new data structures and their associated functions to the language. We detail in this paper our solution, in a functional setting, which is based on techniques similar to those found in *aspect-oriented programming*.

1 Introduction

This work takes place in the `MGS` [11, 16] project[1] which develops new data and control structures for the modelization and simulation of *dynamical systems with a dynamical structure* [14]. These features are embedded in a simple functional language, called also `MGS`, which is used to model various physical and biological processes [30, 31, 13].

The adequacy of `MGS` to its application domain is achieved through the following three features:

1. it embeds a very rich family of data structures used for the representation of the states of dynamical systems;
2. it provides a very large set of functions operating on these data structures;
3. it offers a new way of specifying uniformly functions defined by case on arbitrary data structures, using topological rewriting [12].

An interpreter for the `MGS` language has been implemented in the `OCAML` [21, 28] language. The decisive advantages of `OCAML` for us were that (1) it provides both functional and object-oriented features in the same environment and (2) it produces very effective code [1, 2].

One of the main problems raised by the `MGS` project is the wish to offer easy incremental addition of new data structures and their associated functions to answer the needs expressed by the end-users. As a matter of fact, the initial release of the interpreter did only include the collection data types *sequences*, *sets* and *multisets*.
The current interpreter includes *arbitrary graphs*, *Voronoï tessalation*, *group*

[1] The `MGS` project is available at: `http://mgs.ibisc.univ-evry.fr`

```
type expr =
    Constant of value
  | Apply of expr * expr

and value =
    Int of int
  | Fun of (value -> value)

let print_val = function
  | Int i -> Printf.printf "%d\n" i
  | Fun x -> Printf.printf "<fun>\n"

let inc_val =
  Fun(function (Int i) -> Int (i + 1)
     | _      -> failwith "bad arg")
```

```
let rec eval = function
  | Constant x     -> x
  | Apply (e1, e2) ->
      (match (eval e1) with
       |   Fun x -> x (eval e2)
       | _       -> failwith "apply: type error")

let inc_expr = Constant inc_val
let inc = Apply(inc_expr,
                Apply(inc_expr, Constant (Int 1)))

print_val (eval(inc))
```

Fig. 1. A simple and basic interpreter expressed in a *higher-order syntax* style in ML.

based fields [11] which generalize various kind of arrays, *gmaps* [22], *extensible records* and *maps*, *trees* defined by automata, and many other data types [29]. All the additional data structures (together with their operators) have been added incrementally using the techniques described in this paper.

Usually, the values handled in the *target language* (that is, the language to be implemented, here, `MGS`), are represented through a unified data structure in the *implementation language* (that is the language used to implement the target language, here, `OCAML`). We call this data structure the *value* data structure. Using `OCAML` as the implementation language, there are two choices for the *value* data structure:

1. it can be represented using a *sum type*, following a functional style,
2. or, it can be represented using a *class* following an object-oriented style.

Both approaches have some shortcomings, with respect to the requirement of incremental development. To summarize

1. in the functional approach, it is easy to add new functions but difficult to add new target data structures;
2. on the contrary, in the object-oriented approach, it is easy to add new target data structures but difficult to add new functions.

To overcome these drawbacks, we have developed an original technique, inspired from aspect programming techniques, that consists in weaving both the *value* data structure and their associated functions. This technique has the advantages of:

– allowing new target data structures to be added without modifying the already written implementation files of the interpreter,
– facilitating the addition of new target data structures and functions to the point that even end-users are able to increment the `MGS` interpreter.

The rest of the paper is organized as follows. We briefly describe the MGS language in the next section to give the reader an idea of the complexity raised by the implementation of the rich data types in the interpreter. Section 3 describes the functional and the object-oriented approach used to implement the *value* data structure and details the problem raised by its incremental evolution. The implementation of heavily overloaded target functions are presented in the next section. The software architecture of the final implementation code of the interpreter is sketched in section 5. Section 6 presents how the informations gathered along all implementation files are collected to generate the *value* data type and to implement the multiple dispatch of the target functions. The conclusion summarizes our approach and shortly reviews related works.

2 Functions and Values in the MGS Programming Language

We briefly discuss in this section the values manipulated in the MGS language and their associated functions. Our aim is to show that the technique presented in this paper is required to deal with its complexity and to allow an easy incremental addition of new data structures and their associated functions.

2.1 The Type Hierarchy of the MGS Programming Language

We briefly give in this section an incomplete description of the type hierarchy of the MGS programming language.

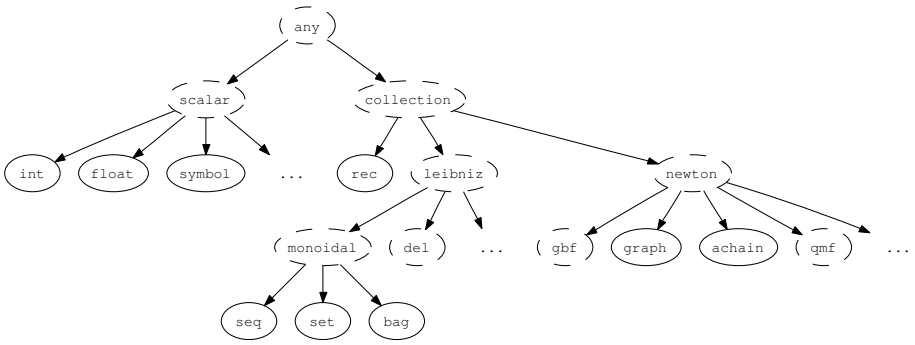


Fig. 2. The type hierarchy of the MGS language.

A graphical representation of the type hierarchy of MGS is given in figure 2. In MGS two main types of values are distinguished: the *scalar values* which are elementary constants and *collections* which allow to organize the values. Example of scalar values are integers, floats, symbols... Example of collection types are

sets, bag, Delaunay graphs, group-based fields [15], quasi-manifolds [22, 23]... Collection values can be any combination of collections and scalar values such as a bag containing symbols and sequences of integers.

In the following example, we define three values equal to collections: `v_seq` which consists in the *sequence* (like a `C` one-dimensional array) composed of a string value (`"str"`), a floating-point value (`3.5`), two integers values (`4, 4`), a boolean value (`true`) and the identity function (expressed as an anonymous lambda-calculus expression: `\x.x`) and the same elements organized as a *set* (`v_set`) and a *bag* (`bag`).

```
1  mgs> v_seq := "str", 3.5, 4, 4, true, (\x.x), seq:();;
2  ("str", 3.500000, 4, 4, true, [funct]):'seq
3
4  mgs> v_set := "str", 3.5, 4, 4, true, (\x.x), set:();;
5  (4, true, 3.500000, "str", [funct]):'set
6
7  mgs> v_bag := "str", 3.5, 4, 4, true, (\x.x), bag:();;
8  (4, 4, true, 3.500000, "str", [funct]):'bag
```

The comma operator is overloaded and used, following the context, to add an element to a collection, to merge two collections of the same type or to create a sequence composed of its elements. For most of the collection types, the empty collection type `xxx` is written `xxx:()` (for the example above, the empty collection for the *sequence* type is namely `seq:()`).

2.2 Functions For the Manipulation of Values

In `MGS`, most of the functions are overloaded to allow an easy handling of complex values. A collection value c has a type $\tau(\mu)$ where τ is the collection type (like `set`, `seq`, `bag`, ...) and μ is the type of the elements of the collection. To allow an easy handling of complex values, most built-in functions are overloaded so that user-defined functions can handle collections of any type $\tau(\mu)$ regardless of τ and μ. That property can be seen as a kind of *polytypism* [5, 19].

For example, the `size` functions, that returns the number of elements in the collection, can be applied to any collection:

```
1  mgs> size(v_seq);;
2  6
3
4  mgs> size(v_set);;
5  5
6
7  mgs> size(v_bag);;
8  6
```

Among all the polytypic functions, we have the classics `map`, `iter`, `fold`, `one_off`, `rest`, `member`... The interested reader should refer to `http://mgs.ibisc.univ-evry.fr/Online_Manual/Collections.html` for the detail of available functions defined on collection types.

2.3 A Short Example

MGS unifies the collection types together with the polytypic functions in a general rewriting scheme. Programs are written as a composition of *transformations*, a very expressive form of rewriting process [12, 13, 17, 30, 32] based on the neighborhood relationship exhibited by each collection type together with a general form of pattern matching.

The following MGS expression returns (if it exists) the Hamiltonian path in a graph G

```
1   trans Hamiltonian =
2    (s* as whole / (size(whole) == size(‘self)) => whole)
```

Pattern `s*` matches any *path* p (that is, a sequence of neighboring values) in G such that each element in p appears only once; the additional requirement that p is of the same size as G ensures that such paths are Hamiltonian. Of course, the complexity of the search remains, but the complexity of its expression is highly reduced.

3 The Implementation of the `value` Data Structure

3.1 The `value` Data Structure in a Functional Setting

In a functional setting, an evaluator consists in a function `eval` that, given an expression of type `expr`, returns a value of type `value`. A toy example of such an interpreter is given in figure 1.

In this example, the type `value` is restricted to integers and functions. The precise application area of MGS does not matter in this paper and detailing the handling of integers and integers operators should be enough to explain our approach.

Functions in the target language rely on the use of functions of the implementation language (see the example of the `inc_val` function at line 16 in figure 1). This mechanism of representing a target function by an implementation function lies at the heart of the *higher-order abstract syntax* [26, 7] approach. For the sake of simplicity, we do not detail here on how to implement user-defined functions. In the current MGS interpreter, this is achieved by using combinators to translate on-the-fly a user-defined lambda expression into a `Fun` value [6]. The same mechanism can be used in the OO approach presented below. With the higher-order syntax approach it is immediate to integrate existing libraries of functions as a predefined kernel of functions: predefined library functions are embedded using the `Fun` constructor. Note that the functions of the kernel have exactly the same status and implementation as the user-defined functions and so they can be arbitrarily mixed “for free” (e.g. using higher-order operators). In the rest of this paper we focus only on the handling of a set of predefined functional constants like `inc_val`.

If one wants to extend the interpreter with a new primitive, like the addition of integers, it only requires to define the corresponding constant

```
#include <iostream>
using namespace std;

struct value;

struct expr { virtual value& eval() =0; };

struct value : public expr {
   value& eval() { return *this; }
   virtual ostream& print(ostream& o) =0;
};

struct Number : public value {
   virtual Number& inc() =0;
};

struct Int : public Number {
   int val;
   Int(int n) : val(n) {}
   Number& inc() { return *(new Int(val + 1));}
   ostream& print(ostream& o) {return o << val
<< "\n";}
};

struct Fun : public value {
   virtual value& operator() (value&) =0;
   ostream& print(ostream& o)
                  {return o << "<fun>\n";}
};

struct Error : public value {
   char* msg;
   Error(char* s) : msg(s) {}
   ostream& print(ostream& o) {return o << msg
<< "\n";}
};

struct Apply : public expr {
   expr& fct;
   expr& arg;
   Apply (expr& f, expr& a) : fct(f), arg(a) {}

   value& eval() {
      if (Fun* f = dynamic_cast<Fun*>(&(fct.eval())))
 return (*f)(arg.eval());
      else
 return *new Error("apply: type error");
   }
};

struct Inc : public Fun {
   value& operator() (value& arg) {
      if (Number* a = dynamic_cast<Number*>(&arg))
 return a->inc();
      else
 return *new Error("bad arg");
   }
};

main()
{
   Int v(1);
   Inc incr;
   Apply tmp(incr, v);
   Apply(incr, tmp).eval().print(cout);
}
```

Fig. 3. A simple and basic interpreter expressed in an OO programming style.

```
1   let add_val =
2     Fun(function (Int v1) ->
3         Fun (function (Int v2) -> Int (v1 + v2)))
```

in a new file and to rely on separate compilation and linking to produce the new interpreter. The new function can be made available to the MGS programmer by registering the previous expression in the global environment under an adequate name.

So, it is straightforward to extend the library of available functions. On the contrary, if we want to extend the available `value` type, for example with floating-points values, we face several problems:

1. the type `value` must be extended accordingly, which implies to edit an existing file,
2. *all* functions defined by case on type `value` have to be updated to take into account the new case.

The second point requires to edit *all existing files* related to the `value` type. For instance, in the context of the MGS project, which represents 50k lines of OCAML code, spread in about 75 files, it would require a huge amount of work.

3.2 The `value` Data Structure in an Object-Oriented Framework

In a object-oriented (OO) framework, the sum type used in the functional approach is replaced by an *abstract class* whose derived classes represent all the cases. *Methods* are used to implement predefined target functions. The corresponding interpreter, in `C++`, is given in figure 3.

The `dynamic_cast<...>(...)` is used for downward casting a class to one of its derived classes in a safe way. Failure to downcast corresponds to type errors during evaluation of `MGS` expressions. `value` are defined as a subtype of `expression`. A class `Number` gathers all classes that admit numerical operations like incrementation. Initially, the only descendant of `Number` is `Int` which represents integers. Despite the syntactic differences, the OO `C++` code mimics closely the functional approach. The `eval` methods applies to any expression and is defined, case by case, on each derived subclasses. The real difference is that the cases are not gathered in one place but scattered in each derived classes. The evaluation of a value is always the identity and so it is defined at the level of the `value` class.

If one wants to extend the interpreter with a new data type, like floating-points values, it only requires to define the corresponding derived class

```
1  struct Float : public Number {
2     float val;
3     Float(float f) : val(f) {}
4
5     value& inc() { return *(new Float(val + 1.0)); }
6
7     ostream& print(ostream& o)
8     { return o << val << "\n"; }
9  };
```

in a new file and to rely on separate compilation and linking to produce the new interpreter.

So, it is straightforward to add new target data structures. On the contrary, if we want to extend the library of available functions, we have to add a virtual function to the mother-class `value` or one of its derived classes. This implies to edit the class value but also *all the derived classes* for which an implementation of the new method is relevant.

arity	number	min cases	average cases	max cases
1	100	1	3.43	24
2	93	1	5.77	40
3	22	1	2.4	14
4	4	1	1	1
5	0			
6	4	1	6	21
7	2	1	12	23

Fig. 4. Statistics summary of overloaded functions in `MGS`.

4 Implementing Overloading

The implementation of an incremental interpreter has also to face an additional problem if we provide to the end-user *overloaded target functions*. In the previous example, the function `inc` has a meaning for both integer and floating-points values. It would be very convenient to offer to the end-user an overloaded function acting on both types. This means that from an `MGS` identifier `inc` and the type of the arguments in an application, some *dispatch* mechanism must be used to call the correct implementation method or function. This problem is not negligible. In the `MGS` context, there are many overloaded functions: figure 4 gives the number, and distribution with respect to their arity, of overloaded target functions available to the end-user.

In the functional framework, the dispatch is easily provided for unary functions, using definition by cases through the pattern matching on the constructors of the `value` data type. In the OO framework, this is also easily achieved using virtual methods.

Things get more complex when we consider functions with multiple arguments. For example, consider the addition of two values. Pattern matching can still be used, but at the price of explicitly writing the Cartesian product of the `value` constructors. For example, in the current `MGS` interpreter, there are 24 available data types. So, overloading the addition comes at the cost of writing 576 cases. Obviously, most of the cases correspond to errors and are handled similarly. Even if this can be done using wild-cards in patterns, there is still a huge number of cases to be written.

In the OO framework, the extension of the overloading of a target function to multiple arguments requires *multiple dispatch* [18]. Multiple dispatch can be implemented (in languages with only single dispatch, like `C++` or `OCAML`) using auxiliary methods [25, item 31]. The number of these functions also grows exponentially with the number of arguments meaningful for the dispatch.

5 An "Incremental" Software Architecture for the `MGS` Interpreter

Our first design decision in `MGS` was to rely on the functional approach. As a matter of fact multiple dispatch is easier to implement in this framework. However, the problems raised in section 3.1 have still to be addressed. Our idea is to split the various cases of an overloaded function into multiple `OCAML` functions spread through the whole set of files. A pre-processing phase gathers all the defined functions and merges them into the actual implementation. A similar process is done for the various constructors of the `value` data type.

Splitting the definition into several files raises the problem of functional dependency. It is hopeless to force the developer to have a correct sequencing of the files when we want to enable at the same time the unconstrained addition of new data types and pieces of code. To solve this problem, we use a well-known technique of forward pointers that are correctly set at run-time (see for example [21, page 150]).

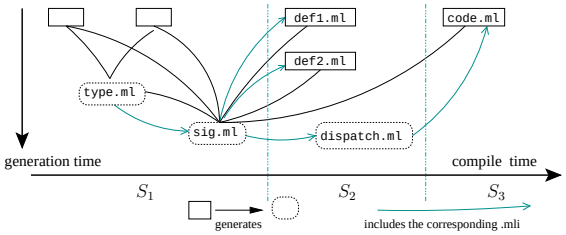


Fig. 5. Organisation of the code: the three phases S_1, S_2 and S_3 are given together with the exact date when each file is produced and the functions are made available.

We detail in the rest of this section the overall software organization through the description of a small example. We assume that the `value` data type is completely defined once and for all. Section 6.1 sketches how this data type can also be generated from informations gathered through all the code. The reader is supposed to be familiar with the `OCAML` language and its compilation tools.

5.1 Organization of the Code

The project consists in three set of files, S_1, S_2 and S_3. A *dispatch* will be computed from the definitions occurring in S_1 and S_2. After S_1, the *signature* of the dispatched functions are available (for the functions defined in S_2 and S_3). That is, the functions are called through a *diversion* mechanism. After S_2, the functions can be directly called since all dispatched functions are known *and initialized* after S_2. Then, the dispatch is effective and the *direct call* to the dispatched functions is possible. Figure 5 shows the clear timing of the operations occurring in the three phases and what files are used.

5.2 S_1: Basic Definitions

The files in S_1 are *definitions*, usually *types* and *functions*, that do not rely on other previous definitions and that will be used everywhere in the project.

It includes a file `types.ml` that defines the type of the values (the `value` type) that are going to be handled. All the functions that will handle values in the code will require to have access to this file. From the `.ml`, a `.mli` *include* file is produced by the `OCAML` compiler. Using this include file through the `open Types` directive all other files are able to define functions on `value`.

```
1  type value =
2      Int of int
3    | Float of float
```

5.3 The Diversion Mechanism: Generation of Forward and Signatures

At this point, it is necessary to give access to the overloaded functions, which raises two problems:

1. since the functions are defined incrementally, there is no global repertory of them;
2. these functions must be made available for code in S_2 and S_3 independently of their actual implementation localization.

These two problems are solved by scanning all implementation files to collect the various function names to generate a unique file `sig.ml` providing the implementation of the forwarding mechanism. The scanning is made possible by enforcing a specific syntax for the function names (see below).

For our example, the generated `sig.ml` file is

```
1   open Types;;
2
3   (* Signature declaration *)
4   let (add_forward : (value -> value -> value) ref) =
5     ref  (function _ -> failwith "unitialized add")
6
7   let (print_forward : (value -> unit) ref) =
8     ref  (function _ -> failwith "unitialized print")
9
10  Printf.printf "Setting the forward pointers\n"
11
12  let add x y = !add_forward x y
13  let print x = !print_forward x
```

The forward mechanism works as follows: an overloaded function `add` is a *wrapper* that applies the value of the imperative variable `add_forward`. This imperative variable is initialized with a dummy function raising an error. This variable will be set later with the correct function (see lines 20 and 21 of the file `dispatch.ml` in section 5.5).

5.4 S_2: Writing of Code

The files in the *second set* S_2 contains the implementation of the various cases of an overloaded function. Suppose that a unary function `XX` is overloaded on two types `p` and `q`. This suppose that the `value` data type has two constructors `P` and `Q` defined like

```
1   type value = ...
2   | P of type_p
3   ...
4   | Q of type_q
5   ...
```

Then the `MGS` implementers have only to provide two functions called `_XX_p` and `_XX_q` both of arity one. The argument of `_XX_p` is of type `type_p`. The naming convention is simple: the name of the constructor (which is constrained to always begin with a capital letter in `OCAML`) is used in small letter in the name of the function case.

The naming convention is straightforwardly extended to handle multiple arguments. A function definition:

$\_XX\_p_1\_\ldots\_p_n$

represents the handling of the arguments of type $\texttt{type\_p}_1$, ..., $\texttt{type\_p}_n$ for the overloaded function `XX`. The types $\texttt{type\_p}_i$ are arguments of constructors of the sum type `value`. Each constructor corresponds to a different `MGS` value type and we assume that the $\texttt{type\_p}_i$ are all different, even if the implementation type are the same by using alias type declaration. This naming convention enables the scanning described in the previous section and the generation of the diversion functions `XX` and `XX_forward` as well as the dispatch function `__XX` described in the next section.

An example of two overloaded functions, `add` and `print` is given in the `def1.ml` file below:

```
1   open Types;;
2   open Sig;;
3
4   let _add_int_int i1 i2     = Int (i1 + i2)
5
6   let _print_int i1          = Printf.printf "%d" i1
7   let _print_float f1        = Printf.printf "%f" f1
```

Note that the definition of `add` is, at this point, not complete. Other cases are specified or will be specified in other files.

All functions are allowed to recursively call any overloaded function. For example, in another file `def2.ml`, the definition of `_add_float_int` uses the overloaded function `add`:

```
1   open Types;;
2   open Sig;;
3
4   let _add_int_float i1 f1   = Float
5                               (f1 +. (float_of_int i1))
6   let _add_float_float f1 f2 = Float (f1 +. f2)
7   let _add_float_int f1 i1   = add (Int i1) (Float f1)
```

Note however that `add` can only be effectively used once the wrapper has correctly been set at run-time. This means that, at this point, only function *definitions*, implying overloaded functions, can occur and *no actual function calls* to overloaded functions.

5.5 Generation of the Overloaded Functions

An overloaded function is implemented using pattern matching to dispatch to the several function cases. The implementation function corresponding to the overloaded function `XX` is called `__XX`. For our example, the generated `dispatch.ml` file is:

```
1   open Types;;
2   open Sig;;
3   open Def1;;
```

```
4   open Def2;;
5
6   let __add x y = match x, y with
7     |  (Int x0), (Int x1)     -> _add_int_int x0 x1
8     |  (Float x0), (Float x1) -> _add_float_float x0 x1
9     |  (Int x0), (Float x1)   -> _add_int_float x0 x1
10    |  (Float x0), (Int x1)   -> _add_float_int x0 x1
11
12  and __print x = match x with
13  | Int x0   -> _print_int x0
14  | Float x0 -> _print_float x0
15
16  Printf.printf "Setting the correct link\n"
17  flush Pervasives.stdout
18
19  Sig.add_forward   := __add
20  Sig.print_forward := __print
```

At the end of the file, the imperative variables used in the wrapper functions are set to their correct value, using the just defined `__XX` functions.

5.6 S_3: Using Dispatched Functions

At this point, all function cases have been gathered, the overloaded functions have been generated and can be used even in the initialization phase, on the contrary to the code in the S_2 set of files. In the `MGS` project, the files in S_3 corresponds to the implementation of transformations, the parsing, the top-level, etc.

To finalize our running example, the file `code.ml` below describes some possible use of the overloaded functions, `add` and `print`:

```
1   open Types;;
2   open Sig;;
3
4   print (add (Float 2.0) (Float 3.0))
5   print_newline()
6   print (add (Float 2.0) (Int 1))
7   print_newline()
8   print (add (Int 2) (Float 1.0))
9   print_newline()
10  print (add (Int 2) (Int 1))
11  print_newline()
```

5.7 Compilation and Execution of the Code

The compilation follows five phases to respect the code organization:

1. in a first phase, all the files in S_1 are compiled;

2. in a second phase, all the files of the project are scanned to automatically generate and compile the `sig.ml` file;
3. in a third phase, all files from S_2 are compiled (which additionally produces the include files required for `dispatch.ml`);
4. in a fourth phase, `dispatch.ml` is generated and compiled;
5. finally, files in S_3 are compiled and the final linking is done.

This process is fully automated by a `Makefile`. The compilation and the execution of our example gives:

```
ibisc 12 > make

ocamlc -c types.ml
ocamlc -c sig.ml
ocamlc -c def1.ml
ocamlc -c def2.ml
ocamlc -c dispatch.ml
ocamlc -c code.ml
ocamlc -o dsal types.cmo sig.cmo def1.cmo def2.cmo\
              dispatch.cmo code.cmo

ibisc 13 > dsal

Setting the forward pointer
Setting the correct link
5.000000
3.000000
3.000000
3
```

6 Weaving the Implementation Code

In this section, we sketch the automatic generation of the `type.ml`, `sig.ml` and `dispatch.ml` files.

6.1 Weaving the `value` Data Structure

In the same way that the function cases are split through several files, the various constructor of the `value` data type are split in several files. This enables to add a new data structure to `MGS` simply by providing a new file introducing the corresponding constructor. The precise syntax used for the constructor declaration does not matter. The first weaving tool scans all the source files to gather all the constructors related to the `value` type and generates the `types.ml` file.

6.2 Weaving the Dispatch on `value` Type

The second weaving tool gathers all the function cases spread among the source files to generate the overloaded functions. The dispatch mechanism presents some

subtleties. In the previous example, all the types used as the arguments of the constructors of the `value` type where incomparable. However, the situation is more complex in the implementation of `MGS`:

- wild-cards are required to handle within the same case function various argument types;
- there is a hierarchy of data types in the `MGS` language that is available to the developer of the `MGS` language.

A simple example of the last kind is the following: `MGS` values are split into *atomic* and *compound* values. Sometimes, cases functions are dispatched on this distinction, and not on the implementation type of the data structure. For example, the primitive function `size` returns `-1` on all atomic values and returns the number of elements in its argument in case of a compound value. Interior nodes of the `MGS` hierarchy type corresponds to several constructor in the `value` type. The type of the argument passed to the dispatched function is then `value` and not the argument type of a constructor.

Having *family* of types produces a hierarchy that has to be taken into account while generating the pattern matching of the overloaded functions. For example, a case on `_XX_int_int` has to appear before the case `_XX_int_atomic`. The partial order relationships between the `MGS` types is used to sort lexicographically the collected cases of an overloaded function.

A "catch-all" case is produced to handle "bad argument types" error. To avoid spurious warnings by the `OCAML` compiler, this case is produced only if required.

7 Conclusion

The software organization and the weaving tools described in this paper have been successfully used in the development of the `MGS` interpreter. This represents over 50k lines of `OCAML` code (there is also over 100k lines of `C++` libraries to provide basic support for sophisticated data structures like Voronoï tessalation, G-Maps, Cayley graphs, ...). The 50k lines of `OCAML` files are scattered over 75 files. The scanning of these files is almost immediate and does not slow down the compilation process. It generates 225 overloaded functions. These results show that our approach is well suited to the development, in a functional setting, of large incremental projects.

One of the originality of this work is the application of aspect weaving techniques in the context of a functional language (`OCAML`). As far as we know, this is the first attempt to merge these two worlds to ease the implementation of a domain-specific language. Our approach relies only on a tailored software architecture, a dedicated `makefile`, some naming conventions and two external tools to parse and collect informations on the various data types entering in the `value` type and on the overloaded functions. It does not involves any changes on the `OCAML` compiler nor sophisticated typing techniques. It is therefore a lightweight solution to the problem of incrementally building an interpreter.

Related Works.

The various techniques implied have already been used in other contexts (for example, wrapper functions are used to overcome the impossibility to have recursively defined modules spun across multiple files) and the problem that we have tried to solve has been coined *the expression problem* in [34] (with an enlightening discussion in [35]). We briefly review, because of space limitation, some similar approaches.

Language Extensions. In [20] is proposed a specific design pattern called the *Extensible Visitor* which is a combination of functional and object-oriented programming methods while our approach is purely functional.

An aspect-oriented programming extension to `OCAML`, very similar to AspectJ [3], is proposed in [24]. It is a highly technical approach that uses the usual features of *join points*, *pointcuts* and *advices declarations* that leads to the definition of the `Aspectual Caml` language while our work do not change the language itself but consists in two additional tools to collect information and produce the dispatch files.

Extensible Interpreters. The conception of extensible interpreters has been considered for example in [33]. However, it requires sophisticated type inference techniques to be implemented that goes beyond standard `ML` type inference.

Multiple dispatch has been considered for overloaded functions in a functional language [4]. As for the previous work, it requires sophisticated types techniques.

Extensible sum data types[8, 9] (which is further extended in [10] by adding private row types to functors) have been proposed and are implemented in `OCAML`. They enable the incremental definition of the `value` data type and of the functions but at the cost of requiring a lot of wrap/unwrap functions that are done for free in our approach. Moreover, since with polymorphic variants a matching case can easily be forgotten in a function definition, we believe that this approach would be too error-prone on a large-scale development like the `MGS` language

Once again, a very technical solution is found in [27] by relying on modules and (higher-order) functors.

Acknowledgements.

The authors thank Julien Cohen of LINA – CNRS FRE 2729 for his comments on the paper.

References

1. Computer language shootout scorecard, June 2003. `http://dada.perl.it/shootout/craps.html`.
2. Gentoo : Intel® pentium® 4 computer language shootout, July 2006. `http://shootout.alioth.debian.org/gp4/index.php`.

3. AspectJ project. Available at `http://www.eclipse.org/aspectj/`.
4. Bourdoncle, F., and Merz, S. Type checking higher-order polymorphic multi-methods. In *Conference Record of POPL'97: The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1997), ACM SIGACT and SIGPLAN, ACM Press, pp. 302–315.
5. Cohen, J. *Intgration des collections topologiques et des transformations dans un langage fonctionnel*. PhD thesis, Université d'Évry, Dec. 2004.
6. Cohen, J. Interprétation par SK-traduction et syntaxe abstraite d'ordre supérieur. In *Journées Francophones des Langages Applicatifs (JFLA 2005)* (2005), O. Michel, Ed., INRIA, pp. 17–34.
7. Cohen, J. Interprétation par syntaxe abstraite d'ordre supérieur et traduction en combinateurs. *Technique et science informatiques* (2007). To appear.
8. Garrigue, J. Programming with polymorphic variants. In *Proc. of 1998 ACM SIGPLAN Wksh. on ML, Baltimore, MD, USA, 26 Sept. 1998*. Oct. 1998.
9. Garrigue, J. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering (FOSE)* (Nov. 2000).
10. Garrigue, J. Private row types: Abstracting the unnamed. In *APLAS* (2006), N. Kobayashi, Ed., vol. 4279 of *Lecture Notes in Computer Science*, Springer, pp. 44–60.
11. Giavitto, J.-L. A framework for the recursive definition of data structures. In *ACM-Sigplan 2nd International Conference on Principles and Practice of Declarative Programming (PPDP'00)* (Montréal, Sept. 2000), ACM-press, pp. 45–55.
12. Giavitto, J.-L. Topological collections, transformations and their application to the modeling and the simulation of dynamical systems. In *Rewriting Technics and Applications (RTA'03)* (Valencia, June 2003), vol. LNCS 2706 of *LNCS*, Springer, pp. 208 – 233.
13. Giavitto, J.-L., Malcolm, G., and Michel, O. Rewriting systems and the modelling of biological systems. *Comparative and Functional Genomics 5* (Feb. 2004), 95–99.
14. Giavitto, J.-L., and Michel, O. Modeling the topological organization of cellular processes. *BioSystems 70*, 2 (2003), 149–163.
15. Giavitto, J.-L., Michel, O., and Cohen, J. Pattern-matching and rewriting rules for group indexed data structures. *ACM SIGPLAN Notices 37*, 12 (Dec. 2002), 76–87.
16. Giavitto, J.-L., Michel, O., Cohen, J., and Spicher, A. Computation in space and space in computation. Tech. Rep. 103-2004, May 2004. 22 p.
17. Giavitto, J.-L., and Spicher, A. *Systems Self-Assembly: multidisciplinary snapshots*. Elsevier, 2006, ch. Simulation of self-assembly processes using abstract reduction systems.
18. Ingalls, D. H. H. A simple technique for handling multiple polymorphism. In *OOPSLA* (1986), pp. 347–349.
19. Jeuring, J., and Jansson, P. Polytypic programming. In *Tutorial Text from 2nd Int. School on Advanced Functional Programming, Olympia, WA, USA, 26–30 Aug 1996*, J. Launchbury, E. Meijer, and T. Sheard, Eds., vol. 1129 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1996, pp. 68–114.
20. Krishnamurthi, S., Felleisen, M., and Friedman, D. P. Synthesizing object-oriented and functional design to promote re-use. In *ECOOP* (1998), E. Jul, Ed., vol. 1445 of *Lecture Notes in Computer Science*, Springer, pp. 91–113.
21. Leroy, X., Doligez, D., Garrigue, J., Rémy, D., and Vouillon, J. *The Objective Caml system, release 3.09*. INRIA, October 2005. available at `http://caml.inria.fr/distrib/ocaml-3.09/`.

22. Lienhardt, P. Topological models for boundary representation : a comparison with n-dimensional generalized maps. *Computer-Aided Design 23*, 1 (1991), 59–82.
23. Lienhardt, P. N-dimensional generalized combinatorial maps and cellular quasi-manifolds. *International Journal on Computational Geometry and Applications 4*, 3 (1994), 275–324.
24. Masuhara, H., Tatsuzawa, H., and Yonezawa, A. Aspectual caml: an aspect-oriented functional language. In *ICFP* (2005), O. Danvy and B. C. Pierce, Eds., ACM, pp. 320–330.
25. Meyers, S. *More Effective C++*. Addison Wesley, 1996.
26. Pfenning, F., and Elliot, C. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation* (1988), pp. 199–208.
27. Ramsey, N. ML module mania: A type-safe, separately compiled, extensible interpreter. *Electr. Notes Theor. Comput. Sci 148*, 2 (2006), 181–209.
28. Remy, D., and Vouillon, J. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems 4*, 1 (1998), 27–50.
29. Spicher, A. *Transformation de collections topologiques de dimension arbitraire. Application la modélisation de systèmes dynamiques.* PhD thesis, Université d'Évry, 2006.
30. Spicher, A., and Michel, O. Using rewriting techniques in the simulation of dynamical systems: Application to the modeling of sperm crawling. In *Fifth International Conference on Computational Science (ICCS'05)* (2005), vol. I, pp. 820–827.
31. Spicher, A., Michel, O., and Giavitto, J.-L. A topological framework for the specification and the simulation of discrete dynamical systems. In *Sixth International conference on Cellular Automata for Research and Industry (ACRI'04)* (Amsterdam, October 2004), vol. 3305 of *LNCS*, Springer.
32. Spicher, A., Michel, O., and Giavitto, J.-L. *Rewriting and Simulation - Application to the Modeling of the Lambda Phage Switch*, vol. Modélisation de systèmes biologiques complexes dans le contexte de la génomique. Genopole, 2006, ch. Modeling of the Lambda Phage Switch.
33. Steele Jr, G. L. Building interpreters by composing monads. In *POPL* (1994), pp. 472–492.
34. Wadler, P. The expression problem. Email to the Java Genericity mailing list, Dec. 1998.
35. Zenger, M., and Odersky, M. Independently extensible solutions to the expression problem. In *The 12th International Workshop on Foundations of Object-Oriented Languages (FOOL 12)* (Long Beach, California, 2005), ACM.

Generic Programming Combinators*

Sebastian Fischer and Frank Huch

Department of Computer Science
University of Kiel, Germany
{sebf,fhu}@informatik.uni-kiel.de

Abstract. We present a novel approach to lightweight generic programming in functional logic languages. No type system extensions like existential quantification or type classes are necessary in order to support it. Function patterns – a recent extension of functional logic programming – give rise to a lazy implementation of our approach. To demonstrate its flexibility, we have developed a library for type-based conversion between algebraic datatypes and XML documents that is available as part of the Curry system PAKCS.

Key words: Generic Programming, Function Patterns, XML Binding

1 Introduction

In strongly typed programming languages like Haskell [1] or Curry [2], it is impossible to write a single function that operates similarly on values of different types. For example, it is not possible to define an equality test that works for arbitrary datatypes. The equality test cannot be polymorphic because it has to inspect its arguments. Defining equality as an overloaded function, requires to redefine it for every datatype. Implementations of the equality test for different datatypes resemble each other a lot. Therefore, programmers long for being able to define a single function that works for all datatypes. Such a function is called *data-generic* or *polytypic* because it operates on values of multiple types.

We present a new approach to defining data-generic functions that relies on function patterns [3] – a recent extension of functional logic programming. For example, we will present a function `prettyPrint` that can be used as follows.

```
> prettyPrint Bool True
True
> prettyPrint (List Char) "IFL 2007"
['I', 'F', 'L', ' ', '2', '0', '0', '7']
> prettyPrint String "IFL 2007"
"IFL 2007"
```

This function can be implemented in Curry without extensions. The first argument resembles the name of the type of the second argument but is an ordinary Curry expression.

* This work was partially supported by the German Research Council (DFG) grant Ha 2457/5-2.

We offer the following contributions:

- We show that generic functions can be implemented via combinators in the functional logic programming paradigm without extensions of the language or the type system (Section 3).
- We show that function patterns, that were originally proposed without generic programming in mind, can be employed to implement the combinators lazily.
- The flexibility of our approach is demonstrated in the context of a generic pretty printer (Section 4) with custom rules for printing strings.
- We present a library for type-based translation between algebraic datatypes and XML documents (Section 5). Our library is remarkable because it is completely lightweight and the XML representation of data terms is not solely determined by their types.

2 The Curry Language

Curry is a declarative programming language that combines concepts from functional and logic programming. It supports lazy evaluation, algebraic datatypes and higher-order functions known from functional programming as well as nondeterminism and free variables known from logic programming. Its syntax is similar to the syntax of Haskell without type classes. We assume that the reader is familiar with the syntax of Haskell and concentrate on Curry specifics. We also discuss data declarations in detail because their structure guides the declaration of our generic programming combinators.

In Curry, a new datatype is defined by the keyword `data`. For example, the datatype for boolean values is predefined as follows.

```
data Bool = False | True
```

The name of the datatype (`Bool`) is followed by an equal sign and a list of constructors (`False` and `True`) that are separated by a vertical bar. Datatypes are called polymorphic if their name has additional arguments. For example the predefined datatype `Maybe` is used to represent optional values of an arbitrary type.

```
data Maybe a = Nothing | Just a
```

Lists have a special syntax in Curry. The type of lists is written `[a]` for an arbitrary element type `a` and its definition is built-in because the following is not valid Curry syntax.

```
data [a] = [] | a : [a]
```

According to this definition, a list is either empty (`[]`) or constructed by the binary constructor `(:)` where the first argument of `(:)` is the head of the list and the second argument of `(:)` is the tail. The datatype `[a]` is *recursive* because it is used in its own definition (in the second argument of `(:)`).

In contrast to Haskell, the names of Curry functions can start with an uppercase letter. As names of functions and names of types are in different namespaces in Curry, we can—indeed, will—define functions with names `Bool` or `Maybe`.

2.1 Logic Features

Curry supports free variables and nondeterminism known from logic programming. A free variable is declared by the keyword `free` and represents an arbitrary value of its type. The following function uses free variables.

```
last :: [a] -> a
last l | l =:= xs++[x] = x
 where x,xs free
```

The part between `|` and `=` is an optional guard that must be satisfied to apply the rule. In the guard, the strict equality operator `(=:=)` is employed to unify the argument list `l` with the result of `xs++[x]`. In order to satisfy this condition, `x` must be bound to the last element of `l`, i.e., `last` yields the last element of `l`.

Unfortunately, `last` evaluates its argument completely, due to the use of `(=:=)` in the guard. For example the application `last [undefined,42]` does not terminate, if `undefined` diverges.[1] This is inappropriate in a lazy programming language and function patterns were proposed to eliminate this drawback. In functional programming languages, patterns are built from constructors and variables only. Function patterns also allow us to use names of defined functions in pattern declarations. With this extension we can redefine the function `last` as follows.

```
last :: [a] -> a
last (xs++[x]) = x
```

Function patterns are lazy, i.e., `last [undefined,42]` evaluates to `42`. Note that `last []` fails because there is no binding for `xs` and `x` such that `[]` equals `xs++[x]`.

In general, guards and function patterns give rise to nondeterminism because there can be more than one possibility to instantiate unknown values. Consider the following definition of `choose`.

```
choose :: [a] -> a
choose (xs++x:ys) = x
```

The operation `choose` uses a function pattern to match an arbitrary element of the argument list. If this list contains more than one element, one of them is returned nondeterministically. Therefore, `choose` is usually called *nondeterministic operation* rather than function. Usually, Curry implementations compute all nondeterministic results of an expression. For example, `choose [1,2,3]` yields three results and `choose []` yields none, i.e., it *fails*.

```
> choose [1,2,3]
Result: 1 ?
Result: 2 ?
Result: 3 ?
No more solutions.
> choose []
No more solutions.
```

[1] `undefined` is defined as `undefined = undefined`

Nondeterminism is not only available indirectly in Curry. A Curry system evaluates every matching rule of an operation nondeterministically instead of choosing the first matching rule. For example, we can define `choose` directly by overlapping rules.

```
choose :: [a] -> a
choose (x:_)  = x
choose (_:xs) = choose xs
```

3 Generic Binary Encoding

As a simple example for generic programming, we discuss the implementation of an operation `showBin` that converts a value of an arbitrary datatype into a list of bits. A bit is either 0 or 1, hence, lists of bits are represented by the following datatypes.

```
data Bit = O | I
type Bin = [Bit]
```

In our approach, a generic function additionally takes a type specification. Our version of `showBin` has the following type.

```
showBin :: ShowBin a -> a -> Bin
```

For example, to convert a list of type `[Maybe Bool]` into a list of bits we define the following specifications.

```
Bool :: ShowBin Bool
Bool = cons0 False ! cons0 True

Maybe :: ShowBin a -> ShowBin (Maybe a)
Maybe a = cons0 Nothing ! cons1 Just a

List :: ShowBin a -> ShowBin [a]
List a = cons0 [] ! cons2 (:) a (List a)
```

We will soon discuss the implementation of the employed combinators. Until then, simply note that the presented type specifications resemble the data declarations of the corresponding types. For each type constructor we define a function with the same name and arguments. Recursive datatypes are translated into recursive functions which is no problem due to lazy evaluation. We can use the specifications as follows.

```
> showBin (List (Maybe Bool)) [Nothing,Just False,Just True]
Result: [I,O,I,I,O,I,I,I,O] ?
No more solutions.
```

The additional argument of type `ShowBin a` may remind a Haskell programmer of a type class constraint. Since Curry does not support type classes, we cannot hide this argument. Later, we see that the explicit type specification increases the flexibility of our approach.

The type specifications presented above will probably be provided by the implementor of `showBin` because they specify predefined types. In order to extend `showBin` to a new type, the user has to define a corresponding specification. For example, consider the datatype `Tree` and its specification of type `ShowBin Tree`.

```
data Tree a = Leaf a | Fork (Tree a) (Tree a)

Tree :: ShowBin a -> ShowBin (Tree a)
Tree a = cons1 Leaf a ! cons2 Fork (Tree a) (Tree a)
```

With this specification the user can convert trees into lists of bits.

```
> showBin (Tree Bool) (Fork (Leaf True) (Leaf False))
Result: [I,O,I,O,O] ?
No more solutions.
```

3.1 Implementation

In this subsection, we present the definition of the type `ShowBin a` and the implementation of the combinators `cons0`, `cons1`, `cons2` and `(!)` that were used to construct the type specifications for the generic operation `showBin`. The type `ShowBin a` is abstract, i.e., not exposed to the user.

```
data ShowBin a = ShowBin (a -> Bin)
```

With this definition of `ShowBin a`, the implementation of the generic operation `showBin` is surprisingly simple.

```
showBin :: ShowBin a -> a -> Bin
showBin (ShowBin sb) = sb
```

The function `showBin` simply unwraps the specification which *is*, in fact, an instance of the generic function for the corresponding type.

The combinator `(!)` is used to combine the specifications of different constructors of the same type. It is defined as follows.

```
(!) :: ShowBin a -> ShowBin a -> ShowBin a
sbO ! sbI = ShowBin sb
 where sb x = O : showBin sbO x
       sb x = I : showBin sbI x
```

We employ nondeterminism in order to combine the converter functions of the two arguments of `(!)`. We expect converter functions to fail, if they are applied to a value that they cannot convert. The converter function defined by `(!)` can convert values that can be converted by either the left or the right argument. The output is extended with the bit `O` or `I` depending on which of the converters is successful. Nondeterminism turns out to be useful in order to elegantly express the choice between different constructors of a datatype in a generic function.

There are combinators `cons`n for every reasonable arity n of possible constructors. We only present the definitions of `cons0`, `cons1` and `cons2` as they are the most interesting ones. The definition of `cons0` takes a constructor of arity 0 and yields a specification for its type.

```
cons0 :: a -> ShowBin a
cons0 cons = ShowBin sb
 where sb x | x =:= cons = []
```

A constructor without arguments is represented as the empty list of bits. We only have to ensure that the argument is indeed the specified constructor (and fail if it is not) and can do this via `(=:=)` before yielding the result.

Now consider the definition of `cons1`. The combinator `cons1` takes a unary constructor along with a specification for its argument type and yields a specification for its result type.

```
cons1 :: (a -> b) -> ShowBin a -> ShowBin b
cons1 cons sba = ShowBin sb
 where sb x | cons a =:= x = showBin sba a where a free
```

This definition uses `(=:=)` to unify `x` with `cons a`. Unfortunately, `x` is completely evaluated if the unification is successful because of the use of strict equality. In order to avoid this, we employ a function pattern to match against the constructor `cons` given as first argument.

```
cons1 :: (a -> b) -> ShowBin a -> ShowBin b
cons1 cons sba = ShowBin sb
 where c x = cons x
       sb (c a) = showBin sba a
```

If `cons` is not the root constructor of the converted argument, this matching fails. If it is, the argument of `cons` is bound to the pattern variable `a` and converted with the given specification `sba`.

Now consider the definition of `cons2`. It takes a binary constructor along with specifications for its argument types and yields a specification for its result type.

```
cons2 :: (a -> b -> c) -> ShowBin a -> ShowBin b -> ShowBin c
cons2 cons sba sbb = ShowBin sb
 where c x y = cons x y
       sb (c a b) = showBin sba a ++ showBin sbb b
```

The implementation is similar to the implementation of `cons1`. Again, a function pattern is used to match the arguments of the value that has to be converted or to fail if this value is not constructed by the given constructor `cons`. The lists of bits that represent the arguments of `cons` are concatenated to compute the result of the constructed converter function. Here is an example evaluation that demonstrates how the combinators work.

```
showBin (Maybe Bool) (Just False)
==> showBin (cons0 Nothing ! cons1 Just Bool) (Just False)
==> showBin (ShowBin sb) (Just False)
    where sb x = O : showBin (cons0 Nothing) x
          sb x = I : showBin (cons1 Just Bool) x
==> sb (Just False)
    where sb x = O : showBin (cons0 Nothing) x
          sb x = I : showBin (cons1 Just Bool) x
==> O : showBin (cons0 Nothing) (Just False)
```

```
==> O : showBin (ShowBin sb) (Just False)
    where sb x | x=:=Nothing = []
==> O : sb (Just true)
    where sb x | x=:=Nothing = []
==> failure
==> I : showBin (cons1 Just Bool) (Just False)
==> I : showBin (ShowBin sb) (Just False)
    where sb (Just a) = showBin Bool False
==> I : sb (Just False)
    where sb (Just a) = showBin Bool False
==> I : showBin Bool False
==> I : showBin (cons0 False ! cons0 True) False
==> I : showBin (ShowBin sb) False
    where  sb x = O : showBin (cons0 False) False
           sb x = I : showBin (cons0 True) False
==> I : sb False
    where   sb x = O : showBin (cons0 False) False
            sb x = I : showBin (cons0 True) False
==> I : O : showBin (cons0 False) False
==> I : O : showBin (ShowBin sb) False
    where sb x | x=:=False = []
==> I : O : sb False
    where sb x | x=:=False = []
==> [I,O]
```

The evaluation continues to convert `False` with `cons0 True` which leads to a failure just like converting `Just True` with `cons0 Nothing` failed in this computation.

In the implementation of the `cons`n combinators, we employ function patterns to match against unknown constructors. Function patterns enable us to write patterns for constructors that are provided as arguments to the combinators. In fact, we could avoid using function patterns and employ `(=:=)` to match a value. However, function patterns allow us to perform the matching without evaluating the matched arguments of the root constructor.

We can only provide a fixed number of `cons`n combinators. Constructors with an arity that is not supported could be handled as follows.

```
data Triple a b c = T3 a b c

Triple :: ShowBin a -> ShowBin b -> ShowBin c -> ShowBin Triple
Triple a b c = cons2 t3 a (cons2 (,) b c)
 where t3 x (y,z) = T3 x y z
```

The datatype `Triple` defines a ternary constructor `T3`. If no combinator `cons3` is provided, the user can employ `cons2` twice to define a specification for `Triple`s.

3.2 Parsing

We have presented combinators to construct functions that convert arbitrary data terms into lists of bits. Representing data as bit list is useless, if you cannot

convert the bits back into the original data. We can extend the presented combinators such that read and show functions are constructed simultaneously. We define a new type for specifications as well as generic read and show functions that take such a specification as first argument.

```
data BinConv a = BinConv (a -> Bin) (Bin -> (a,Bin))

showBin :: BinConv a -> a -> Bin
showBin (BinConv sb _) = sb

readBin :: BinConv a -> Bin -> (a,Bin)
readBin (BinConv _ rb) = rb
```

We exemplarily give definitions for `(!)` and `cons1`. The remaining combinators can be implemented analogously.

```
(!) :: BinConv a -> BinConv a -> BinConv a
bc0 ! bcI = BinConv sb rb
 where sb x = (0:showBin bc0 x) ? (I:showBin bcI x)
       rb (0:bs) = readBin bc0 bs
       rb (I:bs) = readBin bcI bs

cons1 :: (a -> b) -> BinConv a -> BinConv b
cons1 cons bca = BinConv sb rb
 where c x = cons x
       sb (c a) = showBin bca a
       rb bs = let (a,bs') = readBin bca bs in (cons a, bs')
```

The implementations of the show functions are identical to the definitions shown above. The read function inside the definition of `(!)` uses the first or the second specification to parse the remaining bits depending on the first. The read functions inside the definition of the `cons`n combinators simply use the read functions of the argument specifications.

4 Generic Pretty Printing

In the previous section, we presented a minimal example of generic programming in our approach. The presented combinators did not associate any additional information to the given constructors. Sometimes, additional information is necessary. For example, a pretty printer needs information about the names of constructors in order to generate a textual representation of data terms. In this section, we present the definition of a generic pretty printer that employs pretty printing combinators as described, e.g., in [6]. Our pretty printing function has the following type.

```
pretty :: Pretty a -> a -> Doc
```

The argument of type `Pretty a` is a type specification and the result of `pretty` is a document that specifies how to layout the given value of type `a`. Type specifications are defined similarly to those in the previous section.

```
data Pretty a = Pretty (a -> Doc)

pretty :: Pretty a -> a -> Doc
pretty (Pretty pt) = pt

(!) :: Pretty a -> Pretty a -> Pretty a
p1 ! p2 = Pretty pt
 where pt x = pretty p1 x
       pt x = pretty p2 x

cons0 :: String -> a -> Pretty a
cons0 name cons = Pretty (\x -> x=:=cons &> text name)

cons1 :: String -> (a -> b) -> Pretty a -> Pretty b
cons1 name cons pa = Pretty (\ (c a) -> prettyCons name [pretty pa a])
 where c x = cons x

cons2 :: String -> (a -> b -> c) -> Pretty a -> Pretty b -> Pretty c
cons2 name cons pa pb =
  Pretty (\ (c a b) -> prettyCons name [pretty pa a, pretty pb b])
 where c x y = cons x y

prettyCons :: String -> [Doc] -> Doc
prettyCons name args
  = group (nest 1 (text "(" <> text name <> line <>
      foldr1 (\d d' -> d <> line <> d') args <> text ")"))
```

Fig. 1. A Generic Pretty Printer

```
Bool :: Pretty Bool
Bool = cons0 "False" False ! cons0 "True" True

ConsList :: Pretty a -> Pretty [a]
ConsList a = cons0 "[]" [] ! cons2 "(:)" (:) a (ConsList a)
```

However, the `cons`n combinators take constructor names as an additional argument. The implementation of the generic pretty printer is shown in Figure 1. With the help of `pretty` and a pretty printing library, we can define an operation `prettyPrint` that prints arbitrary values. For example, we can print a list of booleans as follows.

```
> prettyPrint (ConsList Bool) [True,False,False]
((:) True ((:) False ((:) False [])))
```

The printed list is far from pretty because the printer displays lists like any other datatype. We want to change the default behavior of the pretty printer for values of type `[a]` for any type `a`. That is easy! We simply define a new specification function for lists.

```
List :: Pretty a -> Pretty [a]
List a = Pretty (prettyList (pretty pa))

prettyList :: (a -> Doc) -> [a] -> Doc
prettyList _ [] = text "[]"
prettyList pt (x:xs) =
  group (nest 1 (text "[" <> pt x <>
    foldr (\y d -> text "," <> line <> pt y <> d) (text "]") xs))
```

With a changed type specification, lists of booleans look much nicer.

```
> prettyPrint (List Bool) [True,False,False]
[True, False, False]
```

But what about strings? We can define a specification for characters and use it together with `List` to display strings as lists of characters.

```
Char :: Pretty Char
Char = Pretty (text . show)

> prettyPrint (List Char) "IFL 2007"
['I', 'F', 'L', ' ', '2', '0', '0', '7']
```

Usually, we would prefer strings to be displayed differently, however. We can define `String` as `Pretty (text . show)` to display strings like one would expect.

```
> prettyPrint String "IFL 2007"
"IFL 2007"
```

In our approach it is not the type of a value that determines how it is printed but an extra *specification* of its type. Although these type specifications are an additional burden for the user, they improve the flexibility of our approach. The increased flexibility is useful to print strings in different formats. It will become even more useful in the following section.

5 Generic XML Conversion

In this section, we present a library to convert arbitrary data terms to XML and vice versa. Both values of algebraic datatypes and XML documents have a tree structure. However, the translation between these tree structures is not always obvious. For example, optional values or repeated elements are modeled differently in both worlds. On the declarative programming side, there are specific algebraic datatypes to model optional and repeated elements, viz., `Maybe`- and list-types. In XML, optional or repeated elements are modeled implicitly. Optional values can simply be left out, repeated elements can be written one after the other without any enclosing construct. Moreover, XML has the notion of *attributes* that has no counterpart in algebraic datatype declarations.

In general, there are many possibilities to represent a value of an algebraic datatype as XML document and vice versa. Providing a fixed translation mechanism is too restrictive in practice. We provide combinators that enable the user

to specify how data terms should be represented in XML and do not dictate a fixed translation scheme. The user can define different converter specifications for the same datatype or different datatypes for the same XML documents.

Our approach is completely lightweight. No additional tool is necessary to read data from or write data to XML documents. The user only has to provide converter specifications that are similar to the type specifications presented in the previous sections. However, the specifications used for XML conversion resemble not only the datatype declarations but also the corresponding XML documents. For example, combinators that start with an `e` represent XML elements while combinators that start with an `a` represent XML attributes. As an example, consider the following datatype for authors and a corresponding converter specification.

```
data Author      = A Name Affiliation
data Name        = N String (Maybe String)
type Affiliation = String

Author :: XmlConv Author
Author = eSeq2 "author" A Name Affiliation

Name :: XmlConv Name
Name = eSeq2 "name" N (aString "last") (opt (aString "first"))

Affiliation :: XmlConv Affiliation
Affiliation = string
```

With this specification, the value

```
A (N "Chitil" (Just "Olaf")) "University of Kent"
```

is represented by the following XML document.

```
<author>
  <name last="Chitil" first="Olaf">
  University of Kent
</author>
```

Note that you cannot tell from the XML document whether the first name is optional or not. If it was `Nothing`, it would simply be left out in the XML representation. The last name, however, cannot be left out according to the specification. The given XML representation is just one possibility to represent values of type `Author`. We could as well define a specification that would relate the same value to another XML document like

```
<author lastname="Chitil" firstname="Olaf">
  <affiliation>University of Kent</affiliation>
</author>
```

by using the following converter specifications:

```
Name = seq2 N (aString "lastname") (opt (aString "firstname"))

Affiliation = eString "affiliation"
```

```
string :: XmlConv String
aString, eString :: String -> XmlConv String
aBool :: String -> String -> String -> XmlConv Bool
eBool :: String -> String -> XmlConv Bool

element :: String -> XmlConv a -> XmlConv a
(!) :: XmlConv a -> XmlConv a -> XmlConv a

seq0 :: a -> XmlConv a
seq1 :: (a -> b) -> XmlConv a -> XmlConv b
seq2 :: (a -> b -> c) -> XmlConv a -> XmlConv b -> XmlConv c

rep :: XmlConv a -> XmlConv [a]
opt :: XmlConv a -> XmlConv (Maybe a)
```

Fig. 2. (Some) Generic XML Converters

Part of the interface of our XML library is summarized in Figure 2. There are combinators to construct converters for primitive types like `String`. The combinator `element` takes a tag name and represents an XML element with the given contents. The combinators `seq`n are used to combine n values of possibly different types in sequence. The combinator `eSeq2` used above is a shortcut for combining `element` and `seq2`.

```
eSeq2 :: String -> (a -> b -> c) -> XmlConv a -> XmlConv b -> XmlConv c
eSeq2 name cons xca xcb = element name (seq2 cons xca xcb)
```

Attributes are associated with the next enclosing element. The combinators `rep` and `opt` represent arbitrary repetitions of elements of the same type or optional occurrences of elements respectively. The operator `(!)` is used to combine two converters of the same type and expresses alternatives. For example, an XML converter for trees (cf. Section 3) can be defined as follows.

```
Tree :: XmlConv a -> XmlConv Tree
Tree a = eSeq1 "leaf" Leaf a ! eSeq2 "fork" Fork (Tree a) (Tree a)
```

The tree `(Fork (Fork (Leaf 1) (Leaf 2)) (Leaf 3))` is represented as

```
<fork>
  <fork>
    <leaf label="1" />
    <leaf label="2" />
  </fork>
  <leaf label="3" />
</fork>
```

by the converter specification `Tree (aInt "label")`.

5.1 Restricting Specifications

The representation of a specific XML document as an algebraic data type is not unique w.r.t. every specification. If we convert a data term to XML and then convert the generated XML back into a data term, we might nondeterministically get multiple results or none at all. For example, consider the following specification for a `Name`-converter.

```
Name :: XmlConv Name
Name = eSeq2 "name" N string (opt string)
```

This specification does not use tags for the first and last name, so the value `(N "Chitil" (Just "Olaf"))` is represented as.

```
<name>Chitil Olaf</name>
```

If we read this back according to the specification, we do not obtain the original value but `(N "Chitil Olaf" Nothing)` because the first and last name are read as a single string. If the first name was mandatory, the parser would even fail because there is only one string. In general, subsequent primitive values cannot be distinguished in the XML representation and must be avoided.

Another problem are repeated optional values. Consider the following definitions.

```
type RepOpt = [Maybe Int]

RepOpt :: XmlConv RepOpt
RepOpt = rep (opt (eInt "value"))
```

The value `[Just 42]` is represented as `<value>42</value>` by `RepOpt`. There are many other terms of type `RepOpt` with this representation because arbitrarily many `Nothing`s can occur in the list without being represented in the XML document. If we convert the XML representation back into a data term, we nondeterministically get every list that contains exactly one `Just 42` and arbitrarily many `Nothing`s.

We want to restrict possible specifications in order to avoid the problems mentioned above. With phantom types [7, 8] we can employ the type system to exclude certain constructions – the details are not in the scope of this paper but can be found in our library that is distributed with the Curry system PAKCS [5]. We pose the following restrictions on specifications.

- In a specification of the form `rep x`, `x` must be *repeatable*. A specification is *repeatable* if it is an element specification or a sequence, i.e., generated by `seq`n, of repeatable specifications.
- As attributes are associated to the next surrounding element, the topmost specification must always be an element specification.

These restrictions are too weak to avoid all problems mentioned above. It is still possible to define a converter with sequences of primitive values. In order to not being overly restrictive, we allow some specifications that might cause trouble. We believe that it is important to be able to access a variety of existing XML

formats and hope that we have found an appropriate balance between safety and flexibility.

6 Related Work

An early pearl by Olivier Danvy serves as a role model for our approach to generic programming. In [9] he presents an ML version of the `printf` function known from C. This function is remarkable because its type depends on a specification given as first argument. Danvy shows how to define such a function in a Hindley-Milner typed language. The `printf` function can be typed and no extensions to the type system are necessary in order to implement it. The trick is to represent the specification as a function rather than a data term (e.g., a string like in C). In fact, the specification *is* the `printf` function, just like the specifications in our approach *are* instances of generic functions. Instead of a generic function that recursively consumes a structural type representation, we define type representations that inductively build instances of generic functions.

Functional Generic Programming: Generic programming [10] has been extensively studied in the recent years. Most approaches to functional generic programming go back to [11]. We only mention selected approaches, see [12] for a comprehensive comparison.

The Haskell community is very successful in modeling generic programming within the Haskell type system, obviating the need for language extensions. Usually, only modest extensions to the Haskell standard are necessary. [4] shows that it is even possible to model generic programming within Haskell98. In [13], existential quantification is employed – a feature that is provided by most Haskell implementations.

In contrast to the mentioned Haskell approaches, our approach does not need type classes or existential quantification. In a functional logic programming language we can define combinators to construct instances of generic functions directly. Haskell approaches employ an intermediate representation of data terms based on a structural representation of their types as *sums of products*. The lack of a common intermediate representation in our approach makes it difficult to combine different generic functions. For example, to combine a pretty printer (Section 4) with an XML converter (Section 5) the user has to supply two different specifications. We could obviate the need for more and more specifications by providing combinators to convert arbitrary data terms into sums of products. All generic functions could then be defined in terms of sums of products and, hence, freely combined. However, this would limit the flexibility of our approach. For example, we could not implement the presented library for XML conversion solely based on sums of products.

Functional XML Bindings: HaXML is a toolkit for XML processing in Haskell. It is based on the ideas described in [14] where two different approaches are compared: a combinator language for generic traversals of untyped XML data

and a tool for the translation of XML document type definitions (DTDs) into Haskell datatypes. [15] provides a formal translation of XML Schema definitions to Haskell datatypes. Both approaches generate new datatypes to represent the XML documents. [16] presents an approach to translating such generated datatypes into user-defined datatypes automatically. The drawback of [14, 15] is that the translation is fully automatic and, hence, fixed. In our approach, the user can control the translation by specifications. We extend the approach outlined in [17]. Our translation is much more flexible because XML and user data are only loosely coupled. Changes in the XML structure of an external document can be reflected directly by changing the converter specification. The internally used datatype declarations – and the program that manipulates the data – usually do not need to be adapted.

7 Conclusions

We present a new approach to lightweight generic programming in the functional logic programming language Curry. Neither type classes nor existential quantification are necessary in order to define generic functions in this programming paradigm. We employ function patterns in order to lazily dismantle values that are built from unknown constructors provided by the user. If we set laziness aside, we can use ordinary unification instead of function patterns – a recent extension of functional logic programming.

Our approach is simple to use and understand. There is a burden to the user of a generic function who has to define specifications for each newly introduced data type. These specifications are passed to a generic function in order to control its behavior. In fact, the specifications *are* the generic functions. This additional burden simultaneously increases the flexibility of our approach because different specifications can be defined for the same type. For example, the same data type can be represented as XML document in multiple ways and our approach to generic programming is flexible enough to support this.

There is also a burden for the implementor of a generic function. The combinators that are used to construct the specifications need to be rewritten for every new generic function. This is no surprise because these combinators are the definition of the generic function. They are the building blocks that are used to assemble functions for concrete data types.

Our XML library is distributed as part of the Curry system PAKCS and has been employed to implement complex converters. An example for a complex data structure is the abstract syntax tree of Curry programs. There is an XML encoding of Curry programs that does not match exactly the corresponding Curry data structure that has evolved over the years. We have re-implemented the translation between Curry programs and XML with our library. The resulting code is shorter (97 instead of 231 lines of code) and much easier to read and maintain. We did not change the Curry representation of Curry programs nor their XML representation but could mimic the original transformation that was completely hand coded.

References

1. Peyton Jones, S., et al.: Haskell 98 - a non-strict, purely functional language (1999)
2. Hanus, M., et al.: Curry: An integrated functional logic language (version 0.8.2). Available at URL `http://www.informatik.uni-kiel.de/~curry` (2006)
3. Antoy, S., Hanus, M.: Declarative programming with function patterns. In: Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05), Springer LNCS 3901 (2005) 6–22
4. Hinze, R.: Generics for the masses. Journal of Functional Programming **16**(4&5) (2006) 451–483
5. Hanus, M., et al.: PAKCS: The Portland Aachen Kiel Curry System (version 1.8.1). Available at URL `http://www.informatik.uni-kiel.de/~pakcs/` (2007)
6. Wadler, P.: A prettier printer. Journal of Functional Programming (1999)
7. Leijen, D., Meijer, E.: Domain specific embedded compilers. In: PLAN '99: Proceedings of the 2nd conference on Domain-specific languages, New York, NY, USA, ACM Press (1999) 109–122
8. Hinze, R.: Fun with phantom types. In Gibbons, J., de Moor, O., eds.: The Fun of Programming. Palgrave Macmillan (2003) 245–262 ISBN 1-4039-0772-2 hardback, ISBN 0-333-99285-7 paperback.
9. Danvy, O.: Functional unparsing. Journal of Functional Programming **8**(6) (1998) 621–625
10. Backhouse, R., Jansson, P., Jeuring, J., Meertens, L.: Generic programming — an introduction. In: LNCS. Volume 1608., Springer-Verlag (1999) 28–115 Revised version of lecture notes for AFP'98.
11. Hinze, R.: A new approach to generic functional programming. In Reps, T.W., ed.: Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00), Boston, Massachusetts, January 19-21. (2000) 119–132
12. Hinze, R., Jeuring, J., Löh, A.: Comparing Approaches to Generic Programming. Lecture Notes in Computer Science. In: Generic programming. Springer-Verlag (2006)
13. Cheney, J., Hinze, R.: A lightweight implementation of generics and dynamics. In Chakravarty, M.M., ed.: Proceedings of the 2002 ACM SIGPLAN Haskell Workshop, ACM-Press (2002) 90–104
14. Wallace, M., Runciman, C.: Haskell and XML: Generic combinators or type-based translation? In: Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99). Volume 34–9., N.Y., ACM Press (1999) 148–159
15. Atanassow, F., Clarke, D., Jeuring, J.: UUXML: A type-preserving XML Schema–Haskell data binding. In Jayaraman, B., ed.: Practical Aspects of Declarative Languages, 6th International Symposium, PADL 2004, Dallas, TX, USA, June 2004, Proceedings. Number 3057 in LNCS, Berlin Heidelberg, Springer–Verlag (2004) 71–85
16. Atanassow, F., Jeuring, J.: Inferring type isomorphisms generically. In Kozen, D., ed.: Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 2004, Proceedings. Number 3125 in LNCS, Berlin Heidelberg, Springer–Verlag (2004) 32–53
17. Fischer, S.: Resource-based web applications. In: TFP '06: Proceedings of the Seventh Symposium on Trends in Functional Programming. (2006)

Supero: Making Haskell Faster

Neil Mitchell and Colin Runciman

University of York, UK, http://www.cs.york.ac.uk/~ndm

Abstract. Haskell is a functional language, with features such as higher order functions and lazy evaluation, which allow succinct programs. These high-level features are difficult for fast execution, but GHC is a mature and widely used optimising compiler. This paper presents a whole-program approach to optimisation, which produces speed improvements of between 10% and 60% when used with GHC, on eight benchmarks.

1 Introduction

Haskell [15] can be used in a highly declarative manner, to express specifications which are themselves executable. Take for example the task of counting the number of words in a file read from the standard input. In Haskell, one could write:

$$\mathsf{main} = \mathsf{print} \circ \mathsf{length} \circ \mathsf{words} =\!\ll \mathsf{getContents}$$

From right to left, the getContents function reads the input as a list of characters, words splits this list into a list of words, length counts the number of words, and finally print writes the value to the screen.

An equivalent C program is given in Figure 1. Compared to the C program, the Haskell version is more concise and more easily seen to be correct. Unfortunately, the Haskell program (compiled with GHC) is also three times slower than the C version (compiled with GCC). This slowdown is caused by several factors:

Intermediate Lists The Haskell program produces and consumes many intermediate lists as it computes the result. The getContents function produces a list of characters, words consumes this list and produces a list of lists of characters, length then consumes the outermost list. The C version uses no intermediate data structures.

Functional Arguments The words function is defined using the dropWhile function, which takes a predicate and discards elements from the input list until the predicate becomes true. The predicate is passed as an invariant function argument in all applications of dropWhile.

Laziness and Thunks The Haskell program proceeds in a lazy manner, first demanding one character from getContents, then processing it with each of the functions in the pipeline. At each stage, a lazy thunk for the remainder of each function is created.

```
int main()
{
        int i = 0;
        int c, last_space = 1, this_space;
        while ((c = getchar()) != EOF) {
                this_space = isspace(c);
                if (last_space && !this_space)
                        i++;
                last_space = this_space;
        }
        printf("%i\n", i);
        return 0;
}
```

Fig. 1. Word counting in C.

Using the optimiser developed in this paper we can eliminate all these overheads. We obtain a program that performs *faster* than the C version. The central idea of the optimiser is to evaluate as much of the program as possible at compile time, leaving a residual program consisting only of actions dependent on the input data.

Our goal is an automatic optimisation that makes high-level Haskell programs run as fast as low-level equivalents, eliminating the current need for hand-tuning and low-level techniques to obtain competitive performance. We require no annotations on any part of the program, including the library functions.

1.1 Roadmap

We first introduce a Core language in §2, on which all transformations are applied. Next we describe our optimisation method in §3. We then give a number of benchmarks, comparing both against C (compiled with GCC) in §4 and Haskell (compiled with GHC) in §5. Finally, we review related work in §6 and conclude in §7.

2 Core Language

All our optimisations operate on a standard Core language, documented in [6]. The expression type is given in Figure 2. A program is a mapping of function names to expressions. Our Core language is higher order and lazy, but lacks much of the syntactic sugar found in Haskell. Pattern matching occurs only in case expressions, and all case expressions are exhaustive. All names are fully qualified. Haskell's type classes have been removed by the dictionary transformation [24].

The Yhc compiler, a fork of nhc [20], can output Core files. Yhc can also link in all definitions from all required libraries, producing a single Core file representing the whole program.

expr $=$	v	variable
	$\mid c$	constructor
	$\mid f$	function
	$\mid x\ y$	application
	$\mid \lambda v \rightarrow x$	lambda abstraction
	$\mid$ **let** $v = x$ **in** y	let binding
	$\mid$ **case** x **of** $\{p_1 \rightarrow y_1; ...; p_n \rightarrow y_n\}$	case expression
pat $=$	$c\ \overrightarrow{vs}$	

Where v ranges over variables, c ranges over constructors, f ranges over functions, x, y and z range over expressions and p ranges over patterns.

Fig. 2. Core syntax

The primary difference between Yhc-Core and GHC-Core [22] is that Yhc-Core is untyped. The Core is generated from well-typed Haskell, and is guaranteed not to fail with a type error. All the transformations could be implemented equally well in a typed Core language, but we prefer to work in an untyped language for simplicity of implementation.

In order to avoid accidental variable name clashes while performing transformations, we demand that all variables within a program are unique. All transformations may assume this invariant, and must ensure it as a postcondition.

3 Optimisation

Our optimisation procedure takes a Core program as input, and produces a new equivalent Core program as output. To improve the program we do not make small local changes to the original, but instead *evaluate it* so far as possible at compile time, leaving a *residual program* to be run.

Each function in the output program is an optimised version of some associated expression in the input program. Optimisation starts at the main function, and optimises the expression associated with main. Once the expression has been optimised, the outermost element in the expression becomes part of the residual program. All the subexpressions are assigned names, and will be given definitions in the residual program. If any expression (up to alpha renaming) already has a name in the residual program, then the same name is used. Each of these named inner expressions is then optimised as before.

Optimisation uses the $\mathcal{O}$ rules in Figure 3, and the simplification rules in Figure 4. We define $\mathcal{O}^*$ to be the result of applying both $\mathcal{O}$ and the simplification rules until no further changes are made. Optimisation is like evaluation, but stops if the expression to reduce is a free variable, a constructor, a primitive, or a CAF (constant applicative form – see §3.3 for more details). The one difference is that in a **let** expression the bound expression and the inner expression are *both*

$$
\begin{array}{lll}
\mathcal{O}[\![\textbf{case } x \textbf{ of } \overrightarrow{alts}]\!] = & \textbf{case } \mathcal{O}[\![x]\!] \textbf{ of } \overrightarrow{alts} \\
\mathcal{O}[\![\textbf{let } v = x \textbf{ in } y]\!] = & \textbf{let } v = \mathcal{O}[\![x]\!] \textbf{ in } \mathcal{O}[\![y]\!] \\
\mathcal{O}[\![x\ y]\!] = & \mathcal{O}[\![x]\!]\ y \\
\mathcal{O}[\![f]\!] = & \mathsf{unfold}\ f, \textbf{where } f \text{ is a non-primitive, non-CAF function} \\
= f & , \mathsf{otherwise} \\
\mathcal{O}[\![v]\!] = & v \\
\mathcal{O}[\![c]\!] = & c \\
\mathcal{O}[\![\lambda v \to x]\!] = & \lambda v \to x
\end{array}
$$

Fig. 3. Optimisation rules.

optimised – see §3.2 for the reasons. The simplification rules are all standard, and similar rules would be found in most optimising compilers.

Example 1

$$\mathsf{main} = \lambda xs \to \mathsf{map}\ \mathsf{inc}\ xs$$

$$
\begin{array}{l}
\mathsf{map} = \lambda f \to \lambda xs \to \textbf{case } xs \textbf{ of} \\
\qquad\qquad\qquad\qquad [\,] \quad \to [\,] \\
\qquad\qquad\qquad\qquad y : ys \to f\ y : \mathsf{map}\ f\ ys
\end{array}
$$

$$\mathsf{inc} = \lambda x \to x{+}1$$

This program defines a main function which increments each value in the list by one. Our main function is not a valid Haskell program, as it has the wrong type, but serves to illustrate the techniques. Note that f is passed around at runtime, when it could be frozen in at compile time. By following the optimisation procedure we end up with:

$$
\begin{array}{l}
\mathsf{main} = \lambda xs \to \textbf{case } xs \textbf{ of} \\
\qquad\qquad\qquad [\,] \quad \to [\,] \\
\qquad\qquad\qquad y : ys \to \mathsf{f0}\ y\ ys
\end{array}
$$

$$\mathsf{f0} = \lambda y \to \lambda ys \to (y{+}1) : \mathsf{main}\ ys$$

And finally by performing some trivial inlining we can obtain:

$$
\begin{array}{l}
\mathsf{main} = \lambda xs \to \textbf{case } xs \textbf{ of} \\
\qquad\qquad\qquad [\,] \quad \to [\,] \\
\qquad\qquad\qquad y : ys \to (y{+}1) : \mathsf{main}\ ys
\end{array}
$$

The residual program is now optimised – there is no runtime passing of the inc function, only a direct arithmetic operation. □

$\mathsf{case}\ (\mathsf{case}\ x\ \mathsf{of}\ \{p_1 \to y_1; ...; p_n \to y_n\})\ \mathsf{of}\ \overrightarrow{alts}$
$\quad \Rightarrow \mathsf{case}\ x\ \mathsf{of}\ \{p_1 \to \mathsf{case}\ y_1\ \mathsf{of}\ \overrightarrow{alts}$
$\qquad\qquad ; ...$
$\qquad\qquad ; p_n \to \mathsf{case}\ y_n\ \mathsf{of}\ \overrightarrow{alts}\}$

$\mathsf{case}\ c\ \overrightarrow{xs}\ \mathsf{of}\ \{...; c\ \overrightarrow{vs} \to y; ...\}$
$\quad \Rightarrow y\ [\overrightarrow{vs}\ /\ \overrightarrow{xs}]$

$\mathsf{case}\ v\ \mathsf{of}\ \{...; c\ \overrightarrow{vs} \to x; ...\}$
$\quad \Rightarrow \mathsf{case}\ v\ \mathsf{of}\ \{...; c\ \overrightarrow{vs} \to x\ [v\ /\ c\ \overrightarrow{vs}]; ...\}$

$\mathsf{case}\ (\mathsf{let}\ v = x\ \mathsf{in}\ y)\ \mathsf{of}\ \overrightarrow{alts}$
$\quad \Rightarrow \mathsf{let}\ v = x\ \mathsf{in}\ \mathsf{case}\ y\ \mathsf{of}\ \overrightarrow{alts}$

$(\mathsf{let}\ v = x\ \mathsf{in}\ y)\ z$
$\quad \Rightarrow \mathsf{let}\ v = x\ \mathsf{in}\ y\ z$

$(\mathsf{case}\ x\ \mathsf{of}\ \{p_1 \to y_1; ...; p_n \to y_n\})\ z$
$\quad \Rightarrow \mathsf{case}\ x\ \mathsf{of}\ \{p_1 \to y_1\ z; ...; p_n \to y_n\ z\}$

$(\lambda v \to x)\ y$
$\quad \Rightarrow \mathsf{let}\ v = y\ \mathsf{in}\ x$

$\mathsf{let}\ v = x\ \mathsf{in}\ (\mathsf{case}\ y\ \mathsf{of}\ \{p_1 \to y_1; ...; p_n \to y_n\})$
$\quad \Rightarrow \mathsf{case}\ y\ \mathsf{of}\ \{p_1 \to \mathsf{let}\ v = x\ \mathsf{in}\ y_1$
$\qquad\qquad ; ...$
$\qquad\qquad ; p_n \to \mathsf{let}\ v = x\ \mathsf{in}\ y_n\}$
$\quad \mathsf{where}\ v$ is not used in$y$

$\mathsf{let}\ v = x\ \mathsf{in}\ y$
$\quad \Rightarrow y\ [v\ /\ x]$
$\quad \mathsf{where}\ x$ is a lambda, variable, or used once in y

$\mathsf{let}\ v = c\ x_1 ... x_n\ \mathsf{in}\ y$
$\quad \Rightarrow \mathsf{let}\ v_1 = x_1\ \mathsf{in}$
$\qquad ...$
$\qquad \mathsf{let}\ v_n = x_n\ \mathsf{in}$
$\qquad y\ [v\ /\ c\ x_1 ... x_n]$
$\quad \mathsf{where}\ v_1 ... v_n$ are fresh

Fig. 4. Simplification rules.

Example 2

Our next example shows how our optimisation rules can carry out list deforestation [23].

$$\mathsf{main}\ xs = \mathsf{map}\ (+1)\ (\mathsf{map}\ (*2)\ xs)$$

$$\begin{array}{l}\mathsf{map}\ f\ xs = \textbf{case}\ xs\ \textbf{of}\\ \qquad\qquad\quad [\,] \quad\ \ \to [\,]\\ \qquad\qquad\quad y : ys \to f\ y : \mathsf{map}\ f\ ys\end{array}$$

The main definition is transformed (after trivial inlining) into:

$$\begin{array}{l}\mathsf{main}\ xs = \textbf{case}\ xs\ \textbf{of}\\ \qquad\qquad [\,] \quad\ \ \to [\,]\\ \qquad\qquad y : ys \to (y*2)+1 : \mathsf{main}\ ys\end{array}$$

The intermediate list has been removed, and the higher order functions eliminated by specialisation. □

3.1 Termination

A problem with the method as presented so far is that it may not terminate. There are several ways that non-termination can arise. We consider, and eliminate, each in turn.

Infinite Unfolding Consider the definition:

$$\mathsf{name} = \lambda x \to \mathsf{name}\ x$$

If the expression $\mathsf{name}\ x$ was being optimised then the optimisation function $\mathcal{O}^*$ would not terminate. We can solve this problem by either bounding the number of unfoldings, or by keeping a list of previously encountered intermediate expressions in $\mathcal{O}^*$. In practice, this situation is rare, and either choice is acceptable. We choose to bound the number of unfoldings. A large limiting value is used, which does not impact either compilation time or memory consumption in the common case.

Accumulating parameters Consider the definition:

$$\begin{array}{l}\mathsf{reverseAcc} = \lambda xs \to \lambda ys \to \textbf{case}\ xs\ \textbf{of}\\ \qquad\qquad\qquad\qquad\qquad [\,] \quad\ \ \to [\,]\\ \qquad\qquad\qquad\qquad\qquad z : zs \to \mathsf{reverseAcc}\ zs\ (z : ys)\end{array}$$

This function is the standard reverse function, with an accumulator. The problem is that successive iterations of the optimisation produce progressively larger subexpressions. A definition is first created for $\mathsf{reverseAcc}\ \_\ \_$, then for $\mathsf{reverseAcc}\ \_\ (\_ : \_)$, then $\mathsf{reverseAcc}\ \_\ (\_ : \_ : \_)$. The residual program is infinite.

The solution is to bound the size of the input expression associated with each definition in the residual program. The size of the expression being optimised can be reduced by lifting subexpressions into a let binding, then placing this let binding in the residual program. By bounding the size of the expression, we bound the number of functions in the residual program.

If the bound is too high, optimisation takes too long and the residual program is excessively large. If the bound is too low then too little is achieved by optimisation. We return to the issue of the size of this bound in §5.2.

Direct Repetition We claim that $\mathcal{O}^*$ terminates with bounded unfoldings and bounded expression size. It is often useful to detect an expression which appears to be repeating, and preemptively bound it. Consider the reverseAcc example – the recursive pattern is an instance of *direct repetition*. Let α be a context, and $\alpha\langle e\rangle$ be the result of substituting e for the hole in the context α. An expression x is directly repeating if $x \simeq \alpha\langle\alpha\langle\beta\rangle\rangle$ where β is an expression, α is a non-empty context and $\simeq$ is equality where all variables are considered equal.

Example 3

The following expressions have direct repetition.

$x : y : xs$ **where** $\alpha = x : \bullet, \beta = xs$
$f\ (f\ x)$ **where** $\alpha = f\ \bullet\ , \beta = x$
case x_1 **of** $\{[\,] \to \mathsf{nil}; y : ys \to$ **case** x_2 **of** $\{[\,] \to \mathsf{nil}; z : zs \to \mathsf{cons}\}\}$
where $\alpha =$ **case** x_1 **of** $\{[\,] \to \mathsf{nil}; y : ys \to \bullet\}, \beta = \mathsf{cons}$

□

If direct repetition is encountered, then the repeating expression is lifted to a top-level let binding, and output directly into the residual program.

Example 4

Take the reverseAcc example. During optimisation, the expression becomes:

reverseAcc $xs\ (y_1 : y_2 : ys)$

The second argument to reverseAcc is an instance of direct repetition, and is lifted to a let binding.

let $v = y_1 : y_2 : ys$
in reverseAcc $xs\ v$

Now the expression bound at the let, and the inner expression, are optimised separately. □

3.2 Let Bindings

The rule for let bindings in Figure 3 may seem curious. The other rules simply follow evaluation order, but the let rule optimises *both* the bound expression and the inner expression. This is a critical choice, which enhances the optimisation performed by the system, without duplicating computation of let bindings.

In the Core language a let expression introduces a binding, which is shared. Given the expression **let** $v = x$ **in** y, even if v is referred to multiple times in y, then the expression x is computed at most once. It is important that sharing of *expensive* functions is preserved. Yet, by inlining *cheap* let expressions, better optimisation can be achieved. Taking the following fragment from a previous example:

let $f = \mathsf{inc}$
in $f\ y : \mathsf{map}\ f\ ys$

If f is not inlined, then the recursive call to map would still contain a functional variable to be passed at runtime. But how can we tell whether inc is cheap enough to be inlined? The solution is to optimise inc first:

let $f = \lambda x \rightarrow x{+}1$
in $f\ y : \mathsf{map}\ f\ ys$

It is now clear that f is a lambda, so no shared computation is lost by inlining it.

3.3 CAF's

A CAF (constant applicative form) is a top level definition of zero arity. In Haskell, CAFs are computed at most once per program run, and retained as long as references to them remain. Consider the program:

$\mathsf{caf} = \mathsf{expensive}$

$\mathsf{main} = \mathsf{caf}{+}\mathsf{caf}$

In this program caf would only be computed once. If a CAF is inlined then this may result in a computation being performed more than would otherwise occur. To ensure that we do not duplicate computations, we never inline CAF's.

4 Performance Compared With C Programs

The benchmarks we have chosen are inspired by the Unix `wc` command – namely character, word and line counting. We require the program to read from the standard input, and write out the number of elements in the file. To ensure that we test computation speed, not IO speed (which is usually determined by the buffering strategy, rather than optimisation) we demand that all input is read

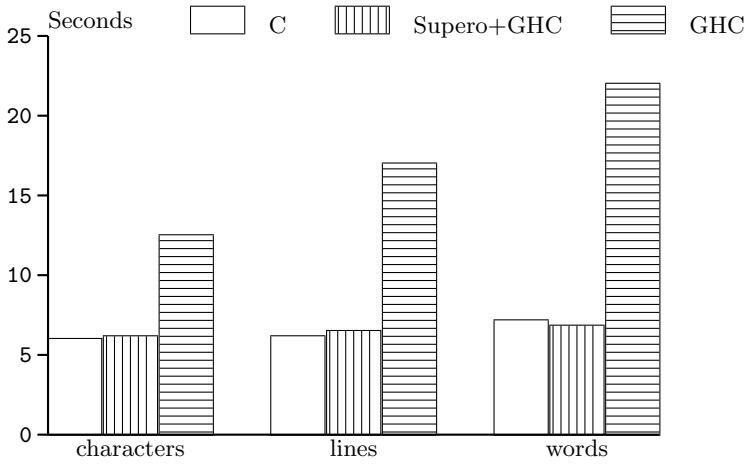


Fig. 5. Benchmarks with C, Supero+GHC and GHC alone.

using the standard C getchar function only. Any buffering improvements, such as reading in blocks or memory mapping of files, could be performed equally in all compilers.

All the C versions are implemented following a similar pattern to Figure 1. Characters are read in a loop, with an accumulator recording the current value. Depending on the program, the body of the loop decides when to increment the accumulator. The Haskell versions all follow the same pattern as in the Introduction, merely replacing words with lines, or removing the words function for character counting.

We performed all benchmarks on a machine running Windows XP, with a 3GHz processor and 1Gb RAM. All benchmarks were run over a 50Mb log file, repeated 10 times, and the lowest value was taken. The C versions used GCC[1] version 3.4.2 with -O3. The Haskell version used GHC [21] 6.6.1 with -O2. The Supero version was compiled using our optimiser, then written back as a Haskell file, and compiled once more with GHC 6.6.1 and -O2.

The results are given in Figure 5. In all the benchmarks C and Supero are within 10% of each other, while GHC trails further behind.

4.1 Identified Haskell Speedups

During initial trials using these benchmarks, we identified two unnecessary bottlenecks in the Haskell version of word counting. Both were remedied before the presented results were obtained.

[1] `http://gcc.gnu.org/`

```
words :: String → [String]
words s = case dropWhile isSpace s of
              [] → []
              x  → w : words y
                   where (w, y) = break isSpace x

words' s = case dropWhile isSpace s of
               []     → []
               x : xs → (x : w) : words' (drop1 z)
                        where (w, z) = break isSpace xs

drop1 []       = []
drop1 (x : xs) = xs
```

Fig. 6. The words function from the Haskell standard libraries, and an improved words′.

Slow isSpace function The first issue is that isSpace in Haskell is much more expensive than isspace in C. The simplest solution is to use a FFI (Foreign Function Interface) [14] call to the C isspace function in all cases, removing this factor from the benchmark. A GHC bug (number 1473) has been filed about the slow performance of isSpace.

Inefficient words function The second issue is that the standard definition of the words function (given in Figure 6) performs two additional isSpace tests per word. By appealing to the definitions of dropWhile and break it is possible to show that in words the first character of x is not a space, and that if y is non-empty then the first character is a space. The revised words′ function uses these facts to avoid the redundant isSpace tests.

4.2 Potential GHC Speedups

We have identified three factors limiting the performance of residual programs when compiled by GHC. These problems cannot be solved at the level of Core transformations. We suspect that by fixing these problems, the Supero execution time would improve by between 5% and 15%.

Strictness inference The GHC compiler is overly conservative when determining strictness for functions which use the FFI (GHC bug 1592). The getchar function is treated as though it may raise an exception, and terminate the program, so strict arguments are not determined to be strict. If GHC provided some way to mark an FFI function as not generating exceptions, this problem could be solved. The lack of strictness information means that in the line and word counting programs, every time the accumulator is incremented, the number is first unboxed and then reboxed [17].

Heap checks The GHC compiler follows the standard STG machine [12] design, and inserts heap checks before allocating memory. The purpose of a heap check is to ensure that there is sufficient memory on the heap, so that allocation of memory is a cheap operation guaranteed to succeed. GHC also attempts to lift heap checks: if two branches of a case expression both have heap checks, they are replaced with one shared heap check before the case expression. Unfortunately, with lifted heap checks, a tail-recursive function that allocates memory only upon exit can have the heap test executed on every iteration (GHC bug 1498). This problem affects the character counting example, but if the strictness problems were solved, it would apply equally to all the benchmarks.

Stack checks The final source of extra computation relative to the C version are stack checks. Before using the stack to store arguments to a function call, a test is performed to check that there is sufficient space on the stack. Unlike the heap checks, it is necessary to analyse a large part of the flow of control to determine when these checks are unnecessary. So it is not clear how to reduce stack checks in GHC.

4.3 Why Supero Outperforms C for the Wordcount Benchmark

The most curious result is that Supero outperforms C on wordcounting, by about 6% – even with the problems discussed! The C program presented in Figure 1 is not optimal. The variable `last_space` is a boolean, indicating whether the previous character was a space, or not. Each time round the loop a test is performed on `last_space`, even though its value was determined and tested on the previous iteration. The way to optimise this code is to have two specialised variants of the loop, one for when `last_space` is true, and one for when it is false. When the value of `last_space` changes, the program would transition to the other loop. This pattern effectively encodes the boolean variable in the program counter, and is what the Haskell program has managed to generate from the high-level code.

However, in C it is quite challenging to capture the required control flow! The program needs two loops, where both loops can transition to the other. Using `goto` turn off many critical optimisations in the C compiler. Tail recursion is neither required by the C standard, nor supported by most compilers. The only way to express the necessary pattern is using nested while loops, but unlike newer imperative languages such as Java, C does not have named loops – so the inner loop cannot break from the outer loop if it reaches the end of the file. The only solution is to place the nested while loops in a function, and use `return` to break from the inner loop. This solution would not scale to a three-valued control structure, and substantially increases the complexity of the code.

5 Performance Compared With GHC Alone

The standard set of Haskell benchmarks is the nofib suite [11]. It is divided into three categories of increasing size: imaginary, spectral and real. Many small

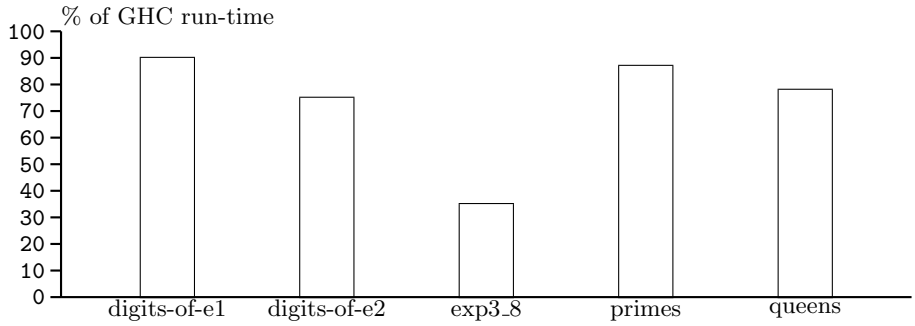


Fig. 7. Runtime, relative to GHC.

Program	Source	Residual	Bound	% GHC Size	% GHC Time
digits-of-e1	521	1676	13	110	90
digits-of-e2	1235	515	12	99	75
exp3_8	380	1138	5	104	35
primes	422	356	12	101	87
queens	637	4265	8	116	78

Program is the name of the program; **Source** is the number of lines of pretty printed source code including all libraries; **Residual** is the number of lines after optimisation; **Bound** is the termination bound used; **Size** is the size of the resultant binary as a percentage of the GHC binary size; **Time** is the runtime as a percentage of GHC run-time.

Table 1. Result on the nofib suite.

Haskell programs increase in size substantially once the libraries are included, particularly when type classes are involved. Because of the relatively large source code size of even small examples, we have limited our focus to five benchmarks drawn from the imaginary section. We have chosen programs which do not perform large amounts of IO.

The benchmarks are: digits-of-e1 and digits-of-e2, both of which compute the digits of e by different methods; exp3_8 computes 3^8 using Peano numbers and the Num class; primes computes a list of prime numbers; and queens counts the safe layouts of queen pieces on a chess board. All benchmarks were run with parameters that require runtimes of between 3 and 5 seconds for GHC.

The results of these benchmarks are given in Figure 7, along with detailed breakdowns in Table 1. In all benchmarks Supero+GHC performs at least 10% faster than GHC alone, and in one case is nearly three times faster. Binaries were at most 10% larger than those from GHC alone, and in one case the binary was even marginally smaller.

5.1 GHC's optimisations

For these benchmarks it is important to clarify which optimisations are performed by GHC, and which are performed by Supero. Core output from Yhc, compiled using GHC without any prior optimisation, would *not* perform as well as the original program compiled using GHC. GHC has two special optimisations that work in a restricted number of cases, but which Supero is unable to take advantage of.

Dictionary Removal Functions which make use of type classes are given an additional dictionary argument. In practice, GHC specialises many such functions by creating code with a particular dictionary frozen in. This optimisation is specific to type classes – a tuple of higher order functions is not similarly specialised. After compilation with Yhc, the type classes have already been converted to tuples, so Supero must be able to remove the dictionaries itself. One benchmark where dictionary removal is critical is digits-of-e2.

List Fusion GHC relies on names of functions, particularly foldr/build [19], to apply special optimisation rules such as list fusion. Many of GHC's library functions, for example iterate, are defined in terms of foldr to take advantage of these special properties. After transformation with Yhc, these names are destroyed, so no rule based optimisation can be performed. One example where list fusion is critical is primes, although it occurs in most of the benchmarks to some extent.

Supero has no special purpose optimisations which rely on named functions or desugaring knowledge. The one benchmark where no GHC specific optimisations apply is exp3_8, which operates solely on Peano numbers – a type GHC has no inbuilt knowledge of. Hence the advantage of Supero in exp3_8: while GHC is limited to basic inline/simplify transformations, Supero is able to remove some intermediate data structures.

5.2 Termination Bound

Table 1 includes a column indicating the size bound that was applied to expressions. Out of the five benchmarks, both primes and queens could be run at any greater bound and would still produce the same program – the direct repetition criteria (see §3.1) bounds the expressions on its own. For the remaining programs, a bound was chosen to ensure that the compilation process was quick (under two seconds). By increasing the termination bound the size of the residual program would increase, but the generated program may execute faster.

The existence of a termination bound requiring different values for different programs is a cause for concern. In a large program it is likely that different parts of the program would require different bounds on the size of the generated expression – something not currently possible. We suspect that the most promising direction is to augment the direct repetition criterion to obtain termination in all practical cases without resorting to a depth bound.

6 Related Work

Partial evaluation There has been a lot of work on partial evaluation [7], where a program is specialised with respect to some static data. The emphasis is on determining which variable can be entirely computed at compile time, and which must remain in the residual program. Partial evaluation is particularly appropriate for specialising an interpreter with an expression tree to generate a compiler automatically, often with an order of magnitude speedup, known as the First Futamura Projection [4]. The difference between our work and partial evaluation is that we fold back definitions, and perform no binding time analysis. Our method is certainly less appropriate for specialising an interpreter, but in the absence of static data, is still able to show improvements.

Deforestation The deforestation technique [23] removes intermediate lists in computations. This technique has been extended in many ways to encompass higher order deforestation [8] and work on other data types [3]. Probably the most practically motivated work on deforestation has come from those attempting to restrict deforestation, in particular shortcut deforestation [5], and newer approaches such as stream fusion [2]. In this work certain named functions are automatically fused together. By rewriting library functions in terms of these special functions, fusion occurs. Shortcut deforestation is limited to cases where the correct underlying function is used – sometimes requiring unnatural definitions.

GRIN The GRIN approach [1] is currently being implemented in the jhc compiler [10], with promising initial results. GRIN works by first translating to a monadic intermediate language, then repeatedly performing a series of optimisations, using whole program transformation. The intermediate language is at a much lower level than our Core language, so jhc is able to perform detailed optimisations that we are unable to express.

Other Transformations One of the central operations within our optimisation in inlining, a technique that has been used extensively within GHC [18]. We generalise the constructor specialisation technique [16], by allowing specialisation on any arbitrary expression, including constructors.

Lower Level Optimisations Our optimisation works at the Core level, but even once optimal Core has been generated there is still some work before optimal machine code can be produced. Key optimisations include strictness analysis and unboxing [17]. In GHC both of these optimisations are done at the Core level, using a Core language extended with unboxed types. After this lower level Core has been generated, it is then transformed in to STG machine instructions [13], before being transformed into assembly code. There is still work being done to modify the lowest levels to take advantage of the current generation of microprocessors [9]. We rely on GHC to perform all these optimisations after Supero generates a residual program.

7 Conclusions and Future Work

We have introduced an optimising front-end which can enhance the results of back-end compilation using GHC – at least for some small programs. Our optimiser is simple – the Core transformation is expressed in just 300 lines of Haskell, yet it replicates many of the performance enhancements of GHC in a more general way. Our initial results are promising, but incomplete. There are three main obstacles that need to be tackled:

Termination We are confident that Supero terminates, but only by use of a crude bound on expression size whose optimal value varies for different programs. To increase the applicability of our optimiser, we would like to remove the depth bound, or at least reduce our reliance upon it.

Benchmarks Eight small benchmarks are not enough. We would like to obtain results for all the remaining benchmarks in the nofib suite.

Performance The performance results presented in §5 are disappointing. Earlier versions of Supero were able to obtain a 50% speed up in the primes benchmark, but decreased performance in other benchmarks. We suspect that much better performance can be obtained.

The Programming Language Shootout[2] has shown that low-level Haskell can compete with with low-level imperative languages such as C. Our goal is that Haskell programs can be written in a high-level declarative style, yet still perform competitively.

Acknowledgements We would like to thank Simon Peyton Jones, Simon Marlow and Tim Chevalier for help understanding the low-level details of GHC, and Peter Jonsson for helpful discussions.

References

1. Urban Boquist and Thomas Johnsson. The GRIN project: A highly optimising back end for lazy functional languages. In *Proc IFL '96*, volume 1268 of *LNCS*, pages 58–84. Springer-Verlag, 1996.
2. Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *Proc ICFP '07*. ACM Press, April 2007.
3. Duncan Coutts, Don Stewart, and Roman Leshchinskiy. Rewriting Haskell strings. In *Proc PADL 2007*, pages 50–64. Springer-Verlag, January 2007.
4. Yoshihiko Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.
5. Andrew Gill, John Launchbury, and Simon Peyton Jones. A short cut to deforestation. In *Proc FPCA '93*, pages 223–232. ACM Press, June 1993.
6. Dimitry Golubovsky, Neil Mitchell, and Matthew Naylor. Yhc.Core - from Haskell to Core. *The Monad.Reader*, (7):45–61, April 2007.
7. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International, 1993.

[2] `http://shootout.alioth.debian.org/`

8. Simon Marlow. *Deforestation for Higher-Order Functional Programs*. PhD thesis, University of Glasgow, 1996.
9. Simon Marlow, Alexey Rodriguez Yakushev, and Simon Peyton Jones. Faster laziness using dynamic pointer tagging. In *Proc. ICFP '07*. ACM Press, October 2007.
10. John Meacham. jhc: John's haskell compiler. `http://repetae.net/john/computer/jhc/`, 2007.
11. Will Partain et al. The `nofib` Benchmark Suite of Haskell Programs. `http://darcs.haskell.org/nofib/`, 2007.
12. Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
13. Simon Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *JFP*, 2(2):127–202, 1992.
14. Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In *Engineering theories of software construction, Marktoberdorf Summer School*, 2002.
15. Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
16. Simon Peyton Jones. Constructor specialisation for Haskell programs. In *Proc. ICFP '07*. ACM Press, October 2007.
17. Simon Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In J. Hughes, editor, *Proc FPCA '91*, volume 523 of *LNCS*, pages 636–666, Cambridge, Massachussets, USA, August 1991. Springer-Verlag.
18. Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *JFP*, 12:393–434, July 2002.
19. Simon Peyton-Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In *Proc. Haskell '01*, pages 203–233. ACM Press, 2001.
20. Niklas Röjemo. Highlights from nhc - a space-efficient Haskell compiler. In *Proc. FPCA '95*, pages 282–292. ACM Press, 1995.
21. The GHC Team. The GHC compiler, version 6.6. `http://www.haskell.org/ghc/`, October 2006.
22. Andrew Tolmach. An External Representation for the GHC Core Language. `http://www.haskell.org/ghc/docs/papers/core.ps.gz`, September 2001.
23. P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proc ESOP '88*, volume 300 of *LNCS*, pages 344–358. Berlin: Springer-Verlag, 1988.
24. Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. POPL '89*, pages 60–76. ACM Press, 1989.

Checking Dependent Types Efficiently

Dirk Kleeblatt

Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
klee@cs.tu-berlin.de

Abstract. Type checkers for dependent types need to evaluate user defined functions during type checking. For this, current implementations typically use an interpreter, which has several drawbacks. We show, how at this stage compiled code can be used for a language with lazy evaluation.

1 Introduction

This article gives an early report on the implementation of Ulysses, a lazy functional language with dependent types. Ulysses is quite similar to Cayenne [1]. One of the similarities is, that there is no sharp distinction between terms and types, the only difference is that some expressions may be used as types while others may not. The consequences of identifying terms and types are significant: Functions can be used not only to construct the usual terms like natural numbers, lists and so on, but also to construct types. Hence, we need to evaluate some user defined functions during type checking time. To circumvent the drawbacks of interpreted code, we use compilation to native machine code instead, which is complicated by the special requirements of evaluation for dependent type checking, namely evaluation under λ abstractions and case analyses. To our knowledge, this is the first implementation using compiled code during type checking for a lazy language. Earlier work exists that is restricted to eager evaluation [2].

In the following, we introduce the language Ulysses (Sect. 2). We adapt former work on strict languages to the needs of lazy evaluation (Sect. 3), and describe the necessary compilation technique (Sect. 4) and runtime system (Sect. 5). While this leads to a working system, it still can be improved (Sect. 6). We give some further notes on our implementation (Sect. 7), a comparison with related work (Sect. 8) and directions for future research (Sect. 9).

2 A description of Ulysses

It is common in functional languages to have type constructors which may be regarded as functions taking types to types. In Haskell, for examples, `Maybe` can be applied to the type `Integer`, yielding a new type `Maybe Integer`. But the possibilities to define type constructors are usually restricted to the definition of

parametric algebraic data types. In Ulysses, such restrictions do not exist. It is possible to write functions that take arbitrary terms to new types, using the full feature set of the underlying language like higher order functions and recursion.[1]

An example for recursive type definitions in Ulysses is shown in Fig. 1. The syntax is similar to Haskell. Line 2 gives the well known definition of Peano numbers. Line 1 gives a type declaration for `nat`, `#0` is the type of all (small) types. More interesting is the declaration and definition of `vector` in lines 4–6. The first argument to `vector` is of type `#0`, so it is a type by itself. The second argument is a natural number, and the result type is `#0` again, so this function computes a type when given a term of type `nat` as an argument. This computed type is defined by recursion, the base case is shown in line 5: A vector of length zero can only be created by the constructor `Nil` which takes no arguments. A vector of length `S x` is created by the constructor `Cons`, which prepends an element of type `a` (a parameter to our definition) to an vector of length `x`. So an expression of type `vector a n` is guaranteed to contain exactly `n` elements of type `a`.

```
1   nat :: #0;
2   nat = data Z | S nat;
3
4   vector :: #0 -> nat -> #0;
5   vector a Z     = data Nil;
6   vector a (S x) = data Cons a (vector a x);
7
8   v :: vector nat (S (S Z));
9   v = Cons Z (Cons Z Nil);
10
11  add :: nat -> nat -> nat;
12  add Z     b = b;
13  add (S a) b = S (add a b);
14
15  append :: forall a :: #0 .
16     forall n :: nat . vector a n ->
17     forall m :: nat . vector a m ->
18     vector a (add n m);
19  append a Z     Nil          m vm = vm;
20  append a (S p) (Cons ft rt) m vm =
21     Cons ft (append a p rt m vm)
```

Fig. 1. Vectors as lists with fixed length

[1] Comparable with Cayenne, type checking Ulysses programs is not decidable. The user has to ensure, that no infinite recursions are used in functions at the type level.

To see an application of `vector`, take a look at lines 8 and 9. Here, `v` is declared to be a vector of length two. To type check the definition, the type checker has to do three reduction steps (note that the `data` keyword can be used to define anonymous types):

```
vector nat (S (S z))
→ data Cons nat (vector nat (S Z))
→ data Cons nat (data Cons nat (vector nat Z))
→ data Cons nat (data Cons nat (data Nil))
```

To perform these reductions, languages that support comparable type definitions typically use an interpreter during type checking. CAYENNE, the language most closely related to ULYSSES, is implemented this way. But this has several disadvantages:

- The first one is the reduced performance of interpreted code compared to compiled code.
- Even neglecting performance reasons, from a software engineering point of view this situation is very unsatisfactory. When writing a compiler, an additional interpreter is needed just for type checking. And when the language is extended later on, two different parts of code have to be adapted: the compiler as well as the interpreter.
- This might get worse in the presence of (even small) differences in the semantics of the interpreter and the compiler: When computations give a different result at run-time than during type checking, the type safety will most probably be violated.

However, replacing an interpreter by a compiler in such a type checker is not as easy as it looks on first sight. The terms that have to be reduced may contain free variables, as illustrated by the rest of our example.

Lines 11–13 define the addition of natural numbers as usual. Lines 15–21 give the definition of the append function for vectors. Using dependent types, its type declaration can give a good specification of this function. The type can be read as follows: for any type `a`, given a natural number `n`, and a vector containing `n` elements of type `a`, and furthermore a second natural number `m` together with a vector of size `m`, return a vector containing `n` plus `m` elements.

In the definition of the recursive case in lines 20 and 21, the resulting type `vector a (add n m)` can be narrowed using the information from the pattern matching, `n` must be equal to `S p`, so we can do the following reductions which are necessary to type check the right hand side:

```
vector a (add (S p) m)
→ vector a (S (add p m))
→ data Cons a (vector a (add p m))
```

In these reductions, all redexes contain not only the free type variable `a`, but also two unknown natural numbers, `p` and `m`. Usually, compilers cannot handle these free variables.

The problem is indeed more general: It is necessary to compute normal forms, while code generated by usual compilers computes only weak head normal forms. Hence, a compiler to be used for a type checker for dependently typed languages must cope with evaluation under λ abstraction and case analyses.

3 Weak Normalization and Readback

Our approach does not compute normal forms of expressions in one big step. Instead, we use a well known (and slightly adopted) weak head evaluator, examine the weak head normal forms computed by this evaluator, and extract remaining redexes. These can then be recursively reduced.

This proceeding is not new, it was introduced by Grégoire and Leroy [2]. However, they restricted their focus on strict evaluation, using the ZAM abstract machine as a weak evaluator. Our work employs lazy evaluation, taking the spineless tagless g-machine by Peyton Jones [3] as a weak evaluator. The differences between ZAM and STG machine are quite significant, so transferring the existing work from strict evaluation to lazy evaluation is nontrivial.

We adopt the definition from [2] of the strong normalization function $\mathcal{N}$ to the needs of the STG machine. This function is defined in terms of two helper functions. The weak evaluation function $\mathcal{V}$ reduces terms to weak head normal forms, the readback function $\mathcal{R}$ scrutinizes the resulting weak head normal form, extracts remaining redexes, and applies $\mathcal{N}$ recursively on unevaluated subterms.

Definition 1. *The normalization function $\mathcal{N}$ is defined as*

$$\mathcal{N}(e) = \mathcal{R}(\mathcal{V}(e)).$$

The weak evaluation of Ulysses expressions is done by compiling them first to STG code and then to native machine code (cf. Sect. 4). This machine code is executed, and the execution stops, when a weak head normal form is reached. Their structure can the be extracted by interpreting the final machine state (cf. Sect. 5).

Definition 2. *A weak head normal form is given by one of four cases:*

$$\begin{aligned} v ::= {} & C\ e_1\ \ldots\ e_n && (1)\\ \mid {} & x\ e_1\ \ldots\ e_n && (2)\\ \mid {} & \lambda x\,.\,e && (3)\\ \mid {} & \Gamma[\,\texttt{case}\ x\ e_1\ \ldots\ e_n\,] && (4) \end{aligned}$$

Note, that the subexpressions e and e_i are *machine representations* of expressions, i.e. closure pointers into the heap.

Line 1 describes the application of a constructor to n argument expressions ($n = 0$ for nullary constructors), while line 2 denotes a free variable x, again applied to n arguments ($n = 0$ for no arguments). Line 3 is a unsaturated function, i. e. a function expecting at least one additional argument.

The last case, number 4, looks most unusual. It is encountered when a case distinction is to be made, and the scrutinee is not a constructor term, but an application of a free variable to zero or more arguments. The context Γ fixes (amongst others) the continuation for each possible constructor (i. e. the result of the case distinction), and is defined by a set of machine stacks and registers.

Next, we focus on the readback function $\mathcal{R}$. It is defined by case analysis on weak head normal forms.

Definition 3. *The readback function $\mathcal{R}$ is defined as*

$$\mathcal{R}(C\ e_1\ \ldots\ e_n) = C\ \mathcal{N}(e_1)\ \ldots\ \mathcal{N}(e_n) \tag{5}$$

$$\mathcal{R}(x\ e_1\ \ldots\ e_n) = x\ \mathcal{N}(e_1)\ \ldots\ \mathcal{N}(e_n) \tag{6}$$

$$\mathcal{R}(\lambda x\,.\,e) = \lambda y\,.\,\mathcal{N}((\lambda x\,.\,e)\ y) \quad (y\ \textit{fresh}) \tag{7}$$

$$\begin{aligned}\mathcal{R}(\Gamma[\,\texttt{case}\ x\ e_1\ \ldots\ e_n\,]) = \texttt{case}\ x\ \mathcal{N}(e_1)\ \ldots\ \mathcal{N}(e_n)\ \texttt{of}\\ \{\ C_i\ \vec{x}_i \to \mathcal{N}(\Gamma[\,C_i\ \vec{x}_i\,])\ \}\\ \textit{(where } C_i \textit{ are the possible constructors}\\ \textit{and } \vec{x}_i \textit{ fresh)}\end{aligned} \tag{8}$$

Equation 5 neatly shows the idea of strong normalization by weak evaluation and readback: When a constructor term has been evaluated, the resulting normal form again is a constructor term. However, the constructor arguments $e_1\ \ldots\ e_n$ have not necessarily been evaluated in the first step, because we employ lazy evaluation. Hence, these arguments have to be normalized by $\mathcal{N}$, which results in their (weak) evaluation and subsequent readback (cf. the definition of $\mathcal{N}$ in Def. 1), which in turn might result in further weak evaluations, and so on.

Equation 6 is completely analogous: when a free variable is applied to zero or more arguments, we have to normalize these arguments.

In equation 7, evaluation under λ is described. We generate a fresh variable y, normalize the expression $(\lambda x\,.\,e)\ y$ where y is free, and the resulting normal form is placed beneath a λ abstraction. This might look complicated at a first glance, a usual definition would involve substitution, resulting in the normalization of $e[y/x]$ instead of the application $(\lambda x\,.\,e)\ y$. However, it is important that the abstraction $\lambda x\,.\,e$ given as argument to $\mathcal{R}$ remains unchanged on the right hand side of the definition. Recall that this abstraction is represented by heap closures of our weak evaluator. It is quite comfortable not being forced to define substitution on these structures, while it is quite easy to push additional arguments on a machine stack.

When evaluation gets stuck because a case analysis of an application of a free variable cannot be reduced further, we have to evaluate all possible branches of the analysis, as shown in Def. 8. We generate all possible constructors C_i, applied to fresh variables $\vec{x}_i$ according to their arity. Which constructors (and with which arity) are needed can be deduced from the type of the free variable x and it's arguments. Details to this can be found in Sect. 5.

$$
\begin{aligned}
e &::= C\ \{x_1\ \ldots\ x_n\} && (9)\\
&\mid\ x\ \{x_1\ \ldots\ x_n\} && (10)\\
&\mid\ \texttt{let}\ bs\ \texttt{in}\ e && (11)\\
&\mid\ \texttt{case}\ e\ \texttt{of}\ as && (12)\\
bs &::= x_1 = lf_1;\ \ldots\ ; x_n = lf_n && (13)\\
lf &::= \{x_1\ \ldots\ x_n\} \setminus \pi\ \{y_1\ \ldots\ y_m\}\ \rightarrow\ e && (14)\\
\pi &::= \texttt{u} \mid \texttt{n} && (15)\\
as &::= a_1;\ \ldots\ ; a_n;\ d && (16)\\
a &::= C\ \{x_1\ \ldots\ x_n\}\ \rightarrow\ e && (17)\\
d &::= \_\ \rightarrow\ e && (18)
\end{aligned}
$$

Fig. 2. Grammar for STG code

Every new constructor application is then executed in context Γ[2]. As mentioned, this context captures the branches for the case analysis, so each $\Gamma[\, C_i\ \vec{x}_i\,]$ is normalized to the corresponding case arm for $C_i\ \vec{x}_i$.

Of course, since we use lazy evaluation, the arguments $e_1\ \ldots\ e_n$ may be unevaluated, so we normalize them using $\mathcal{N}$.

4 The Spineless Tagless G-Machine

Up to now, we treated the weak evaluation function $\mathcal{V}$ fairly abstract. In this section we describe the spineless tagless g-machine, which we used to implement a lazy weak evaluator. A more complete description can be found in [3]. Here, we will focus on the aspects of the STG machine that are relevant for our modifications and the machine state interpretation we detail in Sect. 5.

Every Ulysses expression that has to be normalized is translated to native machine code in two steps: first, we create STG code, which looks like a restricted and annotated functional programming language, and from this we generate target machine assembly code.

4.1 The STG Language

STG code is formed according to the grammar in Fig. 2. We describe a simplified variant of STG code. The original formulation is prepared for *primitive values* to deal e.g. with unboxed integers. Additionally, we do not distinguish recursive and nonrecursive `let` bindings, we treat all bindings as recursive.

[2] executing $\Gamma[e]$ means entering the closure for e in the machine state given by Γ (cf. Sect. 5)

The first form of STG expressions is a constructor application. However, the constructor has to be *saturated*, i.e. all arguments according to the arity of the constructor C have to be present. Moreover, the constructor arguments are restricted to be variables.

A function application is restricted in a similar manner: the function x and all arguments given have to be variables. No anonymous functions exist in this intermediate language, they have to be bound globally or locally. Function applications do not have to be saturated but can be partial.

Local definitions are bound by `let` expressions. Each binding b associates a so-called *lambda form* of the syntactic category lf with a name. A lambda form is annotated with two lists of variables. The lambda form abstracts over the variables $y_1 \ \ldots \ y_m$, so it defines an m-ary function. The list $x_1 \ \ldots \ x_n$ gives the free variables of the body e, excluding the abstracted variables y_i.

Additionally, each lambda form is annotated with an *update flag* π which can be `u` or `n`. These flags are necessary for the implementation of *lazy evaluation*, where each closure is evaluated only when necessary, but at most once. To ensure this, closures are overwritten with their weak head normal form after their first evaluation. However, not every closure has to be overwritten: if the bound expression already is in weak head normal form, or the compiler can prove that it will be evaluated only once anyway, the binding is flagged with `n` to signal that no update code has to be generated. Otherwise, the binding is flagged with `u` to cause the generation of update code. For example, in the expression

```
let compose = {} \n {f g x} →
      let gx = {g x} \u {} → g {x}
      in f {gx}
in ...
```

`compose` is defined in weak head normal form, since it abstracts over `f`, `g` and `x`, and is flagged with `n` accordingly. By contrast `gx` is *not* in weak head normal form, `g` and `x` are merely free variables, so the flag is `u` and the closure of `gx` will be overwritten as soon as it is evaluated the first time[3].

Case analysis can be done on arbitrary expressions, but is restricted to flat patterns without nesting. The default case is expressed using the pattern `_`, matching every expression.

4.2 Translating Ulysses to STG Code

The translation of Ulysses code to the STG language is straight forward. Function arguments that are not yet simple variables are bound by new local variables. The same holds for constructor arguments, furthermore we have to saturate constructors by η expansion: a binary constructor `Pair` applied to a single argument `x` becomes

```
let f = {x} \n {y} → Pair x y in f.
```

[3] Of course, each application of `compose` will create a new closure `gx`.

Nested patterns from Ulysses definitions are flattened by a well known pattern matching compiler, as described in [4].

Besides the usual feature set at the term level, we need a representation of Ulysses types. This is done by introducing a reserved constructor for each predefined type constructor. The simplest case is the function space, a type `a → b` is translated to the constructor application `Fun a b`.

The encoding of type universes and data types makes use of unboxed integers. For instance, we represent `#2` by `Universe 2`, and `data Nothing | Just a` as `Data 1 2 a`. In the latter case, 1 and 2 are the tags for the constructors `Nothing` and `Just`, and their arity can be seen by the number of boxed values after the constructor tag, in this case 0 and 1, respectively.

Dependent product types are represented by the special constructor `Forall`, taking as arguments the representations of argument and result types. To make the necessary substitutions in the result type possible, we use a technique that was used already in [5]. The result type is not stored directly, but as a function taking a member of the argument type to a type representation. The Ulysses type of the identity function `forall t :: #0 . t -> t` is thus represented as

```
let x = {} \u {} → Universe 0;
    y = {} \n {t} → Fun t t
in Forall x y
```

which allows to replace `t` during type checking with a concrete type by extracting `y` from the constructor expression, and applying it to the needed type.

4.3 Executing STG Code on Conventional Machines

STG code can be easily translated to machine code for execution on traditional hardware. We next give an overview of the memory layout and operational behavior of the resulting machine programs.

Our machine state consists of

- a *heap* which contains closures, each consisting of one code pointer and a sequence of pointers to the values of the free variables used in this code,
- a *closure register* $\mathtt{R}_{\mathtt{closure}}$, pointing to the currently evaluated closure,
- an *argument stack*, containing pointers to closures in the heap, for passing arguments to functions,
- a *continuation stack*, holding code pointers, and a *tag return register* $\mathtt{R}_{\mathtt{tag}}$, containing a small integer, for the implementation of case analyses, and
- an *update stack*, containing update frames, for bookkeeping closures that have to be overwritten as soon as they are evaluated to weak head normal form.

Function Application The implementation of function calls follows the push/enter model: we push all arguments onto the argument stack, and enter the function. Entering a function is done in two steps: first, load the address of the function's closure into the closure register, and second jump to the function body. Since

the STG language does not allow nested function applications, this is a tail call, and no return address has to be remembered. So we translate an STG function application `f {x y}` to following pseudo assembler code:

```
push-argument y
push-argument x
enter f
```

Constructors and Case Analyses Constructor applications usually occur as scrutinees within case analyses. When a `case` expression is evaluated, a return address is pushed onto the continuation stack. Next, the evaluation of the scrutinee is started. When the scrutinee is finally evaluated to a constructor application, the constructor tag is loaded into the tag return register and a pointer to a closure containing the constructor arguments is loaded into the closure register. These registers now have to be passed to the code of the case analysis, so a jump to the topmost code pointer on the continuation stack is taken. At the jump target, the continuation is removed from the stack, and the tag is analyzed. The constructor arguments can be accessed through the closure register.

Accordingly, the STG expression

`case` e_1 `of { C x y` $\rightarrow$ e_2`; _` $\rightarrow$ e_3 `}`

is compiled to the following pseudo assembler:

```
   push-continuation l
   «code for e1»
l: pop-continuation
   compare R_tag «tag reserved for C»
   jump-if-not-equal d
   «code for e2»
d: «code for e3»
```

and a corresponding constructor application `C {a b}` is translated to[4]

```
   R_closure := allocate l, {a b}
l: R_tag := «tag reserved for C»
   jump-continuation
```

Local Bindings For `let` bindings, we allocate on the heap a closure for each bound variable. The code pointers of these closures point to the compiled bodies of the lambda forms. The current values of the free variables, which are pointers to other closures, are saved into the corresponding closure fields. After that, evaluation continues with the body of the `let` expression. Note that due to the lazy semantics no evaluation of the bound variables is triggered now.

A binding with update flag `n` as e. g.

`let v = {x y} \n {}` $\rightarrow$ e_1 `in` e_2

[4] Here, `allocate` allocates heap space for a new closure and fills it with the given code pointer and free variables.

is thus translated to

```
      allocate l, {x y}
      «code for e2»
l:    «code for e1»
```

When the closure shall be updated after its first evaluation, the code of the new closure is preceeded by pushing an update frame that contains the current closure pointer (pointing to the memory location to be overwritten), and the current argument stack content[5]. Next, the argument stack is emptied to signal a necessary update to partial function applications. Thus A binding with update flag `u` as e. g.

```
let v = {x y} \u {} → e1 in e2
```

is compiled to

```
      allocate l, {x y}
      «code for e2»
l:    push-update-frame
      empty-argument-stack
      «code for e1»
```

Accordingly, each function has to check whether all expected arguments are present and, if not, call a global routine `updatePAP` that overwrites the closure pointed to by the topmost update frame with a partial applied function closure, removes the update frame from the update stack, restores the argument stack, and finally re-enters the current closure. So we translate

```
let v = {} \n {x y} → e1 in e2
```

to the assembly code

```
      allocate l, {x y}
      «code for e2»
l:    compare-argument-stack-length 2
      jump-if-less updatePAP
      «code for e1»
```

This allows to update closures with function values, but we need to find a way for constructor values, too. This can be done quite elegant by merging the update and the continuation stack, pushing update frames and continuation onto the same stack. Now, we can arrange update frames so that the topmost word on the stack contains a pointer to a routine `updateConstructor` that overwrites the closure pointed to by the update frame with an indirection to the current closure, removes the update frame, and jumps to the now topmost pointer on the stack. Therefore a constructor can simply jump to the topmost pointer on the merged stack, which points either to the update routine, if an update is necessary, or to the case analysis code.

[5] In [3] you can find a description how this can be done by fast pointer manipulations

5 Runtime System

Evaluating type expressions to normal forms by running machine code requires a special runtime system with two main tasks:

- The different forms of weak head normal forms (cf. Def. 2) have to be discriminated, and their components have to be extracted from the machine registers and stacks.
- To evaluate under λ and `case`, we need to generate free variables. They must be carefully designed to fit to the generated machine code, since we don't want to be forced to generate special machine code that deviates from traditional STG compilers and might have poor performance.

5.1 Constructor Expressions

To identify final machine states that constitute constructor expressions we exploit that each constructor, after loading the tag return register and closure pointer, just takes a jump to the topmost address on the continuation stack. So, before we start running any machine code, we just push a special exit continuation on the continuation stack. The corresponding code finds the constructor tag and arguments via the respective registers, and can hand them on to the readback function (cf. Def.3) for recursive evaluation of the constructor arguments.

5.2 Unsaturated Functions

To recognize unsaturated functions, we utilize, that each function starts evaluation by checking whether enough arguments are present on the stack. If this check fails, this usually means that an update has to be performed and the global function update routine is jumped to. When an update frame is found, a closure is overwritten and additional arguments are restored on the stack. But if the update stack is empty, we know that the weak head normal form of the overall expression is an unsaturated function, so we return a λ abstraction to the readback function, which is responsible for creating a fresh free variable, pushing it onto the argument stack and reentering the last evaluated closure.

At this point we have introduced an additional check compared to code generated by traditional compilers using STG intermediate code: When not enough function arguments are present on the argument stack, we have to check whether the update stack is empty. Usually, a restricted top level type ensures that this does not happen. However, this check occurs only at a single code location in `updatePAP`. So it is possible to link this routine during run-time (as opposed to evaluation at type checking time) to a simpler version omitting this check.

5.3 Free Variables

Free variables are the most intricate part of our implementation. They originate from evaluations under λ abstractions, where they were pushed onto the argument stack, or evaluations under `case`, where constructor expressions with free variables in argument positions were created.

Code generated from STG intermediate code has a distinguishing property from many other compilation schemes: Constructor closures and function closures are entered in the same way, but execute very different code. While the former take a jump to a code address left on the continuation stack, the latter expect arguments on the stack and start some computation. So we need an implementation of free variables prepared for both scenarios.

But luckily one invariant exists in both cases: entering a free variable means a weak head normal form has been reached. We just have to find out, whether it is a free variable application (Def. 2, Line 2) or a case analysis of a free variable application (Def. 2, Line 4), and collect eventual arguments from the stack.

In our system, we implemented free variables as follows. Each free variable is represented by a closure on the heap. All free variables share a common code pointer into the runtime system. Every variable closure contains a identification number, to distinguish different occurrences of the same variable. Moreover, every variable is annotated with its type, to allow to select the right behavior when it is entered. As a last component, every free variable closure contains a list of arguments to which it is applied. This list is initially empty, but when a closure is updated with a free variable, some arguments might have been accumulated.

So when a free variable is entered, the runtime system starts a loop that interprets the type annotation of the variable in the current machine state. The following algorithm operates on the entered variable x, the collected arguments $e_1 \ \ldots \ e_n$ found in the variable closure, and the type of x applied to $e_1 \ \ldots \ e_n$, also found in the closure.

1. if the x applied to $e_1 \ \ldots \ e_n$ has a function type, then
   (a) if there is an argument on the argument stack, then
      i. pop this argument, and add it to the accumulated arguments
      ii. compute the result type of the variable's type
      iii. restart at 1 interpreting this result type
   (b) else, if there is an update frame on the update stack
      i. perform an update, overwriting the destination closure with a free variable closure containing all arguments accumulated so far and the type of this application
      ii. restore argument and return stack from the update frame
      iii. restart at 1
   (c) else, the free variable applied to the accumulated arguments constitutes the weak head normal form that is returned to the readback function
2. else, if the variable has a data type
   (a) if there is an continuation on the continuation stack, then
      i. save the current machine state as context Γ
      ii. return a case analysis of a free variable application in context Γ, i.e. $\Gamma[\,\texttt{case}\ x\ e_1\ \ldots\ e_n\,]$, to the readback function
   (b) as in 1b
   (c) as in 1c
3. otherwise, the free variable application x applied to $e_1 \ \ldots \ e_n$ is returned as weak head normal form to the readback function

6 Further Improvements

By now, we have all we need for a fairly efficient type checker for dependent types. Type expressions can be reduced by the strong evaluation function $\mathcal{N}$. It uses a weak evaluator $\mathcal{V}$ that is implemented using a approved compilation scheme via STG to assembly code. The results of the weak evaluation can be extracted from the machine state, and all occurring weak head normal forms can be distinguished. By passing these evaluation result to the readback function $\mathcal{R}$, further redexes can be extracted and again passed to our weak evaluator. However, Ulysses implements two additional improvements.

6.1 Interleaving Type Checking and Evaluation of Types

Instead of reducing all types occurring during type checking in one step to normal forms, it is beneficial to reduce them in the first step to weak head normal form only. This weak head normal form is usually a type constructor applied to not yet normalized type arguments. In this stage, we can check whether the type constructor matches the syntactic construct to check, e. g. a function type and a λ abstraction, and reduce the type arguments only when this check succeeds. When this check fails, e. g. because a λ abstraction shall be checked to have a data type, the redexes in the arguments of the type do not have to be reduced, and some amount of unnecessary work can be avoided.

This idea of reduction to weak head normal form interleaved with type checking and recursive further evaluation has been applied in [6], too.

6.2 Detecting Equivalent Types Early

Our focus is type checking only, we do not consider inferring types. Hence, whenever we have to check a variable against a type, we know, that this variable has been declared with some type, and thus can be found in the type context. So, the variable typing rule in Ulysses looks like follows.

$$\textsc{Var}\frac{(x : \tau_1) \in \Gamma \qquad \tau_1 \equiv_\beta \tau_2}{\Gamma \vdash x : t_2}$$

A variable has type τ_2 if it is declared to have type τ_1 in the context Γ, and τ_1 and τ_2 are equivalent.[6] To check this equivalence, one could reduce both types to normal forms, and compare them for equivalence. But we can do better.

The types τ_1 and τ_2 are represented as closures in the heap. These closures evaluate certainly to the same normal form, if they contain the same pointers in the same places, i. e. the same code pointer and the same pointers for the free variables (cf. Fig. 3 (a)). Moreover, they also evaluate to the same normal form

[6] Here, Γ is a *type* context, as usual in the specification of type systems. It is not to be confused with the context Γ in the definition of the readback function in Def. 3, Line 8, where it denotes the *dynamic* context of an expression, i. e. the machine state in which it is evaluated.

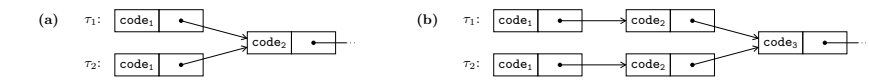


Fig. 3. Detecting equal normal forms without evaluation

if they have the same code pointer, and their free variable pointers are different, but the respective closures they point to have the same code pointers and the same variable pointers (cf. Fig. 3 (b)).

Generalizing this pattern, we reach a variant of bisimilarity, introduced by Park in [7]. The following definition uses the notation c_i to access field i in closure c, $|c|$ for the number of fields of closure c, and $*p$ to dereference a pointer.

Definition 4. *We call a relation R on closures a* bisimulation*, iff for each pair of closures $(s,t) \in R$*

1. *$s_0 = t_0$, i. e. the code pointers are equal, and*
2. *$|s| = |t|$, i. e. the closures have same length, and*
3. *$\forall i \in \{1 \ldots |s|-1\}$:*
   (a) *s_i and t_i are pointers, and $(*s_i, *t_i) \in R$, or*
   (b) *s_i and t_i are non-pointers, and $s_i = t_i$.*

We can consider τ_1 and τ_2 to be equal, if there is a bisimulation R such that $(\tau_1, \tau_2) \in R$. Whether such a relation exists can be checked by a single simultaneous traversal of both closure graphs of τ_1 and τ_2. If this traversal reaches closures with different code pointers, further reductions have to be done.

This notion is appropriate for non-normalizing terms, too. Recursive type definitions as for example `nat = data Z | S nat` can be unfolded infinitely, thus the simple strategy of complete reduction and subsequent equality check is not successful for this type, while the bisimilarity check allows to deal with it.

7 Notes on the Implementation

The Ulysses system has been implemented in Haskell, and is available at `http://uebb.cs.tu-berlin.de/~klee/ulysses`.

7.1 The Overall Type Checking Process

The first step during type checking an Ulysses program is a dependency analysis. It has to ensure that functions, which are used in the types of other function definitions, are type checked before these other definitions. The definitions are sorted accordingly, whereby mutual recursive definitions are clustered together. Then, each definition is first checked and then compiled to machine code. Mutual recursive definitions get checked together and are compiled only after checking

the whole cluster. For compilation, we use Harpy[7], a HASKELL library for runtime code generation. It allows to write x86 machine code into memory buffers, and to directly execute it without external tools.

As soon as type checking is completed, the code of the whole checked ULYSSES file is located in the Harpy code buffer and could be dumped to an object file. However, this is not yet implemented.

7.2 Copy-On-Write

In Sect. 5.3, we had to save the machine state as a context Γ for each case analysis of a free variable. This is no problem for the registers and stacks, as they are fairly small. The heap, however, can be quite large, so unnecessary copies should be avoided when possible. Therefore, we use a copy-on-write approach. Instead of making a copy of the heap, we use the memory management unit to write protect the heap. When the running program tries to modify the heap, a segmentation violation signal is raised. This is handled by making a copy and releasing the protection of the affected page, so only modified pages have to be copied.

8 Related Work

Our work is a transfer of the approach of Grégoire and Leroy [2, 5] from the strict ZAM abstract machine to the lazy STG machine. Therefore, the readback function had to be adopted. For free variable applications and constructor applications, the eager evaluation strategy resulted in completely evaluated constructor arguments, which can be simply read back by $\mathcal{R}$, while we need a recursive call to the weak evaluator.

The implementation of free variables is quite different than described in [2]. The main reason for these differences lies in the different underlying abstract machines, mainly the uniform handling of constructor and function closures which are both *entered*, which is quite specific to the STG machine.

In [8] Crégut presents an abstract machine for strong normalization of λ terms. It differs from ours and Grégoire's in the missing distinction of weak evaluation and readback. This might be faster when reducing to normal forms, but precludes the improvements of Sect. 6.

Another related line of research is partial evaluation, most closely probably type-directed partial evaluation introduced by Danvy in [9]. This evaluator generates constructors expressions for free variables not only when they are scrutinized in case expressions, but whenever functions take arguments of disjoint sum types. Therefore our implementation deals better with recursive data types.

9 Conclusion and Future Work

ULYSSES is a prototype of a language with dependent types, that uses compiled code during type checking where interpreters are used traditionally. Even though

[7] `http://uebb.cs.tu-berlin.de/harpy`

it misses some features that constitute a complete programming language, as e. g. garbage collection and a module system, it shows the feasibility of our approach.

First experiments suggest, that the performance gain is as expected when switching from an interpreted to a compiled system, but further benchmarks have to be done. This is complicated by the fact that it is not at all clear how a reasonable benchmark for a type checker should look like.

The bisimilarity check (cf. Sect. 6.2) allows to deal with recursive types that have no normal form, but not all type equivalences can be detected this way. While it works for many types, as natural numbers, lists and vectors, it needs a great deal of knowledge of system internals to find the reasons why it does not work in some circumstances, leading to a nonterminating type check. A topic for future research is finding a simple criterion which recursive types can be proven equal by bisimilarity after a finite number of reduction steps. Alternatively, integrating an explicit fixpoint operator, as in [2], should improve the situation.

Several optimization techniques for STG code are known. While we believe that most of such techniques can be used in our settings, a closer look is necessary to find possible interactions with the readback scheme and the interpretation of final machine states, as well as with the implementation of free variables.

To be assured that Ulysses is indeed a type safe language, we plan to formalize our reduction and readback scheme and prove its correctness.

References

1. Augustsson, L.: Cayenne – a language with dependent types. In: ICFP '98: Proceedings of the third ACM SIGPLAN International Conference on Functional Programming, ACM Press (1998) 239–250
2. Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. In: ICFP '02: Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming, ACM Press (2002) 235–246
3. Peyton Jones, S.L.: Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. Journal of Functional Programming **2**(2) (1992) 127–202
4. Peyton Jones, S.L.: The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science). Prentice-Hall (1987)
5. Grégoire, B.: Compilation des termes de preuves: un (nouveau) mariage entre Coq et Ocaml. Thése de doctorat, spécialité informatique, Université Paris 7, école Polytechnique, France (2003)
6. Coquand, T.: An algorithm for type-checking dependent types. Science of Computer Programming **26**(1-3) (1996) 167–177
7. Park, D.: Concurrency and automata on infinite sequences. In: Proceedings of the 5th GI-Conference on Theoretical Computer Science, Springer-Verlag (1981) 167–183
8. Crégut, P.: An abstract machine for lambda-terms normalization. In: LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming, ACM Press (1990) 333–340
9. Danvy, O.: Type-directed partial evaluation. In: POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press (1996) 242–257

HW-Hume in Isabelle

Chunxu Liu and Greg Michaelson

School of Mathematical and Computer Science
Heriot-Watt University, Riccarton, Scotland.
Email: {chunxu,greg}@macs.hw.ac.uk

Abstract. HW-Hume is the decidable Hume level oriented to direct implementation in hardware. As a first stage in the development of a verified compiler from HW-Hume to Java, we have implemented the semantics of HW-Hume in the Isabelle/HOL theorem prover, enabling the automatic proof of correctness of programs in a Floyd/Hoare style.

1 Introduction

A *verified* compiler gives guarantees that compilation preserves meaning from source to target, but not that the source program satisfies its specification. That is, given:

$M_S : S \rightarrow D$ - meaning of source programs in language S
$M_T : T \rightarrow D$ - meaning of target programs in language T
$C_{S-T} : S \rightarrow T$ - compiler from S to T

where D is some domain of meanings, we wish to guarantee for program P_S in source language S that:

$$M_T(C(P_S)) = M_S(P_S)$$

In contrast, a verifying compiler gives guarantees that a source program satisfies its specification, but not that the target is a true translation of the source.

Verified compilation has a long but surprisingly thin pedigree, since McCarthy and Painter's seminal work 40 years ago for arithmetic expressions[MP67]. They defined very simple source and target languages and their semantics, gave rules from source to target and proved them correct. Almost all subsequent work has followed this basic approach. However, where McCarthy and Painter used an ad hoc notation and constructed proofs by hand, there has been a growing trend to the use of formal, and possibly executable, notations, and of automated theorem proving technology.

Palsberg[Pal92] designed, implemented and proved the correctness of a compiler generator, called Cantor, that accepts action semantic descriptions.

Stepney[Ste93] discusses compilation from the simple high-level imperative source language Tosca to the low level target language Aida. Prolog and Z are used as the meta-languages to define the denotational semantics of both Tosca and Aida, with translation templates from each source language syntax structure to the equivalent target language structure. While this highly ambitious work for an industrial client, used executable notations to deliver a working verified compiler, almost all proofs were conducted painstakingly by hand.

Stringer-Calvert[SC98] extends Stepney's work in his PhD thesis, presenting an overview of the development of a demonstrably correct compiler by what he terms the

DCC method, which has three components: Specification, Implementation and Proof. The correctness of the DCC compiler is proved mechanically, using PVS[SF06]. of programming language, and targets used an abstract RISC machine language.

Curzon[Cur92,Cur94] presents a formal machine-checked verification of a simple compiler specification using the HOL[GM93] theorem prover, thus combining a unitary notation and proof tool. He has implemented a tool that executes the verified compiler specification using formal proof. He also discusses bootstraping a correct compiler implementation.

Most recently, Klein and Nipkow[KN06] have used Isabelle/HOL[NPW02] to prove the correctness of a compiler from the Java subset Jinja to JVM code.

We are exploring the development of a provably correct or verified compiler from HW-Hume[MH04,MHS05], the decidable but impoverished Hume layer oriented to hardware realisation, to Java. We have already constructed a prototype HW-Hume to Java compiler, discussed in[LM04].

While our work is are strongly influenced by Stepney, and by Klein and Nipkow, there are important differences. Where Stepney proved correctness via denotational semantics, we intend to use an operational semantics. While this enables us to retain close correspondence with our prototype HW-Hume to Java compiler, proofs may be longer than for denotational semantics, as operational semantics includes considerably more detail of evaluation. Similarly, Klein and Nipkow use Isaballe to also prove that source programs are well formed and type correct. We are only concerned with proving the correctness of translation and assume that some prior analysis has established other properties.

As a vital core stage in our work, we have embedded the semantics of HW-Hume in Isabelle, enabling proof of correctness. In the following sections we provide overviews of HW-Hume and it semantics, and of Isabelle/HOL. We then present the realisation of the HW-Hume semantics in Isabelle/HOLs, and discuss the proof of correctness of two HW-Hume exemplars. Finally, we consider how we intend to complete our formally verified HW-Hume to Java compiler.

2 HW-Hume Semantics

2.1 HW-Hume Abstract Syntax

Figure 1 shows the abstract syntax of HW-Hume. A HW-Hume program is built from one or more box(s), one or more wire(s) and optional initial declarations, const declarations and type declarations. The program execution is independent of the order of box, wire and init declarations.

2.2 HW-Hume Execution Model

Figure 2 shows the HW-Hume execution model which is based on non-terminating, round-robin, one-shot scheduling of boxes.

prog	::= $decl_1$ ";" ...";" $decl_n$	$n \geq 1$
decl	::= *box* | *wire* | *init* | *constdecl* | *typedecl*	
box	::= "**box**" *boxid ins outs* "**match**" *matches*	
ins/outs	::= ($ioid_1$ :: $type_1$, ...$ioid_n$ :: $type_1$)	$n \geq 1$
matches	::= $match_1$ "|" ... "|" $match_n$	$n \geq 1$
match	::= *patt* "–>" *expr*	
wire	::= "**wire**" $link_1$ "**to**" $link_2$	
init	::= *link* "=" *value*	
link	::= *boxid* "." *ioid*	
constdecl	::= *constid* "=" *value*	
typedecl	::= *typeid* "=" *type*	
patt	::= *intliteral* | * | _ | _* | *varid* | (*patt*,*patt*)	
expr	::= *intliteral* | * | *varid* | (*expr*,*expr*)	
val	::= *intliteral* | (*val*,*val*)	
type	::= *int* | *typeid* | (*type*,*type*)	

Fig. 1. HW-Hume Abstract Syntax

```
for each box
   state ← RUNNABLE
forever
   for each box
      if state != BLOCKED then
         state ← MATCHFAIL
         for each match
            if some pattern matches input values then
               consume values from input wires
               evaluate associated expression
               generate output wires
               state ← SUCCESS
               stop match loop
   for each box
      if state!=MATCHFAIL then
         if output wires can be established to their input wires then
            establish output wiress
            state ← RUNNABLE
         else
            state ← BLOCKED
```

Fig. 2. HW-Hume Execution Model

3 Isabelle Overview

Isabelle is a popular generic interactive theorem prover which supports a variety of logics. Isabelle/HOL[NPW02] is the specialization for HOL(Higher-Order Logic) that is based on Gordon's HOL system[GM93], which itself is based on Church's original paper[Chu40]. As Tobias Nipkow[NPW02] said:

HOL = Functional Programming + Logic

Isabelle conforms largely to standard mathematical notation. As a generic proof assistant for logic, Isabelle is a powerful system for implementing logic formalisms.

Isabelle Proof General is a generic interface for proof assistants in Isabelle which supports a print mode for X Symbol tokens. In Isabelle, we can write

`1<^bsub>v\<^esub>` or `1<^isub>v` or `1<^sub>v`

In the Isabelle Proof General environment or an Isabelle print document, this will be displayed as 1_v which is easy to read.

In the sequel, we use Isabelle for Isabelle/HOL.

This section introduces Isabelle basic types with their primitive operations and further non-standard notation.

base types:

- *bool* — the type of truth.
- *nat* — the type of natural numbers.

constructor types:

- *list* — the type of lists with the type *'a list*. *nat list* means that every element of list has the type *nat*. Empty list is []. The infix operator @ concatenates two lists. The infix operator # inserts a element to the beginning of a list. *xs*!*n* is the nth-element of *xs* (starting with 0).
- *set* — the type of sets with the type *'a set*. *nat set* means that every element of the set has the type *nat*. Empty set is {}. We write $\emptyset$ instead of {} for friendly reading.
- *option* — defined by
  datatype *'a option = None | Some 'a*
  It adjoins a new element *None* to a type *'a*. We write $\lfloor x \rfloor$ instead of *Some x* for succinctness. *the* $\lfloor x \rfloor = x$, where *the* is a function.
- *Pairs* — ordered pairs. (a_1, a_2) is of type $\tau_1 \times \tau_2$, where a_1 is of type τ_1 and a_2 is of type τ_2. $fst(a_1, a_2) = a_1$, $snd(a_1, a_2) = a_2$, where *fst* and *snd* are functions.
- *Tuples* — defined by *Pairs* nested. (a_1, a_2, a_3) stands for $(a_1, (a_2, a_3))$. So $fst(a_1, a_2, a_3) = a_1$, $fst(snd(a_1, a_2, a_3)) = a_2$.

function types:

- *total function* — denoted by $\Rightarrow$. Like SML, $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3$ means $\tau_1 \Rightarrow (\tau_2 \Rightarrow \tau_3)$. $[\tau_1, \tau_2, ..., \tau_n] \Rightarrow \tau$ is abbreviation of $\tau_1 \Rightarrow \tau_2 \Rightarrow ... \Rightarrow \tau_n \Rightarrow \tau$. Its update is $f(x := y)$ where $f :: 'a \Rightarrow 'b$, x with type *'a* and y with type *'b*.

– *partial function* — defined by $'a \Rightarrow 'b\ option$. *None* represents undefinedness, $f\,x{=}\lfloor x\rfloor$ means that x is mapped to y. The domain of f is defined $dom\ f \equiv \{a \mid f\ \ x \neq None\}$. We write $'a \rightharpoonup 'b$ instead of $'a \Rightarrow 'b\ option$. *empty* is defined by $\lambda x.None$. Its updates is $f(x := \lfloor y\rfloor)$. We abbraviate $f(x := \lfloor y\rfloor)$ to $f(x \mapsto y)$. Such functions are called maps. The infix operator ++ overwrites map m_1 with m_2.
i.e. $m_1 +\!+ m_2 \equiv \lambda x.\ case\ m_2\ x\ of\ None \Rightarrow m_1\ x \mid \lfloor y\rfloor \Rightarrow \lfloor y\rfloor$
– *Inductive definitions* — a method to define a function. In fact, a function from set A to set B is defined by a relation set $C \subseteq A \times B$ when set C satisfies some properties. Many datatype are inductive defined. A structural oprational semantics[Hen90] is inductive definition of an evaluation relation. The inductive definitions[Acz77] specifies the least set R closed under given collection rules. Applying a rule to elements of R yields a result within R. Milner[Mil80] implemented one of the first inductive definitions. Isabelle provides commands for formalizing inductive definitions. Paulson[Pau00] proved a fixedpoint with inductive definitions in Isabelle. Klein and Nipkow[KN06] inductively defined Jinja semantics and proved correctness of Jinja compiler. We mainly present HW-Hume semantics with inductive definitions.

type variables: denoted by $'a$, $'b$ etc. Like SML, they give rise to polymorphis types.

inference rule: $A_1 \Longrightarrow A_2 \Longrightarrow A_3$ means $A_1 \Longrightarrow (A_2 \Longrightarrow A_3)$.
$[\![A_1; A_2; ...; A_n]\!] \Longrightarrow A$ is abbreviation of $A_1 \Longrightarrow (A_2 \Longrightarrow (... \Longrightarrow (A_n \Longrightarrow A)...))$.
It means "If A_1 and A_2 and ... and A_n then A".
i.e. inference rule

$$\frac{A_1\ \ A_2\ \ ...\ \ A_n}{A}$$

4 HW-Hume Semantics in Isabelle

4.1 HW-Hume Abstract Syntax in Isabelle

Figure 3 shows the abstract syntax of HW-Hume in Isabelle. After precompilation, all constid in const declarations are replaced by their values, and all typeid in type declarations are replaced by their types. The program execution is independent of the order of box and wire declarations. So we define a HW-Hume program as
"**types** *HProg = HBox list* × *HWire list*"
"*HInit list*" is initial values. A box comprises a boxid, a series of inputs, a series of outputs and a series of matches. Hence we define a box as
"**types** *HBox = BName* × *HIO list* × *HIO list* × *HMatch list*"
The first *HIO list* is for input and the second is for output.

4.2 HW-Hume States

Figure 4 shows the definition of the HW-Hume states — dynamic environment. Each box in a HW-Hume program has its own input and output. We define *hLocation* = *INOUT* × *BName* × *LName*) to denote them. The states is a map from hLocation to

```
types BName = string                    -- "names for box"
types LName = string                    -- "names for io (link)"
types VName = string                    -- "names for variable"
datatype HP = I | W | WI                -- "ignore,wild,wild ignore."
            | HPInt int                 -- "nat value"
            | HPVar VName               -- "variable"
            | HPPair HP HP              -- "pair"
datatype HE = I                         -- "ignore"
            | HEInt int                 -- "int value"
            | HEVar VName               -- "variable"
            | HEPair HE HE              -- "pair"
datatype HV = HVInt int                 -- "int value"
            | HVPair HV HV              -- "pair"
datatype HT = Z                         -- "nat"
            | HTPair HT HT              -- "pair"
types HIO    = LName × HT
types HMatch = HP × HE
types HBox   = BName × HIO list × HIO list × HMatch list
types HWire  = (BName × LName) × (BName × LName)
types HProg  = HBox list × HWire list
types HInit  = BName × LName × HV
```

Fig. 3. HW-Hume Abstract Syntax

value. At the end of each cycle, a HW-Hume program checks the wire of each box's output and try to transfer their values to some box's input. If there is an output which cannot be transferred into a box, that box is blocked. Conversely, if an output is still mapped to a value, that box is blocked. We define a function *BnoEmpty* to check if a box is blocked or not. The function *Init_hStete* sets the initial states. The function *UpdateS* updates state by transferring each box's output which can be transferred to its target.

4.3 HW-Hume Big Step Semantics

We next present the HW-Hume big step semantics in inductive definitions judgement form.

Cycles is the top-step of running a Hume program. Its goal is to run all boxes N times in repeated cycles. When all cycles finish, the program halts. The global states (*hState*) identify the program states.

We use *RC* defined in Figure 5 as a running result. The possible results of *RunC* are error (*Cer hErr*) or success. When success, the result is *Cs hState*.

We inductively define *RunC* as well. The *RunC B* is the least relation set closed under given rules below.

We define the *RunC* judgement form $P \vdash_c \langle n, s\rangle \Rightarrow \langle rc\rangle$. This is an abbreviation of $(n, s, rc) \in RunC\ P$. The P has type *HProg*; n has type *hCNum*; rc has type *RC*.

We define *Cycle* induction rules below.

Global State:	
datatype *INOUT* = *LI* | *LO*	
types *hLocation* = *INOUT* × *BName* × *LName*	
types *hState* = *hLocation* ⇀ *HV*	
Local State:	
datatype *hErr* = *MPVer* | *MIPer* | *EEer* | *MOEer*	
types *hVar* = *VName* ⇀ *HV*	
types *hCNum* = *nat*	
Running Result States	
datatype *RC* = *Cer hErr* | *Cs hState*	Run Cycles (RunC)
datatype *RB* = *Ber hErr* | *Bs hState*	Run Box and Boxes (RunB, RunBL)
datatype *RM* = *Mer hErr* | *Ms hState*	Run Match (RunM)
datatype *MIP* = *IPer hErr* | *IPf* | *IPs hState* × *hVar*	Match in to pattrn (MatchIP)
datatype *MPV* = *PVer hErr* | *PVf* | *PVs hVar*	Match pattrn to value (MatchPV)
datatype *EE* = *Eer hErr* | *Es HE*	Evalate expression (EvalE)
datatype *MOE* = *OEer hErr* | *OEs hState*	Match out to expression (MatchOE)

Fig. 4. HW-Hume State

– C0_s — Running a program 0 time, i.e. 0 cycle. *s* is not changed.

$$\frac{n = 0}{P \vdash_c \langle n, s\rangle \Rightarrow \langle Cs\ \ s\rangle}$$

– C1_e — Calling *RunBL* error, so running a program 1 cycle error.

$$\frac{n = 1 \quad \vdash_{bl} \langle P2BL\ P, s\rangle \Rightarrow \langle Ber\ \ er\rangle}{P \vdash_c \langle n, s\rangle \Rightarrow \langle Cer\ \ er\rangle}$$

– C1_s — Calling *RunBL* success, running a program 1 cycle success.

$$\frac{n = 1 \quad \vdash_{bl} \langle P2BL\ P, s\rangle \Rightarrow \langle Bs\ \ s'\rangle \ \ s'' = UpdateS\ \ s'\ P}{P \vdash_c \langle n, s\rangle \Rightarrow \langle Cs\ \ s''\rangle}$$

– C_R_e — More than one cycle, recursive, the first cycle error.

$$\frac{n > 1 \ \ P \vdash_c \langle 1, s\rangle \Rightarrow \langle Cer\ \ er\rangle}{P \vdash_c \langle n, s\rangle \Rightarrow \langle Cer\ \ er\rangle}$$

– C_R — More than one cycle, recursive, the first cycle is success. Result depends on remaining cycles.

$$\frac{n > 1 \ \ P \vdash_c \langle 1, s\rangle \Rightarrow \langle Cs\ \ s'\rangle \ \ n' = n - 1 \ \ P \vdash_c \langle n', s'\rangle \Rightarrow \langle rc\rangle}{P \vdash_c \langle n, s\rangle \Rightarrow \langle rc\rangle}$$

We show below the result theorems:

theorem *EvalE_Result*: $bn, s, lv \vdash_e \langle e\rangle \Rightarrow \langle ee\rangle \Longrightarrow$
$(ee = Eer\ \ EEer) \vee (\exists es.(ee = Es\ \ es) \wedge (HEhasVar\ \ es = False))$

consts $RunC :: HProg \Rightarrow (hCNum \times hState \times RC)\ set$ **syntax** $RunC :: HProg \Rightarrow hCNum \Rightarrow hState \Rightarrow RC \Rightarrow bool$ $(\_ \vdash_c \langle \_, \_ \rangle \Rightarrow \langle \_ \rangle \quad [0,0,0,0]\ 81)$ **translations** $P \vdash_c \langle n, s \rangle \Rightarrow \langle rc \rangle \rightleftharpoons (n, s, rc) \in RunC\ P$
consts $RunBL :: (HBox\ list \times hState \times RB)\ set$ **syntax** $RunBL :: HBox\ list \Rightarrow hState \Rightarrow RB \Rightarrow bool$ $(\vdash_{bl} \langle \_, \_ \rangle \Rightarrow \langle \_ \rangle \quad [0,0,0]\ 81)$ **translations** $\vdash_{bl} \langle bl, s \rangle \Rightarrow \langle rb \rangle \rightleftharpoons (bl, s, rb) \in RunBL$
consts $RunB :: HBox \Rightarrow (hState \times RB)\ set$ **syntax** $RunB :: HBox \Rightarrow hState \Rightarrow RB \Rightarrow bool$ $(\_ \vdash_b \langle \_ \rangle \Rightarrow \langle \_ \rangle \quad [0,0,0]\ 81)$ **translations** $B \vdash_b \langle s \rangle \Rightarrow \langle rb \rangle \rightleftharpoons (s, rb) \in RunB\ B$
consts $RunM :: BName \Rightarrow LName\ list \Rightarrow LName\ list \Rightarrow$ $(HMatch\ list \times hState \times RM)\ set$ **syntax** $RunM :: BName \Rightarrow LName\ list \Rightarrow LName\ list \Rightarrow$ $HMatch\ list \Rightarrow hState \Rightarrow RM \Rightarrow bool$ $(\_, \_, \_ \vdash_m \langle \_, \_ \rangle \Rightarrow \langle \_ \rangle \quad [0,0,0,0,0,0]\ 81)$ **translations** $bn, ilnl, olnl \vdash_m \langle ml, s \rangle \Rightarrow \langle rm \rangle \rightleftharpoons (ml, s, rm) \in RunM\ bn\ ilnl\ olnl$
consts $MatchIP :: BName \Rightarrow$ $(LNname\ list \times HP \times hState \times hVar \times MIP)\ set$ **syntax** $MatchIP :: BName \Rightarrow$ $(LNname\ list \Rightarrow HP \Rightarrow hState \Rightarrow hVar \Rightarrow MIP \Rightarrow boot$ $(\_ \vdash_{i-p} \langle \_, \_, \_, \_ \rangle \Rightarrow \langle \_ \rangle \quad [0,0,0,0,0,0]\ 81)$ **translations** $bn \vdash_{i-p} \langle ilnl, p, s, lv \rangle \Rightarrow \langle mip \rangle \rightleftharpoons (ilnl, p, s, lv, mip) \in MatchIP\ bn$
consts $MatchPV :: (HP \times HV \times hVar \times MPV)\ set$ **syntax** $MatchPV :: HP \Rightarrow HV \Rightarrow hVar \Rightarrow MPV \Rightarrow bool$ $(\vdash_{p-v} \langle \_, \_, \_ \rangle \Rightarrow \langle \_ \rangle \quad [0,0,0,0]\ 81)$ **translations** $\vdash_{p-v} \langle p, v, lv \rangle \Rightarrow \langle mpv \rangle \rightleftharpoons (p, v, lv, mpv) \in MatchPV$
consts $EvalE :: BName \Rightarrow hState \Rightarrow hVar \Rightarrow (HE \times EE)\ set$ **syntax** $EvalE :: BName \Rightarrow hState \Rightarrow hVar \Rightarrow HE \Rightarrow EE \Rightarrow bool$ $(\_, \_, \_ \vdash_e \langle \_ \rangle \Rightarrow \langle \_ \rangle \quad [0,0,0,0,0]\ 81)$ **translations** $bn, s, lv \vdash_e \langle e \rangle \Rightarrow \langle ee \rangle \rightleftharpoons (e, ee) \in EvalE\ bn\ s\ lv$
consts $MatchOE :: BName \Rightarrow (LName\ list \times HE \times hState \times MOE)\ set$ **syntax** $MatchOE :: BName \Rightarrow (LName\ list \times HE \times hState \times MOE \Rightarrow bool$ $(\_ \vdash_{o-e} \langle \_, \_, \_ \rangle \Rightarrow \langle \_ \rangle \quad [0,0,0,0,0]\ 81)$ **translations** $bn \vdash_{o-e} \langle olnl, e, s \rangle \Rightarrow \langle moe \rangle \rightleftharpoons (olnl, e, s, moe) \in MatchOE\ bn$

Fig. 5. Inductive Definitions and Judgement Forms

theorem *MatchOE_Result*: $bn \vdash_{o-e} \langle olnl, e, s\rangle \Rightarrow \langle moe\rangle \Longrightarrow$
$(moe = OEer\ MOEer) \vee (\exists oes.(moe = OEs\ oes))$
theorem *MatchPV_Result*: $\vdash_{p-v} \langle p, v, lv\rangle \Rightarrow \langle mpv\rangle \Longrightarrow$
$(mpv = PVer\ MPVer) \vee (mpv = PVf) \vee (\exists lv'.(mpv = PVs\ lv'))$
theorem *MatchIP_Result*: $bn \vdash_{i-p} \langle ilnl, p, s, lv\rangle \Rightarrow \langle mip\rangle \Longrightarrow$
$(mip = IPer\ MPVer) \vee (mip = IPer\ MIPer) \vee (mip = IPf) \vee (\exists ips.(mip = IPs\ ips))$
theorem *RunM_Result*: $bn, ilnl, olnl \vdash_m \langle ml, s\rangle \Rightarrow \langle m\rangle \Longrightarrow (\exists pls.(m = Ms\ pls)) \vee$
$(m = Mer\ MPVer) \vee (m = Mer\ MIPer) \vee (m = Mer\ EEer) \vee (m = Mer\ MOEer)$
theorem *RunB_Result*: $B \vdash_b \langle s\rangle \Rightarrow \langle rb\rangle \Longrightarrow (\exists bs.(rb = Bs\ bs)) \vee$
$(rb = Ber\ MPVer) \vee rb = Ber\ MIPer) \vee rb = Ber\ EEer) \vee rb = Ber\ MOEer)$
theorem *RunBL_Result*: $\vdash_{bl} \langle bls\rangle \Rightarrow \langle rb\rangle \Longrightarrow (\exists bs.(rb = Bs\ bs)) \vee$
$(rb = Ber\ MPVer) \vee rb = Ber\ MIPer) \vee rb = Ber\ EEer) \vee rb = Ber\ MOEer)$
theorem *RunC_Result*: $P \vdash_c \langle n, s\rangle \Rightarrow \langle rc\rangle \Longrightarrow (\exists s'.(rc = Cs\ s')) \vee$
$(rc = Cer\ MPVer) \vee rc = Cer\ MIPer) \vee rc = Cer\ EEer) \vee rc = Cer\ MOEer)$

5 Proving HW-Hume Program Correctness

Now, we prove two HW-Hume programs correct from the HW-Hume semantics in Isabelle.

5.1 Proving Swap

The first example is "*Swap*" which is very simple. There is only one box and one wire. The box Swap's out("*o*") is a link to its in("*i*"). The "*Swap.o*" and "*Swap.i*" are tuples. The matching rule "(x,y) -> (y,x)" will swap the values of "*Swap.i*". Figure 6 depicts the following code:

```
box Swap
  in  (i::(Bit,Bit))
  out (o::(Bit,Bit))
match
  (x,y) -> (y,x);
wire Swap
  (Swap.o initially (0,1))
  (Swap.i);
```

With our HW-Hume abstract syntax in Isabelle, this program is presented as:
constdefs *Swap_P* :: *HProg*
Swap_P ≡
(
[(∗ *Box* ∗)
(
''*Swap*'',
[(''*i*'', $(NAT, NAT)_t$)], (∗ *in* ∗)
[(''*o*'', $(NAT, NAT)_t$)], (∗ *out* ∗)

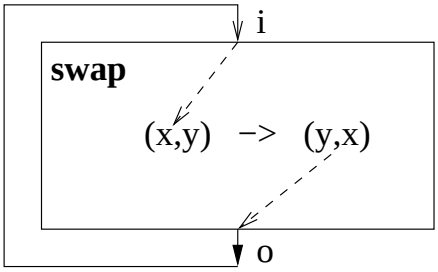


Fig. 6. Swap Diagram

```
        [
         ( (''x''_pv, ''y''_pv)_p,   (''y''_ev, ''x''_ev)_e )
        ]
    )
  ],
  [ (* Wire *)
    ( (''Swap'', ''o''),   (''Swap'', ''i'') )
  ],
)
```

constdefs *Swap_Init* :: *HInit list*

Swap_Init ≡

```
(
  [ (* Init *)
    ( (''Swap'', ''i'', (0_v, 1_v)_v )
  ]
)
```

After initialisation, "*Swap.i*" has value "(0,1)". After running box "*Swap*" once, "*Swap.o*" got value "(1,0)". Then, at the end of the cycle, it will be transfered to "*Swap.i*". So before the second cycle, "*Swap.i*" has value "(1,0)". Before the third cycle, "*Swap.i*" has value "(0,1)", and so on.

In Isabelle, the global states of HW-Hume is defined as *hState = hLocation ⇀ HV*. We can get the initial state by applying function *Init_hState* to *Swap_Init*. We define two states:

constdefs *Swap_S0* :: *hState*

Swap_S0 ≡ *empty*((*LI*, ''*Swap*'', ''*i*'') ↦ $(0_v, 1_v)_v$)

constdefs *Swap_S1* :: *hState*

Swap_S1 ≡ *empty*((*LI*, ''*Swap*'', ''*i*'') ↦ $(1_v, 0_v)_v$)

Simply, we have a lemma:

lemma *SwapInit* : *Init_hState Swap_Init = Swap_S0*

i.e. *Swap_S0* equals the initial state.

With our inductive defined HW-Hume semantics in Isabelle, we prove below lemmas by recursively calling inductive rules.

lemma *Swap_S0_Cycle_0* : $Swap\_P \vdash_c \langle 0, Swap\_S0\rangle \Rightarrow \langle Cs\ Swap\_S0\rangle$
lemma *Swap_S1_Cycle_0* : $Swap\_P \vdash_c \langle 0, Swap\_S1\rangle \Rightarrow \langle Cs\ Swap\_S1\rangle$
lemma *Swap_S0_Cycle_1* : $Swap\_P \vdash_c \langle 1, Swap\_S0\rangle \Rightarrow \langle Cs\ Swap\_S1\rangle$
lemma *Swap_S1_Cycle_1* : $Swap\_P \vdash_c \langle 1, Swap\_S1\rangle \Rightarrow \langle Cs\ Swap\_S0\rangle$
lemma *Swap_S0_Cycle_2* : $Swap\_P \vdash_c \langle 2, Swap\_S0\rangle \Rightarrow \langle Cs\ Swap\_S0\rangle$
lemma *Swap_S1_Cycle_2* : $Swap\_P \vdash_c \langle 2, Swap\_S1\rangle \Rightarrow \langle Cs\ Swap\_S1\rangle$

Finally, we have a theorem:
theorem *Swap_Cycles*:
$Swap\_P \vdash_c \langle 2 * n,\quad Init\_hState\ Swap\_Init\rangle \Rightarrow \langle Cs\ Swap\_S0\rangle$
$Swap\_P \vdash_c \langle 2 * n + 1, Init\_hState\ Swap\_Init\rangle \Rightarrow \langle Cs\ Swap\_S1\rangle$
Based on theorem *Swap_Cycles*, when we run *Swap* $2 * n$ times, "*Swap.i*" has value "(0,1)"; when we run *Swap* $2 * n + 1$ times, "*Swap.i*" has value "(1,0)".

5.2 Proving Adder

The second example is "*Adder*". There are three boxes and four wires. The box "*gen*" outputs from "(0,0,0)" to "(1,1,1)" in each cycle repetitively. This output ("*gen.t*") is linked to the in of box "*adder*". Matching rules of the box "*adder*" calculate full bit addition of "*adder.i*" by truth table. The result "*adder.o*" will be transfered to box "*output*". In the original version, the "*output*" is standard output. Because we cannot at present accommodate I/O in our semantics in Isabelle, we simulate standard output by box "*output*.

In each cycle, "*gen.t*" is a value from "(0,0,0)" to "(1,1,1)". At the end of a cycle, the value is transferred to "*adder.i*". On the next cycle, the full bit addition is stored in "*adder.o*". At the end of the cycle, the value is transferred to "*output.i*". Then in the third cycle, the box "*output*" throws it away. Figure 7 depicts the following code:

```
box gen
in  (i::(Bit,Bit,Bit))
out (o::(Bit,Bit,Bit), t::(Bit,Bit,Bit))
match
  (0,0,0) -> ((0,0,1),(0,0,0)) |
  (0,0,1) -> ((0,1,0),(0,1,0)) |
  (0,1,0) -> ((0,1,1),(1,0,0)) |
  (0,1,1) -> ((1,0,0),(1,1,0)) |
  (1,0,0) -> ((1,0,1),(0,0,1)) |
  (1,0,1) -> ((1,1,0),(0,1,1)) |
  (1,1,0) -> ((1,1,1),(1,0,1)) |
  (1,1,1) -> ((0,0,0),(1,1,1));

box adder
in  (i::(Bit,Bit,Bit))
out (o::(Bit,Bit))
match
  (0,0,0) -> (0,0) |
```

```
  (0,1,0) -> (1,0) |
  (1,0,0) -> (1,0) |
  (1,1,0) -> (0,1) |
  (0,0,1) -> (1,0) |
  (0,1,1) -> (0,1) |
  (1,0,1) -> (0,1) |
  (1,1,1) -> (1,1) ;

box output
in  (i::Bit, eat::(Bit,Bit))
out (o::Bit)
match
  (*,eat) -> (*);

wire gen    (gen.o initially (0,0,0)) (gen.i, adder.i trace);
wire adder  (gen.t)                   (output.eat trace);
wire output (output.o, adder.o)       (output.i);
```

With our HW-Hume abstract syntax in Isabelle, this program is presented as:

constdefs *Adder_P* :: *HProg*

$$
\begin{array}{l}
\mathit{Adder\_P} \equiv \\
(\\
\quad [\ (*\ \mathit{Box}\ *) \\
\quad\quad (\\
\quad\quad\quad ''gen'', \\
\quad\quad\quad [(''i'',\ (\mathit{NAT}, \mathit{NAT}, \mathit{NAT})_t)], \qquad\qquad\qquad\qquad\qquad\qquad (*\ \mathit{in}\ *) \\
\quad\quad\quad [(''o'',\ (\mathit{NAT}, \mathit{NAT}, \mathit{NAT})_t),\ (''t'',\ (\mathit{NAT}, \mathit{NAT}, \mathit{NAT})_t)], \quad (*\ \mathit{out}\ *) \\
\quad\quad\quad [\\
\quad\quad\quad\quad (\ (0_p, 0_p, 0_p)_p, \quad ((0_e, 0_e, 1_e)_e, (0_e, 0_e, 0_e)_e)_e\ ), \\
\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad (\ (1_p, 1_p, 1_p)_p, \quad ((0_e, 0_e, 0_e)_e, (1_e, 1_e, 1_e)_e)_e\ ) \\
\quad\quad\quad] \\
\quad\quad), \\
\quad\quad (\\
\quad\quad\quad ''adder'', \\
\quad\quad\quad [(''i'',\ (\mathit{NAT}, \mathit{NAT}, \mathit{NAT})_t)], \quad (*\ \mathit{in}\ *) \\
\quad\quad\quad [(''o'',\ (\mathit{NAT}, \mathit{NAT})_t)], \qquad (*\ \mathit{out}\ *) \\
\quad\quad\quad [\\
\quad\quad\quad\quad (\ (0_p, 0_p, 0_p)_p, \quad (0_e, 0_e)_e\ ), \\
\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad (\ (1_p, 0_p, 0_p)_p, \quad (1_e, 0_e)_e\ ), \\
\quad\quad\quad\quad \ldots \\
\quad\quad\quad] \\
\quad\quad), \\
\quad\quad (\\
\quad\quad\quad ''output'',
\end{array}
$$

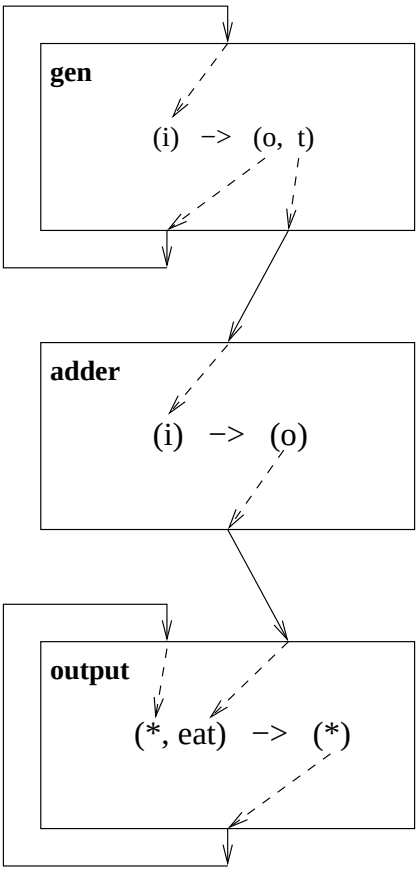


Fig. 7. Adder Diagram

```
        [(''i'', NAT), (''eat'', (NAT, NAT)_t)], (* in *)
        [(''o'', NAT)],                          (* out *)
        [
          ( (HP.I, ''eat''_pv)_p,   HE.I )
        ]
      )
  ],
  [ (* Wire *)
    ( (''gen'',    ''o'' ),   (''gen'',    ''i'' ) ),
    ( (''gen'',    ''t'' ),   (''adder'',  ''i'' ) ),
    ( (''adder'',  ''o'' ),   (''output'', ''eat'') ),
    ( (''output'', ''o'' ),   (''output'', ''i'' ) )
  ],
)
constdefs Adder_Init :: HInit list
Adder_Init ≡
(
  [ (* Init *)
    ( (''gen'',''i'', (0_v, 0_v, 0_v)_v )
  ]
)
```

We define global states of *Adder* in Isabelle as:

```
constdefs Adder_S0' :: hState

  Adder_S0' ≡ empty(
    (LI, ''gen'', ''i'') ↦ (0_v, 0_v, 0_v)_v
  )
constdefs Adder_S0 :: hState

  Adder_S0 ≡ empty(
    (LI, ''gen'', ''i'') ↦ (0_v, 0_v, 0_v)_v,
    (LI, ''adder'', ''i'') ↦ (1_v, 1_v, 1_v)_v,
    (LI, ''output'', ''eat'') ↦ (0_v, 1_v)_v
  )
constdefs Adder_S1' :: hState

  Adder_S1' ≡ empty(
    (LI, ''gen'', ''i'') ↦ (0_v, 0_v, 1_v)_v,
    (LI, ''adder'', ''i'') ↦ (0_v, 0_v, 0_v)_v
  )
constdefs Adder_S1 :: hState

  Adder_S1 ≡ empty(
    (LI, ''gen'', ''i'') ↦ (0_v, 0_v, 1_v)_v,
    (LI, ''adder'', ''i'') ↦ (0_v, 0_v, 0_v)_v,
    (LI, ''output'', ''eat'') ↦ (1_v, 1_v)_v
  )
```

...
constdefs *Adder_S7* :: *hState*
Adder_S7 ≡ *empty*(
(*LI*, ''*gen*'', ''*i*'') ↦ $(1_v, 1_v, 1_v)_v$,
(*LI*, ''*adder*'', ''*i*'') ↦ $(1_v, 0_v, 1_v)_v$,
(*LI*, ''*output*'', ''*eat*'') ↦ $(0_v, 1_v)_v$
)

With our inductively defined HW-Hume semantics in Isabelle, we prove the following lemmas by recursively calling inductive rules.

lemma *AdderInit* : *Init_hState Adder_Init* = *Adder_S0'*
lemma *Adder_S0'_Cycle_1* : $Adder\_P \vdash_c \langle 1, Adder\_S0' \rangle \Rightarrow \langle Cs\ Adder\_S1' \rangle$
lemma *Adder_S1'_Cycle_1* : $Adder\_P \vdash_c \langle 1, Adder\_S1' \rangle \Rightarrow \langle Cs\ Adder\_S2 \rangle$
...
lemma *Adder_S6_Cycle_1* : $Adder\_P \vdash_c \langle 1, Adder\_S6 \rangle \Rightarrow \langle Cs\ Adder\_S7 \rangle$
lemma *Adder_S7_Cycle_1* : $Adder\_P \vdash_c \langle 1, Adder\_S7 \rangle \Rightarrow \langle Cs\ Adder\_S0 \rangle$

Finally, we have a theorem:

theorem *Adder_Cycles*:
$Adder\_P \vdash_c \langle 0, Init\_hState\ Adder\_Init \rangle \Rightarrow \langle Cs\ Adder\_S0' \rangle$
$Adder\_P \vdash_c \langle 1, Init\_hState\ Adder\_Init \rangle \Rightarrow \langle Cs\ Adder\_S1' \rangle$
$Adder\_P \vdash_c \langle 0 + 8 * (n + 1), Init\_hState\ Adder\_Init \rangle \Rightarrow \langle Cs\ Adder\_S0 \rangle$
$Adder\_P \vdash_c \langle 1 + 8 * (n + 1), Init\_hState\ Adder\_Init \rangle \Rightarrow \langle Cs\ Adder\_S1 \rangle$
$Adder\_P \vdash_c \langle 2 + 8 * n, Init\_hState\ Adder\_Init \rangle \Rightarrow \langle Cs\ Adder\_S2 \rangle$
$Adder\_P \vdash_c \langle 3 + 8 * n, Init\_hState\ Adder\_Init \rangle \Rightarrow \langle Cs\ Adder\_S3 \rangle$
$Adder\_P \vdash_c \langle 4 + 8 * n, Init\_hState\ Adder\_Init \rangle \Rightarrow \langle Cs\ Adder\_S4 \rangle$
$Adder\_P \vdash_c \langle 5 + 8 * n, Init\_hState\ Adder\_Init \rangle \Rightarrow \langle Cs\ Adder\_S5 \rangle$
$Adder\_P \vdash_c \langle 6 + 8 * n, Init\_hState\ Adder\_Init \rangle \Rightarrow \langle Cs\ Adder\_S6 \rangle$
$Adder\_P \vdash_c \langle 7 + 8 * n, Init\_hState\ Adder\_Init \rangle \Rightarrow \langle Cs\ Adder\_S7 \rangle$

6 Conclusions

We have presented the embedding of HW-Hume in Isabelle and the proof of correctness of two programs.

The work discussed here is central to our longer term goal of a verified compiler from HW-Hume to Java. We recently decided to adopt Jinja as the target language, to enable an integrated approach in Isabelle. We have also modified our original HW-Hume to Java compiler to generate Jinja, and are currently formalising the translation from HW-Hume to Jinja. Time permitting, the next stages would be to embed this formalisation in Isabelle and to explore automatic support for proof that compilation maintains semantic consistency.

Acknowlegements

This research is partly supported by EU FP6 EmBounded. We would like to thank our colleagues in the wider Hume project for valuable discussion.

References

[Acz77] P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland, Amsterdam, 1977.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Cur92] P. Curzon. Compiler correctness and input/output. Technical report, Computer Laboratory, University of Cambridge, November 1992.

[Cur94] P. Curzon. The Verified Compilation of Vista Programs. Technical report, Computer Laboratory, University of Cambridge, January 1994.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, New York, NY, USA, 1993.

[Hen90] Matthew Hennessy. *The semantics of programming languages: an elementary introduction using structural operational semantics*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[KN06] Gerwin Klein and Tobias Nipkow. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006.

[LM04] C. Liu and G. Michaelson. Translating Hume to Java. In H-W. Loidl, editor, *Draft Proceedings of 5th Symposium on Trends in Functional Programming, Ludwig-Maximillian's Universitat, Munich, Germany*, pages 113–128, November 2004.

[MH04] G. Michaelson and Kevin Hammond. The Hume Language Definition and Report, Version 0.3. Technical report, Heriot-Watt University and University of St Andrews, 2004.

[MHS05] Greg Michaelson, Kevin Hammond, and Jocelyn Sérot. FSM-Hume is finite state. In Stephen Gilmore, editor, *Trends in Functional Programming, Volume 4*, volume 4 of *Trends in Functional Programming*, pages 19–28. Intellect, 2005.

[Mil80] Robin Milner. *How to Derive Inductions in LCF*. Edinburgh University Press, U.K., 1980.

[MP67] J. McCarthy and J. Painter. Correctness of a compiler for Arithmetic Expressions. In J. T. Schwarz, editor, *Proceedings of Symposium in Applied Mathematics, 19, Mathematical Aspects of Computer Science*. American Mathematical Society, 1967.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. `http://www.in.tum.de/~nipkow/LNCS2283`.

[Pal92] J. Palsberg. A provably correct compiler generator. In *Proceedings, ESOP '92, 4th European Symposium on Programming, Rennes, France*, February 1992.

[Pau00] Lawrence C. Paulson. A fixedpoint approach to (co)inductive and (co)datatype definitions. In *Proof, language, and interaction: essays in honour of Robin Milner*, pages 187–211. MIT Press, Cambridge, MA, USA, 2000.

[SC98] D. W. J. Stringer-Calvert. *Mechanical Verification of Compiler Correctness*. PhD thesis, University of York, March 1998.

[SF06] SRI-Formalware. *The PVS Specification and Verification System*. SRI International Computer Science Laboratory, 2006. `http://pvs.csl.sri.com/`.

[Ste93] Susan Stepney. *High Integrity Compilation: A Case Study*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1993.

Static Contract Checking for Haskell

Dana N. Xu[1] and Simon Peyton Jones[2] and Koen Claessen[3]

[1] University of Cambridge nx200@cam.ac.uk
[2] Microsoft Research Cambridge simonpj@microsoft.com
[3] Chalmers University of Technology koen@chalmers.se

Abstract. Program errors are hard to detect and are costly both to programmers who spend significant efforts in debugging, and to systems that are guarded by runtime checks. Static verification techniques have been applied to imperative and object-oriented languages, like Java and C#, but few have been applied to a higher-order lazy functional language, like Haskell. In this paper, we describe a sound and automatic static verification tool for Haskell, that is based on contracts and symbolic execution. Our approach gives precise blame assignments at compile-time in the presence of higher-order functions and laziness.

1 Introduction

Program errors are common in software systems, including those that are constructed from functional languages. For greater software reliability, such errors should be reported accurately and detected early during program development. *Contract checking* (both static and dynamic) has been widely used in procedural and object-oriented languages [19,10,4,2]. The difficulty of contract checking in functional languages lies in the use of higher-order functions. However, dynamic checking of contracts for higher-order functions has been studied by [8,3,7,14]. Recently, static pre/postcondition checking [29] as well as hybrid[1] contract checking [9,18,17,12] for functional languages have also been proposed.

In this paper, we combine the idea of the contract semantics [3] and the idea of the static verification through symbolic execution [29] to propose a sound automatic static contract checking framework for a higher-order lazy functional language, Haskell. Consider:

```
f :: [Int] -> Int
f xs = head xs `max` 0
```

```
head :: [a] -> a
head (x:xs) = x
head [] = error "empty list"
```

If we have a call `f []` in our program, its execution will result in the following error message from the runtime system of the Glasgow Haskell Compiler (GHC):

```
Exception: Prelude.head: empty list
```

This gives no information on which part of the program is wrong except that `head` has been wrongly called with an empty list. This lack of information is compounded by the fact that it is hard to trace the function calling sequence at run-time for lazy languages, such as Haskell.

[1] a static contract checking followed by a dynamic contract checking

The programmer's intention is that `head` should not be called with an empty list. To achieve this, programmers can give a contract to the function `head`. Contracts are implemented as pragamas with notation `{-# CONTRACT <contract> #-}`.

```
{-# CONTRACT head :: {s | not (null s)} -> {r | True} #-}
```

where `not` and `null` are just ordinary Haskell functions. This places the onus on callers to ensure that the argument to `head` satisfies the expected precondition. With this contract, our compiler would generate the following warning (with a counter-example) when checking the definition of `f`:

```
Warning: f [] calls head
         which may fail head's precondition!
```

Suppose we change `f`'s definition to the following:

```
f xs = if null xs then 0 else head xs `max` 0
```

With this correction, our compiler will not give any more warning as the precondition of `head` is now fulfilled.

Our goal is to detect crashes in a program where a *crash* is informally defined as an unexpected termination of a program (i.e. a call to `error`). Divergence (i.e. non-termination) is not a crash. We make the following contributions:

- Compared with the dynamic contract checking work by [8,3,14], our system can detect contract violations early at compile-time.
- Compared with the hybrid contract checking [9,18], we deal with a lazy language instead of a strict one.
- We allow data constructors to be used in constructing contracts so that properties of the subcomponents of a data type can be specified (§2.3). This is not addressed in [8,3,9,18], but in [14].
- Compared with our earlier ESC/Haskell system [29], we can now
  - detect and locate bugs more precisely by giving contracts to higher order function's parameters which themselves may be functions (§2.1);
  - reduce false alarms caused by laziness in an efficient way (§2.2);
  - specify pre- and post-condition in a type-like way and allow contract synonyms to be defined easily (§2.4).
- We develop a concise notation ($\triangleright$ and $\triangleleft$) for describing contract checking (§5), which enjoys many useful properties (§5.3). Thus equipped, we give a *complete* proof of the soundness and completeness of dynamic contract checking that takes care of laziness; this proof is much trickier than it looks [30]!

We implement the idea in one branch of the GHC, which can accept full Haskell and also support separate compilation as the verification is modular.

2 Overview

The type of a function constitutes a partials specification to the function. For example, `inc :: Int -> Int` says that `inc` is a function that takes an integer and returns an integer. A *contract* of a function gives more detailed specification. For example: `{-# CONTRACT inc :: {x | x > 0} -> {r | r > x} #-}` says that the function `inc` takes a positive value and returns a value that is greater

than the input. A contract can therefore be viewed as a refinement to a type, so it is also known as refinement type in [11,6,9].

The constraint used in a contract such as `{x | x > 0}` is an arbitrary boolean-valued Haskell expression (like $\lambda x.x > 0$). Programmers do not need to learn a new language of predicates; instead they can use an arbitrary Haskell expression. In this section, we give the flavour of contracts with various examples; Section 4 makes contracts precise.

2.1 Contracts for Higher Order Functions

Contract notation is more expressive than the `requires`, `ensures` notation used in the ESC/Haskell [29]. We can give a contract to a parameter of a higher-order function. This is not expressible in the ESC/Haskell syntax. Consider an example adopted from [3]:

```
f1 :: (Int -> Int) -> Int
{-# CONTRACT f1 :: ({x | True} -> {y | y>=0}) -> {r | r>=0} #-}
f1 g = (g 1) - 1
f2   = f1 (\x -> x - 1)
```

The contract of `f1` says that if `f1` takes a function, which returns a natural number when given any integer, the function `f1` itself returns a natural number.

The Findler-Felleisen algorithm in [8] (a run-time contract checking algorithm) can detect a violation of the contract of `f1`, however, it cannot tell that the argument of `f1` in the definition of `f2` fails `f1`'s precondition. On the other hand, the Sage system [18] (a hybrid contract checking system) can detect the failure in `f2` statically, and can report contract violation of `f1` at run-time. Our system can report both failures at compile-time with the following messages:

```
Error: f1's postcondition fails
       because (g 1) >= 0 does not imply (g 1) - 1 >= 0
Error: f2 calls f1
       which fails f1's precondition
```

In theory, the idea in [9] could also detect both failures during compile-time.

2.2 Laziness

Laziness causes false alarms, for example:

```
fst (a, b) = a
f3 = fst (5, error "f")
```

Syntactically, the call `fst (5, error "f")` is unsafe because of the existence of a call to `error`. The only static verification tool that caters for laziness is the ESC/Haskell system [29], which can reduce false alarms due to laziness by inlining. In the above case, the function `fst` is inlined, so the call to `fst` in `f3` becomes `5` which is syntactically safe. However, if the size of the lazy function is big, or the function is recursive, the inlining strategy is not ideal. For example:

```
fstN :: (Int, Int) -> Int
{-# CONTRACT fstN :: ({x | True}, Any) -> {r | True} #-}
fstN (a, b) n = if n>0 then fstN (a+1, b) (n-1)
                       else a

g2  = fstN (5, error "fstN") 100
```

We need to inline `fstN` for 100 times to know `g2` is safe.

To reduce the false alarms due to laziness, we introduce a special contract `Any`, which every expression satisfies. The contract of `fstN` says that it does not care what the second component of the argument is, as long as the first component is crash-free, the result is crash-free. Here, with the contract `Any`, without inlining any function, our system can tell that `g2` is safe.

2.3 Contract Constructors

As shown in §2.2, the data constructor `(,)` may be used in constructing a contract. Programmers may want to give a contract to the sub-components of a data constructor. For example, we can use the list constructor (:) to create a contract like this:

$\{x \mid x > 0\} : \{xs \mid$ `all` $(< 0)\ xs\}$

which says that the first element in the list is positive while the rest are all negative (where `all :: (a->Bool)->[a]->Bool`). We can also have this:

$\{x \mid x > 0\} : \{y \mid y > 0\} :$ `Any`

which says the first two elements are positive while the rest can be undefined. In general we allow any user-defined data constructors be used in constructing a contract.

2.4 Contract Synonym

In previous sections, we use the contract `{y | True}` at many places. In our system, we allow programmers to define contract synonyms which are similar to the idea of type synonyms. Contract synonyms are implemented as pragamas with notation `{-# TYPE <contract synonym> #-}`. For example, we may have:

```
{-# TYPE Ok = {x | True} #-}
{-# TYPE Nat = {x | x >= 0} #-}
{-# TYPE NotNull = {xs | not (null xs)} #-}

{-# CONTRACT head :: NotNull -> Ok #-}
head (x:xs) = x
```

2.5 The Plan for Verification

It is all very well for programmers to *claim* that a function satisfies a contract, but how can we *verify* that claim statically (i.e. at compile time)? Our overall plan, which is similar to that of Blume and McAllester [3], is as follows.

– Our overall goal is to prove that the program does not *crash*, so we must first say what programs are, and what it means to "crash" (§3).
– Next, we give a semantic specification for what it means for a function f to "satisfy a contract" t, written $f \in t$, in the first place (§4).
– From the definition $f = e$ we form the term $e \triangleright t$ pronounced "e ensures t". This term behaves just like e except that (a) if e disobeys t then the term crashes; (b) if the context uses e in a way not permitted by t then the term loops. The term $e \triangleright t$ is essentially the wrapper mechanism first described by Findler and Felleisen [8], with some important refinements (§5).
– With these pieces in place, we can write down our main theorem (Section 5.3), namely that

$$e \in t \quad \Longleftrightarrow \quad (e \triangleright t) \text{ is crash-free}$$

We must ensure that everything works properly, even if e diverges, or laziness is involved, or the contract contains divergent or crashing terms.
– Using this theorem, we may check whether $f \in t$ holds as follows: we attempt to prove that $(e \triangleright t)$ cannot crash, regardless of the context in which it is used. We conduct this proof in a particularly straightforward way: we simply perform symbolic evaluation of $(e \triangleright t)$. If we can simplify the term to a new term e', where e' is syntactically safe — that is, contains no crashes whatsoever — then we are done. This test is sufficient, but not necessary; of course, the general problem is undecidable.

3 The Language

The language presented in this paper, named language $\mathcal{H}_{core}$, is simply-typed lambda calculus with case-expression, constructors and integers. In our implementation, we use the GHC Core Language [24], which is similar to System F and includes parametric polymorphism, but the language we use here is simpler.

3.1 Syntax

The syntax of our language $\mathcal{H}_{core}$ is shown in Figure 1. A program is a module that contains a set of data type declarations, contract specifications and function definitions. Expressions include integers, variables, type and term abstractions, type and term applications, constructors and `case` expressions. We omit local `letrec` as well, we only have recursive (or mutually recursive) top-level functions. Moreover, we treat `let`-expressions as syntactic sugar: `let` $x = e_1$ `in` $e_2 \equiv_s (\lambda x.e_2)\ e_1$. There are two unusual expressions adopted from [29]:

`BAD` is an expression that *crashes*. A program crashes if and only if it evaluates to `BAD`. For example, a user-defined function `error` can be defined as: `error s = BAD`. A preprocessor ensures that source programs with missing cases of pattern matching are explicitly replaced by the corresponding equations with `BAD` constructs.

`UNR` (short for "unreachable") is an expression that gets stuck. This is *not* considered as a crash, although execution comes to a halt without delivering a result. A program that loops forever also does not crash, and does not

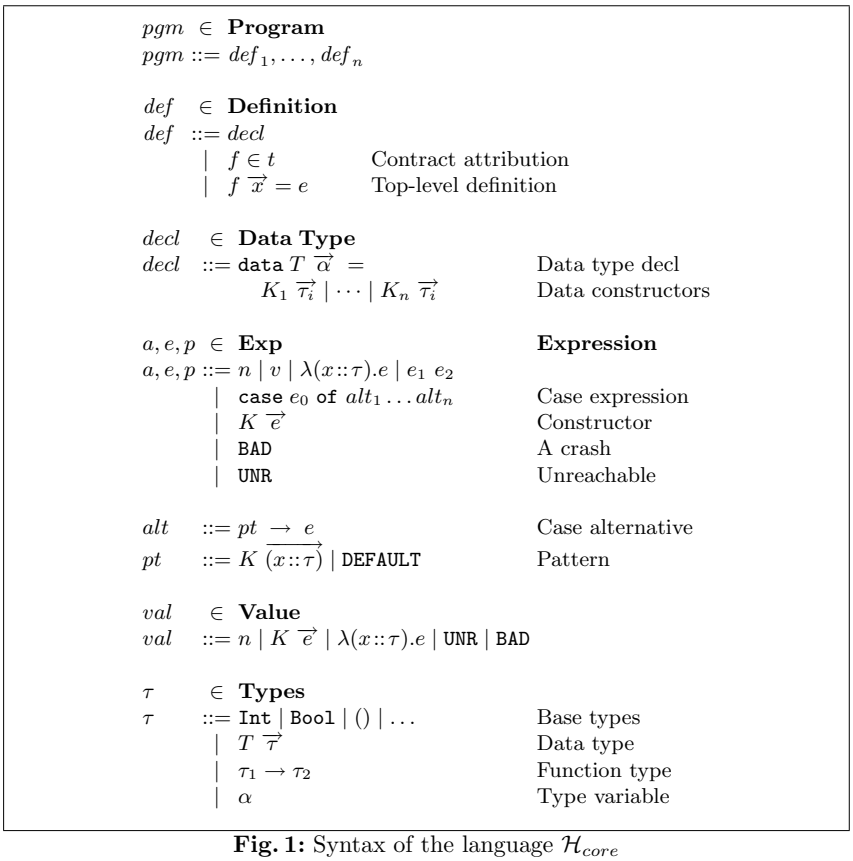


Fig. 1: Syntax of the language $\mathcal{H}_{core}$

deliver a result, so you can think of `UNR` as a term that simply goes into an infinite loop.

These two constructs are for internal usage and hidden from programmers. Their behaviour is made precise by the operational semantics in Figure 2 in §3.2.

The top-level declaration $f \in t$ is the claim that f satisfies contract t. We discuss contracts in §4. We assume all functions and their contracts in a program are well-typed before being verified. Details of type checking are omitted here, but can be found in [30].

3.2 Operational Semantics

Our language's semantics is given by the confluent, non-deterministic rewrite rules in Figure 2. We use a reduction-rule semantics, rather than (say) a de-

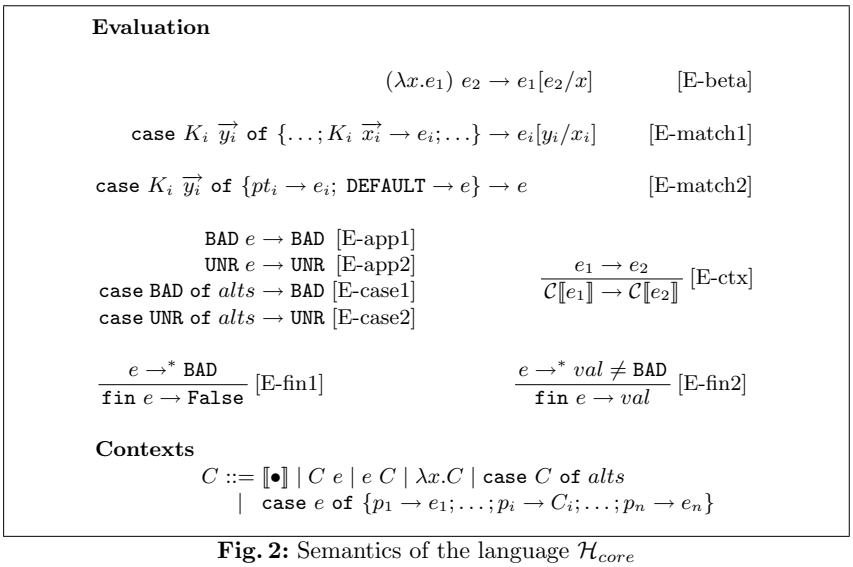


Fig. 2: Semantics of the language $\mathcal{H}_{core}$

terministic more machine-oriented semantics, because the more concrete the semantics becomes, the more involved the proofs become too.

The rule [E-beta] performs the standard β-reduction. When a scrutinee of a `case` expression is a data constructor that matches one of the patterns, it is also a redex, shown in the rules [E-match1] and [E-match2]. The group of rules [E-app1]-[E-case2] deal with the propagation of the exceptional values `BAD` and `UNR`. Whenever the current redex is `BAD` (or `UNR`), the evaluation terminates immediately and the value `BAD` (or `UNR`) is returned. Finally, [E-ctx] allows a rewrite to apply at an arbitrary place, defined by context $\mathcal{C}$.

Evaluation proceeds by repeatedly replacing the current redex with its corresponding 1-step reduction until a value is reached. (Note that `BAD` and `UNR` are considered as values.) The relation $e_1 \to e_2$ performs a single step reduction and the relation $\to^*$ is the reflexive-transitive closure of $\to$.

The special function `fin` converts crashing boolean expression to `False` shown in [E-fin1] and has no effect on other values shown in [E-fin2]. The `fin` is only for internal usage, not exposed to programmers. Moreover, the reduction rules involving `fin` are not used in a real execution of a program either, but very useful in proving our grand theorem (Theorem 1) for crashing contracts. If a boolean expression in a contract crashes, we say the contract crashes.

3.3 Crashing

A program is correct if and only if the `main` function in a program does not *crash*. Our technique can only guarantee *partial* correctness: a diverging program does not crash. We can define a diverging function like this: `bot = bot`.

Definition 1 (Crashes) *A closed term* e crashes *iff* $e \rightarrow^* \texttt{BAD}$.

Definition 2 (Diverges) *A closed expression* e diverges*, written* $e\uparrow$*, iff either* $e \rightarrow^* \texttt{UNR}$*, or there is no value val such that* $e \rightarrow^* val$.

In many cases, the special constructor `UNR` behaves the same as looping, so it can be seen as an identifiable divergence. This definition of divergence, which includes `UNR`, saves repetition later.

During compile-time, the only way to check the safety of a program is to see whether the program is syntactically safe.

Definition 3 (Syntactic safety) *A (possibly-open) expression* e *is* syntactically safe *iff* $\texttt{BAD} \notin e$. *Similarly, a context* $\mathcal{C}$ *is syntactically safe iff* $\texttt{BAD} \notin \mathcal{C}$.

The notation $\texttt{BAD} \notin e$ means `BAD` does not syntactically appear anywhere in e, similarly for $\texttt{BAD} \notin \mathcal{C}$. For example, $\lambda x.x$ is syntactically safe while $\lambda x.\,(\texttt{BAD}, x)$ is not. However, in some cases, a syntactically non-safe expression cannot crash because the `BAD`s in the expression cannot be reached.

Definition 4 (Crash-free Expression) *A (possibly-open) expression* e *is crash-free iff :* $\forall \mathcal{C}.\ \texttt{BAD} \notin \mathcal{C},\ \vdash \mathcal{C}[\![e]\!] :: (),\ \mathcal{C}[\![e]\!] \not\rightarrow^* \texttt{BAD}$

The notation $\vdash \mathcal{C}[\![e]\!] :: ()$ means $\mathcal{C}[\![e]\!]$ is closed and well-typed. Definition 4 says that if an expression does not crash in all safe contexts, which are like probes for `BAD`, then the expression cannot crash regardless whether there is any `BAD` syntactically appearing in it because all of them are unreachable. That means a crash-free expression may not be syntactically safe, for example:

```
\x-> case x*x >= 0 of {True -> x+1; False -> BAD}
```

The tautology $x * x >= 0$ is always true, so the `BAD` can never be reached. On the other hand, $(\texttt{BAD}, 3)$ is not crash-free because there exists a context, $\texttt{fst}\ [\![\bullet]\!]$, such that: $\texttt{fst}\ (\texttt{BAD}, 3) \rightarrow \texttt{BAD}$.

In short, crash-freeness is a *semantic* concept, and hence undecidable, while syntactic-safety is *syntactic* and readily decidable. Certainly, *a syntactically safe expression is crash-free.*

4 Contract Syntax and Semantics

t	$\in$ **Contract**	
t	$::= \{x \mid p\}$	Predicate Contract
	$\mid\ x\colon t_1 \rightarrow t_2$	Dependent Function Contract
	$\mid\ (t_1, t_2)$	Tuple Contract
	$\mid\ \texttt{Any}$	Polymorphic Any Contract

Fig. 3: Syntax of contracts

4.1 Syntax

The syntax of contracts is given in Figure 3, which is similar to those in [8,3,9]. A predicate contract $\{x \mid p\}$ can be viewed as a boolean-valued function $\lambda x.p$ where p is an arbitrary expression in $\mathcal{H}_{core}$. We use syntax $x\colon t_1 \to t_2$ for a dependent function contract where x can be used in t_2. If x is not used in t_2, it can be omitted. We adopt this notation from [9,1], which is equivalent to $\Pi x\colon t_1 \to t_2$.

In §2, we have seen an example of using tuple contract. We restrict ourselves to tuple contracts in this paper. In our full implementation, we deal with arbitrary user-defined data constructor contracts.

We introduce a special polymorphic contract named `Any` which every expression satisfies. The *any* contract in [8,7] corresponds to our $\{x \mid \texttt{True}\}$, which is different from `Any`.

4.2 Contract Satisfaction

We give the semantics of contracts by defining e satisfies t (written $e \in t$) in Figure 4. Since divergence is not considered as a crash, if $e\uparrow$, then e satisfies all contracts. The notation, $\vdash e :: \tau$ and $\vdash_c t :: \tau$ say that both expression e and its contract t are closed and well-typed (detailed typing rules are in [30]).

Given $\vdash e :: \tau$ and $\vdash_c t :: \tau$, we define $e \in t$ as follows:

$$e \in \{x \mid p\} \iff e\uparrow \text{ or } (e \text{ is crash-free and } p[e/x] \not\rightarrow^* \{\texttt{BAD}, \texttt{False}\}) \quad \text{[A1]}$$

$$e \in x\colon t_1 \to t_2 \iff e\uparrow \text{ or } \forall e_1 \in t_1.\ (e\ e_1) \in t_2[e_1/x] \quad \text{[A2]}$$

$$e \in (t_1, t_2) \iff e\uparrow \text{ or } (e \to^* (e_1, e_2) \text{ and } e_1 \in t_1, e_2 \in t_2) \quad \text{[A3]}$$

$$e \in \texttt{Any} \iff \texttt{True} \quad \text{[A4]}$$

Fig. 4: Contract Satisfaction

Predicate Contract In [A1] we say an expression e has contract $\{x \mid p\}$ written $e \in \{x \mid p\}$, we mean e either diverges or it is a crash-free expression e that satisfies the predicate p in the contract. The predicate p may give four possible outcomes: $\uparrow$, `True`, `False` and `BAD`. We consider $\uparrow$ and `True` to be safe while the outcomes `False` and `BAD` to be unsafe. To say e satisfies p, we require $p[e/x] \not\rightarrow^* \{\texttt{BAD}, \texttt{False}\}$, which means $p[e/x]\uparrow$ or $p[e/x] \to^* \texttt{True}$. If $e \in \{x \mid p\}$, the contract $\{x \mid p\}$ can be seen as e's postcondition.

Any crash-free expression satisfies contract $\{x \mid \texttt{True}\}$ and $\{x \mid e\}$ where $e\uparrow$. For example, $\lambda x.\ x \in \{x \mid \texttt{True}\}$ and $(3, 5) \in \{x \mid \texttt{True}\}$. Only diverging expressions satisfy contracts $\{x \mid \texttt{False}\}$ and $\{x \mid \texttt{BAD}\}$, for example, $\texttt{UNR}, \texttt{bot} \in \{x \mid \texttt{False}\}$ and $\texttt{UNR}, \texttt{bot} \in \{x \mid \texttt{BAD}\}$.

Function Contract In [A2], we expect the expression to evaluate to a lambda-expression. We say an expression e has dependent function type $x\colon t_1 \to t_2$, when e is applied to any argument e_1 that satisfies the contract t_1, it produces a result that satisfies the contract $t_2[e_1/x]$. The t_1 and t_2 themselves can be function contracts.

If both of them are predicate contracts, t_1 is a precondition and t_2 is a postcondition. If the precondition diverges or evaluates to `True`, we expect the function body to satisfy the postcondition:

$$\lambda x.\ x \quad \in \{x \mid \texttt{bot}\} \to \{r \mid \texttt{True}\}$$
$$\lambda x.\ \texttt{BAD} \in \{x \mid \texttt{bot}\} \to \texttt{Any}$$
$$\lambda x.\ \texttt{BAD} \notin \{x \mid \texttt{bot}\} \to \{r \mid \texttt{True}\}$$

If the postcondition diverges, we expect the function body to be crash-free:

$$\lambda x.\ x + 1 \in \{x \mid \texttt{True}\} \to \{r \mid \texttt{bot}\}$$
$$\lambda x.\ x + 1 \notin \texttt{Any} \to \{r \mid \texttt{bot}\}$$

In this case, if the argument has contract, $\{x \mid \texttt{True}\}$, we know the it is crash-free, so $x + 1$ is crash-free. Since any crash-free expression satisfies $\{x \mid \texttt{bot}\}$, $x + 1$ satisfies the postcondition. However, if the precondition is `Any`, it means `BAD` could be the argument, and `BAD` $+ 1$ crashes, then it does not satisfy the postcondition $\{x \mid \texttt{bot}\}$. Moreover, a lambda-expression satisfies only a function contract, for example: $\lambda x.\ \texttt{BAD} \notin \texttt{Any}$ and $\lambda x.\ \texttt{BAD} \in \texttt{Any} \to \texttt{Any}$.

Tuple Contract In [A3], we expect the expression to evaluate to a tuple and each component satisfies its corresponding contract. A non-crash-free tuple expression satisfies only a tuple contract. For example:

$$(\texttt{BAD}, 3) \notin \{x \mid (\texttt{snd}\ x) > 0\} \qquad (\texttt{BAD}, 3) \in (\texttt{Any}, \{x \mid x > 0\})$$

However, we can have: $(\texttt{True}, 2) \in \{x \mid (\texttt{snd}\ x) > 0\}$. This coincides with the definition in [A1] which requires a crash-free expression.

Any Contract If we only have [A1]-[A3], the expression `BAD` does not satisfy any contract because we expect crash-free expressions to satisfy those contracts. In [A4], we introduce a special contract, named `Any` which any expression satisfies including `BAD`. For example: $\lambda x.6 \in \texttt{Any} \to \{x \mid \texttt{True}\}$. Due to laziness, even given an argument that crashes, the function $\lambda x.6$ returns a crash-free value. In general, a function, whose return-contract is `Any`, is a function that crashes. The function `error` may have the following contract:

```
{-# CONTRACT error :: {x | True} -> Any #-}
```

Furthermore, `BAD` only satisfies the contract `Any` because it fails the constraints stated in [A1]-[A3]. For example: $\texttt{BAD} \notin (\texttt{Any}, \texttt{Any})$ and $\texttt{BAD} \notin \texttt{Any} \to \texttt{Any}$.

4.3 Alternatives

An obvious alternative design choice for contract satisfaction would be to drop the "e is crash-free" condition in the predicate contract case:

$$e \in \{x \mid p\} \iff e\uparrow \text{ or } p[e/x] \not\rightarrow^* \{\texttt{BAD}, \texttt{False}\}$$

Then we could get rid of `Any`, because `{x | True}` would do instead. On the

other hand a polymorphic contract meaning "crash-free" is extremely useful in practice, so we would probably need a new contract `Ok` defined thus:

$$e \in \texttt{Ok} \iff e\uparrow \text{ or } e \text{ is crash-free}$$

This all seems quite plausible, but it has a fatal flaw: *we could not find a definition for $\triangleright$ that validates our main theorem.* That is, our chosen definition for $\in$ makes the forthcoming Section 5 work out, whereas the otherwise-plausible alternative appears to prevent it doing so.

4.4 Open Expressions and Open Contracts

So far we have only said what it means for a *closed* term to satisfy a *closed* contract. It is easy to extend the definition to open terms and open contracts. A contract judgement has the form $\Delta \vdash e \in t$ where Δ is a mapping from variable to its contract and the definition (i.e. expression) it denotes.

Definition 5 (Contract judgement) *We write $\Delta \vdash e \in t$ to mean that e has contract t assuming that any free variable in e has contract given by Δ and any free variable in t has definition given by Δ. Suppose $\Delta = \{f_1 \mapsto (t_1, e_1), \ldots, f_n \mapsto (t_n, e_n)\}$, we have definition:*

$$\Delta \vdash e \in t \iff \lambda f_1 \ldots . f_n . e \in t_1 \to \cdots \to t_n \to t$$

This means, in theory (e.g. in the formalization of the verification), we only need to deal with closed expressions; in practice (e.g. in our examples), we may refer to the environment Δ when necessary.

5 Contract Checking

The goal of contract checking is to check whether the `main` function satisfies the contract $\{x \mid \texttt{True}\}$. As mentioned in §2.5, we convert this satisfaction checking problem to crash-freeness checking problem, by making use of our main theorem:

$$e \in t \quad \iff \quad (e \triangleright t) \text{ is crash-free}$$

Our goal is to define $e \triangleright t$ such that Theorem 1 holds.

In fact, we define two expression constructors, $e \triangleright t$ and $e \triangleleft t$, where e is an expression and t is a contract. These two forms are not part of the syntax of expressions (Figure 1); rather they are thought of as macros, which expand to a particular expression. Informally:

- $e \triangleright t$, pronounced "e `ensures` t", crashes if e does not satisfy t.
- $e \triangleleft t$, pronounced "e `requires` t", crashes if the context does not satisfy t.

The definition of the projection is given in Figure 5. The expression constructors $\triangleright$ and $\triangleleft$ are dual to each other, so we define $e \triangleright t$ and $e \triangleleft t$ through a combined constructor: $e \overset{r}{\bowtie} t$ which takes 3 inputs namely e, t and r and produces a term that behaves like e if $e \in t$; otherwise, it behaves like r or $\neg r$. The r denotes a dummy result we would like to give to an expansion. It can be either `BAD` or

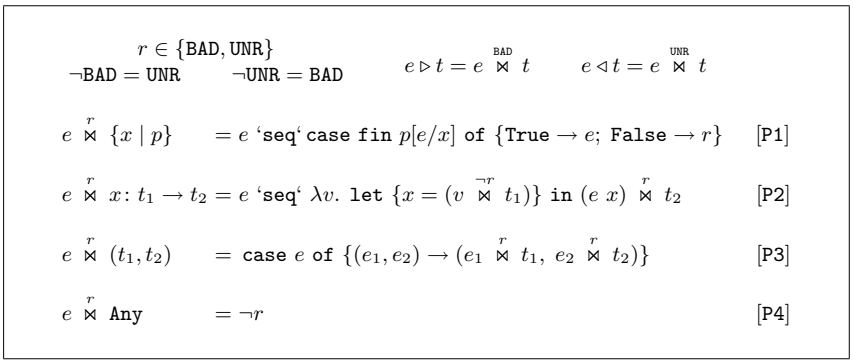


Fig. 5: Expression Constructors $\triangleright$ and $\triangleleft$

`UNR`, which are complementary to each other as shown in the top-left corner of Figure 5.

The beauty of the projections lies in [P1] and [P2]. Let us ignore the decoration `seq` and `fin` first, and focus on the gist of each projection. In [P1], we have:

$$e \triangleright \{x \mid p\} = \texttt{case}\ p[e/x]\ \texttt{of}\ \ \texttt{True} \rightarrow e;\ \ \texttt{False} \rightarrow \texttt{BAD}$$

$$e \triangleleft \{x \mid p\} = \texttt{case}\ p[e/x]\ \texttt{of}\ \ \texttt{True} \rightarrow e;\ \ \texttt{False} \rightarrow \texttt{UNR}$$

The $e \triangleright \{x \mid p\}$ says that if e fails to satisfy the predicate p, it is e's fault because we would like e to **ensure** the property p and we signal this fault with the `BAD`. That means if $e \triangleright \{x \mid p\}$ is crash-free, then the `BAD` is not reachable so we know the predicate p is satisfied. On the other hand, the $e \triangleleft \{x \mid p\}$ says that if e fails to satisfy the predicate p, the rest of the code should be unreachable because we **require** the caller of e to have the property p before e is called. In [P2], we swap the direction of the triangles for the parameter:

$$e \triangleright t_1 \rightarrow t_2 = \lambda v.\ ((e\ (v \triangleleft t_1)) \triangleright t_2) \qquad e \triangleleft t_1 \rightarrow t_2 = \lambda v.\ ((e\ (v \triangleright t_1)) \triangleleft t_2)$$

The $e \triangleright t_1 \rightarrow t_2$ says that in order to ensure the postcondition to be t_2, we require the argument to satisfy the precondition t_1. The $e \triangleleft t_1 \rightarrow t_2$ says that if the caller of e (i.e. the argument given to e) cannot ensure t_1, the `BAD` introduced by the $\triangleright$ signals this failure.

It becomes more interesting when we have higher order functions, the direction of the triangle swaps back and forth:

$$\begin{aligned} & (\lambda x.e) \triangleright (t_1 \rightarrow t_2) \rightarrow t_3 \\ = {} & \lambda v_1.\ (((\lambda x.e)\ (v_1 \triangleleft t_1 \rightarrow t_2)) \triangleright t_3) \\ = {} & \lambda v_1.\ (((\lambda x.e)\ (\lambda v_2.\ ((v_1\ (v_2 \triangleright t_1)) \triangleleft t_2))) \triangleright t_3) \end{aligned}$$

Recall the higher-order function `f1 g = (g 1) - 1` in §2.1, we have:

$$
\begin{aligned}
&\texttt{f1} \triangleright (\{x \mid \texttt{True}\} \to \{y \mid y >= 0\}) \to \{r \mid r >= 0\} \\
&= \ldots \\
&= \lambda v_1.\ \texttt{case}\ (v_1\ 1) >= 0\ \texttt{of} \\
&\qquad \texttt{True} \to \texttt{case}\ (v_1\ 1) - 1 >= 0\ \texttt{of} \\
&\qquad\qquad \texttt{True} \to (v_1\ 1) - 1 \\
&\qquad\qquad \texttt{False} \to \texttt{BAD} \\
&\qquad \texttt{False} \to \texttt{UNR}
\end{aligned}
$$

As $((v_1\ 1) >= 0)$ does not imply $((v_1\ 1) - 1 >= 0)$, the residual `BAD` indicates a postcondition failure. This illustrates how we get the first error message in §2.1. As `f1`'s definition does not satisfy its contract, at call sites of `f1`, we only use its contract. As a result, in `f2`, as $(x - 1 \geq 0)$ is not true for all x, we get the second error message in §2.1.

In [P2], if we inline the `let`, we have:

$$e \overset{r}{\bowtie} x\colon t_1 \to t_2 = e\ \texttt{`seq`}\ \lambda v.\ ((e\ (v \overset{\neg r}{\bowtie} t_1)) \overset{r}{\bowtie} t_2[(v \overset{\neg r}{\bowtie} t_1)/x])$$

We need to guard v by $v \overset{\neg r}{\bowtie} t_1$ in the contract t_2 if x is used in t_2. This is because we allow partial functions to be used in contracts. Without this guard, we have:

$$e \overset{r}{\bowtie} x\colon t_1 \to t_2 = e\ \texttt{`seq`}\ \lambda v.\ ((e\ (v \overset{\neg r}{\bowtie} t_1)) \overset{r}{\bowtie} t_2[v/x]) \qquad [Q2]$$

With [Q2], our Theorem 1 does not hold and a counterexample is as follows.

```
{-# CONTRACT h :: {x | not (null x)} -> {r | head x == r} #-}
h (y:ys) = y
```

5.1 The Function `seq`

The function `seq` is defined as follows:

$$e_1\ \texttt{`seq`}\ e_2 = \texttt{case}\ e_1\ \texttt{of}\ \{\texttt{DEFAULT} \to e_2\}$$

The function `` `seq` `` ensures two things:

- If $e \to^* \texttt{BAD}$, and $t \neq \texttt{Any}$, then $e \triangleright t \to^* \texttt{BAD}$.
- If $e\!\uparrow$, then $(e \triangleright t)\!\uparrow$.

For example, in `[P1]`, if e is `UNR` and t is $\{x \mid \texttt{False}\}$, without the e `` `seq` ``, we have: $(\texttt{UNR} \triangleright \{x \mid \texttt{False}\}) \to^* \texttt{BAD}$ which fails the Theorem 1 because $\texttt{UNR} \in \{x \mid \texttt{False}\}$ while $\texttt{UNR} \triangleright \{x \mid \texttt{False}\}$ crashes. For another example, in `[P2]`, the e `` `seq` `` ensures that $\texttt{BAD} \not\equiv \lambda x.\texttt{BAD}$ because `BAD` denotes a crash while $\lambda x.\texttt{BAD}$ is a value which will only crash when applied to an argument. Moreover, in [P3], we expect the expression e to evaluate to a tuple. Recall the example $(\texttt{BAD}, 3) \notin \{x \mid \texttt{snd}\ x > 0\}$, we can verify it by checking crash-freeness of $((\texttt{BAD}, 3) \triangleright \{x \mid \texttt{snd}\ x > 0\}) \to^* (\texttt{BAD}, 3)$ which is not crash-free. Similarly, we can verify that $(\texttt{BAD}, 3) \in (\texttt{Any}, \{x \mid x > 0\})$ as $\texttt{BAD} \triangleright \texttt{Any} = \texttt{UNR}$ which is crash-free and $(3 \triangleright \{x \mid x > 0\}) \to^* 3$ which is also crash-free.

5.2 The function `fin`

The purpose of introducing this special function `fin` is to deal with crashing contracts. If we do not convert crashing predicate to `False`, we cannot prove the

Theorem 1, for example: $\lambda x.x \in \{x \mid \texttt{BAD}\} \to \{x \mid \texttt{True}\}$ but $(\lambda x.x \triangleright \{x \mid \texttt{BAD}\} \to \{x \mid \texttt{True}\}) \to^* \lambda v.\ v$ `` `seq` BAD``, which is not crash-free.

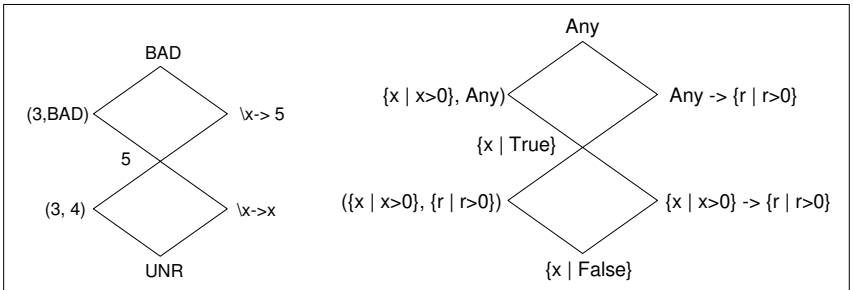
5.3 Properties

We order expressions with a relation named crashes-more-often. The notation $\vdash \mathcal{C}[\![e_i]\!] :: ()$ means $\mathcal{C}[\![e_i]\!]$ is closed and well-typed.

Definition 6 (Crashes more often) *e_1 crashes more often than e_2, written $e_1 \preceq e_2$, iff for all contexts $\mathcal{C}$, such that $\forall i = 1, 2.\ \vdash \mathcal{C}[\![e_i]\!] :: ()$*

$$\mathcal{C}[\![e_2]\!] \to^* \texttt{BAD} \quad \Rightarrow \quad \mathcal{C}[\![e_1]\!] \to^* \texttt{BAD}$$

A lattice for expressions is shown in Figure 6(a). The lower diamond contains all the crash-free expressions while the upper diamond contains all the expressions that may crash. A corresponding ordering for contracts is given in Figure 6(b).



(a) lattice for expressions (b) lattice for contracts

Fig. 6: Lattice

The $\preceq$ relation, the satisfaction $\in$ and the two constructors $\triangleright$ and $\triangleleft$ enjoy many nice properties:

Congruence	$\forall e_1, e_2.\ e_1 \preceq e_2 \iff \forall \mathcal{C},\ \texttt{fin} \notin \mathcal{C},\ \mathcal{C}[\![e_1]\!] \preceq \mathcal{C}[\![e_2]\!]$
Projection (w.r.t. $\preceq$,$\succeq$)	If $e \in t$, then (a) $e \triangleleft t \preceq e$; (b) $e \triangleright t \succeq e$.
Key Lemma	If a closed e is crash-free, then $e \triangleleft t \in t$.
Monotonicity of $\in$	If $e_1 \in t_1$ and $e_1 \preceq e_2$, then $e_2 \in t$
Idempotence	(a) $e \triangleright t \triangleright t \equiv e \triangleright t$ (b) $e \triangleleft t \triangleleft t \equiv e \triangleleft t$
Projection Pair	$\forall e \in t.\ e \triangleright t \triangleleft t \preceq e$
Closure Pair	$\forall e \in t.\ e \preceq e \triangleleft t \triangleright t$

These lemmas form a basis for proving our main result:

Theorem 1 (Soundness and Completeness of Contract Checking) *For all closed expression e, closed contract t,*

$$(e \triangleright t) \text{ is crash-free} \quad \iff \quad e \in t$$

Theorem 1 is similar to the soundness and completeness theorems in [3]. We have a proof of the theorem for non-dependent function contracts (i.e. $x\colon t_1 \to t_2$ where x is not used in t_2), under the assumption that the terms in contracts are terminating. Diverging contracts are addressed in §6. We have not yet completed the proof for dependent function contracts.

5.4 Soundness of Static Verification

To achieve the plan in §2.5, our static contract checking consists of 3 steps:

1. Construct the expression $e \triangleright t$.
2. Simplify $e \triangleright t$ as much as possible, to e', say.
3. See if `BAD` is syntactically in e'; if not, e' is crash-free.

Our static contract checking is *sound* because if e' is syntactically safe, we know e' is crash-free. As the simplification process preserves the semantics of an expression, we know $e \triangleright t$ is crash-free. By Theorem 1, we know $e \in t$. Step 2 and 3 are the same as the symbolic simplification idea proposed in the ESC/Haskell [29] so we omit them here.

Our static verification is *not complete*, which means even if $e \in t$, our algorithm may not be able to tell because we cannot detect divergence. We cannot tell that the expression `case bot of {DEFAULT -> BAD}` diverges i.e. crash-free. So we give a false alarm because it is not syntactically safe.

6 Discussion – Diverging Contracts

So far we describe contract satisfaction and contract checking for under the assumption that contracts converged. But since contracts contain arbitrary Haskell terms, there is no way to enforce this assumption. If we drop the assumption, the main theorem no longer holds, for a tiresome reason. Consider $e =$ `(BAD,BAD)` and $t =$ `{x|bot}`. Then $e \notin t$ (since e is not crash-free); but $e \triangleright t$ reduces to this term: `case fin bot of {True -> (BAD, BAD); False -> BAD}` Annoyingly, this term *is* crash-free because the `(fin bot)` term diverges. Interestingly, our static evaluation engine does not attempt to detect divergence, and therefore does not reduce this term to `bot` (although doing so would be correct). So our implementation will not eliminate the `BAD`s in this program and hence does validate the theorem. All we need is to do is to make the theory match the implementation!

Doing so is tiresome but not difficult. We enhance the special function `fin` to $\texttt{fin}_n$, which converts divergence to `True` in n steps shown in [E-fin3] and [E-fin4]. The idea is that we initially create the terms $e \triangleright_n t$ with $\texttt{fin}_n$ at every point. The n is like "fuel": $\texttt{fin}_i$ permits i reduction steps on its argument before giving up and returning `True`.

$$\frac{e \to e' \quad n > 0}{\texttt{fin}_n\ e \to \texttt{fin}_{n-1}\ e'}\ \text{[E-fin3]} \qquad \frac{}{\texttt{fin}_0\ e \to \texttt{True}}\ \text{[E-fin4]}$$

Now the main theorem becomes:

$$e \in t \quad \iff \quad \exists N.\ e \triangleright_N t \text{ is crash-free}$$

That is $e \in t$ if and only if $e \triangleright_N t$ can be show crash-free for some *finite* amount of "fuel" N.

7 Related Work

In [3], Blume and McAllester introduce the concept of contract semantics and prove the sound-and-completeness of the dynamic contract checking algorithm

in [8] with respect to the contract semantics. We use a similar approach, but aimed at static checking. We use a lazy language, and support constructor contracts; on the other hand they deal with recursive contracts which we do not. There are many other differences of detail, and we believe that our proof (not presented here) is rather simpler than theirs.

Also inspired by [8,3], Hinze et. cl. [14] implement contracts as a library in Haskell and also provide contract constructors such as pairs, lists, etc. Compared with [8,3,14], we apply a theory similar to those but to static contract checking so that we can detect bugs early. Another dynamic contract checking work is the Camila project [25] which use monads to encapsulate the pre/post-conditions checking behaviour.

In [7], Findler and Blume discover that contracts are pairs of projections. Our projections use crashes-more-often as the partial ordering rather than the specificness of the contract. Moreover, we also discover the *projection pair* property which plays a crucial role in our proof and makes our proof a bit different from the one in [3].

The hybrid contract checking framework [9,18,17,12], in theory, can be as powerful as our system. But in practice, our symbolic execution strategy adopted from [29] gives more flexibility to the verification as illustrated in §2.1. In [26], Wadler and Findler show how contracts fit with hybrid types and gradual types by requiring casts in the source code. The casts are similar to our $\triangleright$ and $\triangleleft$. Compared with [9,18,17,12,26], we deal with a lazy language and also handle constructor contracts.

Compared with the dependent type approaches [28,5,23,27], we separate type and contract declarations so that type related work (e.g. type inference) and contract related techniques can be developed independently.

In Hoare Type Theory (HTT) [22,21], higher-order predicates and inductive predicates can be used in specifications. We share the same expressiveness from these two aspects as we allow higher-order functions and recursive functions to be used in contracts. But our contracts do not contain quantifiers.

The Programatica toolset [13] verifies Haskell programs with P-Logic [16] while we use Haskell itself as the specification language and rely on sound symbolic evaluation for its reasoning. Our approach eliminates the effort of inventing and learning a new logic together with its theorem prover.

In [15], a compositional assertion checking framework has been proposed with a set of logical rules for handling higher order functions. Their assertion checking technique is primarily for postcondition checking and is currently used for manual proofs. Apart from our focus on automatic verification, we also support precondition checking that seems not to be addressed in [15].

In [20], pattern matching failures can be inferred during compile time without specifications from programmers. Notably they use a language of regular expressions for contracts, which is incomparable with ours, and the technical details are very different.

8 Conclusion

We have presented a sound and automatic static contract checking framework for Haskell. Based on contract semantics and symbolic execution, our approach gives precise blame assignments at compile-time in the presence of higher-order functions and laziness. We have implemented this idea in GHC and experimented with various small examples and are in the process of verifying larger programs.

Acknowledgements The first author would like to thank Microsoft Research Cambridge for a scholarship as well as a grant for attending conferences.

References

1. Lennart Augustsson. Cayenne - language with dependent types. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 239–250, New York, NY, USA, 1998. ACM Press.
2. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. *CASSIS*, LNCS 3362, 2004.
3. Matthias Blume and David McAllester. A sound (and complete) model of contracts. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 189–200, New York, NY, USA, 2004.
4. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll. An overview of jml tools and applications, 2003.
5. Sa Cui, Kevin Donnelly, and Hongwei Xi. Ats: A language that combines programming with theorem proving. In Bernhard Gramlich, editor, *FroCos*, volume 3717 of *Lecture Notes in Computer Science*, pages 310–320. Springer, 2005.
6. Rowan Davies. Refinement-type checker for standard ml. In *AMAST '97: Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology*, pages 565–566, London, UK, 1997. Springer-Verlag.
7. Robert Bruce Findler and Matthias Blume. Contracts as pairs of projections. In *Functional and Logic Programming*, pages 226–241. Springer Berlin / Heidelberg, 2006.
8. Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 48–59, New York, NY, USA, 2002. ACM Press.
9. Cormac Flanagan. Hybrid type checking. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 245–256, New York, NY, USA, 2006. ACM Press.
10. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press.
11. Tim Freeman and Frank Pfenning. Refinement types for ml. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 268–277, New York, NY, USA, 1991. ACM Press.
12. Jessica Gronski and Cormac Flanagan. Unifying hybrid types and contracts. In *Eighth Symposium on Trends in Functional Programming*, April 2007.
13. Thomas Hallgren, James Hook, Mark P. Jones, and Richard Kieburtz. An overview of the Programatica toolset. In *High Confidence Software and Systems Conference*, 2004.

14. Ralf Hinze, Johan Jeuring, and Andres Löh. Typed contracts for functional programming. In *FLOPS '06: Functional and Logic Programming: 8th International Symposium*, pages 208–225, 2006.
15. Kohei Honda and Nobuko Yoshida. A compositional logic for polymorphic higher-order functions. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 191–202, New York, NY, USA, 2004. ACM Press.
16. Richard B. Kieburtz. P-logic: property verification for haskell programs. *Draft*, 2002.
17. Kenneth Knowles and Cormac Flanagan. Type reconstruction for general refinement types. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007*. Springer-Verlag, April 2007.
18. Kenneth Knowles, Aaron Tomb, Jessica Gronski, Stephen N. Freund, and Cormac Flanagan. SAGE: Unified hybrid checking for first-class types, general refinement types, and dynamic (extended report. *http://sage.soe.ucsc.edu/sage-tr.pdf*, 2006.
19. K. Rustan M. Leino and Greg Nelson. An extended static checker for Modular-3. In *CC '98: Proceedings of the 7th International Conference on Compiler Construction*, pages 302–305, London, UK, 1998. Springer-Verlag.
20. Neil Mitchell and Colin Runciman. Unfailing Haskell: A static checker for pattern matching. In *TFP '05: The 6th Symposium on Trends in Functional Programming*, pages 313–328, 2005.
21. Aleksandar Nanevski, Amal Ahmed, Greg Morrisett, and Lars Birkedal. Abstract predicates and mutable adts in hoare type theory. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 189–204. Springer, 2007.
22. Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in hoare type theory. In John H. Reppy and Julia L. Lawall, editors, *ICFP*, pages 62–73. ACM, 2006.
23. Tim Sheard. Languages of the future. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 116–119, New York, NY, USA, 2004. ACM Press.
24. The GHC Team. *The Glasgow Haskell Compiler User's Guide*. www.haskell.org/ghc/documentation.html, 1998.
25. J Visser, J. N. Oliveira, Barbosa L. S., J. F. Ferreira, and A. Mendes. CAMILA revival: VDM meets haskell. In Nico Plat and Peter Gorm Larsen, editors, *Overture Workshop (co-located with FM'05)*, 2005.
26. Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Workshop on Scheme and Functional Programming*, Sept 2007.
27. Edwin M. Westbrook, Aaron Stump, and Ian Wehrman. A language-based approach to functionally correct imperative programming. In Olivier Danvy and Benjamin C. Pierce, editors, *ICFP*, pages 268–279. ACM, 2005.
28. Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227, New York, NY, USA, 1999. ACM Press.
29. Dana N. Xu. Extended static checking for haskell. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 48–59, New York, NY, USA, 2006. ACM Press.
30. Dana N. Xu, Simon Peyton Jones, and Koen Claessen. Sound Haskell - Technical Report. *http://www.cl.cam.ac.uk/users/nx200/research/hvs.ps*, 2007.

Debugging Lazy Functional Programs by Asking the Oracle*

Holger Siegel, Bernd Braßel

CAU Kiel usw

Abstract. The complexity of lazy evaluation forbids classic debugging techniques like a simple step-by-step representation of the buggy program run. Therefore, most sophisticated tools for searching bugs in lazy functional programs try to display the run as if the program's underlying semantics was strict. In order to provide such a strict representation current approaches gather much information about the executed program. We utilized a new technique to drastically reduce the amount of gathered data and show how to use the reduced information to implement a debugging tool which supports declarative debugging as well as a strict step-by-step tracer.

1 Introduction

The task of designing tools to find bugs in lazy functional programs is demanding. On one hand, the sophisticated strategy employed by the underlying implementation enables the programmer to write code on a high level of abstraction. On the other hand the same sophisticated strategy makes it very hard to understand how a given program is executed step-by-step. Most successful approaches to debugging solve this problem by collecting enough data to represent the program's execution as if the underlying strategy was strict, which is much easier to understand. Examples for such approaches are declarative debugging, cf. [4, 5], observations, cf. [3], and redex trailing, cf. [7]. The most comprehensive tool, HAT [2], comprises all three approaches among others.

In order to present information about the program in a simple way, the above approaches collect data during its execution. This is especially true, the more powerful the tool is, e.g., the HAT system collects megabytes of data in many cases. In [1], we developed an approach to collect considerably less data and still be able to provide the user with a strict view on the execution of his program. The basic idea is that the critical information to replay a lazy computation as if the underlying semantics was strict is when unneeded redexes are discarded. Therefore, we only count the number of strict steps between such discarding steps. We call the resulting list of numbers an *oracle*. Different debugging tools can then be realized as strict monadic versions of the original program correctly consuming the number of steps in an oracle. [1] contains a soundness proof for this technique, this paper presents the implementation of a debugging tool based on the oracle technique.

* This work has been partially supported by DFG grant Ha 2457/5-2.

```
module Example where

import Prelude hiding (length)

length []     = 0
length (_:xs) = length xs

exp = length (take 2 (fiblist 0))

fiblist x = fib x : fibs (x+1)

fib :: Int -> Int
fib _ = error "this will not be evaluated"
```

Fig. 1. Example program

2 Example Sessions

So far, our debugging tool supports two modes. The first is an implementation of the well known declarative debugging method, described in Section 2.1. The second is a step-by-step tracer allowing us to follow a program's execution as if the underlying semantics was strict, skipping uninteresting sub computations. In addition, the tool gives some support to search bugs in programs employing I/O. This approach to "virtual I/O" is presented along with the step-by-step mode in Section 2.2.

2.1 Declarative Debugging

Figure 1 shows a small example program containing an intentional error to demonstrate the declarative debugging mode. The function `fiblist` creates a potentially infinite list of delayed calls to function `fib`. Due to laziness, `fib` is never evaluated in our example, so we omit its definition. The infinite list is cut to the first three elements by a call to function `take`, which is defined in the usual way. On top level, function `length` is applied to count the elements of the resulting list. It is to be expected that the program returns the number 2.

```
> :l Example
Example> exp
0
```

We see that running the program reveals the result `0`, which indicates that there must be a bug somewhere. Therefore, we switch on the debug mode and execute the program once again.

```
Example> :set +d
Example> exp
```

In the background, our example program (along with all of its imported modules) is transformed to a program `OracleExample`.

```
Example> exp
processing: OracleExample
up-to-date: OraclePrelude
```

As we see, our `OraclePrelude` is still up to date, and thus not generated again. The programs resulting from this transformation behave exactly like the original program. The only difference is that – as a side effect – it will produce an "oracle". Before continuing with the debugging session we take a look at this oracle.

After evaluating `exp` in `OracleExample` has successfully terminated, the current directory contains a file called `Example.steps`:

```
$ cat Example.steps
[2,0,1,0,0,23]
```

These numbers compactly represent the laziness information. If every expression in the program was evaluated there would have been only a single number. This number indicates how many steps that evaluation would have taken. The fact that there are six numbers for this example tells us that five expressions have been discarded without evaluation. (Two calls to `fib`, two to `(+)` and one to `fiblist`). For more details about the oracle format, how it is produced and why it can be utilized to correctly execute the traced program strictly, cf. [1].

The user does not see anything of the oracle or the steps file. Directly after the steps file has been produced, the debugging tool proceeds by applying a second transformation on the modules.

```
up-to-date: StrictPrelude.hs
generating ./StrictExample.hs
```

The second transformation produces Haskell modules named `Strict*.hs`. These Haskell modules contain the definitions to execute the original program with an underlying strict semantics. The details of this transformation will be presented in Section 3 on page 7.

After the second transformation the actual debugging session is started.

```
 ____    ____    _____
(  _ \  (_  _)  (  _  )   Believe
 ) _ <   _)(_    )(_)(    in
(____/()(____)()(_____)() Oracles
--------type ? for help----------

exp
```

Initially, we only see a call to function `exp` which represents the whole program. By pressing `i` we turn on *inspect mode*. In inspection mode, the result of every sub computation is directly shown and can be "inspected" by the user, i.e., rated as correct or wrong. Inspection mode corresponds therefore to the declarative

debugging method. But as we will see in the next section, showing the results of sub computations can be turned on and off at will. (Of course, there is a help menu available, showing a list of all possible inputs.)

After pressing `i` the debugger evaluates the expression and displays the result.

```
exp ~> 0
```

We expected main to have value `2`, but the program delivered value `0`. Thus, we enter `w` (*wrong*) in order to tell the debugger that the result was wrong. The debugging tool stores this choice as explained in Section 3. As the value of `exp` depends on several function calls on the right hand side of its definition, the tool now displays the first of these calls in a left-most innermost order:

```
fiblist 0 ~> _ : (_ : _)
```

The line above shows that the expression `fiblist 0` has been evaluated to a list that has at least two elements. This might be correct, but we are not too sure, since this result depends strongly on the evaluation context. A "don't know" in declarative debugging actually corresponds to the skipping of sub computations in the step-by-step mode, as described in the next section. We therefore press `s` (*skip*).

```
take 2 (_ : (_ : _)) ~> [_,_]
```

Actually, this looks quite good. By entering `c` (*correct*) we declare that this sub computation meets our expectation. Now the following calculation is displayed:

```
length [_,_] ~> 0
```

The function `length` is supposed to count the elements of a list. Since the argument is a two-element list, the result should be `2`, but it is actually `0`. By pressing `w` we therefore state that this calculation is erroneous. Now the debugger asks for the first sub computation leading to this result:

```
length [_]  ~> 0
```

This is wrong, too, but for the sake of demonstration we delay our decision. By pressing the space bar (*step into*) we move to the sub expressions of `length [_]`. We now get to the final question:

```
length [] ~> 0
```

The length of an empty list `[]` is zero, so by pressing `c` (*correct*) we state that this evaluation step is correct. Now we have reached the end of the program execution, but a bug has not been isolated yet. We have narrowed down the error to the function call `length [_,_]`, but still there are unrated sub computations which might have contributed to the erroneous result. The tool asks if the user wants to restart the debugging session re-using previously given ratings:

```
end reached. press 'q' to abort or any other key to restart.
```

```
module IOExample where

import Prelude hiding (getLine)

getLine :: IO String
getLine = getChar >>= testEOL

testEOL :: Char -> IO String
testEOL c = if c=='\n' then return []
                       else getLine >>= \ cs -> return (c:cs)

main = getLine >>= writeFile "userInput"
```

Fig. 2. I/O example

After pressing `<SPACE>`, the debugger restarts and asks for the remaining function calls. There is only one unrated call left within the erroneous sub computation:

```
length [_] ~> 0
```

Now we provide the rating we previously skipped. After entering `w` (*wrong*) it is evident which definition contains the error:

```
found bug in rule:
  lhs = length [_]
  rhs = 0
```

2.2 Step-by-Step Debugging and Virtual I/O

A further interesting advantage of our approach to reexecute the program with a strict evaluation strategy is the possibility to include "virtual I/O". During the execution of the original program, all externally defined I/O-actions with non-trivial results, i.e., other than `IO ()`, are stored in a special file. These values are retrieved during the debugging session. In addition, selected externally defined I/O-actions, e.g., `putChar`, are provided with a "virtual implementation". To show what this means, we demonstrate how the `main` action of the program found in figure 2 is treated by our debugging tool. As described in the previous section, the program is executed to obtain the oracle in the file `IOExample.steps`. As this program contains user interaction, we also have to enter a line. We type `abc` for this demonstration. Meanwhile, along with the file containing the steps, a file `IOExample.ext` was written, containing only the sequence of values of `getChar` (and their size).

```
~/oracle> cat IOExample.ext
3,'a'3,'b'3,'c'4,'\n'
```

There is no need to identify the different calls to external functions, since I/O-actions will be executed in the strict version in exactly the same order as in the original program. This is of course essential for correctness. We now start the debugging tool, and look st single steps by typing `<SPACE>` twice. This is, what the tool displays:

```
main
getLine
getLine ~> getChar >>= testEOL
main ~> (getChar >>= testEOL) >>= writeFile "userInput"
initial action computed. press any key to execute it
```

In step-by-step mode, we only get to see results when a subcomputation is finished. The above lines mean that the evaluation of both, `getLine` and `main` is now complete. The results are partial calls of the bind operator `(>>=)` waiting for the world, so to speak. We press an arbitrary key to start the action followed by a `<SPACE>` to make on more single step and get:

```
getChar >>= testEOL
getChar
```

When we hit `<SPACE>` now, two things happen at once. First, the value `'a'` is retrieved from the file and, second, the GUI called `B.I.O.tope` is started, which represents the virtual I/O environment. The `B.I.O.tope` is told that someone has typed an `a` on the console. This is the "virtual I/O-action" we connected with `getChar`. The window coming up is shown in Figure 3. Meanwhile on the

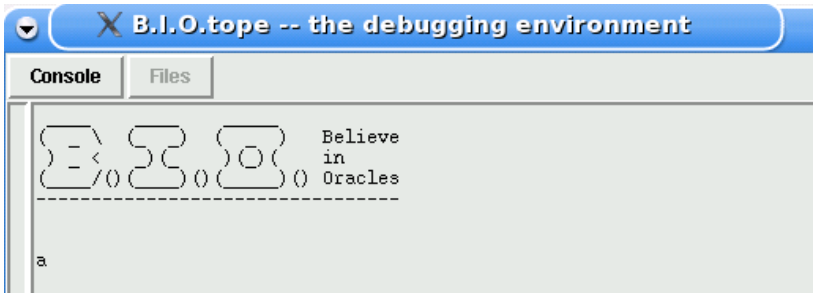


Fig. 3. The B.I.O.tope Virtual I/O Environment

console we see the call `testEOL 'a'`, which we skip by typing `s`. We directly see the result:

```
testEOL 'a' ~> (getChar >>= testEOL) >>= testEOL_lambda 'a'
(getChar >>= testEOL) >>= testEOL_lambda 'a'
```

Admittedly, the expression `testEOL_lambda 'a'` shows that the source code binding is improvable. Now we wonder, whether or not the current sub computation is interesting or not. We type `r` to have a look at the result and get:

```
(getChar >>= testEOL) >>= testEOL_lambda 'a' ~> IO "abc"
```

This is fine, so we decide to skip the computation by pressing `s`. Note, that as soon as a result is shown, we could also rate the sub computation, i.e., tell the tool that this result is correct or wrong. This information will then be considered if we restart the debugging session in inspection mode, cf. Section 2.1. It is also noteworthy that the virtual I/O commands are never issued twice although even, if we would have decided on going into the sub computation instead of skipping it.

The final action of our program is:

```
writeFile "userInput" "abc"
```

Executing this action brings another change to the `B.I.O.tope` as shown in Figure 4. There we can see the GUI has switched to the file dialog. It contains a list of files which have been read (`R:`) or written (`W:`) during the debugging session and clicking a file in this list makes the file contents visible as they are at the current point of the debugging session.

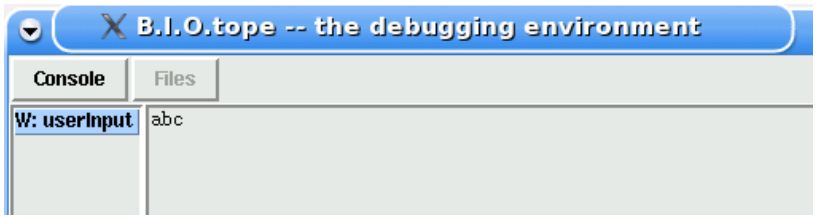


Fig. 4. Files in the B.I.O.tope Virtual I/O Environment

3 Implementation

In this section we present the runtime library `StrictSteps.hs` and its interaction with the programs that have been generated by the transformation introduced in Section 2.1. Reconsider the program in Figure 1. For this program the transformation creates a Haskell program called `StrictExample.hs`. Each generated module uses the functions that are exported by the library `StrictSteps.hs`. It also imports the transformed versions of its original imports. In this case the only such module is `StrictPrelude`. Finally, some functions from the original Haskell `Prelude` are needed. All in all, the module head of the generated module looks like this:

```
module StrictExample where

import StrictSteps
import Prelude (Maybe(..),(.),Eq(..),Show(..),Ordering(..),
                Either(..),String,Bool(..),Char(..),Float(..))
import qualified Prelude (IO,return,(>>=))
import StrictPrelude
```

3.1 Encoding of Oracles

Conceptually, an oracle is a list of logical values. This list is being consumed while an instrumented program is evaluated according to a leftmost innermost strategy. If the next entry of the oracle has value `True`, the next reduction step according to strict evaluation order will be evaluated. If the next entry has value `False`, then the next redex will be skipped and the result will be replaced by a placeholder value called `underscore`. In order to represent this oracle in a compact way it is encoded as a list of natural numbers. A number n stands for a list of n entries whose value is `True` followed by one entry of value `False`.

```
type Oracle = BoolStack
type BoolStack = [Int]
```

The function `popBoolStack` removes the first entry from an encoded oracle and returns the resulting oracle paired with the value of the removed entry. The function `pushBoolStack` appends an entry to the head of an oracle and returns the resulting oracle.

```
popBoolStack  :: BoolStack -> (BoolStack, Bool)
pushBoolStack :: BoolStack -> Bool -> BoolStack
```

An empty oracle is constructed according to the following declaration:

```
emptyBoolStack :: BoolStack
emptyBoolStack = [0]
```

For example, the oracle produced in Section 2.1 to evaluate `exp` in Figure 1 was `[2,0,1,0,0,23]`. It stands for a list containing 31 entries: the first two entries have value `True`, then there are two `False` followed by one `True`, then three `False` and finally 23 `True`. Having value `True`, the first entry indicates that expression `exp` has to be evaluated. The next entry tells us that the next leftmost innermost call in Figure 1, i.e., `fiblist 0`, is also unfolded. The next two entries have value `False` and thereby indicate that the leftmost innermost calls `fib x` and `x+1` must not be evaluated but replaced by an `underscore` (also denoted by `_`) as a placeholder value. Replacing `x+1` by `_` the next call is `fibs _`. This call is then also evaluated whereas the next free expressions, `fib _`, `x+1` and another `fibs _` are discarded. The final 23 entries tell us that the remaining computation is totally strict (and 23 steps long).

3.2 The Representation of `underscore`

Expressions which are not evaluated are replaced by the special value `underscore`. Therefore, at least conceptually, every data type has to be augmented with a new element which represents that value. From a semantical point of view `underscore` resembles the undefined value $\perp$. If we were only interested in computing the same result with a strict semantics, we could actually represent `underscore` by a call to the Haskell function `undefined` as discarded expressions are guaranteed to be never needed in the evaluation. However, we want the debugging tool to print intermediate results and, therefore, we need to be able to tell undefined values from defined ones.

Another option is to add a new constructor to every data declaration. However, this would make the inclusion of external functions and data types much more complicated. In order to, e.g., call `(+)`, we would need to convert to and fro for each argument and the result. In addition, much of the behavior already provided would have to be duplicated, e.g., the way strings of characters are shown.

Therefore a trick is applied, which makes use of the fact that `underscore` is never evaluated outside of the display routines. We represent `underscore` by an exception:

```
underscore :: a
underscore = throw NonTermination
```

Since the oracle that guides the evaluation indicates which expressions are needed to run the program, it is guaranteed that this value will never be accessed while the program is being evaluated. It will be accessed when the debugger tries to print out a data value. As the whole evaluation is monadic, cf. the next sub section, the exception can be safely caught whenever values are printed.

3.3 Representing Values

The debugger must be able to display a textual representation of the arguments and results of the function calls being debugged. In order to provide more flexibility for the debugging tool, we represent expressions in a term structure rather than a simple string. This makes it possible to, e.g., restrict the depth in which expressions are shown and enables pretty printing. Therefore, the data type `Term` is introduced:

```
data Term = Term String [Term] | Underscore | Fail String
```

The constructor `Term` contains the name of the applied symbol and a term representation of its arguments. As discussed above, `Underscore` stands for those expressions, which were not evaluated. Finally, `Fail` represents exceptions that occurred during the execution along with an error message. The implementation of the corresponding mechanism is, however, not finished by now.

In order to retrieve term representations of a given expression, each data type has to be an instance of the class `ShowTerm`:

```
class ShowTerm a where
  showCons   :: a -> DebugMonad Term
```

These instances are automatically generated by the transformation. For example, the following instance declaration is generated for lists:

```
instance ShowTerm a => ShowTerm [a] where
  showCons [] = return (consTerm "[]" [])
  showCons (x1:x2) = do sx1 <- showTerm x1
                        sx2 <- showTerm x2
                        return (consTerm ":" [sx1,sx2])
```

The generated instances depend on the generic function `showStep`, which is responsible for catching the exception thrown by `underscore`:

```
showTerm :: ShowTerm a => a -> DebugMonad Term
showTerm x = liftIO (catch (x `seq` return Nothing) (return . Just)) >>=
             maybe (showCons x) ( e -> case e of
                                    NonTermination -> return Underscore
                                    ErrorCall s    -> return (Fail s))
```

3.4 The Debug Monad

The whole evaluation of the generated program takes place in a monad, the `DebugMonad`. This monad is a state monad, managing the following state:

```
data DebuggerState = DebuggerState { oracle      :: Oracle,
                                     displayMode :: IORef DMode,
                                     skipped     :: BoolStack,
                                     unrated     :: BoolStack }
```

`oracle` contains the part of the oracle that has not yet been consumed.

`displayMode` contains display options like the verbosity level and the depth up to which terms should be printed. In addition this field contains a flag indicating, which of the two debugging modes introduced in Section 2 is active.

`skipped, unrated` are encoded the same way as oracles. They indicate which functions are still unrated (`True`) or have been rated as correct (`False`). The list `skipped` holds the ratings of the functions that have already been displayed in the current program run, whereas `unrated` holds the ratings of the function calls that still have not been displayed in the current program run, but might have been rated in a former evaluation cycle. The purpose of `skipped` and `unrated` will be explained in greater detail in Section 3.5.

In addition to the `DebugState`, three *evaluation modes* are encoded in data type `StepMode`:

`StepBackground` The expression is evaluated without user interaction. For every sub expression previously given information about its correctness will be preserved by moving the corresponding entries from `unrated` to `skipped`.

`StepInteractive` Function calls are displayed and rated by the user. To display its result, an expression is evaluated in mode `StepBackground`. When the user presses `<SPACE>`, meaning that he wants to inspect a sub computation in greater detail, the calls of that sub computation are re-evaluated in mode `StepInteractive`.

`StepCorrect` Like in background mode, the expression is evaluated without user interaction. When an expression is rated as correct, all its subexpressions are considered correct, too. Thus, its subexpressions are evaluated with evaluation mode `StepCorrect`, and the information stored in `skipped` and `unrated` will remain unchanged.

The monad `DebugMonad a` is used by the debugger to manage its internal state while evaluating an expression that has result type `a`.

```
type DebugMonad a
  = StateT DebuggerState (ErrorT (Maybe BugReport) IO) a
```

Values of type `BugReport` are used to report an erroneous program rule which is represented by two constructor terms, i.e., the call along with arguments and its result.

The result delivered by the debugger is lifted into monad `IO`, because the debugger has to interact with the user via IO actions while it evaluates the expression.

This monad is transformed by the monad transformer `ErrorT`, so that not only the result can be returned, but also the evaluation can be truncated reporting the location of an error (`Just bug`) or indicating that the user has aborted the debugging session (`Nothing`). As soon as a program error has been pinned down to a single program rule, the evaluation is truncated and that program rule is reported to the user.

Going one step further, this monad is transformed by the monad transformer `StateT` in order to let the debugger read and write its state while executing a program.

```
type Step a = StepMode -> DebugMonad a
```

`Step a` is the type of an evaluation step with result type `a`. Since the evaluation of an expression depends on its evaluation mode, an evaluation step consists of an expression of type `DebugMonad a` that is parameterized with an evaluation mode of type `StepMode`.

The function `eval` consumes one entry from the current oracle. Depending on the value of this entry, it either evaluates its argument and returns the result of this evaluation, or it refrains from evaluating its argument and returns `underscore` as a placeholder.

```
eval :: ShowTerm a => Step a -> Step a
eval a mode = do state <- get
                 let (orc, needed) = popBoolStack (oracle state)
                 put (state {oracle = orc})
```

```
                if needed then a mode
                          else return underscore
```

The following pair of functions is used to combine evaluation steps:

```
(>>>=) :: ShowTerm a => (Step a) -> (a -> Step b) -> Step b
a >>>= b = \ mode -> do a' <- eval a mode
                        b a' mode

return' :: ShowTerm a => a -> Step a
return' x = \ _ -> return x
```

At first, function `(>>>=)` calls `eval` with parameter `a` as its argument. Depending on the current entry of the oracle, it either evaluates this parameter and returns the result, or it returns a placeholder value `underscore`. Then the second argument is applied to the value returned by `eval`. Since this application is done by function `(>>=)` of type `DebugMonad`, managing the debugger state by `StateT` as well handling exceptions by `ErrorT` is done in background by the monad `DebugMonad`.

The function `return'` turns its argument into an evaluation step which returns this argument as a result when evaluated.

Assuming that all entries of the current oracle have value `True`, type `Step a`, together with functions `(>>>=)` and `return'`, form a monad.

For example, the types of the transformed versions of the functions in Figure 1 are:

```
length :: ShowTerm a => [a] -> Step Int
exp :: Step Int
fiblist :: Int -> Step [Int]
fib :: Int -> Step Int
```

In the original program `exp` is a value of type `Int`, but in the transformed program every evaluation takes place in the debug monad. The resulting function is an evaluation step which may return a value of type `Int` or a bug report. Similarly, the transformed version of `length` is still a function taking two arguments, but now it yields an evaluation step that has to be executed in the debug monad to retrieve its result.

The transformation also adds a function `main`, an action of type `IO ()`. This action contains the whole debugging session from loading the oracle from disk to debugging the program interactively. The details of this transformation will be explained in Section 3.6.

3.5 The Function `traceFunCall`

The function `traceFunCall` interfaces the instrumented program with the interactive debugger. Every top level declaration is augmented with a call to this function, which has the following type signature:

```
traceFunCall :: ShowTerm a => DebugMonad Term -> Step a -> Step a
```

The first argument contains an action to retrieve the term representation of the function call about to be evaluated, cf. Section 3.3. The second argument is the function body that has been lifted into the debugging monad as described in the previous section. The class context `ShowTerm a =>` makes sure that the result can be displayed to the user. There is a third argument hidden in the resulting type `Step a` as given in the type declaration above: the current evaluation mode of the debugger. Depending on the evaluation mode, `traceFunCall` shows one of the following behaviors:

Mode ***`StepCorrect`*** If a function call is rated as correct, all its sub computations are considered as correct, too, so that the components `skipped` and `unrated` of the global state do not contain entries for those calls. Therefore, `skipped` and `unrated` will not be changed while evaluation a sub expression with mode `Stepcorrect`.

Mode ***`StepBackground`*** The resulting value is calculated without uesr interaction. Since `unrated` might contain ratings for sub computations and those ratings shall be preserved, with mode `StepBackground` every call of `traceFunction` takes one rating from `unrated` and puts it onto `skipped`. Therefore at the end of a debugging cycle, the entries of `skipped` can be put back onto `unrated`, so that in the next debugging cycle those ratings will be reused and there is no need to re-state the correctness of the function calls that have already been rated.

There is one exception from this rule: If the entry that was taken from `unrated` is `True`, the current function call has been rated as correct in a former evaluation, and this function call is evaluated with mode `StepCorrect`.

Mode ***`StepInteractive`*** All interaction between the user and the debugger takes place with mode `StepInteractive`. In inspection mode, cf. Section 2.1, function `traceFunCall` first evaluates the function body with mode `StepBackground`. Then the function call is – preliminary – rated as correct by appending value `False` to the component `skipped` of the debugger state. After that the resulting value is displayed and the user is asked whether it is correct.

- Since for every function call that has been rated as as correct all its subfunction calls are considered as correct, too, their rating as well as the preliminary rating of the current function call are removed from `skipped` and from `unrated`. They are replaced by a single entry of value `True` that states the correctness of the whole subexpression.
- If the user has rated the resulting value as wrong, the debugging session will confine itself to searching the bug in the current expression. If it finds a bug in one of them, then in turn it restricts itself to searching the bug in that subexpression. If it finds no bug in any of its subexpressions, it is clear that the definition of the currently called function is erroreous, and the current function call will be returned as the result of the debugging session.

– If rating the current function call is skipped, the ratings that had already been present in `unrated` – and have been moved to `skipped` while evaluating the function call with mode `StepBackground` – are kept, so that they will be available in subsequent evaluation cycles.
– If the user moves to evaluating the subexpressions of the current function calls (*step into*), the body of the function currently called is evaluated with mode `StepInteractive`. The rating of the current expression, that has already been appended to `skipped`, will be kept, since it resembles the fact that the current function call is still unrated.

If the debugging tool is in step-by-step mode, cf. Section 2.2, `traceFunCall` will not calculate the result of the current function call until the user requests it. Instead it starts by displaying only the function call and giving the user an opportunity to move forward to rating its subexpressions without having to evaluate the whole function first.

We have now explained all the components introduced by the transformation and can now give show how, for example, function `fiblist` of Figure 1 is transformed:

```
fiblist ::  Int -> Step [Int]
fiblist x1 = traceFunCall (do sx1 <- showTerm x1
                              return (Term "fiblist" [sx1]))
                          (fib x1 >>>= \x4 ->
                           x1 + 1 >>>= \x2 ->
                           fiblist x2 >>>= \x3 ->
                           return' (x4 : x3))
```

3.6 Starting the Debugger

The functions `traceWithStepfile` and `traceProgram` are used to start a debugging session. The function `traceProgram` has the following type signature:

```
traceProgram :: ShowTerm a => Step a -> DebuggerState -> IO ()
```

As parameters it receives both an expression of type `Step a`, which represents the program to be debugged, and the state containing an oracle, which will guide the evaluation of this program.

The program will be repeatedly executed, and the evaluation is stopped when the location of a bug is found, when every function call is rated as correct, or when the user cancels the debugging session.

The function `traceWithStepfile` has the following type signature:

```
traceWithStepfile :: ShowTerm a => String -> Step a -> IO ()
```

It loads an oracle from a file whose name is given in the first parameter, and then it debugs the program given in the second parameter by calling `traceProgram`. The name of the file from which the oracle is loaded consists of the string given in the first argument followed by a suffix `".steps"`.

The expression to be evaluated by the debugger is included in function `main :: IO ()`. In addition to the transformed expression, the resulting declaration of `main` also contains a call to function `traceProgram`. The first parameter, which indicates the name of the oracle file, is derived from the name of the program file being transformed, and the second parameter is the transformed expression of type `Step a`, where `a` is the type of the expression to be evaluated.

For example, the function `main` added to the program in Figure 1 in order to evaluate the expression `exp` is:

```
main :: IO ()
main = traceWithStepfile "Example"
         (return (traceFunCall "main" []))
         exp
```

Since `main` has type `IO ()`, the transformed program can now be compiled by the Haskell compiler `ghc`. Also one can start a debugging session by first loading the transformed program into a Haskell interpreter and then calling `main`.

4 Conclusion and Future Work

We have presented the usage and implementation of a debugging tool utilizing the oracle technique developed in [1]. Up to now, the debugger features a declarative debugging mode as well as a step-by-step mode corresponding to a leftmost innermost evaluation strategy. In addition, a virtual I/O environment gives the user the opportunity to see side effects issued by the program. Up to now, this environment features console output and file access. An extension to other often used I/O actions like `IORef`s and sockets is straight forward. The main limitation of the approach as it is developed by now is the lack of treating run-time errors. Improving this is clearly important for debugging purposes.

Other room for improvement is of course adding to the list of debugging features. Many useful techniques are easy to integrate into the framework like spy points, trusted functions and remembering questions already asked. We plan to include some of the features described in [6] as well as those provided by HAT [2], as far as they fit into the framework.

A thorough benchmarking comparison with HAT [2] is also future work. First impressions are that tracing computations is about five times faster than HAT due to the fact that our approach needs much less space. The size of programs that HAT and our approach can handle in declarative debugging mode seem to be roughly the same. The size of programs manageable in step-by-step mode is only limited in the size of programs that can be traced and we have not yet seen a program that could be successfully executed but not be traced due to memory limitations.

References

1. B. Braßel, S. Fischer, M. Hanus, F. Huch, and G. Vidal. Lazy call-by-value evaluation. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP'07)*, 2007. To be published.
2. O. Chitil, C. Runciman, and M. Wallace. Freja, hat and hood – a comparative evaluation of three systems for tracing and debugging lazy functional programs. In *Proc. of the 12th International Workshop on Implementation of Functional Languages (IFL 2000)*, pages 176–193. Springer LNCS 2011, 2001.
3. Andy Gill. Debugging Haskell by observing intermediate datastructures. *Electronic Notes in Theoretical Computer Science*, 41(1), 2001.
4. H. Nilsson and J. Sparud. The Evaluation Dependence Tree as a Basis for Lazy Functional Debugging. *Automated Software Engineering*, 4(2):121–150, 1997.
5. B. Pope. Declarative Debugging with Buddha. In V. Vene and T. Uustalu, editors, *Advanced Functional Programming, 5th International School, AFP 2004*, volume 3622 of *Lecture Notes in Computer Science*, pages 273–308. Springer Verlag, September 2005.
6. Josep Silva. A comparative study of algorithmic debugging strategies. In Germán Puebla, editor, *LOPSTR*, volume 4407 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2006.
7. J. Sparud and C. Runciman. Tracing Lazy Functional Computations Using Redex Trails. In *Proc. of the 9th Int'l Symp. on Programming Languages, Implementations, Logics and Programs (PLILP'97)*, pages 291–308. Springer LNCS 1292, 1997.

Uniqueness Typing Simplified

Edsko de Vries[1]*, Rinus Plasmeijer[2], and David M Abrahamson[1]

[1] Trinity College Dublin, Ireland, {devriese,david}@cs.tcd.ie
[2] Radboud Universiteit Nijmegen, Netherlands, rinus@cs.ru.nl

Abstract. We present a uniqueness type system that is simpler than both Clean's uniqueness system and than the system we proposed previously. At the same time, our new type system is more expressive, is straightforward to implement or add to existing compilers, and can easily be extended with advanced features such as higher rank types and impredicativity. We have integrated this type system in *Morrow*, an experimental functional language with a very advanced type system that supports higher rank types and impredicativity.

1 Introduction to Uniqueness Typing

An important property of pure functional programming languages is *referential transparency*: the same expression used twice must yield the same result. This makes reasoning about programs easier, since we do not need to know global properties about the state of the program to deduce local properties about a small part of the program. Imperative languages, and even some functional languages, do not have this property. For example, in the following C fragment,

```
int f(FILE* file)
{
  int a = fgetc(file); // Read a character from 'file'
  int b = fgetc(file);
  return a + b;
}
```

it is understood that `a` and `b` can (and typically will) have different values, even though we are applying the same function (`fgetc`) to the same input (`file`). The problem is that even though the input is syntactically identical in both statements, the structure denoted by `file` is modified by each call to `fgetc` (the file pointer is advanced): `fgetc` has a side effect.

Nevertheless, even in a pure functional programming language, we would like to be able to read from files or perform other actions with side effects. Key to understanding how we can add a function such as `fgetc` without sacrificing referential transparency is noting that in the small C example above, there would be no problem with referential transparency *if there was only a single reference to `file`*: a side effect on a variable (`file`) is okay as long as that variable is never used again. Put another way, it is okay for a function to modify its input

* Supported by the Irish Research Council for Science, Engineering and Technology.

if the input is not *shared*. Referential transparency then trivially holds because the same expression (the same function applied to the same input) never occurs more than once.

The same function `f` implemented in a functional language using uniqueness typing gives

```
f file0 = let (a, file1) = fgetc file0
              (b, file2) = fgetc file1
          in (a + b, file2)
```

Rather that just returning the read character, `fgetc` returns a pair consisting of the read character *and a new file*, `file1`. Even though `file0` and `file1` point to the same file on disk, they are conceptually *and* syntactically different, and thus it is clear that a and b may have different values (because `fgetc` is applied to two different inputs). The uniqueness type system guarantees that `fgetc` is never applied to an argument which is used again (shared). For example, the type checker would complain if we had written

```
f file0 = let (a, file1) = fgetc file0
              (b, file2) = fgetc file0
          in (a + b, file0)
```

Sharing information is recorded as an attribute on the type of a term. This attribute is either $\bullet$ for unique (guaranteed not to be shared) or $\times$ for non-unique (may or may not be shared). For example, $\texttt{File}^{\bullet}$ is the type of files that are guaranteed not to be shared, and the type of `fgetc` might be

$$\texttt{fgetc} :: \texttt{File}^{\bullet} \xrightarrow{\times} (\texttt{Char}^{\times}, \texttt{File}^{u})^{v}$$

The uniqueness attribute u on the file in the result of `fgetc` means that it is up to the programmer to decide if she wants to treat it as unique or shared. This is discussed in more detail in Section 7.

2 Contributions of This Paper

The type system we present in this paper is based on that of the programming language *Clean* [1, 2]. However, Clean's type system has a number of drawbacks.

- Types and attributes are regarded as two different entities, which limits expressiveness and makes retrofitting uniqueness typing to existing compilers more difficult.
- Types in Clean often involve implications between uniqueness attributes. For example, the function `const` has type

  $$\texttt{const} :: t^{u} \xrightarrow{\times} s^{v} \xrightarrow{w} t^{u}, [w \leq u]$$
  ```
  const x y = x
  ```

  The constraint $[w \leq u]$ denotes that if u is unique, then w must be unique (u implies w)[3]. The need for this constraint will be explained in Section 4,

[3] Perhaps the choice of the symbol $\leq$ is unfortunate. In logic $a \leq b$ denotes a implies b, whereas here $u \leq v$ denotes v implies u. Usage here conforms to *Clean* conventions.

but the presence of constraints complicates the work of the type checker (the heart of the typechecker is a unification algorithm, and unification cannot deal with inequalities) and makes extending the type system to support modern features such as arbitrary rank types difficult.
- Clean distinguishes between non-unique terms, unique terms, and *necessarily unique* terms that are never allowed to become non-unique. Moreover, Clean's type system has a subtyping relation between unique and non-unique terms. Both these features make the type system unnecessarily complicated.

In this paper, we make the following contributions.

- Section 3 shows that we can regard uniqueness attributes as type constructors of a special kind, rather than regarding types and uniqueness attributes as two different syntactic categories. If we do that, we gain expressive power in the type system, types become more readable, and both the presentation and the implementation of uniqueness typing is greatly simplified.
- Section 5 shows how we can recode inequalities as equalities (and thus removing the need for constraints) if we allow for arbitrary boolean expressions as uniqueness attributes. This makes it much easier to extend the type system with advanced features, and enables the use of unification to solve relations between attributes.
- Section 7 shows that by distinguishing between non-unique and necessarily unique (and no notion of "unique, but not necessarily unique"), we avoid the need for subtyping. We argued a similar point in a previous paper [3], where we distinguished between non-unique and unique instead (and no notion of necessarily unique). Unfortunately, that approach requires a second uniqueness attribute on the function arrow, which offsets the advantage of removing subtyping somewhat. Our new approach does not have this disadvantage.
- We describe our implementation of uniqueness typing in *Morrow* in Section 8. Morrow is an experimental functional language developed by Daan Leijen, and has an advanced type system supporting, amongst other things, higher rank types and impredicativity. Adding support for uniqueness typing to Morrow required only a few changes to the compiler. This provides strong evidence for our claim that retrofitting uniqueness typing to an existing compiler, and extending uniqueness typing with advanced features such as higher rank types and impredicativity, becomes straightforward with the techniques in this paper. As far as the authors are aware, this is also the first substructural type system that has these features.

3 Attributes are Types

In this section, we show that we can regard types and attributes as one syntactic category. This simplifies both the presentation and implementation of a uniqueness type system, makes types more readable, and increases the expressive power of the type system.

Up to now, we have regarded types and attributes as two very different entities. Although it is certainly possible to formalize a uniqueness type system this way, it makes things more complicated than necessary. If types and attributes are completely different, then we need both type variables and attribute variables, and we need to be able to quantify ($\forall$) over both type variables and attribute variables. The status of arguments to algebraic datatypes (such as `List a`) is also not very clear: are they types, attributes, or do they correspond to a type *with* an attribute?

Perhaps counter-intuitively, everything becomes much clearer when we regard types and attributes as a single syntactic category. Thus, `Int` is a type, `Bool` is a type, but so is $\bullet$ (unique) and $\times$ (non-unique). We regard $\texttt{Int}^{\bullet}$ as the application of a special type constructor (denoted infix by the superscript operation) to two arguments, `Int` and $\bullet$. To make that slightly more explicit, we can give the operator a name, `Attr`, and write $\texttt{Int}^{\bullet}$ as `Attr Int` $\bullet$.

There are no values of type $\bullet$, nor are there values of type `Int`, because `Int` is lacking a uniqueness attribute. There are however values of type $\texttt{Int}^{\times}$; 1, 2, and 1024 are all examples.

Types that do not classify values are nothing new. For example, they arise in Haskell as *type constructors*; a well-known example is `[]` (the list type constructor). We can make precise which types classify values and which do not by introducing a kind system [4]. Kinds can be regarded as the "types of types". By definition, the kind of those types that classify values is denoted by $*$. In Haskell, we have `Int :: *`, `Bool :: *`, but `[] :: * → *`; it takes a type as argument, and returns a type as result.

But here is where we deviate from Haskell. Since we have already said that we do not regard `Int` as a type classifying values in our type system, its kind cannot be $*$. Instead, we introduce two new kinds, $\mathcal{T}$ and $\mathcal{U}$, classifying "base types" and uniqueness attributes, respectively. So, we have `Int` $::\ \mathcal{T}$, `Bool` $::\ \mathcal{T}$, $\bullet :: \mathcal{U}$ and $\times :: \mathcal{U}$. Since `Attr` combines a base type and an attribute into a type of kind $*$, *its* kind must be

$$\texttt{Attr} :: \mathcal{T} \to \mathcal{U} \to *$$

The kind language and some type constructors along with their kinds are listed in Figure 1. At this point it is useful to introduce the following convention.

> **(Syntactic convention.)** Type variables[4] of kind $\mathcal{T}$ will be denoted by t, s. Type variables of kind $\mathcal{U}$ will be denoted by u, v, and type variables of kind $*$ will be denoted by a, b.

We will discuss the consequences of treating uniqueness attributes as types of kind $\mathcal{U}$ for the implementation of a compiler in Section 8; here we will focus on the consequences for the programmer. The first consequence is that we can simplify the way we write types. We can write the type of the identity function as

[4] Strictly speaking, these are meta variables, not object language type variables. Our core language does not include universal quantification.

Kind language

κ	::=		kind
		$\mathcal{T}$	base type
		$\mathcal{U}$	uniqueness attribute
		$*$	base type together with a uniqueness attribute
		$\kappa_1 \to \kappa_2$	type constructors

Type constants

`Int`	::	$\mathcal{T}$	
`Bool`	::	$\mathcal{T}$	
$\to$	::	$* \to * \to \mathcal{T}$	function space
$\bullet$	::	$\mathcal{U}$	unique
$\times$	::	$\mathcal{U}$	non-unique
$\vee$	::	$\mathcal{U} \to \mathcal{U} \to \mathcal{U}$	disjunction
$\wedge$	::	$\mathcal{U} \to \mathcal{U} \to \mathcal{U}$	conjunction
$\neg$	::	$\mathcal{U} \to \mathcal{U}$	negation
`Attr`	::	$\mathcal{T} \to \mathcal{U} \to *$	combine a base type and attribute

Syntactic conventions

t^u	$\equiv$	`Attr` t u
$a \xrightarrow{u} b$	$\equiv$	`Attr` $(a \to b)$ u

Fig. 1. The kind language and some type constructors with their kinds

`id` :: $t^u \xrightarrow{\times} t^u$

(the attribute on the arrow will be explained in Section 4). But now we can also write it as

`id` :: $a \xrightarrow{\times} a$

In the first case we are using two type variables (t of kind $\mathcal{T}$ and u of kind $\mathcal{U}$), whereas in the second case we are using only a single type variable (a of kind $*$). Both types mean the same thing, but the second is easier to read than the first. If we adopt Clean's convention of denoting "non-unique" by the absence of an attribute, we can even write the type of `id` as

`id` :: $a \to a$

Although we will not discuss algebraic datatypes in much detail in this paper, this way of treating types and attributes also gives more expressive power when defining new datatypes. For example, we can define the following three datatypes:

```
newtype X a = X a
newtype Y t = Y t•
newtype Z u = Z Int^u
```

The first datatype `X` is parametrized by an attributed type (a type of kind $*$), the second by a base type (a type of kind $\mathcal{T}$), and the third by a uniqueness attribute (a type of kind $\mathcal{U}$). The kinds of `X`, `Y` and `Z` are therefore $* \to \mathcal{T}$, $\mathcal{T} \to \mathcal{T}$ and $\mathcal{U} \to \mathcal{T}$, respectively. The co-domain is $\mathcal{T}$ in all cases, because `X`

`Int`$^{\times}$ still lacks an attribute; `(X Int`$^{\times}$`)`$^{\bullet}$ on the other hand (which we can also write as `X`$^{\bullet}$ `Int`$^{\times}$) is a unique `X` containing a non-unique `Int`.

In Clean, we can only define the first of these three datatypes, so we have gained expressive power. What is more, although we have used syntactic conventions to give a visual clue about the kinds of the type variables, the kinds of these types can automatically be inferred by the kind checker, so the expressive power comes at no cost to the programmer.

4 Partial Application

Dealing correctly with partial application is probably the most subtle aspect of uniqueness typing. For example, this particular aspect is not dealt with correctly in a recent paper comparing usage analysis and uniqueness typing [5]. So before we delve into the details of the type system in the next section, we must look into this issue.

In Haskell, the function `const` is defined as follows.

```
const :: a → b → a
const x y = x
```

Temporarily ignoring the attributes on arrows, the type of `const` in a uniqueness type system is exactly the same. But now consider the following program:

```
dup x = (\f -> (f 1, f 2)) (const x)
```

What is the type of `dup`? Given the type of `const`, above, it would seem that the type of `dup` is

`dup` :: $a \to (a, a)$

But that cannot be correct, because this type of `dup` tells us that if we pass in a single unique a to `dup`, it will return a pair of two unique as, an impressive feat. However, the full type of `const` in our type system is

`const` :: $t^u \xrightarrow{\times} b \xrightarrow{u} t^u$

If you pass in a unique t, you get a *unique* function from b to t: a function that can only be used once. An alternative reading of this type is also possible: *if* you use a partial application of `const` more than once, the argument to `const` must be non-unique. The type of `dup` is therefore

`dup` :: $t^{\times} \xrightarrow{\times} (t^{\times}, t^{\times})^u$

In general, a function must be unique (and can therefore be applied only once) if it has any unique elements in its closure (the closure of a function is the environment that binds the free variables in the function body).

e	::=		expression
		$x^{\odot}$	variable (used once)
		$x^{\otimes}$	variable (used more than once)
		$\lambda x \cdot e$	abstraction
		$e\, e$	application
τ_k	::=		type
		c_k	constant
		$\tau_{k' \to k}\, \tau_{k'}$	(type) application

Fig. 2. Expression and type language for the core system

5 Removing Constraints

In this section we show that by allowing arbitrary boolean expressions[5] as uniqueness attributes, we can recode implications between uniqueness attributes as equalities. That makes it easier to extend the type system with advanced features such as arbitrary rank types, and simplifies the implementation. In fact, this change makes the uniqueness type system so similar to the standard Hindley/Milner type system that standard type inference algorithms can be applied. The only change required is in the unification algorithm, as we shall see in Section 6.

The expression language and type language are defined in Figure 2 (types have been indexed by kinds k). Both are almost entirely standard, except that we assume that a sharing analysis has annotated variable uses with $\odot$ or $\otimes$. A variable x marked as $x^{\odot}$ is used only once within its scope, but a variable marked as $x^{\otimes}$ is used more than once. The typing rules are listed in Figure 3. The typing relation takes the form

$$\Gamma \vdash e : \tau|_{fv}$$

which reads as "in environment Γ, expression e has type τ; the attributes on the types of the free variables in e are fv". We represent fv as a relation $Var \times Attribute$; it is used in rule Abs to determine whether a function needs to be unique (this is discussed in more detail in Section 4).

The rules are very similar to the standard Hindley/Milner type system, except that they maintain some extra information about uniqueness. The underlying base system is unchanged, so that uniqueness typing can be seen as an "add-on". In the rest of this section we will explain the rules in more detail.

5.1 Variables

We need to distinguish variables that are used once in their scope and variables that are used multiple times. The rule for variables that are used only once

[5] Although the typing rules only introduce disjunctions between uniqueness attributes, more complicated expressions can be introduced by boolean unification when unifying two boolean expressions.

$$\frac{}{\Gamma, x : t^u \vdash x^{\odot} : t^u|_{(x,u)}}\ \textsc{Var}^{\odot} \qquad \frac{}{\Gamma, x : t^{\times} \vdash x^{\otimes} : t^{\times}|_{(x,\times)}}\ \textsc{Var}^{\otimes}$$

$$\frac{\Gamma, x : a \vdash e : b|_{fv} \qquad fv' = {\lhd}_x\, fv}{\Gamma \vdash \lambda x \cdot e : a \xrightarrow{\bigvee fv'} b|_{fv'}}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash e_1 : a \xrightarrow{u} b|_{fv} \quad \Gamma \vdash e_2 : a|_{fv'}}{\Gamma \vdash e_1\, e_2 : b|_{fv \cup fv'}}\ \textsc{App}$$

Fig. 3. Typing rules for the core lambda calculus

($\textsc{Var}^{\odot}$) is identical to the normal Hindley/Milder rule, and we simply look up the type of the variable in the environment. Note that even when a variable is used only once, that does not automatically make its type unique. For example, there is only one use of `x` in the identity function:

```
id x = x⊙
```

but when a shared term is passed to `id`, it will still be shared when it is returned from `id`. On the other hand, if a variable is used more than once (rule $\textsc{Var}^{\otimes}$), its type must be non-unique (shared). This happens for example in `dup`:

```
dup x = (x⊗, x⊗)
```

Both rules also record the uniqueness attribute on the type of the variable.

5.2 Abstraction and Application

The rule for abstractions is the same as the Hindley/Milner rule, except that we must determine the values of the attribute on the arrow. As discussed in Section 4, a function must be unique if it has any unique elements in its closure. The closure of a function $\lambda x \cdot e$ consists of the free variables in the body e of the function, minus x. The attributes on the free variables in the body of the function are recorded in fv; using $fv' = {\lhd}_x\, fv$ (domain subtraction) to denote fv with x removed from the domain of fv, we use the disjunction $\bigvee fv'$ of all the attributes in the range of fv' as the uniqueness attribute on the arrow.

The rule for application is the normal one, except that we collect the free variables. The attribute on the arrow is ignored (we can apply both unique and shared functions).

5.3 Encoding Constraints

In general, we can always recode a type of the form

$$\ldots \square^u \quad \ldots \square^v \ldots, [u \leq v]$$

using a disjunction

$$\ldots \square^{u \vee v} \ldots \square^v \ldots$$

When v is unique, $u \vee v$ reduces to unique, but when v is non-unique, $u \vee v$ reduces to u (a free variable), so this faithfully models the implication. For example, in Clean the function `fst` that extracts the first element of a pair has the type

```
fst :: (a^u, b^v)^w → a^u, [w ≤ u]
fst (x, y) = x
```

which we can recode as

```
fst :: (a^u, b^v)^(w∨u) → a^u
```

However, in many cases we can do slightly better. For example, suppose the typing rule for pairs is

$$\frac{\Gamma \vdash e_1 : a|_{fv_1} \qquad \Gamma \vdash e_2 : b|_{fv_2}}{\Gamma \vdash (e_1, e_2) : (a, b)^u|_{fv_1 \cup fv_2}} \quad \textsc{Pair}$$

then for every derivation of $e :: (a, b)^\bullet$, there is also a derivation of $e :: (a, b)^\times$ (because the typing rule leaves the attribute on the pair free). That means that we can simplify the type of `fst` to

```
fst' :: (a^u, b^v)^u → a^u
```

The only pairs accepted by `fst` but rejected by `fst'` are unique pairs, but since the type checker will never infer a pair to be unique, that situation will never arise. We took advantage of the same principle in the rule for abstraction, where we recoded a type

$$\ldots \xrightarrow{u} \ldots, [u \leq v, u \leq w, \ldots]$$

as

$$\ldots \xrightarrow{v \vee w \vee \ldots} \ldots$$

This will force some functions to be non-unique which would otherwise be polymorphic in their uniqueness, but that cannot cause any type errors: the rule for function application ignores the uniqueness attribute on the function, and non-unique functions can be used multiple times.

6 Boolean Unification

One advantage of removing constraints from the type language is that standard inference algorithms (such as algorithm $\mathcal{W}$ [6]) can be applied without any modifications. The inference algorithm will depend on a unification algorithm, which must be modified to use boolean unification when unifying two terms of kind $\mathcal{U}$. The rest of this section explains how boolean unification works.

Suppose we have two terms g and h

$$g :: a^\bullet \xrightarrow{\times} \ldots \qquad h :: a^{u \vee v}$$

Should the application $g\,h$ be allowed? If so, we must be able to unify $u \vee v$ and $\bullet$. Of course, this equation has many solutions, for example

$$\begin{bmatrix} u \mapsto \bullet \\ v \mapsto v \end{bmatrix} \qquad \begin{bmatrix} u \mapsto u \\ v \mapsto \bullet \end{bmatrix} \qquad \begin{bmatrix} u \mapsto \bullet \\ v \mapsto \bullet \end{bmatrix}$$

(Recall that we treat attributes as boolean expressions.) Unfortunately none of the solutions listed above is most general, and it not obvious that the equation $u \vee v = \bullet$ even has a most general unifier, which means we would lose principal types. Fortunately, unification in a boolean algebra is unitary [7]. In other words, if a boolean equation has a solution, it has a most general solution. In the example, one most general solution is

$$\begin{bmatrix} u \mapsto u \\ v \mapsto v \vee \neg u \end{bmatrix}$$

There are two well-known algorithms for unification in a boolean algebra, known as Löwenheim's formula and successive variable elimination. For the core system from Section 5 either algorithm will work, but when arbitrary rank types are introduced and we need to use skolemization [8], only successive variable elimination is practical[6]. The description of successive variable elimination we give here combines the methods from [7] and [9]. Temporarily using the more common 0 for false (not unique) and 1 for true (unique), to unify two terms p and q of a boolean algebra it suffices to unify

$$t = (p \wedge \neg q) \vee (\neg p \wedge q) = 0$$

This is implemented by $\texttt{unify}_0$, shown in Figure 4, which gets a term t in a boolean algebra a and the list of free variables in t as input, and returns a substitution and the "consistency condition", which will be zero if unification succeeded.

7 On Subtyping

In this section we compare our approach to subtyping to that of Clean [2] and to that of our previous paper on the topic [3]. Readers not familiar with either of those two type systems can skip this section if they wish.

Consider again the function `dup`:

[6] Löwenheim's formula maps any unifier to a most general unifier, reducing the problem of finding an MGU to finding a specific unifier. For the two-element boolean algebra, that is very simple (just try all possible instantiations of the variables) but it is not so easy in the presence of skolem constants. Skolem constants introduce new elements into the boolean algebra, making it much more difficult to guess ground unifiers. For example, assuming that u_R and v_R are skolem constants, and w is a uniqueness variable, the equation $u_R \vee v_R \simeq w$ has an obvious solution $[w \mapsto u_R \vee v_R]$, but we can no longer guess this solution by instantiating all variables to either true ($\bullet$) or false ($\times$).

$$
\begin{aligned}
&\texttt{unify}_0 :: \texttt{BooleanAlgebra}\ a \Rightarrow [Var] \to a \to (\texttt{Subst}\ a, a)\\
&\texttt{unify}_0\ []\ t = ([], t)\\
&\texttt{unify}_0\ (x : xs)\ t = (s_t \cup s_e, cc)\\
&\quad \texttt{where}\\
&\qquad s_t = [x \mapsto s_e\, t_0 \vee (x \wedge s_e\, (\neg t_1))]\\
&\qquad (s_e, cc) = \texttt{unify}_0\ xs\ (t_0 \wedge t_1)\\
&\qquad t_0 = [x \mapsto 0]\ t\\
&\qquad t_1 = [x \mapsto 1]\ t
\end{aligned}
$$

Fig. 4. Boolean unification

$$\texttt{dup} :: t^{\times} \xrightarrow{\times} (t^{\times}, t^{\times})^{u}$$
```
dup x = (x, x)
```

In Clean, `dup` gets the same type, but that type is interpreted slightly differently. Clean's type system uses an explicit subtyping relation: a unique type is considered a subtype of a non-unique type. That is, we can pass in something that is unique (such as a unique `Array`) to a function that is expecting a non-unique type (such as `dup`).

The fact that a unique array can become non-unique is an important feature of a uniqueness type system. A non-unique array can no longer be updated, but can still be read from. However, adding subtyping to a type system leads to considerable additional complexity, especially when considering a contravariant/covariant system (such as Clean's). It becomes simpler when considering an invariant subtyping relation such as the one proposed in [5], but we feel that subtyping is not necessary at all.

In our previous paper, we argued that the type of `dup` should be

$$\texttt{dup} :: t^{u} \xrightarrow[\times]{u_f} (t^{\times}, t^{\times})^{v}$$

The (free) uniqueness variable on the t in the domain of the function indicates that we can pass unique or non-unique terms to `dup`. Since it is always possible to use a uniqueness variable in lieu of a non-unique attribute, an explicit subtyping relation is not necessary (this can be compared to the use of phantom types in *wxHaskell* to encode inheritance [10]).

But there is a catch. As we saw in Section 4, functions with unique elements in their closure must be unique, and must *remain* unique: they can only be applied once. In Clean, this is accomplished by regarding unique functions as *necessarily unique*, and the subtyping is adjusted to deal with this third notion of uniqueness: a necessarily unique type is *not* a subtype of a non-unique type. Hence, we cannot pass functions with unique elements in their closure to `dup`.

Unfortunately, when `dup` gets the type from our previous paper it *can* be used to duplicate functions with unique elements in their closure. To remedy that problem, we introduced a second uniqueness attribute on the function arrow,

indicating whether the function had any unique elements in its closure. The typing rule for application then enforced that functions with unique elements in their closure (second attribute) were unique (first attribute). That means that functions with unique elements in their closure can be duplicated, but once duplicated can no longer be applied. While this removed the need for subtyping, that advantage was offset by the additional complexity introduced by the second uniqueness attribute on arrows.

An important contribution of the current paper is the observation that we can avoid this additional complexity (without reintroducing an explicit subtyping relation) by disallowing sharing unique elements, *but giving the programmer the option of treating a term as unique or non-unique*. For example, a function that returns a new empty array should get the type

$\texttt{newArray} :: \texttt{Int} \xrightarrow{\times} \texttt{Array}^{u}$

rather than

$\texttt{newArray} :: \texttt{Int} \xrightarrow{\times} \texttt{Array}^{\bullet}$

Similarly, the function that resets all elements of an array back to zero should get the type

$\texttt{resetArray} :: \texttt{Array}^{\bullet} \xrightarrow{\times} \texttt{Array}^{u}$

rather than

$\texttt{resetArray} :: \texttt{Array}^{\bullet} \xrightarrow{\times} \texttt{Array}^{\bullet}$

An `Array` that is polymorphic in its uniqueness can be passed to `resetArray` as easily as it can be passed to `dup` (but of course, a shared array still cannot be passed to `resetArray`). If we are careful never to *return* a unique array from a function, we will always be able to share arrays. We still do not have an explicit subtyping relation but we get the same functionality: the subtyping is encoded in the type of `Array`, rather than in the type of `dup`.

Besides simplifying the type system (we no longer need the additional uniqueness attribute on arrows, nor do we need subtyping), we also gain expressive power. While for some types sharing makes sense (such as `Array`s), for others it does not. For example, many functions with side effects in Clean have a type such as

$\texttt{fun} :: \cdots \rightarrow (\texttt{World}^{\bullet} \rightarrow \texttt{World}^{\bullet})$

where the `World` is a token object representing the world state. It never makes sense to duplicate the world, and we can now enforce that by returning a unique `World` (rather than a `World` which is polymorphic in its uniqueness)[7].

[7] In a sense, Clean already has this expressiveness, as it has both the notions of unique and necessarily unique. However, only unique functions are regarded as necessarily unique; it is not possible to define a function that returns a necessarily unique `World`, for example.

8 Implementation in Morrow

We have integrated our type system in *Morrow*[8], an experimental functional language developed by Daan Leijen in Microsoft Research, Redmond. Morrow's type system is HMF [11], which is a Hindley/Milner-like type system that supports first class polymorphism (higher rank types and impredicativity). As such, it is an alternative to both Boxy Types [12] and MLF [13]. However, unlike boxy types, it is presented as a logical system which makes it more predictable, and at the same time it is much simpler than MLF.

HMF derives much of its simplicity from being invariant. Many higher rank type systems define a subsumption relation, which can be seen as a subtyping relation between type schemes. For example, $\forall ab.a \to b \to a$ is a "subtype" of $\forall a.a \to a \to a$ in the sense that whenever a function of the second type is expected, we can also pass in a function of the first. Since HMF is invariant, it does not use subtyping, which makes it a very natural fit with with the type system we propose[9].

As it turns out, the implementation of our type system in Morrow is agreeably straightforward. This provides strong evidence for our claim that adding uniqueness typing to an existing compiler, and more importantly, extending uniqueness typing with advanced features such as higher rank types and impredicativity, poses little difficulty when using the techniques from this paper. We outline the most important changes we had to make to Morrow:

- We implemented sharing analysis, annotating variables with information on how often they are used within their scope (once or more than once)
- We modified the rules for variables and abstraction, so that shared variables must be non-unique, and abstractions become unique when they have unique elements in their closure. To be able to do the latter, all the typing rules had to be adapted to return the *fv* structure from Section 5.
- Let bindings had to be adapted to remove the variables bound from *fv*. Moreover, the type of every binding in a recursive binding group must be non-unique (as is standard in a uniqueness type system [2]).
- Most of the work was in modifying the types of the built-in functions and the kinds of the built-in types, and adding the appropriate type constants (such as `Attr`) and kind constants ($\mathcal{T}, \mathcal{U}$). However, all of these changes were local and did not affect the rest of the type checker.
- Unification had be adapted to do boolean unification, as explained in Section 6. In addition, it is necessary to simplify boolean expressions, so that for example $a^{u \vee \times}$ is simplified to a^u. This is not really optional, because if no simplification is used, the boolean expressions can quickly get complicated. Fortunately, we can use an independent module for boolean unification and simplification. When unifying $a \simeq b$, it suffices to check the kinds of a and

[8] Our implementation can be downloaded from `http://www.cs.tcd.ie/~devriese`.

[9] Nevertheless, it is quite possible to integrate our type system in other type systems, too. For example, we have a prototype implementation of a variant on the type system of this paper that uses the arbitrary rank type system from [8].

b, and if they are $\mathcal{U}$, to call the boolean unification module. In Morrow, this is implemented as

```
unify t1 t2 | getKind t1 == kindU && getKind t2 == kindU
  = case BooleanAlgebra.try_unify t1 t2 of
      Nothing    -> unifyError NoMatchType
      Just subst -> return (subNew subst, ksubNull)
```

The details do not matter, but we want to emphasize that the boolean unification does not in any way complicate the unification algorithm of the type checker.

– Morrow uses System F (with pattern matching) as its typed internal language. Although the "attributes are types" approach of Section 3 means that the internal language does not need to change at all, Morrow also includes a System F type checker to ensure that the various phases of the compiler generate valid code. This type checker had be adapted in a similar way to the main type checker (alternatively, it would also have been possible not to include any uniqueness information in the System F types).

The most important point, perhaps, is that the majority of these changes were local (did not require any significant refactoring of the compiler), and none of the changes were very complicated. No new subtleties arise when scaling our core type system up to include advanced features.

9 Conclusions

By treating uniqueness attributes as types of a special kind $\mathcal{U}$, the presentation of a uniqueness type system is much simplified, types become more readable, and we gain expressiveness in the definition of algebraic datatypes. We can recode attribute inequalities (implications between uniqueness variables) as equalities if we allow for arbitrary boolean expressions as uniqueness attributes. This makes type inference easier (unification cannot deal with inequalities, but *can* deal with equalities between boolean expression using a well-known technique of boolean unification). Finally, no explicit subtyping relation is necessary when we are clever in the types we assign to functions: we require that unique terms must never be shared, but make sure that functions never return unique terms (but rather terms that are polymorphic in their uniqueness).

Taken together these observations lead to an expressive and yet very simple uniqueness type system, which can easily be extended with advanced features such as higher rank types and impredicativity. We have integrated our type system in Morrow, an experimental programming language with a very advanced type system. The implementation required only minor changes to the compiler, which provides strong evidence for our claim that retrofitting our type system to existing compilers is straightforward.

Acknowledgments

We thank Daan Leijen, John Gilbert and Wendy Verbruggen for their insightful and helpful comments on various drafts of this paper. Daan Leijen was also the first to question whether it would be possible to remove subtyping without making `dup` more polymorphic. We eventually realized that this is possible if we are careful when specifying the return type of functions (Section 7).

References

1. Barendsen, E., Smetsers, S.: Conventional and uniqueness typing in graph rewrite systems. Technical Report CSI-R9328, University of Nijmegen (December 1993)
2. Barendsen, E., Smetsers, S.: Uniqueness typing for functional languages with graph rewriting semantics. Mathematical Structures in Computer Science **6** (1996) 579–612
3. De Vries, E., Plasmeijer, R., Abrahamson, D.: Uniqueness typing redefined. In Horváth, Z., Zsók, V., Butterfield, A., eds.: Revised selected papers from IFL 2006, LNCS 4449. (2007)
4. Jones, M.P.: A system of constructor classes: overloading and implicit higher-order polymorphism. In: FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture, New York, NY, USA, ACM Press (1993) 52–61
5. Hage, J., Holdermans, S., Middelkoop, A.: A generic usage analysis with subeffect qualifiers. In: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming. (To appear).
6. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM Press (1982) 207–212
7. Baader, F., Niphow, T.: Term Rewriting and All That. Cambridge University Press (1998)
8. Peyton Jones, S., Vytiniotis, D., Weirich, S., Shields, M.: Practical type inference for arbitrary-rank types. Journal of Functional Programming **17**(1) (Jan 2007) 1–82
9. Brown, F.M.: Boolean Reasoning, The Logic of Boolean Equations. Dover Publications, Inc. (2003)
10. Leijen, D.: wxHaskell – a portable and concise GUI library for Haskell. In: ACM SIGPLAN Haskell Workshop (HW'04), ACM Press (September 2004)
11. Leijen, D.: HMF: Simple type inference for first-class polymorphism. Technical Report MSR-TR-2007-118, Microsoft Research, Redmond
12. Vytiniotis, D., Weirich, S., Peyton Jones, S.: Boxy types: inference for higher-rank types and impredicativity. In: ICFP '06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming, New York, NY, USA, ACM Press (2006) 251–262
13. Botlan, D.L., Rémy, D.: ML^F: raising ML to the power of System F. In: ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming, New York, NY, USA, ACM Press (2003) 27–38

Tabular Expressions and Total Functional Programming

Baltasar Trancón y Widemann and David Lorge Parnas

Software Quality Research Laboratory (SQRL)
University of Limerick, Ireland
http://www.sqrl.ie

Abstract. Tabular expressions are a multidimensional structured notation for complex mathematical definitions of relations or functions. In order to create tools to check and evaluate tabular expressions, we have investigated functional programming as an implementation paradigm that reflects mathematical semantics faithfully. We explain why and how the restriction to total functions improves the semantic correspondence substantially, and describe the basic design and capabilities of our total functional programming tools for tabular expressions.

1 Introduction

1.1 Context

Our research group is developing methods of producing practical reference documentation for software products and components. Our document contents are defined by a relational model in which each document is required to be a representation of a specified relation. In effect, we are using mathematical descriptions of relations to provide specifications and descriptions of programs written in conventional programming languages.

Key to making these documents readable is a multidimensional form of expressions, which we call *tabular expressions* or just *tables*. These parse complex expressions into arrays of simpler expressions allowing readers to "look up" the information that they seek without understanding the whole expression.

Tools that check and evaluate these expressions would be very useful when these methods are applied and we are looking for effective and efficient implementations of such tools.

1.2 This Work

This paper reports on our experiences with applying the functional programming paradigm to the construction of tools for tabular expressions. Functional programming is a natural choice because

1. The tasks of checking and evaluating tabular expressions are typical examples of side-effect-free processing and interpretation of structured data.

2. The formal semantic model of tabular expressions, as presented to some degree in the earlier work [1] and more generically in the forthcoming [2], is given largely in terms of functions.
3. The intended application of these expressions is software documentation using a relational model [3] but the relations are described by their characteristic predicate and those are always functional.

Our intent is to give a reference implementation of the formal model that is not only executable, but also mirrors the intended semantics and the model's theoretical properties as faithfully as possible. We shall argue that our goals can almost, but not quite, be achieved by using a universal functional language, and describe an alternative.

1.3 Related Work

This is not the first time that the relation between tabular expressions and functional programming has been noticed or exploited. In [4], Kahl presents an inductive approach to tables of certain regular types that is compositional in table content and semantics at the same time. He provides an implementation of table constructors and inductive interpretation in Haskell, and corresponding formal proofs in the proof system Isabelle. Because of the restricted set of constructors, the resulting theory is compact and elegant.

In contrast, our current work is intended to implement the more generic table model of [2], that allows *all* constructs of a mathematical base language to be used freely in content and semantics of tables. This paper discusses the requirements of such a generic view, and presents preliminary results from the approach we have taken.

2 Example Tabular Expression

We shall use a simple tabular expression taken from [5] as the running example for explaining the basic usage of tables and the services we expect from an evaluation tool.

$PwrCnd(Prev : bool; Power, K_{in}, K_{out} : real) : bool =$

$Power \leq K_{out}$	$K_{out} < Power < K_{in}$	$Power \geq K_{in}$
false	*Prev*	*true*

Table 1. Power Conditioning (Specification)

The tabular expression depicted as table 1 is a small, but real example.[1] It specifies a family of control functions of a nuclear reactor shutdown system. As some of the status monitoring logic is only applicable when the reactor is operating near its maximum output power level, they need to be "conditioned in" (activated) above a certain power level, and "conditioned out" (deactivated) below. To avoid *jitter* (many changes separated by very short intervals), hysteresis is simulated by setting the threshold for conditioning in (K_{in}) slightly higher than that for conditioning out (K_{out}). In between, the previous state (*Prev*) is maintained. A graph illustrating some change of power over time is depicted in figure 1. The relevant state transitions and their effects are marked.

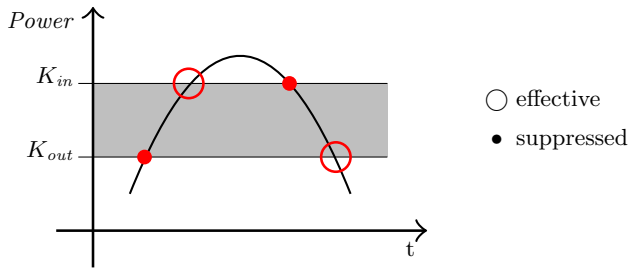


Fig. 1. Power Conditioning (Example Graph)

2.1 Meaning of a Tabular Expression

The concrete syntax for this table is deceivingly straightforward; for multidimensional or irregular tables, there may not be such an obvious graphical representation. Hence the mathematical table model only represents the abstract syntax of the *content* of the table as an indexed set (aka family or map) of *grids*. Each grid is in turn an indexed set of *cells*, each of which contains a (conventional or nested tabular) expression. A table *type* complements the content to make the tabular expression semantically self-contained. The table type, that may be shared by many similar tables, comprises

1. an *evaluation term*, i.e., an algorithm for evaluating the table's content, depending on a valuation of free variables,
2. a *restriction predicate*, i.e., a well-formedness condition that a table's content must satisfy for the evaluation algorithm to be applicable.

The table type is an integral part of the table expression. One can consider it as an instance of dynamic typing, or as semantically rigorous meta-data.

[1] Although we have chosen the simplest possible real example to illustrate these expressions, many much more complex tables were used in the inspection of the Darlington Nuclear Power Generation Station described in [5].

The given example table is an instance of the one-dimensional *normal function table* type:

1. It contains two grids of three cells each.
   (a) The upper grid is a *header* grid that contains predicate expressions.
   (b) The lower grid is a *main* grid that contains value expression (also of type *bool* in this case).
2. To evaluate the table, choose an index to the header grid, such that
   (a) the selected predicate expression evaluates to *true*,
   (b) then evaluate only the corresponding cell of the main grid.

   (For more than one dimension, one index for each header grid would be chosen independently.)
3. The table is well-formed, if
   (a) the main grid has the same indexes as the header grid (for higher dimensions, the index set of the main grid must be the Cartesian product of the index sets of the header grids), and
   (b) each header grid partitions the set of possible variable valuations, i.e., exactly one is found to be true in any case.

See [1, 2] for more exact definitions of the normal function table type and other types of tabular expressions.

2.2 Tool Requirements

We expect an evaluation tool to enable us to

1. evaluate a tabular expression for a given variable valuation, by applying the evaluation term specified by the table's type to its content,
2. check the restriction predicate, distinguishing two parts for practical reasons:
   (a) clauses that do not depend on variable values (called the *static* restriction), to be checked universally for the table's content,
   (b) clauses that do depend on variable values (called the *dynamic* restriction), to be checked specifically for the table's content and a given variable valuation,

all with reasonable efficiency.

We do not expect an evaluation tool to support checking dynamic restrictions universally for all possible variable valuations. This is a task for a theorem proving system, and may involve much more complex computations. In [5], based on earlier work [6], the authors show how a flaw in the specified table has been discovered by the automatic theorem prover PVS: the header cells are only a partition of the valuation space, if the (intuitive, but unstated) assertion $K_{out} < K_{in}$ holds. Otherwise, the first and third columns overlap, and the table does not specify a function.

3 The Logic Behind Tabular Expressions

If tabular expressions are to be used for describing real problems, they must be able to deal with partial functions. Partial functions can lead to undefined expressions, and there are many ways to handle undefinedness in logic, e.g., by having three or more truth values.

The meaning of partial functions in tabular expressions here is the one defined in [7]. It was chosen to give the shortest possible expressions in the table cells. It can be summarized as follows:

1. There is a special value $(*)$, distinct from all proper values of interest. This value is assigned to the application of a partial function to arguments outside its domain.
2. The domain of partial functions never contains $(*)$. This implies that the result of a partial function is $(*)$ whenever one of its arguments is $(*)$, i.e., functions are strict. A partial function is being treated as if it were a *total* function whose range includes $(*)$.
3. Predicates are treated differently from functions. A predicate is simply *false* if any of its arguments is $(*)$. Consequently, the truth value of a formula is always *true* or *false*, but never $(*)$. I.e., predicates are non-strict.

Note that $(=)$ is also a predicate, so (by the third rule) the seemingly trivially true predicate expression $f(x) = f(x)$ is *not* true if x is outside the domain of f. On the other hand, the equation $f(x) = y$ is logically equivalent to $F(x, y)$ where F is the characteristic predicate for f. It has been argued in [7] and later work that this interpretation of partial functions is particularly concise and useful for writing software descriptions and specifications in the tabular notation.

This semantic decision has consequences for the construction of an effective universal evaluation algorithm for tabular expressions. The intuitively appealing representation of $(*)$ by the element $(\perp)$ of standard domain-theoretic semantics does not work as intended: Since $(\perp)$ is also assigned to expressions that cannot be evaluated effectively, e.g., a nonterminating recursive function application, writing a program that would evaluate any predicate of the formalism becomes as hard as solving the halting problem, i.e., impossible without restrictions.

1. The *pragmatic* approach is to use a universal language to implement the model, accepting some semantic deviations. It is impossible to preclude undetermined predicate expressions in this case; so the responsibility is placed on the programmer to find the appropriate termination arguments.
2. The *rigorous* approach is to use a restricted language with the "right" semantics to implement the model. If we have to decide whether an expression evaluates to $(*)$, it has to be an proper value in a calculus of total functions. The advantage of this approach is that properties of the implementation are closely related to (and not much more complex to prove than) properties of the formal model. The price is that one has to obey the restrictions of the implementation language.

4 Total Functional Programming

In [8], Turner expresses similar, albeit more fundamental concerns regarding the relation of universal functional programming calculi and mathematical functions:

> *The driving idea of functional programming is to make programming more closely related to mathematics. [...] The existing model of functional programming [...] is compromised to a greater extent than is commonly recognized by the presence of partial functions.*

He strives for a language that abolishes partial functions, but retains as much as possible of the notational ease of Miranda or Haskell.

A quite different approach to total functions is taken by total function calculi in the style of Martin-Löf's type theory or Coquand's *calculus of constructions*. These are closely connected to higher-order logic (via the Curry-Howard isomorphism), a fact that is exploited in constructive proof systems like Coq.

We have chosen a "middle road", employing a rigorous explicit type system like the latter, but focusing on computation (rather than logic) like the former. The result is FCN[2], the design and implementation of a practical programming language for pure total functions. Like other total languages, it is characterized by the absence of general recursion: The syntax forbids recursive definitions, and the type system forbids fixpoint operators.

5 Functional Programming Techniques Applied

Limited space prohibits the detailed description of the FCN language. The following subsections can provide only brief examples of how our requirements have been mapped successfully onto features of the functional paradigm.

5.1 Partial Functions

The logical rules concerning partial functions and predicates can be implemented in a completely explicit way using a simple *error monad* [9].

1. For each type A, a *dubious* type $A?$ is defined to contain one additional element:
$$\texttt{type}\ A? = A + *$$
Readers familiar with Haskell will easily recognize this as the *Maybe* functor.
2. A partial function $f : A \rightharpoonup B$ is represented as a total function $f' : A \to B?$. Consider another partial function $g : B \rightharpoonup C$, totalized as $g' : B \to C?$. The composition $g' \circ f'$ is not type-correct, so a canonical transformation
$$\mathit{bind} : \forall\, B, C.\, (B \to C?) \to (B? \to C?)$$

[2] Functional Core Notation

is inserted. It satisfies the strictness law

$$bind(g')(x) = \begin{cases} * & x = * \\ g'(x) & x \neq * \end{cases}$$

such that the composition of total functions $bind(g') \circ f'$ correctly implements the composition of partial functions $g \circ f$. The operation *bind* can be extended to the functorial operation of (?) to deal with total functions:

$$lift : \forall\, B, C.\, (B \to C) \to (B? \to C?)$$

3. For predicates, a different canonical transformation

$$prim : \forall\, A.\, (A \to bool) \to (A? \to bool)$$

is used to compose them with partial functions. It satisfies the non-strictness law

$$prim(p)(x) = \begin{cases} false & x = * \\ p(x) & x \neq * \end{cases}$$

Apart from reflecting the intended semantics precisely, this approach has several additional benefits:

1. Algebraic simplification laws that do not hold for the original implicit notation are restored. These include the aforementioned $f(x) = f(x) \iff true$, as well as general β-reduction.
2. A single boolean-valued function can be re-used to define both a partial function and a total predicate, by exchanging *lift* and *prim*.
3. There is no ambiguity which symbols are primitive predicates and thus subject to the non-strictness rule: they are explicitly qualified with *prim*.

5.2 Cells and Variables

Tabular expressions are used to define functions and relations, hence they are likely to contain (free) variables. In a language with first-order functions, the grid structure of a table and the functionality of individual cells can be separated cleanly by *closure conversion*, aka *lambda lifting*: The open expression in each cell is turned into a function of the table's variables, which can then be stored in a data structure. In the given example table, the effect is that the phrase

$$\lambda\, Prev : bool; Power, K_{in}, K_{out} : real. \cdots$$

is prepended to each cell expression. A variable valuation then takes the form of an argument vector that is applied uniformly to all cells of the table.

5.3 Table Interpretation

Table content is structured as grids and cells, organized in list-like collections. All typical access operations required to define a table type, such as the normal function table type described above, are easily defined in terms of primitive recursion, applying a recursion operator (aka *fold*) to a non-recursive step function. So far, we have not encountered any problem in this specific domain that would have required recursion support from the language.

6 Tool Support

Programming in FCN is supported by tools, most notably a parser, type checker and compiler. All of these are written in Java. The compiler produces Java code that runs on the JVM, together with a small runtime library. Figure 2 shows a compiled version of the running example table. It is controlled by a GUI that is derived directly from the function signature, and will be generated automatically by a future version of the tools.

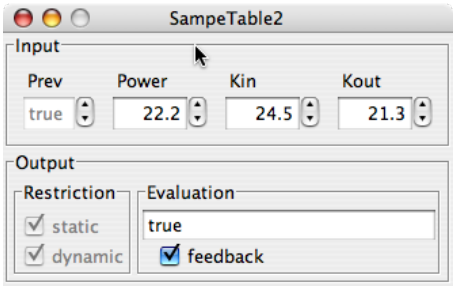


Fig. 2. Power Conditioning (Simulation Screenshot)

A library of about 1000 lines of FCN code defines the ubiquitous basic types and operations: booleans, natural numbers, tuples, lists monads, etc. The powerful type system of the calculus of constructions allows the definition of all of these in terms of the λ operator only; no additional primitive constructs are needed or used.

A second level of library code, about 500 lines of FCN, defines the table model in terms of standard functional data structures and operations, as well as some common table types, including the multidimensional normal function table type used in the example. This library will be extended in the future to support other table types.

A tabular expression is simply data structured according to the model, containing functions at the cell level. Evaluation and restriction checking are completely generic operations, because all semantic information is explicit in the "type" part of the table data.

Tabular expressions that describe software behaviour can be "animated" with compiled FCN code to produce simulations, test oracles or prototypes.

6.1 Total Functions and Theorem Proving

There is ongoing work [10] to represent the formal model of tabular expressions as a theory in the proving system PVS. This would complement the services of the evaluation tools by allowing to prove properties of tables universally for a class of variable valuations.

Because of the close similarity between the calculi of total functions and the higher-order logic of PVS, and because of the explicit treatment of partiality issues in the FCN implementation, large parts of the implementation's design carry over to PVS directly. The FCN type checker has proved a valuable tool for quick consistency checking in the design process, helping to keep consistency proof obligations in the PVS theory tractable.

7 Conclusion

The work described in this paper is an experimental use of functional programming in the creation of software engineering tools. The approach has provided a formulation of the mathematical model of tabular expressions that can reflect semantics precisely, but is also directly and effectively executable. The strict type system has proven a valuable consistency check. The features of functional programming that are supposed to support abstraction and reuse in functional programming, namely parametric polymorphism and higher-order functions, have found essential use, e.g., as primitive recursion operators and monadic liftings.

We have also found that the notion of total functional programming, that is looked upon with some scepticism by most of the community, is quite feasible for this specific application. The absence of general recursion does not impede the construction or interpretation of tables unduly. The pervasive use of recursion operators even encourages a point-free programming style.[3]

The absence of infinite evaluation branches greatly simplifies the choice of, and encourages experiments with, evaluation strategies. This applies both at run-time, where no semantic difference between eager and lazy evaluation exists, and also at compile-time to program specialization by partial evaluation. Both cases are investigated in ongoing work.

Finally, we have found that a calculus of total functions greatly reduces the impedance mismatch between the implementation of a formalism and its formalization in a proof system, making it attractive for projects that involve both evaluation and verification.

[3] An obvious benefit from the viewpoint of the functional programmer, but of questionable merit for the software engineer.

Acknowledgements

Thanks to Dennis Peters, Mark Lawford and other SQRL members for helpful discussions.

References

1. Parnas, D.L.: Tabular representation of relations. CRL Report 260, McMaster University (1992)
2. Balaban, A., Bane, D., Jin, Y., Parnas, D.L.: Mathematical model of tabular expressions. SQRL draft (2007) available for review.
3. Parnas, D.L., Madey, J., Iglewski, M.: Precise documentation of well-structured programs. IEEE Trans. Softw. Eng. **20**(12) (1994) 948–976
4. Kahl, W.: Compositional syntax and semantics of tables. SQRL Report 15, McMaster University (2003)
5. Lawford, M., Froebel, P., Moum, G.: Application of tabular methods to the specification and verification of a nuclear reactor shutdown system. Submitted to Formal Methods in System Design (2000)
6. Jing, M.: A table checking tool. SERG Report 384, McMaster University (2000)
7. Parnas, D.L.: Predicate logic for software engineering. IEEE Trans. Softw. Eng. **19**(9) (1993) 856–862
8. Turner, D.A.: Total functional programming. Universal Computer Science **10**(7) (2004) 751–768
9. Spivey, M.: A functional theory of exceptions. Sci. Comput. Program. **14**(1) (1990) 25–42
10. Peters, D.K., Lawford, M., Trancón y Widemann, B.: An IDE for software development using tabular expressions. In: CASCON 2007. (2007) to appear.

Positive Supercompilation for a higher order call-by-value language

Peter A. Jonsson and Johan Nordlander

{pj,nordland}@csee.ltu.se
Luleå University of Technology

Abstract. Previous deforestation and supercompilation algorithms may introduce termination when applied to call-by-value programs. This hides looping bugs from the programmer, and changes the behaviour of a program depending on whether it is optimized or not. We present a supercompilation algorithm for a higher-order call-by-value language that preserves termination properties. This algorithm utilizes strictness information for deciding whether to substitute or not and compares favorably with previous call-by-name transformations.

1 Introduction

Intermediate lists in functional programs allow the programmer to write clear and concise programs, but carry a cost at run time since list cells need to be both allocated and garbage collected. Both deforestation [19] and supercompilation [14] are automatic program transformations that remove many of these intermediate structures. In a call-by-value context these transformations are unsound, and might hide looping bugs from the programmer. Consider the program

$$(\lambda x.y)\,\bot.$$

Applying Wadler's deforestation algorithm [19] to the program will result in y, which is sound under call-by-name or call-by-need. The non-termination under call-by-value in the original program has been removed, and hence the meaning of the program has been altered by the transformation. Removal of intermediate structures in a strict language is still desirable though, perhaps even more than in a lazy language since the entire intermediate structure has to stay alive during the entire computation.

We show how to construct a meaning-preserving supercompilation scheme for call-by-value languages. It might seem like such a result should be easily obtainable from a call-by-name algorithm by simply delaying beta-reduction until every function argument has been specialized to a value. However, it turns out that this strategy misses even simple opportunities to remove intermediate structures. The explanation is that eager specialization of function arguments risks destroying *fold* opportunities that might otherwise appear, something which may even prohibit complexity improvements to the resulting program.

The novelty of our supercompilation algorithm is that it concentrates all call-by-value dependencies to a single rule that relies on the result from a separate *strictness analysis* for correct behavior. In effect, our algorithm delays transformation of function arguments past inlining, much like a call-by-name scheme does, although only as far as is legal with respect to call-by-value semantics. The result is an algorithm that is able to improve a wide range of illustrative examples like the existing algorithms do, but without the risk of introducing superficial termination. The specific contributions of our work are:

- We provide an algorithm for positive supercompilation including folding, for a strict and pure higher-order functional language (Section 4).
- Section 5 outlines extensions to improve transformational power.
- We prove that a restricted version of our algorithm preserve call-by-value semantics (Section 6).

We start out with some examples in Section 2 to give the reader an intuitive feel of how the algorithm behaves. Our language of study is defined in Section 3 right before the technical contributions are presented.

2 Examples

We call our transformation algorithm $\mathcal{D}$ and let $\mathcal{D}[\![\, e]\!]$ denote the result of transforming an expression e. The transformation takes an expression and a set of global definitions as input, and returns a new expression together with a new set of global definitions.

The inherent conflict between having call-by-value semantics and delaying evaluation of arguments as long as possible is the core of the problem we face when trying to construct an equally powerful call-by-value deforestation algorithm as the previous call-by-name ones. Consider a slightly contrived example, $\mathcal{D}[\![\, \mathit{skipFst}\ \bot\ \mathit{True}\,]\!]$. Under call-by-name this evaluates to 1. However, this result is not possible to reach under call-by-value semantics, since there is obvious non-termination in the original code snippet. The call-by-value semantics force worse performance; the best transformation result that still preserves semantics is $\mathbf{let}\ x\ =\ \bot\ \mathbf{in}\ 1$.

$$
\begin{array}{lcl}
\mathit{skipFst}\ x\ \mathit{cond} & = & \mathbf{case}\ \mathit{cond}\ \mathbf{of}\ \{\mathit{True}\ \rightarrow\ 1;\ \mathit{False}\ \rightarrow\ 2\} \\
\mathit{append}\ \mathit{xs}\ \mathit{ys} & = & \mathbf{case}\ \mathit{xs}\ \mathbf{of}\ \{[]\ \rightarrow\ \mathit{ys}; (x : \mathit{xs})\ \rightarrow\ x : \mathit{append}\ \mathit{xs}\ \mathit{ys}\}
\end{array}
$$

Wadler [19] uses the example $\mathit{append}\ (\mathit{append}\ \mathit{xs}\ \mathit{ys})\ \mathit{zs}$ and shows that his deforestation algorithm transforms the program so that its complexity is reduced from $2|xs| + |ys|$ to $|xs| + |ys|$, thereby saving one traversal of the first list.

If we naïvely change Wadler's algorithm to call-by-value semantics by eagerly attempting to transform arguments before attacking the body, we do not achieve this improvement in complexity. An example from a fictional driving algorithm that attacks arguments first is:

$\mathcal{D}[\![\ append\ (append\ xs'\ ys')\ zs'\ ]\!]$

(Drive $append\ xs'\ ys'$ in the context $append\ []\ zs'$)

$$= \mathcal{D}[\![\ \textbf{case}\ xs'\ \textbf{of}\ \begin{array}{l}[] \rightarrow\ append\ ys'\ zs' \\ (x : xs) \rightarrow\ append\ (x : append\ xs\ ys')\ zs'\end{array}]\!]$$

(Drive each branch, we focus on the (*x:xs*) case)

$$= \mathcal{D}[\![\ append\ (x : append\ xs\ ys')\ zs'\ ]\!]$$

(The expression has a previously seen expression embedded in it.
$x : append\ xs\ ys'$ is extracted for separate driving)

$$= \mathcal{D}[\![\ x : append\ xs\ ys'\ ]\!]$$
$$= \mathcal{D}[\![\ x :\ \textbf{case}\ xs\ \textbf{of}\ \begin{array}{l}[] \rightarrow\ ys' \\ (x' : xs') \rightarrow\ x' : append\ xs'\ ys'\ ]\!]\end{array}$$

(A renaming of a previous expression in the (x':xs') branch)

The end result from this driving is:

$$\mathcal{D}[\![\ append\ (append\ xs'\ ys')\ zs'\ ]\!] = \begin{array}{l}\textbf{case}\ xs'\ \textbf{of} \\ \quad [] \rightarrow\ h_1\ ys'\ zs' \\ \quad (x : xs) \rightarrow\ h_1\ (h_2\ x\ xs\ ys')\ zs'\end{array}$$

$$\begin{array}{l} h_1\ xs\ ys\ \ = \textbf{case}\ xs\ \textbf{of}\ \{[] \rightarrow\ ys; (x' : xs') \rightarrow\ x' : h_1\ xs'\ ys\} \\ h_2\ x\ xs\ ys = x : \textbf{case}\ xs\ \textbf{of}\ \{[] \rightarrow\ ys; (x' : xs') \rightarrow\ h_2\ x'\ xs'\ ys\}\end{array}$$

The intermediate structure in the input is still there after the transformation, and the complexity remains at $2|xs| + |ys|$!

However, doing the exact opposite — that is, carefully delaying transformation of arguments to a function past inlining of its body — actually leads to the same result as Wadler obtains after transforming *append* (*append xs ys*) *zs*.

Notice that the primitive operations ranged over by $\oplus$ in this language are supposed to be strict with regards to both arguments. This stands in contrast to ordinary functions, which can be inlined and partially evaluated for a couple of steps even if the arguments are unknown. Our algorithm leaves a primitive operation in place if any of its arguments fails to specialize to a primitive value.

If we had a perfect strictness analysis and could decide whether an arbitrary expression will terminate or not, the only difference in results between our algorithm and a call-by-name counterpart would be for the non-terminating cases. In practice, we have to settle for an approximation, such as the simple analysis defined in Figure 3. One may speculate whether the transformations thus missed will have adverse effects on the usefulness of our algorithm in practice. We believe

we have seen clear indications that this is not the case, and that crucial factor instead is the ability to inline function bodies irrespective of whether arguments are values or not.

We claim that our algorithm compares favorably with previous call-by-name transformations, and proceed with demonstrating the transformation of common examples. The results are equal to those of Wadler [19]. Our first example is transformation of *sum* (*map square ys*). The functions used in the examples are defined as:

$$\begin{aligned} &\mathit{square}\ x\ =\ x * x \\ &\mathit{map}\ f\ \mathit{xs} = \mathbf{case}\ \mathit{xs}\ \mathbf{of}\ \{[]\ \rightarrow\ []\mathrm{;}\ (x' : \mathit{xs}')\ \rightarrow\ (f\ x') : (\mathit{map}\ f\ \mathit{xs}')\ \} \\ &\mathit{sum}\ \mathit{xs}\ \ \ = \mathbf{case}\ \mathit{xs}\ \mathbf{of}\ \{[]\ \rightarrow\ 0\mathrm{;}\ (x' : \mathit{xs}')\ \rightarrow\ x'\ +\ \mathit{sum}\ \mathit{xs}'\ \} \end{aligned}$$

We start our transformation by allocating a new fresh function name (h_0) to this expression, inlining the body of *sum* and substituting *map square ys* into the body of *sum*:

$$\mathcal{D}[\![\mathbf{case}\ (\mathit{map}\ \mathit{square}\ \mathit{ys})\ \mathbf{of}\ \{[]\ \rightarrow\ 0\mathrm{;}\ (x' : \mathit{xs}')\ \rightarrow\ x'\ +\ \mathit{sum}\ \mathit{xs}'\ \}]\!].$$

After inlining *map* and substituting the arguments into the body the result becomes:

$$\begin{aligned} &\mathcal{D}[\![\ \mathbf{case}\ (\mathbf{case}\ \mathit{ys}\ \mathbf{of}\ \{[]\ \rightarrow\ []\mathrm{;}\ (x' : \mathit{xs}')\ \rightarrow\ (\mathit{square}\ x') : (\mathit{map}\ \mathit{square}\ \mathit{xs}')\})\ \mathbf{of} \\ &\qquad [] \rightarrow 0 \\ &\qquad (x' : \mathit{xs}')\ \rightarrow\ x'\ +\ \mathit{sum}\ \mathit{xs}'\ ]\!] \end{aligned}$$

We duplicate the outer case in each of the inner case's branches, using the expression in the branches as head of that case-statement. Continuing the transformation on each branch with ordinary reduction steps yields:

$$\mathbf{case}\ \mathit{ys}\ \mathbf{of}\ \{[]\ \rightarrow\ 0\mathrm{;}\ (x' : \mathit{xs}')\ \rightarrow\ \mathcal{D}[\![\ \mathit{square}\ x'\ +\ \mathit{sum}\ (\mathit{map}\ \mathit{square}\ \mathit{xs}')\ ]\!]\})$$

Now inline the body of the first square and observe that the second argument to (+) is similar to the expression we started with. A most specific generalization (msg) between the starting expression and the current expression is calculated, effectively splitting the expression in several parts. These expressions are transformed separately. We replace the second parameter to (+) with $h_0\ \mathit{xs}'$. The result of our transformation is $h_0\ \mathit{ys}$, with h_0 defined as:

$$\begin{aligned} h_0\ \mathit{ys} = \ &\mathbf{case}\ \mathit{ys}\ \mathbf{of} \\ &\qquad [] \rightarrow 0 \\ &\qquad (x' : \mathit{xs}')\ \rightarrow\ x' * x'\ +\ h_0\ \mathit{xs}' \end{aligned}$$

This new function only traverses its input once, and no intermediate structures are created. If the expression *sum* (*map square xs*) or a renaming thereof is detected elsewhere in the input, a call to h_0 will be inserted there instead.

The following examples are due to Ohori and Sasano [8]. We need the following new function definitions:

$$
\begin{array}{lll}
mapsq\ xs & = \textbf{case}\ xs\ \textbf{of}\ \{\ [\ ] & \rightarrow\ [];\ (x' : xs') \rightarrow\ (x' * x') : (mapsq\ xs')\ \} \\
f\ xs & = \textbf{case}\ xs\ \textbf{of}\ \{\ [\ ] & \rightarrow\ [];\ (x' : xs') \rightarrow\ (2 * x') : (g\ xs') \\
g\ xs & = \textbf{case}\ xs\ \textbf{of}\ \{\ [\ ] & \rightarrow\ [];\ (x' : xs') \rightarrow\ (3 * x') : (f\ xs')
\end{array}
$$

Evaluating $\mathcal{D}[\![\ mapsq\ (mapsq\ xs)]\!]$ will inline the outer *mapsq*, substitute the argument in the function body and inline the inner call to *mapsq*:

$$
\begin{array}{l}
\mathcal{D}[\![\ \textbf{case}\ (\textbf{case}\ xs\ \textbf{of}\ \{[\ ] \rightarrow\ [];\ (x' : xs') \rightarrow\ (x' * x') : (mapsq\ xs')\})\ \textbf{of} \\
\quad [\ ] \rightarrow\ [] \\
\quad (x' : xs') \rightarrow\ (x' * x') : (mapsq\ xs')\ ]\!]
\end{array}
$$

As previously, we duplicate the outer case in each of the inner case's branches, using the expression in the branches as head of that case-statement. Continuing the transformation on each branch by ordinary reduction steps yields:

$$
\textbf{case}\ xs\ \textbf{of}\ \{[] \rightarrow\ [];(x' : xs') \rightarrow\ \mathcal{D}[\![\ (x' * x' * x' * x') : (mapsq\ (mapsq\ xs'))\ ]\!]\ \}
$$

This will fold in a few steps, and the final result of our transformation is $h_1\ xs$, with the new residual function h_1 that only traverses its input once:

$$
h_1\ xs = \textbf{case}\ xs\ \textbf{of}\ \{[\ ] \rightarrow\ [];\ (x' : xs') \rightarrow\ ((x' * x') * (x' * x')) : (h_1\ xs'))\ ]\!]\ \}
$$

For an example of transforming mutually recursive functions, consider the transformation $\mathcal{D}[\![\ sum\ (f\ xs)]\!]$. Inlining the body of *sum*, substituting its arguments in the function body and inlining the body of f yields:

$$
\begin{array}{l}
\mathcal{D}[\![\ \textbf{case}\ (\textbf{case}\ xs\ \textbf{of}\ \{[\ ] \rightarrow\ [];\ (x' : xs') \rightarrow\ (2 * x') : (g\ xs')\})\ \textbf{of} \\
\quad [\ ] \rightarrow\ 0 \\
\quad (x' : xs') \rightarrow\ x' + sum\ xs'\ ]\!]
\end{array}
$$

Moving down the outer case into each branch, performing reductions to end up with:

$$
\textbf{case}\ xs\ \textbf{of}\ \{[\ ] \rightarrow\ 0;\ (x' : xs') \rightarrow\ \mathcal{D}[\![\ (2 * x') + sum\ (g\ xs')\ ]\!]\ \}
$$

We notice that unlike in previous examples, $sum\ (g\ xs')$ is not similar to what we started transforming. For space reasons, we focus on the transformation of the expression in the last branch, $\mathcal{D}[\![(2 * x') + sum\ (g\ xs')\ ]\!]$, while keeping the functions already seen in mind. We inline the body of *sum*, perform the substitution of its arguments and inline the body of g:

$$
\begin{array}{l}
\mathcal{D}[\![\ (2 * x') + \textbf{case}\ (\textbf{case}\ xs'\ \textbf{of}\ \{[\ ] \rightarrow\ [];\ (x'' : xs'') \rightarrow\ (3 * x'') : (f\ xs'')\})\ \textbf{of} \\
\qquad\qquad [\ ] \rightarrow\ 0 \\
\qquad\qquad (x' : xs') \rightarrow\ x' + sum\ xs'\ ]\!]
\end{array}
$$

We now move down both $(2 * x')$ and the outer case into each branch, and perform reductions:

$$
\begin{array}{l}
\textbf{case}\ xs'\ \textbf{of} \\
\quad [\ ] \rightarrow\ (2 * x') + 0 \\
\quad (x'' : xs'') \rightarrow\ \mathcal{D}[\![\ (2 * x') + (3 * x'') + sum\ (f\ xs'')\ ]\!]
\end{array}
$$

We notice a familiar expression in $sum\ (f\ xs'')$, and fold when reaching it. Adding it all together gives a new function h_2:

$$\begin{array}{l} h_2\ xs = \mathbf{case}\ xs\ \mathbf{of} \\ \qquad [] \to 0 \\ \qquad (x' : xs') \to \mathbf{case}\ xs'\ \mathbf{of} \\ \qquad\qquad [] \to (2 * x') + 0 \\ \qquad\qquad (x'' : xs'') \to (2 * x') + (3 * x'') + h_2\ xs'' \end{array}$$

Kort [4] studied a ray-tracer written in Haskell, and identified a critical function in the innermost loop of a matrix multiplication, called *vecDot*:

$$vecDot\ xs\ ys = sum\ (zipWith\ (*)\ xs\ ys)$$

This is simplified by our positive supercompiler to:

$$\begin{array}{l} vecDot\ xs\ ys = h_1\ xs\ ys \\ h_1\ xs\ ys \quad = \mathbf{case}\ xs\ \mathbf{of} \\ \qquad (x' : xs') \to \mathbf{case}\ ys\ \mathbf{of} \\ \qquad\qquad (y' : ys') \to x' * y' + h_1\ xs'\ ys' \\ \qquad\qquad \_ \to 0 \\ \qquad \_ \to 0 \end{array}$$

The intermediate list between *sum* and *zipWith* is transformed away, and the complexity is reduced from $2|xs| + |ys|$ to $|xs| + |ys|$ (since this is matrix multiplication $|xs| = |ys|$).

3 Language

Our language of study is a strict, higher-order, functional language with let-bindings and case-expressions. Its syntax for expressions, values and patterns is:

$$\begin{array}{lll} e, f & ::= & n \mid x \mid g \mid f\,\overline{e} \mid \lambda\overline{x}.e \mid k\,\overline{e} \mid e_1 \oplus e_2 \mid \mathbf{let}\,x = f\,\mathbf{in}\,e \\ & \mid & \mathbf{case}\,e\,\mathbf{of}\,\{p_i \to e_i\} \\ p & ::= & n \mid k\,\overline{x} \\ v & ::= & n \mid \lambda\overline{x}.e \mid k\,\overline{v} \end{array}$$

We let constructor symbols be denoted by k. Let g range over a set $\mathcal{G}$ of global definitions whose right-hand sides are all values. Recursion is only allowed at the top level but this restriction is not particularly burdensome – if a local function needs to be recursive, it can always be lambda-lifted [3] to the top level.

The language contains integer values n and arithmetic operations $\oplus$, although these meta-variables can preferably be understood as ranging over primitive values in general and arbitrary operations on these. We let $+$ denote the semantic meaning of $\oplus$.

All functions have a specific arity and all applications must be saturated; hence $\lambda x.map\ (+1)\ x$ is legal whereas $map\ (+1)$ is not. We abbreviate a list of expressions $e_1 \ldots e_n$ as $\overline{e}$, and a list of variables $x_1 \ldots x_n$ as $\overline{x}$.

Reduction contexts

$$\mathcal{E} ::= [\,] \mid \mathcal{E}\,\overline{e} \mid (\lambda\overline{x}.e)\,\overline{\mathcal{E}} \mid k\,\overline{\mathcal{E}} \mid \mathcal{E} \oplus e \mid n \oplus \mathcal{E} \mid \mathbf{let}\, x = \mathcal{E}\, \mathbf{in}\, e \mid \mathbf{case}\ \mathcal{E}\ \mathbf{of}\ \{p_i \to e_i\}$$

Evaluation relation

$$\begin{array}{lll}
\mathcal{E}\langle g\rangle & \mapsto \mathcal{E}\langle v\rangle, & \text{if } (g = v) \in \mathcal{G} \quad \text{(Global)} \\
\mathcal{E}\langle(\lambda\overline{x}.e)\,\overline{v}\rangle & \mapsto \mathcal{E}\langle[\overline{v}/\overline{x}]e\rangle & \quad \text{(App)} \\
\mathcal{E}\langle\mathbf{let}\, x = v\, \mathbf{in}\, e\rangle & \mapsto \mathcal{E}\langle[v/x]e\rangle & \quad \text{(Let)} \\
\mathcal{E}\langle\mathbf{case}\, k\,\overline{v}\ \mathbf{of}\ \{k_i\,\overline{x}_i \to e_i\}\rangle & \mapsto \mathcal{E}\langle[\overline{v}/\overline{x}_j]e_j\rangle, & \text{if } k = k_j \quad \text{(KCase)} \\
\mathcal{E}\langle\mathbf{case}\, n\ \mathbf{of}\ \{n_i \to e_i\}\rangle & \mapsto \mathcal{E}\langle e_j\rangle, & \text{if } n = n_j \quad \text{(NCase)} \\
\mathcal{E}\langle n_1 \oplus n_2\rangle & \mapsto \mathcal{E}\langle n\rangle, & \text{if } n = n_1 + n_2 \quad \text{(Arith)}
\end{array}$$

Fig. 1. Reduction semantics

A program is an expression with no free variables except those defined in $\mathcal{G}$. The intended operational semantics is given in Figure 1, where $[\overline{e}/\overline{x}]e'$ is the capture-free substitution of expressions $\overline{e}$ for variables $\overline{x}$ in e'.

A reduction context $\mathcal{E}$ is a term containing a single hole $[\,]$, which indicates the next expression to be reduced. The expression $\mathcal{E}\langle e\rangle$ is the term obtained by replacing the hole in $\mathcal{E}$ with e. $\overline{\mathcal{E}}$ denotes a list of terms with just a single hole, evaluated from left to right. We use $\equiv$ to denote equality up to renaming of variables.

If a variable appears no more than once in a term, that term is said to be *linear* with respect to that variable. Like Wadler [19], we extend the definition slightly for linear **case** terms: no variable may appear in both the selector and a branch, although a variable may appear in more than one branch. The definition of *append* is linear, although *ys* appears in both branches.

4 Higher Order Positive Supercompilation

Our supercompiler is defined as a set of rewrite rules that pattern-match on expressions. This algorithm is called the *driving* algorithm, and is defined in Figure 2. Two additional parameters appear as subscripts to the rewrite rules: a memoization list ρ and a driving context $\mathcal{R}$. The memoization list holds information about expressions already traversed and is explained more in detail in Section 4.1. The driving context $\mathcal{R}$ is smaller than $\mathcal{E}$, and is defined as follows:

$$\mathcal{R} ::= [\,] \mid \mathcal{R}\,\overline{e} \mid \mathbf{case}\,\mathcal{R}\ \mathbf{of}\ \{p_i \to e_i\} \mid \mathcal{R} \oplus e \mid e \oplus \mathcal{R}$$

Interestingly this definition coincides with the evaluation contexts for a call-by-name language. The reason our algorithm still preserves a call-by-value semantics is that beta-reduction (rule R8) results in a let-binding, whose further specialization in rule R11 depends on whether the body expression f is strict in the bound variable x or not.

In principle, an expression e is strict with regards to a variable x if it eventually evaluates x; in other words, if $e \mapsto \ldots \mapsto \mathcal{E}\langle x\rangle$. Such information is in

$$
\begin{array}{lll}
\mathcal{D}[\![n]\!]_{\mathcal{R},\rho} & = \mathcal{R}\langle n\rangle & \text{(R1)}\\
\mathcal{D}[\![k\,\overline{e}]\!]_{[],\rho} & = k\,\mathcal{D}[\![\overline{e}]\!]_{[],\rho} & \text{(R2)}\\
\mathcal{D}[\![x\,\overline{e}]\!]_{\mathcal{R},\rho} & = \mathcal{R}\langle x\,\mathcal{D}[\![\overline{e}]\!]_{[],\rho}\rangle & \text{(R3)}\\
\mathcal{D}[\![\lambda\overline{x}.e]\!]_{[],\rho} & = (\lambda\overline{x}.\mathcal{D}[\![e]\!]_{[],\rho}) & \text{(R4)}\\
\mathcal{D}[\![n_1 \oplus n_2]\!]_{\mathcal{R},\rho} & = \mathcal{D}[\![\mathcal{R}\langle n\rangle]\!]_{[],\rho}, \text{ where } n = n_1 + n_2 & \text{(R5)}\\
\mathcal{D}[\![e_1 \oplus e_2]\!]_{\mathcal{R},\rho} & = \mathcal{D}[\![e_1]\!]_{[],\rho} \oplus \mathcal{D}[\![e_2]\!]_{[],\rho}, \text{ if } e_1 \oplus e_2 = a & \text{(R6)}\\
 & \quad \mathcal{D}[\![e_2]\!]_{\mathcal{R}\langle e_1 \oplus [] \rangle,\rho}, \text{ if } e_1 = n \text{ or } e_1 = a & \\
 & \quad \mathcal{D}[\![e_1]\!]_{\mathcal{R}\langle [] \oplus e_2 \rangle,\rho}, \text{ otherwise} & \\
\mathcal{D}[\![g\,\overline{e}]\!]_{\mathcal{R},\rho} & = \mathcal{D}_{app}[\![g\,\overline{e}]\!]_{\mathcal{R},\rho} & \text{(R7)}\\
\mathcal{D}[\![(\lambda\overline{x}.f)\,\overline{e}]\!]_{\mathcal{R},\rho} & = \mathcal{D}[\![\mathbf{let}\,\overline{x} = \overline{e}\,\mathbf{in}\,f]\!]_{\mathcal{R},\rho} & \text{(R8)}\\
\mathcal{D}[\![e\,\overline{e}]\!]_{\mathcal{R},\rho} & = \mathcal{D}[\![e]\!]_{\mathcal{R}\langle [] \,\overline{e}\rangle,\rho} & \text{(R9)}\\
\mathcal{D}[\![\mathbf{let}\,x = v\,\mathbf{in}\,f]\!]_{\mathcal{R},\rho} & = \mathcal{D}[\![\mathcal{R}\langle [v/x]f\rangle]\!]_{[],\rho} & \text{(R10)}\\
\mathcal{D}[\![\mathbf{let}\,x = e\,\mathbf{in}\,f]\!]_{\mathcal{R},\rho} & = \mathcal{D}[\![\mathcal{R}\langle [e/x]f\rangle]\!]_{[],\rho}, \text{ if } e = g|x \text{ or } x \in \mathit{strict}(f) & \text{(R11)}\\
 & \quad \mathbf{let}\,x = \mathcal{D}[\![e]\!]_{[],\rho}\,\mathbf{in}\,\mathcal{D}[\![\mathcal{R}\langle f\rangle]\!]_{[],\rho}, \text{ otherwise} & \\
\mathcal{D}[\![\mathbf{case}\,x\,\mathbf{of}\,\{p_i \to e_i\}]\!]_{\mathcal{R},\rho} & = \mathbf{case}\,x\,\mathbf{of}\,\{p_i \to \mathcal{D}[\![\mathcal{R}\langle [p_i/x]e_i\rangle]\!]_{[],\rho}\} & \text{(R12)}\\
\mathcal{D}[\![\mathbf{case}\,k_j\,\overline{e}\,\mathbf{of}\,\{k_i\,\overline{x}_i \to e_i\}]\!]_{\mathcal{R},\rho} & = \mathcal{D}[\![\mathbf{let}\,\overline{x}_j = \overline{e}\,\mathbf{in}\,e_j]\!]_{\mathcal{R},\rho} & \text{(R13)}\\
\mathcal{D}[\![\mathbf{case}\,n_j\,\mathbf{of}\,\{n_i \to e_i\}]\!]_{\mathcal{R},\rho} & = \mathcal{D}[\![\mathcal{R}\langle e_j\rangle]\!]_{[],\rho} & \text{(R14)}\\
\mathcal{D}[\![\mathbf{case}\,a\,\mathbf{of}\,\{p_i \to e_i\}]\!]_{\mathcal{R},\rho} & = \mathbf{case}\,\mathcal{D}[\![a]\!]_{[],\rho}\,\mathbf{of}\,\{p_i \to \mathcal{D}[\![\mathcal{R}\langle e_i\rangle]\!]_{[],\rho}\}, & \text{(R15)}\\
\mathcal{D}[\![\mathbf{case}\,e\,\mathbf{of}\,\{p_i \to e_i\}]\!]_{\mathcal{R},\rho} & = \mathcal{D}[\![e]\!]_{\mathcal{R}\langle \mathbf{case}\,[]\,\mathbf{of}\,\{p_i \to e_i\}\rangle,\rho} & \text{(R16)}\\
\mathcal{D}[\![\overline{e}]\!]_{[],\rho} & = \mathcal{D}[\![e_1]\!]_{[],\rho}, \ldots, \mathcal{D}[\![e_n]\!]_{[],\rho} & \text{(R17)}
\end{array}
$$

Fig. 2. Driving algorithm

general not computable, although call-by-value semantics allows for reasonably tight approximations. One such approximation is given in Figure 3, where the strict variables of an expression e are defined as all free variables of e except those that only appear under a lambda or not inside all branches of a case.

$$
\begin{array}{ll}
\mathit{strict}(x) & = \{x\}\\
\mathit{strict}(n) & = \emptyset\\
\mathit{strict}(g) & = \emptyset\\
\mathit{strict}(k\,\overline{e}) & = \mathit{strict}(\overline{e})\\
\mathit{strict}(\lambda\overline{x}.e) & = \emptyset\\
\mathit{strict}(f\,\overline{e}) & = \mathit{strict}(f) \cup \mathit{strict}(\overline{e})\\
\mathit{strict}(\mathbf{let}\,x = e\,\mathbf{in}\,f) & = \mathit{strict}(e) \cup (\mathit{strict}(f) \backslash \{x\})\\
\mathit{strict}(\mathbf{case}\,e\,\mathbf{of}\,\{p_i \to e_i\}) & = \mathit{strict}(e) \cup (\bigcap(\mathit{strict}(e_i) \backslash \mathit{fv}(p_i))\\
\mathit{strict}(e_1 \oplus e_2) & = \mathit{strict}(e_1) \cup \mathit{strict}(e_2)
\end{array}
$$

Fig. 3. The strict variables of an expression

There is an ordering between rules; i.e., all rules must be tried in the order they appear. Rules R9 and R16 are the default fallback cases which extend the given driving context $\mathcal{R}$ and zoom in on the next expression to be driven. Meta-variable a in rules R6 and R15 stands for an "annoying" expression; i.e., an expression that would be further reducible were it not for a free variable getting

in the way. The grammar for annoying expressions is as follows:

$$a ::= x \mid n \oplus a \mid a \oplus n \mid a \oplus a \mid a\,\overline{e}$$

It is important to note that we allow empty argument vectors for all rules. Further, we assume that rule R7 matches the largest possible redex. The first alternative for rule R6 will only be selected if $\mathcal{R}$ is the empty context since rule R15 catches any $\oplus$s that are in the head of a case statement.

Some expressions should be handled differently depending on context. If a constructor application appears in an empty context, there is not much we can do but to drive the argument expressions (rule R2). On the other hand - if the application occurs at the head of a case expression, we may choose a branch on basis of the constructor and leave the arguments unevaluated in the hope of finding fold opportunities further down the syntax tree (rule R13).

The argumentation is analogous for lambda abstractions: if there is a surrounding context we perform a beta reduction, otherwise we drive its body.

The algorithm only performs substitutions of expressions in one rule (R11), hence there is only one possible source of code duplication. Duplicating code forces evaluation of the same expression multiple times, in addition to the increase in code size.

If duplication must be avoided, restrict substitution in R11 to the cases where the body of f is linear with respect to the bound variable x. However, this issue is a bit more complicated, as it might sometimes be beneficial to duplicate code that enables further transformations. The current algorithm has no means of finding a suitable trade-off, so it ignores the problem and always performs substitution when strictness so allows. Obtaining a more refined behavior in this respect is left for future work.

4.1 Application Rule

In the driving algorithm rule R7 refers to $\mathcal{D}_{app}[\![\,]\!]$, defined in Figure 4. This algorithm destructively updates a set *defs* with new function definitions as a side-effect. The input definitions are also accessed as the implicit parameter $\mathcal{G}$. $\mathcal{D}_{app}[\![\,]\!]$ can be inlined in the definition of the driving algorithm, it is merely given a separate name for improved clarity of the presentation.

$$\begin{array}{lll}
\mathcal{D}_{app}[\![g\,\overline{e}]\!]_{\mathcal{R},\rho} = h'\,\overline{x}, & & \text{if } \exists (t,h') \in \rho \,.\, t \equiv \mathcal{R}\langle g\,\overline{e}\rangle \\
\qquad\qquad \theta_2'\mathcal{D}[\![t_g]\!]_{[],\rho}, & & \text{if } \exists (t,h') \in \rho \,.\, t \trianglelefteq \mathcal{R}\langle g\,\overline{e}\rangle \text{ and } t_g \neq x \\
\qquad\qquad \theta_3\mathcal{D}[\![\mathcal{R}\langle y\rangle]\!]_{[],\rho}, & & \text{if } \exists (t,h') \in \rho \,.\, t \trianglelefteq \mathcal{R}\langle g\,\overline{e}\rangle \text{ and } t_g = x \\
\qquad\qquad h\,\overline{x},\ \mathit{defs} := \mathit{defs} \cup (h, \lambda\overline{x}.e'), & & \text{if } h \in fv(e') \\
\qquad\qquad e', & & \text{otherwise} \\
\text{where } (g = v) \in \mathcal{G},\ e' = \mathcal{D}[\![\mathcal{R}\langle v\,\overline{e}\rangle]\!]_{[],\rho'},\ (t_g, \theta_1, \theta_2) = msg(t, \mathcal{R}\langle g\,\overline{e}\rangle) & & \\
\qquad \overline{x} = \mathrm{fv}(\mathcal{R}\langle g\,\overline{e}\rangle),\ \rho' = \rho \cup (\mathcal{R}\langle g\,\overline{e}\rangle, h),\ h, y \text{ and } \overline{x}' \text{ fresh}, & & \\
\qquad \theta_2' = \mathcal{D}[\![\theta_2]\!]_{[],\rho},\ \theta_3 = [\mathcal{D}[\![g\,\overline{x}']\!]_{[],\rho}/y, \mathcal{D}[\![\overline{e}]\!]_{[],\rho}/\overline{x}'] & &
\end{array}$$

Fig. 4. Driving applications

Care needs to be taken to ensure that recursive functions are not inlined forever. The driving algorithm keeps a record of previously seen applications in the memoization list ρ; whenever it detects an expression that is equivalent (up to alpha conversion) to a previous expression, the algorithm creates a new recursive function h_n for some n. Whenever such an expression is encountered again, a call to h_n is inserted. This is not sufficient to guarantee termination of the algorithm, but the mechanism is crucial for the complexity improvements mentioned in Section 2.

To ensure termination, we use the homeomorphic embedding relation $\trianglelefteq$ to define a predicate called "the whistle". The intuition is that when $e \trianglelefteq f$, f contains all subterms of e, possibly embedded in other terms. For any infinite sequence $e_0, e_1, \ldots$ there exists i and j such that $i < j$ and $e_i \trianglelefteq e_j$. This condition is sufficient to ensure termination.

We need a definition of uniform terms analogous to the work by Sørensen and Glück [13], with some small adjustments specific to our language.

Definition 1 (Uniform terms). *Let s range over a the set $G \cup X \cup K \cup \{\mathbf{caseof}, \mathbf{letin}\}$, and let $\mathbf{caseof}(\overline{e})$ ($\mathbf{letin}(\overline{e})$) denote a case (let) expression containing subexpressions $\overline{e}$. The set of small terms T is the smallest set of arity respecting symbol applications $s(\overline{e})$.*

Definition 2 (Homeomorphic embedding). *Define $\trianglelefteq$ as the smallest relation on T satisfying:*

$$x \trianglelefteq y, n_1 \trianglelefteq n_2, \frac{e \trianglelefteq f_i \text{ for some } i}{e \trianglelefteq s(f_1, \ldots, f_n)}, \frac{e_1 \trianglelefteq f_1, \ldots, e_n \trianglelefteq f_n}{s(e_1, \ldots, e_n) \trianglelefteq s(f_1, \ldots, f_n)}$$

Whenever the whistle blows, our algorithm splits the current input expression into strictly smaller terms that are driven separately in the empty context. We split the expression by calculating the most specific generalization of the input and the embedded expression. The most specific generalization entails the smallest possible loss of knowledge, and is defined as:

Definition 3 (Most specific generalization).

- *An* instance *of a term* e *is a term of the form θe for some substitution θ.*
- *A* generalization *of two terms* e *and* f *is a triple $(t_g, \theta_1, \theta_2)$, where θ_1, θ_2 are substitutions such that $\theta_1 t_g \equiv e$ and $\theta_2 t_g \equiv f$.*
- *A* most specific generalization *(msg) of two terms* e *and* f *is a generalization $(t_g, \theta_1, \theta_2)$ such that for every other generalization $(t'_g, \theta'_1, \theta'_2)$ of* e *and* f *it holds that t_g is an instance of t'_g.*

We refer to t_g as the ground term. For background infromation and an algorithm to compute most specific generalizations, see Lassez et. al [5]. An example of the homeomorphic embedding and the msg is:

e		f	t_g	θ_1	θ_2
e	$\trianglelefteq$	$Just\ e$	x	$[e/x]$	$[Just\, e/x]$
$Right\ e$	$\trianglelefteq$	$Right\ (e,\ e')$	$Right\ x$	$[e/x]$	$[(e, e')/x]$
$fac\ y$	$\trianglelefteq$	$fac\ (y\ -\ 1)$	$fac\ x$	$[y/x]$	$[(y-1)/x]$

4.2 Positive Supercompilation versus Deforestation

The additional strength in positive supercompilation comes from propagating information about the pattern of each branch in a case statement (rule R12 in our definition). It is not hard to craft a program by hand that benefits from this extra information propagation. It turns out that this information is useful for real programs as well. An example collected from Conway's Life in the nofib benchmark suite [9] is the expression *concat* (*map star xs*) with star defined as:

$$star\ n = \ \textbf{case}\ n\ \textbf{of}\ \{0\ \rightarrow\ ''\,'';\ 1\ \rightarrow\ ''o''\}$$

Evaluating $\mathcal{D}[\![\ concat\ (map\ star\ xs)]\!]$ results in the function $h_1'\ xs$:

$$\begin{array}{lll}
h_1'\ (x:xs) = & \textbf{case}\ x\ \textbf{of} & \\
& 0\ \rightarrow\ \textbf{case}\ xs\ \textbf{of} & \\
& \quad [] \rightarrow [' \,'] & \\
& \quad (x':xs') \rightarrow \textbf{let}\ h_2 = h_1'\ xs\ \textbf{in}\ [' \,',\ h_2] & \\
& 1\ \rightarrow\ \textbf{case}\ xs\ \textbf{of} & \\
& \quad [] \rightarrow [' \,',' \,o']) & \\
& \quad (x':xs') \rightarrow \textbf{let}\ h_3 = h_1'\ xs\ \textbf{in}\ [' \,',\ 'o',\ h_3] & \\
h_1\ (x:xs) = & \textbf{case}\ x\ \textbf{of} & \\
& 0\ \rightarrow\ \textbf{case}\ xs\ \textbf{of} & \\
& \quad [] \rightarrow \textbf{case}\ x\ \textbf{of}\ \{\ 0\ \rightarrow\ [' \,'];\ 1\ \rightarrow\ [' \,',\ 'o']\ \} & \\
& \quad (x':xs') \rightarrow \textbf{let}\ h_2 = h_1\ xs\ \textbf{in case}\ x\ \textbf{of} & \\
& \qquad\qquad\qquad\qquad 0\ \rightarrow\ [' \,',\ h_2] & \\
& \qquad\qquad\qquad\qquad 1\ \rightarrow\ [' \,',\ 'o',\ h_2] & \\
& 1\ \rightarrow\ \textbf{case}\ xs\ \textbf{of} & \\
& \quad [] \rightarrow \textbf{case}\ x\ \textbf{of}\ \{\ 0\ \rightarrow\ [' \,'];\ 1\ \rightarrow\ [' \,',\ 'o']\ \} & \\
& \quad (x':xs') \rightarrow \textbf{let}\ h_3 = h_1\ xs\ \textbf{in case}\ x\ \textbf{of} & \\
& \qquad\qquad\qquad\qquad 0\ \rightarrow\ [' \,',\ h_3] & \\
& \qquad\qquad\qquad\qquad 1\ \rightarrow\ [' \,',\ 'o',\ h_3] &
\end{array}$$

In comparison, the function h_1 is the output from our algorithm if we remove the extra information propagation from rule R12. There are two case-statements in this function that pattern matches on x. The second one is not necessary, since the value of x must be either 0 or 1 depending on which branch it has previously taken.

5 Extended Let-rule

The driving algorithm can be extended in various ways that will make it more powerful. We show that with an extended Let-rule in combination with a disabled whistle for closed expressions we can evaluate arbitrary closed expressions.

If the let-expression contains no free variables we could drive either the right hand side or the body and see what the result is. We could augment rule R11

with a second and third alternative:

$$\begin{aligned}\mathcal{D}[\![\mathbf{let}\ x = e\ \mathbf{in}\ f]\!]_{\mathcal{R},\rho} = {} & \mathcal{D}[\![\mathcal{R}\langle [e/x]f\rangle]\!]_{[],\rho}, \text{if } e = g, e = x \text{ or } x \in \mathit{strict}(f)\\ & \mathcal{D}[\![\mathcal{R}\langle [v/x]f\rangle]\!]_{[],\rho}, \text{if } \mathcal{D}[\![e]\!]_{[],\rho} = v\\ & \mathcal{D}[\![\mathcal{R}\langle [e/x]f'\rangle]\!]_{[],\rho}, \text{if } \mathcal{D}[\![f]\!]_{[],\rho} = f', \text{and } x \in \mathit{strict}(f')\\ & \mathbf{let}\ x = \mathcal{D}[\![e]\!]_{[],\rho}\ \mathbf{in}\ \mathcal{D}[\![\mathcal{R}\langle f\rangle]\!]_{[],\rho}, \text{otherwise}\end{aligned}$$

The reasoning behind this is that a closed expression contains all information that is needed to evaluate it, thus a fold should be unnecessary. If the expression diverges, then so does the final program.

The immediate question following from the above is whether this is beneficial for expressions that are not closed, a question we have no definite answer to. An example of the benefit of driving the body is *bodyEx* as shown below.

$$\mathit{bodyEx} = \mathbf{let}\ x\ =\ e\ \mathbf{in\ case}\ []\ \{[]\ \rightarrow\ x;\ (x : xs)\ \rightarrow\ y\}$$

$$\begin{aligned}& \mathcal{D}[\![\ \mathit{bodyEx}\ ]\!]\\ &= \mathcal{D}[\![\ \mathbf{let}\ x\ =\ e\ \mathbf{in\ case}\ []\ \mathbf{of}\ \{\ []\ \rightarrow\ x;\ (x : xs)\ \rightarrow\ y]\!]\ \} && \text{(R7)}\\ &= \mathcal{D}[\![\ e\ ]\!] && \text{(R11)}\end{aligned}$$

We can see how the body becomes strict after driving it, which opens up for further transformations.

Disabling the whistle for closed expressions in combination with the let rule above yields 6 from $\mathcal{D}[\![\ \mathit{fac}\ 3\ ]\!]$. This works for arbitrary closed expressions that are in the language, so, for example $\mathcal{D}[\![\ \mathit{sum}\ [1..3]\ ]\!]\ = 6$.

6 Total Correctness

The problem with previous deforestation and supercompilation algorithms in a call-by-value context is that they might change termination properties of programs. In this section, we prove that a slightly restricted version of our supercompiler does not alter whether a program terminates or not.

We define the standard notions of operational approximation and equivalence. A general context C which has zero or more holes in the place of some subexpressions is introduced.

Definition 4 (Operational approximation, Operational Equivalence).

- *e operationally approximates e', $e \mathrel{\tilde{\sqsubseteq}} e'$, if for all contexts C such that C[e], C[e'] are closed, if evaluation of C[e] terminates then so does evaluation of C[e'].*
- *e is operationally equivalent to e', $e \cong e'$, if $e \mathrel{\tilde{\sqsubseteq}} e'$ and $e' \mathrel{\tilde{\sqsubseteq}} e$*

The correctness of deforestation in a call-by-name setting has previously been shown by Sands [10] using his improvement theory. Notice that improvement $\trianglerighteq$ below is not the same as the homeomorphic embedding $\trianglelefteq$ defined previuosly. We use Sands's definitions for improvement and strong improvement:

Definition 5 (Strong Improvement, Cost equivalent).

- *e is improved by e', $e \trianglerighteq e'$, if for all contexts C such that C[e], C[e'] are closed, if computation of C[e] terminates using n function calls, then computation of C[e'] also terminates, and uses no more than n function calls.*
- *e is strongly improved by e', $e \trianglerighteq_s e'$, iff $e \trianglerighteq e'$ and $e \cong e'$.*
- *e and e' are cost equivalent, $e \trianglelefteq\trianglerighteq e'$ iff $e \trianglerighteq e'$ and $e' \trianglerighteq e$*

Cost equivalence implies strong improvement. With these definitions in place, total correctness for a transformation can be stated:

Theorem 1 (Sands). *If $e \trianglerighteq_s e'$, a transformation that replaces e by e' is totally correct.*

Improvement theory in a call-by-value setting requires Sands's operational metatheory for functional languages [11], where the improvement theory is a simple corollary of improvement induction over the well fonded resource structure $\langle \mathbb{N}, 0, +, \geq \rangle$.

Our set $\mathcal{G}$ corresponds to Sands's global recursive constants. We therefore label evaluation in one step using the Global-rule with the resource 1. We use $e \mapsto^k v$ to denote that e will evaluate to v using the Global rule k times, and any other rule as many times as it needs.

The expression $\mathbf{let}\, x = e\, \mathbf{in}\, (x, x)$ is not improved by (e, e), so we restrict rule R11 to only allow substitution if the body is both strict and linear with respect to the variable bound. We let $v\, R^\dagger\, v'$ for some relation R mean that $v = (\lambda x.e), v' = (\lambda x.e')$ and $e\, R\, e'$ or $v = v'$.

Proposition 1 (Total Correctness). *For all well-formed and well-typed expressions e, $e \trianglerighteq_s \mathcal{D}[\![e]\!]_{[],\rho}$.*

Proof (Proof sketch for $\trianglerighteq$ of R11).

Let R be the relation containing $\equiv$, together with all pairs of closed expressions where f is linear and strict with respect to x on the form $(\mathcal{R}\langle \mathbf{let}\, x = e\, \mathbf{in}\, f\rangle, \mathcal{R}\langle [e/x]f\rangle)$. Assume $f \mapsto^l \mathcal{E}\langle x\rangle$, $e \mapsto^k v$, and notice that $\mathcal{R}\langle \mathbf{let}\, x = []\, \mathbf{in}\, f\rangle$ is a reduction context.

Evaluation yields $\mathcal{R}\langle \mathbf{let}\, x = e\, \mathbf{in}\, f\rangle \mapsto^k \mathcal{R}\langle \mathbf{let}\, x = v\, \mathbf{in}\, f\rangle$ which will evaluate to $\mathcal{R}\langle [v/x]f\rangle$, and proceed $\mathcal{R}\langle [v/x]f\rangle \mapsto^l \mathcal{E}\langle v\rangle$. There are matching evaluation steps (in different order) in $\mathcal{R}\langle [e/x]f\rangle$: $\mathcal{R}\langle [e/x]f\rangle \mapsto^l \mathcal{E}\langle e\rangle \mapsto^k \mathcal{E}\langle v\rangle$. From this we conclude that $\mathcal{R}\langle \mathbf{let}\, x = e\, \mathbf{in}\, f\rangle \mapsto^n v$, for some $n \geq k + l$.

The remaining conditions for improvement simulation are satisfied since $v \equiv^\dagger v$, which implies $v\, R^\dagger\, v$ as required.

The proof for rule R7 is similar in structure to the proof by Sands [10, p. 24], and all other rules are cost equivalences, which implies strong improvement. The structure of the proofs for cost equivalence closely resembles the example by Sands [10, p. 14].

7 Termination

The homeomorphic embedding is a sufficient condition for ensuring termination. When the whistle blows, we split our expression into several strictly smaller expressions that are driven in the empty context. Termination can be proved using the framework by Sørensen [16].

Our proof is similar in structure to Sørensen's termination proof for supercompilation [16]. It is however simpler since we avoid partitioning nodes into trivial and non-trivial nodes and use a single predicate (the homeomorphic embedding) to control inlining.

8 Related work

There exists much literature concerning algorithms that remove intermediate structures in functional programs. Most of it is however in a call-by-name or call-by-need context which makes it a different, yet difficult, problem.

8.1 Transformations for Strict Languages

The seminal work by Turchin on Supercompilation [17] is closely related to our work. The supercompiler was originally intended for the strict functional language Refal [18]. The supercompiler could therefore introduce termination in Refal programs, the problem we set out to solve with our work.

More recently Ohori and Sasano presented a lightweight fusion algorihtm [8], which they applied to variant of Standard ML. They limit their algorithm to inlining each function at most once, thereby avoiding termination problems. This gives the unfortunate side effect that it does not handle two successive applications of the same function, nor mutually recursive functions as our examples in Section 2.

8.2 Transformations for Lazy Languages

Focusing on the closest relatives to our algorithm will bring out Wadler's deforestation algorithm [19] for a first order language as the start for deforestation research. This was later extended to a higher-order language by Marlow and Wadler [6]. Refinements to this was made by Marlow [7] and Hamilton [1, 2].

Work by Sørensen et. al. on the positive supercompiler [14] deserves a special mention. Their work clearly explains the difference between deforestation and supercompilation, giving lots of illustrative examples. While the context is a lazy language, many of their insights apply to our work as well. Secher and Sørensen later refined this work to propagate both negative and positive information in the perfect supercompiler [12].

9 Conclusion

We have presented a positive supercompiler for a higher-order call-by-value language. It is proven correct for all input.

The adjustment to the algorithm for preserving call-by-value semantics is new and works surprisingly well for many examples that were intended to show the usefulness of call-by-name transformations.

9.1 Future work

We believe that the restriction of rule R11 in the proof of correctness is not necessary for the soundness of our algorithm, but have not found a way to prove it yet. We will investigate how the concept of an inline budget may be used to obtain good balance between code size and inlining benefits.

The current termination strategy could be improved with respect to the transformation result, the current definition inlines a little too much. The algorithm should start over and calculate a msg at the expression t when the whistle blows, just like Supercompilation [15] does.

More work could be done on strictness analysis component of our supercompiler. We do not intend to focus on that subject, though; instead we hope that the modular dependency on strictness analysis will allow our supercompiler to readily take advantage of general improvements in the area.

Acknowledgements

The authors would like to thank Simon Marlow, Duncan Coutts and Neil Mitchell for valuable discussions. We would also like to thank Viktor Leijon for reading earlier drafts of this paper and providing useful comments for improvements.

References

1. G. W. Hamilton. Higher order deforestation. In *PLILP '96: Proceedings of the 8th International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 213–227, London, UK, 1996. Springer-Verlag. ISBN 3-540-61756-6.
2. G. W. Hamilton. Higher order deforestation. *Fundam. Informaticae*, 69(1-2):39–61, 2006.
3. T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *FPCA*, pages 190–203, 1985.
4. J. Kort. Deforestation of a raytracer. Master's thesis, University of Amsterdam, 1996.
5. J-L. Lassez, M. Maher, and K. Marriott. Unification revisited. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, 1988.
6. S. Marlow and P. Wadler. Deforestation for higher-order functions. In John Launchbury and Patrick M. Sansom, editors, *Functional Programming*, Workshops in Computing, pages 154–165. Springer, 1992. ISBN 3-540-19820-2.

7. S. D. Marlow. *Deforestation for Higher-Order Functional Programs*. PhD thesis, Univ. of Glasgow, 27 April 1995.
8. A. Ohori and I. Sasano. Lightweight fusion by fixed point promotion. In M. Hofmann and M. Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 143–154. ACM, 2007. ISBN 1-59593-575-4.
9. W. Partain. The nofib benchmark suite of haskell programs. In John Launchbury and Patrick M. Sansom, editors, *Functional Programming*, Workshops in Computing, pages 195–202. Springer, 1992. ISBN 3-540-19820-2.
10. D. Sands. Proving the correctness of recursion-based automatic program transformations. *Theoretical Computer Science*, 167(1–2):193–233, 30 October 1996.
11. D. Sands. From SOS rules to proof principles: An operational metatheory for functional languages. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, January 1997.
12. J. P. Secher and M. H. Sørensen. On perfect supercompilation. In D. Bjørner, M. Broy, and A. Zamulin, editors, *Proceedings of Perspectives of System Informatics*, volume 1755 of *Lecture Notes in Computer Science*, pages 113–127. Springer-Verlag, 2000.
13. M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J.W. Lloyd, editor, *International Logic Programming Symposium*, pages 465–479. Cambridge, MA: MIT Press, 1995.
14. M. H. Sørensen, R. Glück, and N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
15. M.H. Sørensen and R. Glück. Introduction to supercompilation. In *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School*, pages 246–270, London, UK, 1999. Springer-Verlag.
16. M.H. Sørensen. Convergence of program transformers in the metric space of trees. *Sci. Comput. Program*, 37(1-3):163–205, 2000.
17. V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.
18. V. F. Turchin. *Refal-5, Programming Guide & Reference Manual*. Holyoke, MA: New England Publishing Co., 1989.
19. P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990. ISSN 0304-3975.

The Simple Category of Modules

Mikołaj Konarski
mikon@mimuw.edu.pl

Institute of Informatics, University of Warsaw,
Banacha 2, 02-097 Warszawa, Poland

Research and Development Division, Comarch SA
Puławska 525, 02-844 Warszawa, Poland

Abstract. Dule is a module system for functional programming languages, modeled using elementary category theory and straightforwardly implemented on an simple categorical abstract machine. The focus of Dule is the ease of maintenance of complete programs at the cost of marginalizing code-reuse mechanisms. Fine-grained modularization without prohibitive programming overhead is made possible by introducing mechanisms inspired by category theory, such as default name-driven sharing and implicit module composition.
In this paper we concentrate on the semantics/implementation of our module system in the abstract categorical machine, via Simple Category of Modules (SCM), where modules are morphisms, signatures are objects, module composition is SCM composition and categorical domain lists module parameters. The construction of SCM from a simple 2-categorical model of the abstract machine provides set-theoretic models for Dule and ensures that its implementation is independent of the core language; in particular, it does not require second-order polymorphism nor dependent types (nor even function types). SCM is cartesian and has enough limits to model type sharing; the constructive proof provides implementation of the related module operations. The semantics of Dule is compositional and constructive, enabling a straightforward, faithful implementation that follows and verifies the semantics.

Key words: module systems, type sharing, category theory, semantics of programming languages, categorical abstract machines

1 Introduction

1.1 Background

Modular programming is necessary due to the growing size, complication and the requirement of modifiability of computer programs. The most useful approach appears to be the integration of rigorous module systems into high level programming languages, as in Standard ML [16] or OCaml [14]. However, strictly modular programming style, with explicitly defined modular dependencies, can be sustained only in small programming projects written in these languages [10].

In large projects, managing the many layers of abstraction, introduced with the module hierarchy, turns out to be harder even than manual tracing of each particular dependency in unstructured code. Abstract and fine-grained modular programming is cumbersome, because the headers to be written are complicated and module applications often trigger global type sharing errors [20]. When a module has n parameters, $O(n^2)$ potential typing conflicts await at each application, making not only creation of modules, but also their maintenance and reuse very costly.

Another problem is the tension between abstraction, expressiveness and applicability of a module. In simplification: the more abstract a signature of a module result is, the easier it is to implement the module, but the harder it is to use the module. Without specialized module system mechanisms, this global tension can only be solved by making each module excessively powerful and general. However, the usual practical approach is to gradually sacrifice abstraction as the program grows, whereas, especially for ensuring correctness of large programs, the abstraction is crucial [4]. Huge collections of interdependent modules themselves require modularization, lest managing the modules becomes as tedious as tending the mass of individual entities in non-modular programs. Grouping of modules performed using the mechanism of submodules often overloads the programmer with type sharing bureaucracy or requires a violation of the abstraction guarantees. Other solutions, some of them using external tools, tend to be cheaper, but are usually very crude and incur the risk of name-space clashes or ignore type abstraction.

1.2 The Module System

The Dule project is an experiment in large-scale fine-grained modular programming employing a terse notation based on an elementary categorical model. The Dule module system remedies the known bureaucratic, debugging and maintenance problems of functor-based modular programming (SML, OCaml) by introducing simple modular operations with succinct notation inspired by the simplicity of its semantic categorical model and a modularization methodology biased towards program structuring, rather than code-reuse [1] (the latter is partially supported through an auxiliary layer of the module system, not described here). The same categorical model and its natural extensions induce an abstract machine [5] based on their equational theories and inspire novel functional core language features that sometimes complement nicely the modular mechanisms and sometimes are language experiments on their own (not described here either; see the formal definition of Dule [12]).

The assets of the Dule project are gathered on its homepage at `http://www.mimuw.edu.pl/~mikon/dule.html`, where the formal definition of the current version of the language as well as an experimental compiler with a body of Dule programs can be obtained. The version of code designed specifically for this presentation is at `http://www.mimuw.edu.pl/~mikon/Dule/download/dule-src.2007-08-31.tgz`. The compiler works adequately, but it still does not bootstrap and the lack of low-level libraries precludes practical applications. There is 10000

lines of fine-grained modular Dule code, including a fine-grained modular rewrite of the main parts of the compiler itself, but surely much more will be needed to discover all scaling problems and fine-tune the modularization methodology.

```
spec DrawChart =
~YearTable ~Picture ->
  sig
    value draw : Picture.t
  end
```

```
DrawChart =
  struct
    value draw =
      if YearTable.patents_expired
        then Picture.ok
      else Picture.crash
  end
```

Fig. 1. An example signature and module in the full, sugared version of Dule. The application of the module to its arguments (which should be defined earlier, not shown here) is implicit; notice also the small signature and module headers.

Below we give an overview of the features of the full version of Dule module system. We refer the reader to the long informal tutorial of Dule available from its homepage that accompanies the Dule formal definition for an illustrated overview. Main highlights of Dule:

- module composition with both transparent and non-transparent versions (the latter is the categorical composition from the semantic model)
- various module grouping mechanisms (including module categorical product operations)
- no sharing equations (nor '`with`' clauses nor type abbreviations); default sharing by names
- no substructures (submodules)
- pseudo-higher-order notation for module signatures (flattened, because no higher-order modules, but type dependencies on parameters remain; not discussed here)
- recursive signatures (flattened in a more complex way; not discussed here)
- (co)inductive modules (mutually dependent modules [6] constructed using inductive types instead of recursive types; not discussed here)
- default implicit module composition
- signatures of transparent functor applications [13] are expressible (actually all signatures of module operations are expressible)
- compositional semantics ensures separate compilation
- no references to environments in the semantics ensures the types from module parameters are abstract; on the other hand the abstraction can be sidestepped in a controlled way and limited scope, if necessary
- no dependencies between declared core language level entities (all dependencies are expressed through modules)

1.3 The Abstract Machine

The compiler of Dule, after type and signature reconstruction, computes the semantics of modular programs in the language of our categorical abstract machine. The semantics, that is the machine code, can then be executed by the machine, yielding results as expected in a programming language. The elementary notion of 2-category [11] with products is the categorical model of the abstract machine. An equational theory of 2-categories is the basis for the typed combinator reduction engine [7] that powers the machine.

The abstract mathematical model of the machine helped in proving its properties, in particular type-soundness and confluence. Preserving type information in the machine language offers many benefits, among them the ability to verify type-correctness of any received piece of machine code. The machine is very simple, especially considering that even advanced module operations can be performed exclusively on the machine language module representations, without the need to consult any additional annotations or modules' sources.

It is possible to extend the machine (and its mathematical model) with function types, sum types, inductive and coinductive types and general recursion (the reduction rules of our basic machine are listed in the Appendix, rules for the extended machine can be found in Appendix A.1.4 of [12]). The resulting machine language, with some syntactic sugar and type reconstruction, makes for an interesting core programming language, making available to the user, e.g., raw categorical composition (explicit substitution) and rigorously typed built-in structured (co)recursion. The construction of the module system on top of the abstract machine carries over to such extensions. The machine can also be extended to execute fully typed OCaml or Standard ML code. Inversely, simply typed λ-calculus with products, system F and ML can be presented as 2-categories with products, as needed for modeling our module system. In such presentations, the vertical composition in the 2-categories would be substitution and the composition of 1-morphisms would be type instantiation.

For our purposes, let's define 2-category as comprising of two ordinary categories and, additionally, a family of categories $C(c, e)$. The first of the two categories is the underlying category U, that is, the category of objects and 1-morphisms with the composition of 1-morphisms. Then, for each pair of objects c, e, there is a category $C(c, e)$ of all 1-morphisms with source c and target e as objects and all 2-morphisms between them as morphisms with their vertical composition. Horizontal composition yields the category of objects and 2-morphisms H, with the identity on object c equal to the 2-identity on the 1-identity on object c.

Cat, the category of all (small) categories is an example of a 2-category. By analogy to **Cat** we will call objects 'categories', 1-morphisms 'functors' and 2-morphisms 'transformations'. The 2-categories that are models of our abstract machine have a distinguished category (object) $*$ and finite products in the U and H categories and in all categories $C(c, *)$. If we take **Set** (the category of sets and functions) for the distinguished category $*$, then **Cat** has all the required products, so it is a model of our abstract machine. We will denote a finite U-product

of categories $c_1, \ldots, c_n$ labeled $i_1, \ldots, i_n$, respectively, by $\texttt{<}i_1 \texttt{ - } c_1\texttt{; } \ldots \texttt{; } i_n \texttt{ - } c_n\texttt{>}$. Products in $C(c, *)$ will be written $\texttt{\{}i_1 \texttt{ : } f_1\texttt{; } \ldots \texttt{; } i_n \texttt{ : } f_n\texttt{\}}$.

Below we present our chosen syntax and typing of basic operations of 2-categories with products, which is also the typing of the combinators of the abstract machine. The 'record' operations that appear below are generalized labeled tuples. First, we assign the U-source and U-target categories to functors (which can be seen as types of the core language).

$$\text{(U-identity)}\ \frac{}{\texttt{F\_ID}(c) : c \to c} \qquad \frac{f : c \to d \qquad g : d \to e}{f \texttt{ . } g : c \to e}\ \text{(U-composition)}$$

$$\frac{}{\texttt{F\_PR}(lc, i) : \texttt{<}i \texttt{ - } c\texttt{; } \ldots\texttt{>} \to c}\ \text{(U-projection)}$$

$$\frac{f_1 : c \to e_1 \quad \cdots \quad f_n : c \to e_n}{\texttt{<}i_1 \texttt{ : } f_1\texttt{; } \ldots\texttt{; } i_n \texttt{ : } f_n\texttt{>} : c \to \texttt{<}i_1 \texttt{ - } e_1\texttt{; } \ldots\texttt{; } i_n \texttt{ - } e_n\texttt{>}}\ \text{(U-record)}$$

$$\frac{f_1 : c \to * \quad \cdots \quad f_n : c \to *}{\texttt{\{}i_1 \texttt{ : } f_1\texttt{; } \ldots\texttt{; } i_n \texttt{ : } f_n\texttt{\}} : c \to *}\ \text{(C(c, *)-product)}$$

Now we present the $C(c, e)$-domains and codomain of transformations (generalized values of a programming language). Many of the rules below have additional, unwritten premises ensuring that the terms that appear in them have compatible source and target categories. For example, in rule (H-comp) we require that the target of functor f_1 is equal to the source of functor f_2.

$$\text{(H-id)}\ \frac{}{\texttt{T\_ID}(c) : \texttt{F\_ID}(c) \to \texttt{F\_ID}(c)} \qquad \frac{t_1 : f_1 \to h_1 \qquad t_2 : f_2 \to h_2}{t_1 * t_2 : f_1 \texttt{ . } f_2 \to h_1 \texttt{ . } h_2}\ \text{(H-comp)}$$

$$\text{(C(c, *)-id)}\ \frac{}{(\texttt{:}\ g) : g \to g} \qquad \frac{t : f \to g \qquad u : g \to h}{t \texttt{ . } u : f \to h}\ \text{(C(c, *)-comp)}$$

$$\text{(H-pr)}\ \frac{}{\texttt{T\_PR}(lc, i) : \texttt{F\_PR}(lc, i) \to \texttt{F\_PR}(lc, i)} \qquad \frac{}{i : \{i \texttt{ : } f\texttt{; } \ldots\} \to f}\ \text{(C(c, *)-pr)}$$

$$\frac{t_1 : f_1 \to h_1 \quad \cdots \quad t_n : f_n \to h_n}{\texttt{<}i_1 \texttt{ = } t_1\texttt{; } \ldots\texttt{; } i_n \texttt{ = } t_n\texttt{>} : \texttt{<}i_1 \texttt{ : } f_1\texttt{; } \ldots\texttt{>} \to \texttt{<}i_1 \texttt{ : } h_1\texttt{; } \ldots\texttt{>}}\ \text{(H-record)}$$

$$\frac{t_1 : f \to h_1 \quad \cdots \quad t_n : f \to h_n}{\texttt{\{}i_1 \texttt{ = } t_1\texttt{; } \ldots\texttt{; } i_n \texttt{ = } t_n\texttt{\}} : f \to \texttt{\{}i_1 \texttt{ : } h_1\texttt{; } \ldots\texttt{; } i_n \texttt{ : } h_n\texttt{\}}}\ \text{(C(c, *)-record)}$$

In the rules, many combinators are written in abbreviated form and their notation does not contain all the typing information, e.g., the C(c, *)-projection. Others are written including full typing annotations, e.g., the H-projection. Later we may sometimes switch between the abbreviated and not abbreviated notation. The $C(c, *)$-composition plays the role of substitution in a programming language and $C(c, *)$-projections can be used as variables, but all these operations are combinators — there are no free variables anywhere — otherwise it wouldn't be an abstract machine, but an interpreter.

2 Simple Category of Modules

The heart of my module system is its mathematical model, the Simple Category of Modules (SCM), which can be based on any 2-category with products that models the core language to be used inside modules. Consequently, no dependent products or sums [18] or second-order polymorphism [2] or even function types are needed for the construction of SCM nor of the operations of the module system, to be presented in the next section. The construction of Simple Category of Modules (SCM) should look quite intuitive to a programmer with a minimal categorical background. The objects of this category are module signatures and the morphisms are modules themselves.

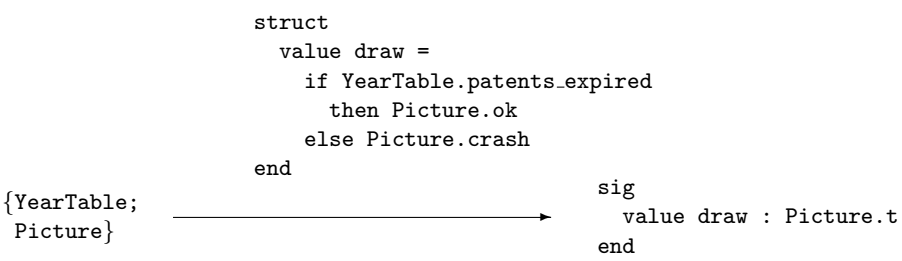


Fig. 2. Objects and morphisms in SCM.

The domain of a module is the signature of its parameters and the codomain is the result signature of the module. The identity morphism of SCM is the identity module and the composition is the operation of supplying implementation of parameters to a module. Other kinds of module composition are categorically definable, as will be shown later.

Fig. 3. Composition in SCM.

Categorical products model parameters (for example, the domain of a module is usually the product of signatures of parameter modules) allowing the programmer to express a kind of 'module variables' as projections in SCM. Moreover there is enough equalizers (limits) in SCM to model type sharing specifications. Complex limits built using equalizers model type sharing among parameter modules and also between parameters and the result signature. The kind of equalizers to be used for the semantics of our module language has a simple construction in SCM.

Now we will proceed with the formal definition of SCM. For the rest of this paper let us fix an arbitrary 2-category with products. Objects and morphisms of the SCM will be built from the morphisms of the fixed 2-category. Since

the category can be, in particular, **Cat** with `*` equal to **Set**, SCM has a set-theoretic semantics [17] and the signatures and modules can be thought of as (quite complex but not higher-order) functions. Our account here is somewhat simplified. For a totally strict and detailed account see [12].

2.1 Objects and morphisms

Objects of SCM are called signatures and are defined as follows.

Definition 1 *A functor* f `:` $\texttt{<}i_1$ `-` c_1`;` $\ldots$`;` i_k `-` $c_k\texttt{>} \to$ `*` *of the fixed 2-category is a signature if it can be presented as* $\texttt{\{}i_1$ `:` f_1`;` $\ldots$`;` i_n `:` $f_n\texttt{\}}$ *for some categories* $\texttt{lc} = i_1$ `-` c_1`;` $\ldots$`;` i_k `-` c_k *and functors* $\texttt{lf} = f_1, \ldots, f_n$ `:` $\texttt{<}i_1$ `-` c_1`;` $\ldots$`;` i_k `-` $c_k\texttt{>} \to$ `*`*. The indexed list of categories* `lc` *is called the type part of* f*, while the indexed list* `lf` *is called the value part of* f*.*

This simple form of signatures resembles the syntax of simple Standard ML or OCaml signatures where `lc` would correspond to names of types and `lf` to types of values in module signature. In general, beside the names of types, `lc` will usually contain names and kinds of parameter modules with nested names of types (in our syntax for base module signatures the information about parameters is passed around in so-called context signatures, see Section 3.1 below). Also, when f is the signature of a group (record) of modules, `lf` is an indexed list of types of value parts of the modules, as defined below. Anyway, our 2-category product operations are enough to capture the diversity.

Definition 2 *A module is a triple of a functor* `f`*, a transformation* `t` *and a signature* `s` *such that there is a signature* `r` *satisfying the following conditions:*

1. `f` : `src r` $\to$ `src s`
2. `t` : `r` $\to$ `f . s`

where `src` *produces the source category of a functor and the dot in the second condition is the U-composition. The (uniquely determined) signature* `r` *is the categorical domain of the above module seen as a morphism of SCM and the signature* `s` *is the categorical codomain (also uniquely determined, because given in the triple). Functor* `f` *is called the type part of the module and* `t` *is called the value part.*

We will use the notation `m` : `r` $\to$ `s` to mark the categorical domain and the categorical codomain of module `m` in SCM. Concrete examples of triples constituting modules are given below.

2.2 Identity and composition

The identity module on signature `s` in SCM is the triple $(\texttt{F\_ID}(c), \texttt{(: s)}, \texttt{s})$, where category c is the source of `s`. So, for example, an identity on an empty signature will be $(\texttt{F\_ID(<>)}, \texttt{(: \{\})}, \texttt{\{\}})$.

Composition in SCM is the operation of supplying implementation of parameters to a module. Let us look at the OCaml code from the Dule compiler that generates the abstract machine code for the module composition operation `m_Comp`. The code should be self-explanatory except for `f_COMP f1 f2`, which is an abstract syntax notation for U-composition written `f1 . f2` in our concrete syntax, `t_FT f1 t2`, which is the multiplication from the left by a functor `f1`, that is H-composition (: `f1`) $*$ `t2` and `t_comp t1 it2`, which is the composition of transformations `t1 . it2`.

```
let m_Comp m1 m2 = (* : r1 -> s2 *)
  let f1 = Dule.type_part m1 (* : r1 -> s1 *) in
  let t1 = Dule.value_part m1 in
  let f2 = Dule.type_part m2 (* : r2 -> s2 *) in
  let t2 = Dule.value_part m2 in
  let f = f_COMP f1 f2 in (* s1 = r2 *)
  let it2 = t_FT f1 t2 in
  let t = t_comp t1 it2 in
  let s2 = Dule.codomain m2 in
  Dule.pack (f, t, s2)
```

The following drawing shows the domains and codomains of the transformations appearing in the code of `m_Comp`, according to Definition 2. Transformations are here depicted by arrows. U-compositions are denoted by horizontal bars (double minuses).

```
     t1    f1         t2    f2         t    f1
r1 ------> --    r2 ------> --    r1 -----> --
           s1               s2              f2
                                            --
                                            s2
```

The drawing below illustrates the value part of the result of module composition. H-composition is here represented by placing the first transformation above the second. Vertical composition is represented by sharing a common domain/codomain. Transformation (: `f1`) is the identity on `f1`.

```
               (: f1)
     t1    f1 ------> f1
r1 ------> --         --
                      f2
           s1 ------> --       s1 = r2
                t2    s2
```

Modularly speaking: `m_Comp` instantiates the values of `m2` with the concrete implementation of the types given in `m1` and then applies the instantiated procedure to the values of `m1`. Consider the following composition written in the concrete syntax of our module language (to be defined shortly):

```
{M = :: {} -> sig type t1 value v1 : t1 end
     struct type t1 = {} value v1 = {} end}
.
:: {M : sig type t1 value v1 : t1 end} ->
     sig value v2 : M.t1 end
struct value v2 = M.v1 end
```

When the generated machine code is executed by the abstract machine, the implementation of value `v2`, which is the composition of projections `M.v1`, is multiplied from the left by the implementation of the types of the first module, that is

```
<M : <t1 :{}>>
```

resulting again in the composition of projections. The composition is then vertically composed (as the second operand) with the implementation of values of the first module, that is

```
{M : {v1 = {}}}
```

resulting in transformation `{}`, which is the value of `v2` in the outcome module.

After some more computation, the result of composition is seen to be expressible in our module language, using the mechanism of context signatures (the signatures written just after `sig`, automatically reconstructed in full Dule, see Section 3.1 below) to retain the signature `M`, as follows.

```
:: {} ->
     sig {M : sig type t1 value v1 : t1 end}
       value v2 : M.t1
     end
struct value v2 = {} end
```

Theorem 3 *The above definitions result in a category, where signatures are objects, modules are morphisms and* `m_Comp` *is the composition.*

Proof. The domains and codomains of modules are well defined. The equalities about identity as the neutral element of composition follow promptly from the properties of identities in 2-categories. The associativity of composition is easy to establish, again using the properties of 2-categories.

2.3 Products

Products are necessary in our model to give semantics to modules that depend on many arguments. It turns out that SCM has (labeled) products.

Theorem 4 *Simple Category of Modules is cartesian.*

The proof (that we omit) is constructive, by defining in OCaml, similarly as we defined `m_Comp`, module operations of product of signatures, projection module and record module. The three operations are well defined; in particular when given correct operands they produce signatures and modules as required in Definition 1 and 2.

Once we have products, we can easily model in SCM a module system similar to the one of Standard ML but with no sharing requirements. Sharing requirements are used in modular programming to ensure that certain types, possibly appearing in distant modules, are equal. In a framework enabling abstraction, such as ours, guarantees of type equality are crucial to enable inter-operation between modules. However, sharing can be difficult to model. In particular, the ordinary labeled products of SCM do not suffice for this task.

2.4 Equalizers

Pullback of signatures (categorical limit of diagrams consisting of morphisms with a common codomain) is a good model of a signature of a pair of modules with some sharing between the two. Consider the following example, in which we write '`sharing type`' to mark an ad hoc notation for a sharing equation that, in this case, requires two product types to be equal (and if the product operation is injective in the fixed 2-category, then `M1.t` is equal to `M2.t` and `M2.u` is equal to `{}`).

```
{M1 : sig type t end;
 M2 : sig type t type u end;
 sharing type
   {t : M1.t; u : {}}
   = {t : M2.t; u : M2.u}}
```

Such a signature can be interpreted as a pullback of two morphisms (modules) from signatures `M1` and `M2`, respectively, to a common target (e.g., signature `sig type c end`). The morphisms determine the types to be identified. The first of the morphisms could look as follows:

```
struct type c = {t : M1.t; u : {}} end
```

and the second as follows:

```
struct type c = {t : M2.t; u : M2.u} end
```

However, pullbacks are not adequate for a similar task in case of many signatures. A sharing requirement may refer to components that are absent in some of the (possibly numerous) signatures, while the pullback construction is based on diagrams with morphisms from all the objects. We could overcome the problem by considering multiple sharing equations and adding a trivial equation for each signature absent from the main equation, but there are more elegant and efficient solutions.

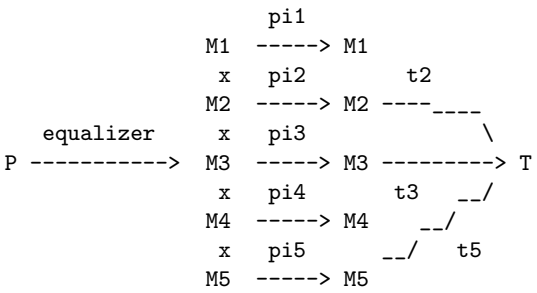
A pullback of given morphisms can be constructed as an equalizer (generalized from two morphisms to a family of morphisms) of the morphisms prefixed with projections [19]. The projections come from the product of sources of the morphisms. This representation is valid for the labeled pullbacks as well, and involves labeled products and equalizers. If we allow the equalizer to be taken of a smaller family than the one used in the product, we can capture the sharing of components of only the chosen signatures. In fact, this construction is just the

construction of the general limit of a diagram ([15], Theorem V.2.1). It turns out that a product of signatures with some sharing is just the limit of the diagram formed by all the signatures and the morphisms representing the sharing requirement.

Let's suppose we are to represent categorically a collection of five signatures with a type shared among three of them.

```
{M1 : sig value v : {} end;
 M2 : sig type t end;
 M3 : sig type t end;
 M4 : sig type u type w end;
 M5 : sig type t end;
 sharing type
   M2.t = M3.t = M5.t}
```

First, we can represent the types to be shared as three morphisms (`t2`, `t3`, `t5`, in this case just identities) into a common target `T`. Then, to construct the limit, we take the product of the five signatures and compose the respective projections (`pi2`, `pi3`, `pi5`) with the three morphisms. The equalizer of the family of the three compositions is the sought limit signature `P`.



If there are several sharing equations, the sought signature is again the limit of the diagram, this time containing several families of morphisms, each sharing a codomain. In the construction using product and equalizer, the equalizers representing consecutive sharing equations have to be composed. For a formalization of the main concepts see Section 5.1.4 of [12]. Here we only cite the result that has a constructive proof that guides the implementation of type sharing in our module system.

Lemma 5 *Let us fix an SCM over an arbitrary 2-category. For each categorical diagram in the SCM representing sharing requirement between whole type parts (the collection of all types) of modules, if the type names agree, a product signature with sharing represented by the diagram exists.*

In our module system we will apply a variant of the whole type parts sharing requirements, in which the modules to be equated are determined by names of nested signatures contained in a product. The sharing should take place between type parts of module signatures having the same labels. For instance, the following signature (featuring context signatures, as described in the next section) requires values `M1.v1` and `M2.v2` to have the same type.

```
{M1 : sig {M : sig type c end} type t1 value v1 : M.c end;
 M2 : sig {M : sig type c end} value v2 : M.c end}
```

In the following example, type `M.c` occurring in three context signatures and one main signature of the product will be shared in all four main signatures.

```
{M1 : sig {M : sig type c end} type t1 end;
 M2 : sig {M : sig type c end} end;
 M3 : sig {M : sig type c end} value v : M.c end;
 M : sig type c end}
```

The context signature of `M1` indicates that `M` is imported into `M1`, or rather that module `M1` is supposed to be built from `M`. The name `M`, assigned to one of the main signatures of the product, is interpreted as marking the same module that was used in construction of `M1`. The module `M` is, in this particular case, required to be provided separately as the fourth component of module record, perhaps to be used for building other modules later on.

Our operation, equating whole type parts of modules with the same names, has the same syntax as for the ordinary labeled product, as no explicit sharing specifications need to be written (notice the absence of explicit '`sharing type`' requirements in the above examples). The operation will be called product with name-driven sharing or (ambiguously but succinctly) just product. Every product with name-driven sharing is a categorical limit of a simple sharing diagram and can be constructed as a composition of a number of equalizers of whole type parts of modules (slightly generalized to allow nesting), taken on the product of signatures. The operations of projection with name-driven sharing and record with name-driven sharing are expressible in an analogous way using the operations of the ordinary product and the equalizer. The proof of Lemma 5 shows how to construct the equalizer operations using the abstract machine code.

3 Semantics/Implementation of the Module System

After constructing and analyzing the SCM we use it to develop a module system — a programming-oriented language for SCM (though without signature reconstruction, etc., it is not yet user-friendly, see [12]). If we fix a 2-category for the core language, we can construct the unique SCM built upon that 2-category. Our module system is an extension of the SCM (treated as a partial algebra) by several module operations, most of which are partial. The carriers are not extended and we don't need any additional assumptions on the core language, in particular we do not require function types.

The semantics of our module system is compositional and environment-free, which implies that the modules may be compiled separately. The correctness and the result of a module operation depend only on the target code (SCM morphisms, abstract machine code) obtained from the operands, so the original source code can be compiled only once and then forgotten. The lack of any environments also ensures that module parameters are abstract. Upon supplying arguments the abstraction can be retained or overcome, depending on the operations used (composition vs. instantiation).

3.1 Basic operations

We start an overview of the typing and semantics of operations of our module system. In the typing rules we do not formulate additional definedness side conditions, hence the rules do not completely capture definedness properties of the operations. They only indicate whether a raw term belongs to the language and what its domain and codomain are. The complete definedness conditions are given and discussed in detail in the formal definition of Dule [12], where also the abstract machine code for the operations is given.

In the formal definition we also argue that each case of partiality of any of the modular operations corresponds to a class of modular programming errors. We prove that the semantics of the operations is well defined; in particular, their results belong to the set of objects and morphisms of the SCM. We check that the declared parameter and result signatures of modules coincide with SCM domains and codomains of the morphisms the module expressions denote. The proof of well-definedness of the semantics of the module language constitutes a major part of the proof of correctness of the Dule compiler.

Here are the rules for the already defined identity and composition operations and for base modules that use elements available from an argument satisfying signature r to define core language types and values as specified in s.

$$\text{(identity)}\ \frac{}{(\texttt{:}\ s) : s \to s} \qquad \frac{m_1 : r_1 \to s \qquad m_2 : s \to s_2}{m_1\ \texttt{.}\ m_2 : r_1 \to s_2}\ \text{(composition)}$$

$$\frac{s = \texttt{sig}\ r'\ \texttt{type}\ i\ \ldots\ \texttt{value}\ j\ \texttt{:}\ f\ \ldots\ \texttt{end}}{(\texttt{::}\ r\ \texttt{->}\ s\quad \texttt{struct type}\ i\ \texttt{=}\ g\ \ldots\ \texttt{value}\ j\ \texttt{=}\ t\ \ldots\ \texttt{end}) : r \to s}\ \text{(base)}$$

Signatures of the same for as s above are called base signatures and r' inside them can specify less components than r. Such r', occurring inside s, is called a context signature, on which types of values in s may depend (if r is `{}`, it is usually omitted in writing; in sugared Dule syntax r disappears altogether). Base signatures can be viewed as products with name-driven sharing (as defined in the next subsection) of micro-signatures $f, \ldots$, while base modules are isomorphic to records with name-driven sharing of micro-modules that only define one type or one value.

3.2 Cartesian structure

As told in Section 2.3, SCM is cartesian, which allows for parameterization by named modules. Inside a modular expression with product domain the parameter modules are treated as 'locally' abstract. When several module expressions are put within a module record with a product domain, they all share the same locally abstract arguments. However, this abstraction does not prevent specifying equalities between individual types.

For the semantics of our module system we choose to employ implicit sharing of whole type parts of modules, determined by names of components, as illustrated in Section 2.3. When grouping signatures in a product, all top-level context types (types inherited from context signatures) with the same name are

to be shared. In the result, top level type components of a product of signatures are the sets of types defined in the signatures themselves indexed by the signature names, together with all their context types.

This particular merger of labeled product and equalizer, will be called product with name-driven sharing, or just product. The notation for the operations is the same as for the ordinary labeled categorical product. This setup results in concise syntax, with somewhat limited expressiveness, which is recoverable with the help of instantiation operation (e.g., to rename module parameters), described later on.

$$\frac{}{i : \{i \ : \ r ; \ldots\} \to r} \text{ (projection)}$$

$$\frac{m_1 : r \to s_1 \quad \cdots \quad m_n : r \to s_n}{\{i_1 \texttt{ = } m_1 \texttt{;} \ \ldots \texttt{;} \ i_n \texttt{ = } m_n\} : r \to \{i_1 \ : \ s_1 \texttt{;} \ \ldots \texttt{;} \ i_n \ : \ s_n\}} \text{ (record)}$$

Whenever the product operations are defined, they satisfy all the equalities required of a categorical product. Moreover, whenever they are undefined their signature operands show that the programmer tried to impose a contradicting sharing requirement, or their module operands show that the programmer violated the sharing requirements he had imposed earlier.

Observation 6 *The product signature with the projection modules form a categorical cone over the diagram of sharing requirement representing name-driven sharing among the product operands.*

3.3 Specialization, instantiation and trimming

There are two modular operations, instantiation and trimming, without clear categorical meaning in the context of SCM, though they have a simple mathematical definition in terms of 2-categories with products.

$$\text{(instantiation)} \ \frac{m_1 : r_1 \to s \qquad m_2 : s \to s_2}{m_1 \ \texttt{|} \ m_2 : r_1 \to m_1 \ \texttt{|} \ s_2} \qquad \frac{m_1 : r_1 \to s_1}{m_1 \ \texttt{:>} \ r_2 : r_1 \to r_2} \ \text{(trimming)}$$

The operation m_1 `|` s_2 specializes signature s_2 to a narrower field of use — restricted to the types of module m_1. The operation m_1 `|` m_2 instantiates module m_2 so that it has more concretely defined manner of operation and more strictly specified field of operation, determined by module m_1.

The instantiation operates on the value parts of operand modules in exactly the same way as the module composition `m_Comp` does. Yet instantiation is not a good candidate for an alternative composition in SCM: it is not always defined even if codomain of the first operand agrees with the domain of the second. A second phenomenon differentiating instantiation from composition is that the codomain of the whole operation may differ from the codomain of the second operand. This implies that instantiation is not associative.

Despite not having the pleasant properties of composition, instantiation has a profound programming meaning. While composition corresponds to non-transparent functor application [8], where arguments are used as tools only, instantiation corresponds to transparent application [13], where arguments specialize the output signature of the functor.

The trimming operation performs a coercion of a module to a given signature. If necessary, some type and value components of an operand module are removed, and only those mentioned in the operand signature remain. Ideally trimming should be absent from the language, but it is indispensable when we want to present an instantiated module as if it had not been instantiated. Moreover, when a larger module is used in a role of a smaller one, trimming relieves the user from writing the interface module.

3.4 Linking

The linking operation is the Dule standard way of supplying modules with arguments. It is defined (just as two other, simpler grouping operations, omitted here due to space constraints) using the already described operations of our module system. This witnesses the power of the system, as well as greatly simplifies proving properties of linking, in particular the proof of its well-definedness and the adequacy of its declared domains and codomains.

$$\frac{\begin{array}{c} m_1 : \{i_{k_1} \,:\, s_{k_1}\,;\,\ldots\,;\, j_1 \,:\, r_1\,;\,\ldots\} \to s_1 \\ \vdots \\ m_n : \{i_{k_n} \,:\, s_{k_n}\,;\,\ldots\,;\, j_n \,:\, r_n\,;\,\ldots\} \to s_n \end{array}}{\begin{array}{c} \texttt{link}\ \{i_1 = m_1\,;\ \ldots\,;\ i_n = m_n\} : \\ \{j_1 \,:\, r_1\,;\ \ldots\,;\, j_n \,:\, r_n\,;\ \ldots\} \to \{i_1 \,:\, s_1\,;\ \ldots\,;\, i_n \,:\, s_n\} \end{array}}\ \text{(linking)}$$

The domains of linking operands have to be product signatures, and needn't be strictly equal to each other. It suffices if there are equal components at the same labels. The domains may contain components not present in the domain of the whole linking expression but each of these has to be a codomain of one of the operand modules, indexed by the name of the module. These components will not be propagated from the domain of the linking operation, but will be the places at which compositions with other operands occur. Like in the module record operation, the codomain of the linking expression is just the product of codomains of all operand modules.

The linking operation eases the grouping of modules (already enabled by the module record operation) that alleviates the need for submodules, as known from conventional module systems. The linking operation fits well with our choice of sharing mechanism. The module product operation assumes that components with the same names are shared, while the linking operation assumes parameters are implemented by modules of the same names. Inside linking expression, supplying implementation of the parameters is automatically performed with multiple compositions and the user is free from writing the compositions by hand. There is also no need to artificially introduce submodules for the purpose of specifying their sharing with others; together with implicit composition this greatly reduces common modular bureaucracy. Linking facilitates naming parameterized modules and applying named modules, but the language remains first-order, readily compilable to the abstract machine language and its implementation avoids copying or storing the source code of modules.

References

1. Anya Helene Bagge, Martin Bravenboer, Karl Trygve Kalleberg, Koen Muilwijk, and Eelco Visser. Adaptive Code Reuse by Aspects, Cloning and Renaming. Technical Report UU-CS-2005-031, Institute of Information and Computing Sciences, Utrecht University, 2005.
2. E. S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(1):35–64, January 1990.
3. Dave Berry. Lessons from the design of a Standard ML library. *Journal of Functional Programming*, 3(4):527–552, October 1993.
4. Michel Bidoit, Donald Sannella, and Andrzej Tarlecki. Architectural specifications in CASL. *Formal Aspects of Computing*, 13:252–273, 2002.
5. G. Cousineau, P. L. Curien, and M. Mauny. The Categorial Abstract Machine. *Science of Computer Programming*, 8:173–202, 1987.
6. Karl Crary, Robert Harper, and Sidd Puri. What is a Recursive Module? In *SIGPLAN Conference on Programming Language Design and Implementation*, 1999.
7. P.-L. Curien. Categorical Combinators. *Information and Control*, 69(1-3):189–254, 1986.
8. Robert Harper and Mark Lillibridge. A Type-Theoretic Approach to Higher-Order Modules with Sharing. In *Proceedings of the ACM Conference on Principles of Programming Languages*, pages 123–137, Portland, Oregon, January 1994.
9. Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-Order Modules and the Phase Distinction. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, CA, January 1990.
10. Robert Harper and Benjamin C. Pierce. Advanced module systems: a guide for the perplexed. In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00)*, volume 35.9 of *ACM Sigplan Notices*, pages 130–130, N.Y., September 18–21 2000. ACM Press.
11. C. B. Jay. An introduction to categories in computing. Technical Report UTS-SOCS-93.9, University of Technology, Sydney, 1993.
12. Mikołaj Konarski. *Application of Category-Theory Methods to the Design of a System of Modules for a Functional Programming Language*. PhD thesis, MIMUW, 2007.
13. Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proc. 22nd symp. Principles of Programming Languages*, pages 142–153. ACM Press, 1995.
14. Xavier Leroy. The Objective Caml system: Documentation and user's manual, 2000. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Available from `http://caml.inria.fr`.
15. S. MacLane. *Categories for the Working Mathematician.* Springer-Verlag, 1971.
16. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.
17. John C. Reynolds. Polymorphism is not set-theoretic. In Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, Berlin, 1984. Springer-Verlag.
18. Claudio V. Russo. Non-dependent Types for Standard ML Modules. In *Principles and Practice of Declarative Programming*, pages 80–97, 1999.
19. D. Sannella and A. Tarlecki. Category theory. In *Foundations of Algebraic Specifications and Formal Program Development*, chapter 3. Cambridge University Press, to appear. See `http://wwwat.mimuw.edu.pl/~tarlecki/book/`.

20. Perdita Stevens. Experiences with the ML Module System, or, Why I Hate ML. Transparencies for a talk given to the Edinburgh ML Club and Glasgow Functional Programming group. `http://www.dcs.ed.ac.uk/home/pxs/talksEtc.html`, 1998.

Appendix: Abstract Machine Execution

Below, we abuse notation, writing multiplications by a functor, that is horizontal compositions with $C(c,e)$-identity, without the identity term constructor, as in $f * \texttt{<}i = u\texttt{; } \ldots\texttt{>}$, which is intended to mean $\texttt{T\_id}(f) * \texttt{<}i = u\texttt{; } \ldots\texttt{>}$. Our abstract machine executes machine code by combinator reduction as follows, where we list the reduction rules for functors first.

$$f \texttt{ . F\_ID}(d) \to f \tag{1}$$
$$\texttt{F\_ID}(c) \texttt{ . } g \to g \tag{2}$$
$$f \texttt{ . } (g_1 \texttt{ . } g_2) \to (f \texttt{ . } g_1) \texttt{ . } g_2 \tag{3}$$
$$\texttt{<}i \texttt{ : } f_i\texttt{; } \ldots\texttt{>} \texttt{ . F\_PR}(ld, i) \to f_i \tag{4}$$
$$f \texttt{ . <}i \texttt{ : } g\texttt{; } \ldots\texttt{>} \to \texttt{<}i \texttt{ : } f \texttt{ . } g\texttt{; } \ldots\texttt{>} \tag{5}$$
$$f \texttt{ . \{}i \texttt{ : } g\texttt{; } \ldots\texttt{\}} \to \texttt{\{}i \texttt{ : } f \texttt{ . } g\texttt{; } \ldots\texttt{\}} \tag{6}$$

These are the reduction rules for transformations.

$$f * \texttt{<}i = u\texttt{; } \ldots\texttt{>} \to \texttt{<}i = f * u\texttt{; } \ldots\texttt{>} \tag{7}$$
$$f * \texttt{T\_id}(g) \to \texttt{T\_id}(f \texttt{ . } g) \tag{8}$$
$$f * (u_1 \texttt{ . } u_2) \to (f * u_1) \texttt{ . } (f * u_2) \tag{9}$$
$$t * \texttt{F\_ID}(d) \to t \tag{10}$$
$$t * (f \texttt{ . } g) \to (t * f) * g \tag{11}$$
$$\texttt{<}i = t_i\texttt{; } \ldots\texttt{>} * \texttt{F\_PR}(ld, i) \to t_i \tag{12}$$
$$\texttt{T\_id}(f) * \texttt{F\_PR}(ld, i) \to \texttt{T\_id}(f \texttt{ . F\_PR}(ld, i)) \tag{13}$$
$$(t_1 \texttt{ . } t_2) * \texttt{F\_PR}(ld, i) \to (t_1 * \texttt{F\_PR}(ld, i)) \texttt{ . } (t_2 * \texttt{F\_PR}(ld, i)) \tag{14}$$
$$t * \texttt{<}i \texttt{ : } g\texttt{; } \ldots\texttt{>} \to \texttt{<}i = t * g\texttt{; } \ldots\texttt{>} \tag{15}$$
$$\texttt{T\_ID}(c) \to \texttt{T\_id}(\texttt{F\_ID}(c)) \tag{16}$$
$$\texttt{T\_PR}(lc, i) \to \texttt{T\_id}(\texttt{F\_PR}(lc, i)) \tag{17}$$
$$t \texttt{ . T\_id}(g) \to t \tag{18}$$
$$\texttt{T\_id}(f) \texttt{ . } t \to t \tag{19}$$
$$t \texttt{ . } (u_1 \texttt{ . } u_2) \to (t \texttt{ . } u_1) \texttt{ . } u_2 \tag{20}$$
$$\texttt{\{}i = t_i\texttt{; } \ldots\texttt{\}} \texttt{ . T\_pr}(lg, i) \to t_i \tag{21}$$
$$t \texttt{ . \{}i = u\texttt{; } \ldots\texttt{\}} \to \texttt{\{}i = t \texttt{ . } u\texttt{; } \ldots\texttt{\}} \tag{22}$$
$$f * \texttt{T\_pr}(i \texttt{ : } g\texttt{; } \ldots, j) \to \texttt{T\_pr}(i \texttt{ : } f \texttt{ . } g\texttt{; } \ldots, j) \tag{23}$$
$$f * \texttt{\{}i = u\texttt{; } \ldots\texttt{\}} \to \texttt{\{}i = f * u\texttt{; } \ldots\texttt{\}} \tag{24}$$
$$t * u \to (f * u) \texttt{ . } (t * h) \tag{25}$$

Polytopes & Polytypes: Generic Isosurfacing & Functional Programming
◇
(DRAFT IN PROGRESS)
◇

Colin Runciman[1], David Duke[2], Rita Borgo[2], and Malcolm Wallace[1]

[1] Department of Computer Science, The University of York, Heslington, York, YO10 5DD, UK,
{colin,malcolm}@cs.york.ac.uk
[2] School of Computing, University of Leeds, Leeds, LS2 9JT, UK,
{djd,rborgo}@comp.leeds.ac.uk

Abstract. Isosurface extraction is a fundamental tool in data visualization, usually implemented by a family of table-driven algorithms. We show how these algorithms can be realised as instances of a single generic scheme, which we implement in Haskell. The way the behaviour of the generic function depends on the spatial dimension and geometric structure of the dataset is richer than the type-structure dependencies exploited in current generic or polytypic programming systems, and presents several challenges. We discuss three inter-related issues: (1) representation types for polytopes and the extent to which they allow a combination of security and genericity; (2) the form of specification, based on predicates over higher-ranked sets, and the extent to which it supports both reasoning about correctness and feasible evaluation; (3) the necessary refinement methods to derive an efficient state-based implementation.

1 Introduction

Scalar fields arise in many branches of science and engineering, for example atmospheric temperature, gas density, height above sea-level, tissue density, or pressure over an airfoil. Data about such phenomena are typically collected, by measurement or simulation, as discrete samples within some space. One method used to understand such phenomena is to plot the points at which the field takes on certain values. Familiar 2D examples include contour lines on geographical maps, and isobar lines on weather charts. In 3D, contour lines become surfaces, generically referred to as *isosurfaces*, and higher-dimensional surfaces arise, for example, from time-varying phenomena and in branches of physics. In this paper we shall use the term "isosurface" whatever the dimension.

Our goal is a generic isosurfacing algorithm. One that can be applied, for example, to fields sampled over a 2D sampling grid, or a 3D grid of tetrahedra,

or a 4D grid organised as hypercubes, without the need to spell out, case-by-case, how to interpret different grid organizations. Reaching this goal is surprisingly hard as The dependency between the structure of the isosurface and the dataset is complex. To make progress we chose to write an executable specification in Haskell. We call this program *Polycell* and it is based on a set of axiomatic requirements characterising the required surface.

Section 2 of the paper gives a brief account of isosurface extraction through dataset traversal, and the fundamental dependency between the algorithm and the structural organization of the dataset. Section 3 shows that by using type classes to abstract over features of the dataset, much of the work to compute an isosurface can be expressed generically. However, by using type classes, we are requiring the implementor to provide certain critical type-specific information for each kind of dataset. We wish to define this information as a function of the dataset organization; in other words, we want a type-indexed function — or rather, a polytope-indexed function. Section 4 sets out some representation issues in Polycell. Section 5 describes our approach to specification, which makes extensive use of the test-and-generate paradigm. Although the specification is an executable prototype, it is inefficient. Section 6 describes an approach to systematic refinement of the specification, lifting tests into generators by re-expressing them as constraining state-machines. Section 7 discusses some related work on generic isosurfacing. Section 8 concludes by briefly summarising our achievements so far and the work that remains.

2 Polytopes and Isosurfaces

The starting point for visualization is the dataset. Data may be sampled on grids that are geometrically and topologically regular or irregular, resulting in a range of dataset organizations [1]. As isolated values do not lend themselves to visualization, it is usual to extrapolate between samples using an interpolation function [2]. But this is expensive. For isosurfacing, rather than interpolate the sample for every point required on the output, a widely used technique is to approximate the true surface by decomposing the physical space into geometric "cells" representing local neighbourhoods bounded by sample points. The only constraint on the structure of the cell is that it be a convex *polytope* — in 2D a convex polygon, in 3D a convex polyhedron, etc.

For each cell, the intersection (if any) between the cell edges and and a given isolevel can be found by interpolating between the samples at the edge end-points. Now the aim is to compute a set of simplices whose vertices are the edge-intersection points, and which approximate the true interpolant of the surface within the cell. In 2D, the simplices are straight line segments; in 3D, they are triangles; and in 4D, tetrahedra. Figure 1 illustrates isosurface extraction on a small 2D dataset: imagine that the values are heights above sea-level, and we are interested in the "5m" contour. Part (a) of the figure shows the original sample data, and part (b) the 'real' contour line.

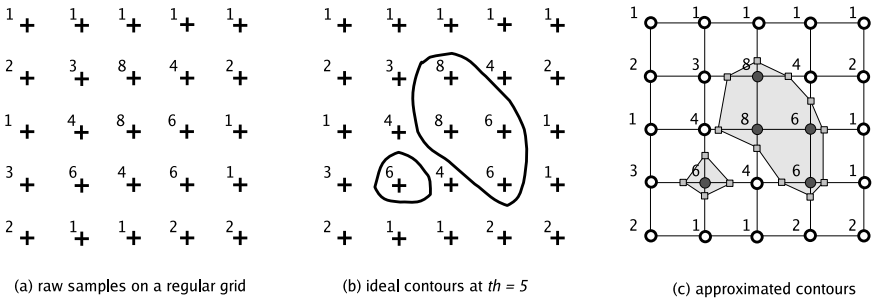


Fig. 1. Cell-based approximation of isolines

In part (c), the grid has been subdivided into square cells. Each cell vertex with a value above the required threshold (5m) has been marked with a filled circle; vertices below the threshold are marked with an empty circle. Considering each cell in turn, we look at edges which involve one marked and one unmarked vertex; these edges will be intersected by the contour at points found by interpolation and marked as a small square in the figure. In this simple 2D case, the simplices are just line-segments formed by joining appropriate pairs of edge intersections.

More generally, while the geometry of simplices approximating an isosurface depends on the threshold level and sample values of a cell's vertices, their topology depends only on the *pattern* of vertices above and below the threshold. This observation underlies the widely implemented *marching cubes* algorithm [3] for isosurface extraction. As traversal proceeds, each cell is inspected and the threshold compared against the sample at each vertex. The resulting boolean vector is then used to index a table of simplex templates, yielding a list of cell edges involved in each constituent simplex. Full tables for even low-dimensional cases are sizeable (e.g. 256 cases for the cube), but use of reflectional and rotational symmetries reduces this number (to 14 canonical cases for the cube). Historically such tables have been constructed *by hand*. Figure 2 gives the canonical table for square cells, and illustrates one simple case for the cubic cell, where the surface is approximated by two triangles.

Higher-dimensional cases are difficult to develop without automatic support, because of the size of the tables and the difficulty of identifying symmetries.

3 A Generic Pipeline

In a previous paper [4] we showed how the marching *cubes* algorithm can be expressed in Haskell as a fine-grained pipeline of simple stream operations, with encouraging performance on large-scale datasets. But our program was specialised to regular cubic grids. We have since been working to remove this constraint in a generic isosurface implementation, still preserving features of the lazy pipeline approach. A complete account of the generic pipeline is beyond the scope of

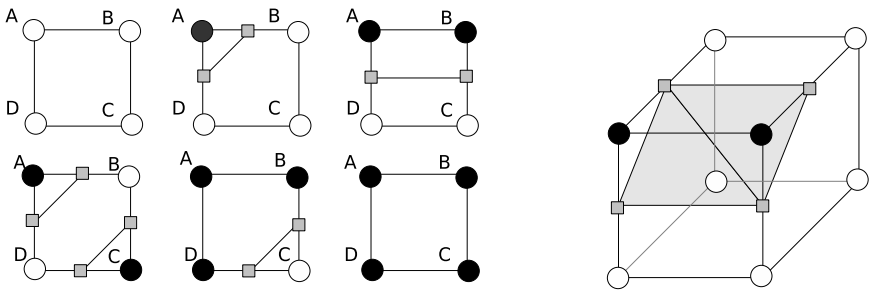


Fig. 2. Case table for squares, and a cube case.

this paper, but an outline will serve both to illustrate the informal introduction given in the preceding section, and to provide the context in which the Polycell program is intended to operate.

At the top level, a Haskell implementation of generic surface extraction is pleasingly concise. There is one function for each of the three conceptual operations: dataset traversal, cell processing, and simplex generation. This brevity is obtained in part through use of type classes and parametric polymorphism that abstracts away from details of dataset and cell organization; the following are all parameters to the function definitions:
`a`: the numeric type of sample values;
`v`: the type used to identify sample points/vertices;
`pos`: the numeric type underlying geometric coordinates;
`c`: the structure of cells;
`Dataset`: the organization of cells into datasets;
and the dimensionality of the space in which data is embedded.

Isosurfacing is a function that, given a sample threshold and a dataset of samples, embedded within some geometric space, and topologically organized into cell neighbourhoods, yields a list of simplices. Each simplex is defined by a list of coordinates in the same geometric space as the original samples.

```
isosurface :: (Real pos, Num a, Cell c v, Ord v, Geom g) =>
              a -> Dataset c v g a pos -> [[g a]]
isosurface th (DataSet coords_values) =
    concatMap (uncurry (mcube th)) coords_values
```

The output surface is the union of the simplex sets obtained from each cell, where each simplex set is generated by interpolating between end-point coordinates of cell edges intersected by the surface. To find these edges, the samples at cell vertices are compared with the threshold. From the resulting boolean structure a function `mc_case` determines the appropriate index by which to access the case table.

```
mcube :: (Real pos, Num a, Cell c v, Ord v, Geom g) =>
         a -> c v (g pos) -> c v a ->[[g pos]]
mcube th points values =
    map (map $ interp th points values) $ mc_case $ fmap (>th) values
```

Interpolation between vertex coordinates is standard, but a notable detail here is the use of the `Geom` class to abstract from the specific geometry in which the data are located.

```
interp :: (Real pos, Num a, Cell c v, Ord v, Geom g) =>
          a -> c v (g pos) -> c v a -> (v,v) -> g pos
interp thresh points values (v0,v1)
  = c0 .+. (p .*. (c1 .-. c0))
    where
      p       = realToFrac (thresh - sa) / realToFrac (sb - sa)
      sa      = select v0 values
      sb      = select v1 values
      c0      = select v0 points
      c1      = select v1 points
```

A full account of dataset and cell abstractions is beyond the scope of this paper. But note the `mc_case` function used within `mcube`. The signature of this function, defined in the `cell` class, is:

```
mc_case  :: c v Bool -> [[(v,v)]]
```

That is, given a cell of boolean values, the function returns a list of vertex identifier pairs, with each pair identifying an edge of the cell cut by the surface. For example, in the fourth of the square cases shown in Figure 2, the resulting entry would be

```
[[(A,B), (A,D)], [(B,C), (C,D)]]
```

that is, the intersection is approximated by two simplices, each of which is a line segment between points lying on two cell edges and that will be found by interpolating between the corresponding coordinates.

> In order to isosurface a dataset, we require an instance of the `Cell` class for the dataset's topology. And to do this, we need an implementation of `mc_case` for the cell.

In the remainder of the paper, we explore the development of a function that, given a polytope representing the topological structure of a cell, returns an appropriate implementation of `mc_case`. Eventually, this function will be generic with respect to the implementation of cell; we start however by developing an encoding of cell structure, and consider the specification of what constitutes a valid simplicial complex for a given cell marking.

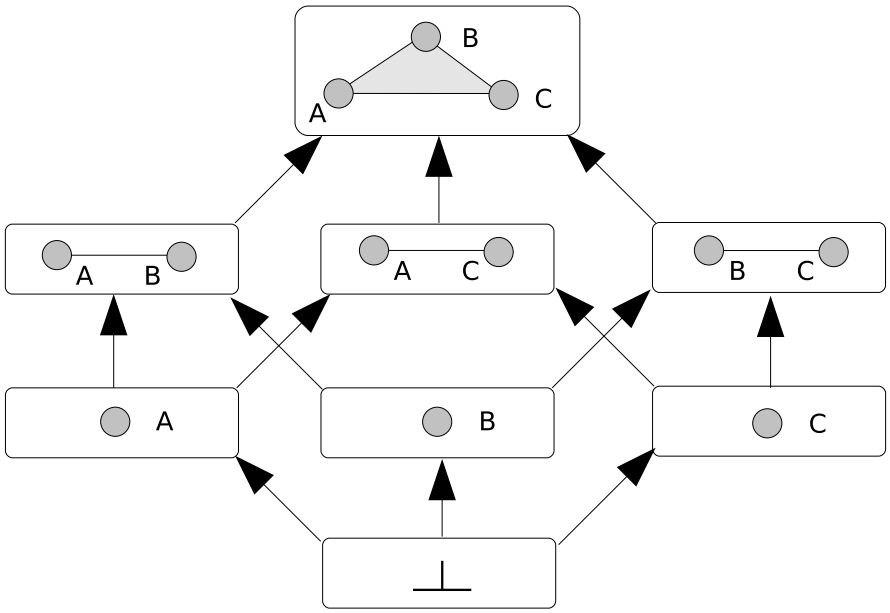


Fig. 3. The polytope lattice of the triangle ABC.

4 Representation: Polytopes and Polytypes

The text-book representation of an n-dimensional polytope [5] is a lattice of height $n + 1$. Elements at height $d + 1$ correspond 1-1 with the d-dimensional components of the polytope. The unique top element corresponds to the entire polytope, and the bottom element to the undefined polytope of dimension -1. For example, Figure 3 shows the lattice for a triangle ABC. As a triangle is also a simplex, for this example we could represent each element of the lattice by a set of vertices (with the empty set for $\perp$) and use the subset relation as the lattice ordering. But in general this will not do: a square ABCD, for example, does not have edges AC or BD. We therefore define:

```
data Poly = V {vertex: Vert}
          | F {facets: Set Poly}
```

This definition raises at least two issues: (1) how to treat dimensions, and (2) how to define sets.

Polytopes and dimensions

Just as lists of the same type may vary in length, so `Poly` data structures may have vertices at varying depths. Indeed, depths at which vertices occur could vary even within a single `Poly` value; there is nothing in the `Poly` type to say that n-dimensional parents may have only $(n - 1)$-dimensional children.

We can express the intended dimensional invariant in the definition of a boolean function.

```
dim :: Int -> Poly -> Bool
n `dim` V v  = (n==0)
n `dim` F fs = forAll fs $ \f -> (n-1) `dim` f
```

Now applications such as `(2 `dim` square)` helpfully record intentions that can be checked when the program runs, or in program properties tested using a tool such as QuickCheck. But dimensional properties are not checked statically, as we should prefer.

Class-level encoding?

The dependent typing of polytopes with dimensions can be "faked" [6] by re-defining natural numbers (for dimensions) and polytopes as *classes* and their alternative constructions as distinct types:

```
class Nat

data Zero = Zero
instance Nat Zero

data Succ n = Succ n
instance Nat n => Nat (Succ n)

class Poly c d | c -> d ...

instance Poly Vert Zero ...

data Facets c d = Facets (Set c)
instance Poly c d => Poly (Facets c d) (Succ d) ...
```

If the program can be reformulated to use this framework, and successfully typechecked, then run-time assertions about dimensionality can be superceded by static guarantees. However, in our experience this sort of class-level encoding is a tricky business. The necessary extensions to the type-system, such as multi-parameter type classes (MPCs) with *functional dependencies* [7], are subject to subtle restrictions. Some functions become unreadably obfuscated by this kind of encoding. Others exceed the limits of what the type-checker can do. It is possible, but adds yet further to the cost, to include *escape* methods mapping type classes and types down to the original dimensionless types and constructors, but there is no way back.

Dimension-carrying Phantom Types

A less sophisticated approach to type-checked dimensions uses a wrapper type with a *phantom* type parameter as the dimension.

```
newtype PolyD d = PolyD {poly :: Poly}
```

The `PolyD` constructor itself is hidden. Other functions for construction and deconstruction of `PolyD` values enforce the dimensional invariant. Natural number dimensions are again coded as types. For example:

```
mkV :: Vert -> Poly Zero
mkV = Poly . V

mkF :: Nat n => Set (Poly n) -> Poly (Succ n)
mkF = Poly . F . mapS poly

polyFacets :: Nat n => Poly (Succ n) -> Set (Poly n)
polyFacets = mapS Poly . facets . poly

polyVertex :: Poly Zero -> Vert
polyVertex = vertex . poly
```

As the dimensions are carried by a phantom type there is no explicit mention of dimensions in the value equations. The type-checker does the work.

Although some class-level encoding is needed for this approach too, it is less problematic. For example, take the `edges` function, defined on the dimensionless `Poly` type by case analysis of the dimension as computed by `dimOf`. Assuming that the `Poly` argument obeys the dimensional invariant, `dimOf` can use any branch to a vertex to compute its result.

```
dimOf :: Poly -> Int
dimOf (Vcell _)  = 0
dimOf (Fcell fs) = 1 + dimOf (minS fs)
```

The `edges` function is defined as follows. We can see that result must be a set of 1-dimensional edges, but this is not reflected in the type signature.

```
edges :: Poly -> Set Poly
edges p = case dim p of
          0 -> emptyS
          1 -> singleS p
          _ -> unionMapS edges (facets p)
```

We want a `PolyD` version of `edges` defined generically for all dimensions. As its behaviour depends on the dimension of the polytope, and dimensions are types, we need a type class. This class will have instances only for the two `Nat` types, but there are three cases — the behaviour for dimension `(Succ n)` depends on `n`. Our solution is an auxiliary method.

```
type One = Succ Zero
type Edge = PolyD One
```

```
class Nat n => PolyEdges n where
  polyEdges :: Poly n -> Set Edge
  polyEdges' :: Poly (Succ n) -> Set Edge

instance PolyEdges Zero where
  polyEdges  = const emptyS
  polyEdges' = singleS

instance PolyEdges n => PolyEdges (Succ n) where
  polyEdges  = polyEdges'
  polyEdges' = unionMapS polyEdges . polyFacets
```

Capturing such dimensional properties in the types of an n-dimensional application seems inherently desirable in principle. Questions in practice include: Is it possible? Is it convenient? Does it catch mistakes? Does it affect performance?

We have already indicated what is possible, and we do not find it too inconvenient, despite the need for devices such as the auxiliary `polyEdges'` method. If any part of a program should prove problematic for dimensional checking, a cheap escape route is always at hand: the selector `poly :: PolyD d -> Poly` takes us directly to the dimensionless world if we need to go there; it is quite straightforward to mix checked and unchecked components.

We have also found that dimensional checking catches mistakes. A routine for checking paths between transitional edges, as originally defined over the dimensionless `Poly` type, incorrectly passed polytopes of arbitrary dimension as the `edge` argument to the function:

```
transitional :: Marking -> Poly -> Bool
transitional marked edge = sizeS (verts e /\ m) == 1
```

The mistake went undetected as the program gave correct results for a few test polytopes. Happily, the correction of the mistake also made the program faster — see later discussion of cardinality.

As to performance, construction and selection for newtypes has a zero runtime overhead. What is more, explicit calculations of dimensions for `Poly` values are no longer needed as the equivalent work is done at compile-time.

Set Representations for Polytopes

Polytopes of positive dimension have *sets* of facets, where each facet is again a polytope. How shall we represent these sets? The main factors in this decision are support for needed operations, and the feasibility of executing the specification as a prototype. Mainstream Haskell libraries offer either raw (unordered) lists or a clever implementation using balanced ordered trees. An intermediate option would be to represent sets as ordered lists. Complexities of common set operations in each case are shown in Table 1. There appears to be no real contest. Surely we should use the balanced trees? Actually, it is *much* better to use ordered lists: Polycell runs about twenty times faster. Here is why:

Table 1. Complexity of set operations for different representations.

	lists	ordered lists	balanced trees
$\in$	$O(n)$	$O(n)$	$O(\log n)$
$\cup$, $\cap$, $\setminus$, $\subseteq$	$O(m \times n)$	$O(m+n)$	$O(m+n)$
$\sharp$	$O(n)$	$O(n)$	$O(1)$
min	$O(n)$	$O(1)$	$O(\log n)$

The tree-based library provides almost no support for *higher-ranked* sets. Sets are permitted as elements, but functions such as `powerset` are not directly provided. The cheapest way to define them in terms of available operations is by first extracting the list of elements and performing a list computation anyway. The complexity of extracting an element list from a tree is $O(n)$.

Almost every operation on sets involves `element comparison`. The complexities listed above assume that comparison is just an $O(1)$ primitive operation. But consider the comparison of (say) vertex sets from two components of a polytope: the common case is that the sets are not equal, and a comparison of just their least elements is sufficient to settle the outcome; now least element comparison is $O(1)$ for ordered lists, but $O(\log n)$ for trees.

Constraints on *cardinalities* occur quite a bit in the isosurface specification. So isn't the maintenance of cardinality information, with $O(1)$ access, an advantage of the tree-based library? No: providing an integer-valued cardinality requires a set to be fully evaluated, and is unnecessarily precise when the cardinality is only needed for comparison. Even if the comparison is an equality, it may be better to define and use a function such as:

```
forExactly :: Int -> Set a -> (a->Bool) -> Bool
```

The *speculative strict evaluation* in the balanced-tree set library is helpful if sets are typically long-lived structures that are frequently "updated" and eventually needed in full. But none of these assumptions is true for Polycell. Lazily evaluated ordered lists fare much better for transient sets about which only partial information is needed.

Lastly, what about *membership* tests? Surely here is the big advantage of balanced trees over lists? For some applications, yes. But although some sets in Polycell computations are quite large (a few hundred elements) these are typically generated sets in generate-and-test definitions, not arguments in membership tests. Polycell does perform a large number of membership tests and related operations such as structural inclusion. But the sets involved are comparatively small, so the advantage of $O(\log n)$ over $O(n)$ is less than might be expected. Overall, Polycell spends about a quarter of its time performing these operations.

5 Specification

The Polycell program is written as an *executable specification*. It is intended (1) as a reference to define correct tables of isosurface intersections for any marked polytope, (2) as a prototype for computing such tables, and (3) as the starting point for a more refined implementation.

The specification style uses set-based operators to express *generate and test* definitions of functions. For example, one generator from a given base set gives the set of all subsets of a specified size:

```
elemSubsetsOf :: Ord a => Int -> Set a -> Set (Set a)
```

Tests are applied by a set-filtering operator:

```
(<|) :: Ord a => (a->Bool) -> Set a -> Set a
```

Here's an illustration of these two functions used in combination. The context is an expression in the body of the `entryFor` function for a list of valid sets of simplices that form an isosurface within a marked cell:

```
entryFor :: Poly -> Marking -> Set Simplex
entryFor c m =
  ...
  [ validSimSet c m <| n `elemSubsetsOf` allValidSims c m
  | n <- [minNoOfSims..maxNoOfSims] ]
  ...
```

The details of `allValidSims` reflect a similar style — there are three further `<|` applications in the body of `allValidSims` itself representing more instances of generate-and-test. Test predicates are often expressed using quantifiers over sets, such as:

```
forAll, thereExists :: Ord a => Set a -> (a->Bool) -> Bool
forExactly :: Ord a => Int -> Set a -> (a-> Bool) -> Bool
```

Figure 5 illustrates the use of these functions to define `validSimSet`. The infix operator `<<=` is the containment relation on `Poly`.

6 Refining generate-and-test to state-space search

Our aim is a systematic refinement of the executable specification to an efficient program, giving confidence in its correctness. A formal derivation would be better still, but would be a demanding project in its own right. The main kind of refinement we have investigated transforms generate-and-test into a form of state-space search — we have a fast state-based implementation and our goal is a rational reconstruction of it.

```
validSimSet :: Poly -> Marking -> Set Simplex -> Bool
validSimSet c m ss =
  -- Simplex facets lying within a cell facet occur in
  -- exactly one simplex; others occur in exactly two.
  ( forAll simFacets $ \sf ->
      forExactly (if inCellFacet sf then 1 else 2) ss $ \s ->
        sf <<= simPoly s ) &&
  -- Each transitional edge is used at least once.
  ( forAll simVerts $ \sv ->
      thereExists simEdges $ \se -> sv <<= se )
  where
  inCellFacet = facial (facets c)
  simEdges    = unionMapS (faces . simPoly) ss
  simFacets   = unionMapS (facets . simPoly) ss
  simVerts    = transitional m <| edges c
```

Fig. 4. Predicates testing the validity of candidate simplex sets.

Filter promotion using state machines

The efficiency of generate-and-test routines can often be improved by promoting the test so that it is built into the generator. Instead of generating a set S of candidates and afterwards filtering out the subset S_P with property P, we generate only the S_P subset in the first place. Efficiency is gained at the expense of complexity as the S_P generator is typically more complex than either the S generator or the P test. Systematic refinement of this kind is a long-established strategy, sometimes called *filter promotion*. In our set-based specification, the generated candidates are typically subsets (of edges, simplices etc), so we seek some systematic and flexible way of constraining the generation of subsets.

Even in the prototype specification, one form of filter promotion has already been done. Instead of an expression such as `(size n <| subsets s)` we use a generator already constrained to give only subsets of a specified size:

```
elementSubsetsOf :: Ord a => Int -> Set a -> Set (Set a)
```

One view of the size value here is that it is an initial state. As each element is generated for a candidate subset there is a size-decrementing transition to a generator for the rest of the subset; finally, the empty subset is generated exactly when the size-state is zero.

Hence the following generalisation. We define a generator for just those subsets whose element sequences are recognised by a given finite-state acceptor.

```
constrainedSubsetsOf :: Ord a => FSA a s -> Set a -> Set (Set a)
```

The `FSA` type of accepting machines is defined as follows.

```
data FSA a s = FSA { start :: s,
                     trans :: (a -> s -> Maybe s),
                     final :: s -> Bool }
```

For example, we have the following embedding:

```
elementSubsetsOf n = constrainedSubsetsOf FSA {
  start = n,
  trans = \e n -> if n > 0 then Just e else Nothing,
  final = \n -> n == 0 }
```

Generator-constraining FSAs can replace tests quantified over the elements of candidate subsets. For example:

```
forallFSA :: (a->Bool) -> FSA a ()
forallFSA p = FSA {
  start = (),
  trans = \e () -> if p e then Just () else Nothing,
  final = \() -> True }

existsFSA :: (a->Bool) -> FSA Bool ()
existsFSA p = FSA {
  start = False,
  trans = \e b -> Just (b || p e)
  final = id }
```

Compound and predicates can be replaced by composite acceptors. Conjunctions by FSA products, for example:

```
(&&&) :: FSA a s1 -> FSA a s2 -> FSA a (s1,s2)
fsa1 &&& fsa2 = FSA {
  start = (start fsa1, start fsa2),
  trans = \e (s1, s2) ->
          case (trans fsa1 e s1, trans fsa2 e s2) of
          (Just s1', Just s2') -> Just (s1', s2')
          (_       , _       ) -> Nothing
  final = \(s1, s2) -> final fsa1 s1 && final fsa2 s2 }
```

If FSA constrained generators label each generated subset with its associated final state then predicates over nested sets of different ranks can also be combined.

We know that the *order* in which states are explored can be critical for the efficiency of state-based search, but we have not yet systematised ordering refinements. The element type in every set is required to be of class `Ord`, and our basic generators simply consider elements in this standard order.

7 Related Work

As scalar fields arise in many application domains, and isosurfaces are a fundamental tool for their exploration, there is unsurprisingly a large body of results concerning both strategies for surface extraction and, more fundamentally, different views on what constitutes the isosurface itself. However, work on case-table

generation is comparatively limited. Bhaniramka et.al. [8] set out an approach that computes simplices starting from a *geometric* model of the polytope. Their work exploits a connection between the cell-surface intersection and the convex hull of a set of points formed by the 'marked' (> threshold) vertices and the midpoints of those edges cut by the surface. Although the authors set out a proof of correctness, the connection between the implementation and the underlying mathematical model is not straightforward. More recently, Banks et.al. [9] have applied group theory to explore the size of case tables, not just for the boolean-valued cell/surface case, but for a range of related algorithms involving more complex vertex markings. Their work explains how reported differences in table size are accounted for by differences in the symmetry groups exploited in the construction process.

Although not discussed in our specification, we expect to make similar use of symmetries when defining case tables, and have already explored a particular branch of group theory, Coxeter groups, that provides elegant machinery for constructing many, though not all, of the polytope structures that commonly arise as cell topology in visualization data [10]. Putting aside issues of genericity, a feature of our approach is the clean separation between specification and implementation, and the audit trail, albeit informal, that the refinement strategy provides. Such an approach is unusual in visualization. To date, there is little published material exploring the correctness of visualization algorithms, with most work on novel techniques exploring the utility of the visual representation, and relying on visual inspection of the output and/or comparison with benchmark images to provide confidence in both the technique and its implementation. However, a recent major NIH/NSF report [11] notes the growing impact of visualization across a range of areas (in particular medicine), and the increasing need for novel and powerful techniques for exploring large volumes of data. The work reported here is an illustration of how development of a fundamental graphical technique can benefit from high-level programming constructs and methodology, in particular, as we ourselves have found, the use of expressive type systems to capture subtle errors.

A more fundamental study of isosurfaces is presented in the thesis of Carr [2]. Returning to the mathematics of fields, Carr explores the relationship between interpolation and various strategies for approximating the surface. Drawing on results from topology, he develops a model of contour evolution expressed through join/split graphs, and demonstrates how this can be used to construct particular case tables. This is an interesting and particularly sound, however it would be infeasible to implement for arbitrary polytopes as it relies on finding solutions to interpolation functions, and for higher-level interpolants the necessary closed form solutions do not exist.

8 Conclusions and Future Work

(Yet to be written!)

References

1. Haber, R., Lucas, B., Collins, N.: A data model for scientific visualization with provision for regular and irregular grids. In: Proceedings of Visualization'91, IEEE Computer Society Press (1991)
2. Carr, H.: Topological Manipulation of Isosurfaces. PhD thesis, University of British Columbia (2004)
3. Lorensen, W., Cline, H.: Marching cubes: A high resolution 3d surface construction algorithm. In: Proceedings of SIGGRAPH'87, ACM Press (1987) 163–169
4. Duke, D., Wallace, M., Borgo, R., Runciman, C.: Fine-grained visualization pipelines and lazy functional languages. Transactions on Visualization and Computer Graphics **12**(5) (2006) 973–980
5. McMullen, P., Schulte, E.: Abstract Regular Polytopes. Cambridge University Press (2002)
6. McBride, C.: Faking it — simulating dependent types in haskell. Journal of Functional Programming **12**(5) (2002) 375–392
7. Jones, M.P.: Type classes with functional dependencies. In: ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems, London, UK, Springer-Verlag (2000) 230–244
8. Bhaniramka, P., Wenger, R., Crawfis, R.: Isosurfacing in higher dimensions. In Ertl, T., Hamann, B., Varshney, A., eds.: Proceedings Visualization 2000, IEEE Computer Society Press (2000) 267–273
9. Banks, D., Linton, S., Stockmeyer, P.: Counting cases in substitope algorithms. Transactions on Visualization and Computer Graphics **10**(4) (2004)
10. Borgo, R., Duke, D., Runciman, C., Wallace, M.: Mathematical foundations for generic surfacing. In: Visualization 2007 Conference Compendium: Posters, IEEE Computer Society (2007)
11. Johnson, C.R., Moorehead, R., Munzner, T., Pfister, H., Rheingans, P., Yoo, T.S., eds.: NIH-NSF Visualization Research Challenges Report. 1st edn. IEEE Press (2006) http://tab.computer.org/vgtc/vrc/index.html.

Meta<Fun> – Towards a Functional-Style Interface for C++ Template Metaprograms*

Ádám Sipos, Zoltán Porkoláb, Norbert Pataki, and Viktória Zsók

Eötvös Loránd University, Faculty of Informatics, Dept. of Programming Languages
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
shp@inf.elte.hu, gsd@elte.hu, patakino@elte.hu, zsv@inf.elte.hu

Abstract. Template metaprogramming is an emerging new direction in C++ programming for executing algorithms in compilation time. Despite that template metaprogramming has a strong relationship with functional programming, existing template metaprogram libraries do not follow the requirements of the functional paradigm. In this paper we discuss the possibility to enhance the syntactical expressivity of template metaprograms using an embedded functional language. Clean – a general-purpose purely functional lazy programming language was chosen as embedded language. The graph-rewriting system of Clean has been implemented as a compile-time template metaprogram library using standard C++ language features. Lazy evaluation of infinite data structures is implemented to demonstrate the feasibility of the approach.

1 Introduction

Programming is primarily a human activity to understand a problem, make design decisions, and express our intentions to computer. In most cases the last step is writing code in a certain programming language according to its specific syntactical and semantical rules. Writing programs today is largely supported by automatic tools, but still considerably influenced by personal experiences, traditions, conventions, and customs. The syntax and semantics of a programming language is a major factor. It is not easy – although possible – to write structured programs in FORTRAN, or to write object-oriented code in PL/I. Such attempts, however, frequently lead to obscure and unmanagable code. As the spoken language has an important impact on human perception, the applied programming language may drive the style of the programs to be written.

Template metaprogramming is an emerging new direction in C++ programming in order to execute algorithms in compilation time. The relationship between C++ template metaprograms and functional programming is well-known: most properties of template metaprograms are very close to those we identify for functional programming paradigm. On the other hand, C++ has a strong heritage of imperative programming (namely from C and Algol68) influenced by

* Supported by GVOP-3.2.2.-2004-07-0005/3.0 and Stiftung Aktion Österreich-Ungarn, Pr.N.: 66öu2

object-orientation (Simula67). Furthermore the syntax of the C++ templates is especially ugly. As a result, C++ template metaprograms are often hard to read, and hopeless to maintain.

Ideally, the programming language interface has to match the paradigm the program is written in. *Meta<Fun>* is a running project at the Department of Programming Languages and Compilers at the Eötvös Loránd University, Budapest. The long-term goal of the project is to define and implement a clear and maintanable, purely functional-style interface for C++ template metaprograms. For this purpose, template metaprograms are written in a functional language and embedded in C++ programs. This code is translated into classical template metaprograms by a translator. The result is a native C++ program complies with the ANSI standard [3].

The *Clean* language has been chosen as an ideal embedded language. Clean is a general-purpose purely functional lazy programming language[11]. Clean's main features include a uniqueness typing system, higher order functions, and a powerfull constructor-based syntax for data structure generation. Clean also supports infinite data structures via delayed evaluation.

In this article we overview the most important properties of the functional paradigm, and evaluate their possible translation techniques into C++ metaprograms. The graph-rewriting system of Clean has been implemented as a C++ template metaprogram library. With the help of the engine, embedded Clean programs can be translated into C++ template metaprograms as clients of this library and can be evaluated in a semantically equivalent way. Delayed evaluation of infinite data structures are also implemented and presented by examples.

The rest of the paper is organized as follows: In section 2 we give a technical overview of the C++ templates and the basics of C++ template metaprograms. Relationship between C++ template metaprograms and functional programming is discussed in section 3. Lazy data stuctures, evaluation, and the template metaprogram implementation of the graph rewriting system of the Clean functional language is described in section 4. In section 5 the core engine of the Meta<Fun> system is discussed in detail. We discuss related works and future plans in section 6.

2 C++ Template Metaprogramming

In this section we give an overview of the most important technical details we use in the rest of the paper.

2.1 Typedef

Typedef is a C++ language tool to give a new name for an existing type. In this sense, typedef do not creates a new type, just provides an alias for an existing one. These alias types are compatible and interchangeable therefore parameter overloading is not possible on typedef created typenames.

Typedef is frequently used in C++ for convenience.

2.2 Template

Templates are key elements of C++ programming language [21]. They enable data structures and algorithms be parameterized by types thus capturing commonalities of abstractions in compile time without performance penalties in runtime [26]. *Generic programming* [19], a recently emerged programming paradigm for writing reusable components – most cases data structures algorithms – is implemented in C++ with heavy use of templates.

Consider a simple example regarding the implementation of a templated list data structure. The abstractions behind a list containing integer numbers, or strings is essentially the same, it is only the *type* of the contained objects that differs. With templates we can capture this abstraction, thus the generic language constructs aid code reuse, and the introduction of higher abstraction levels. Let us consider the following code:

```
template <class T>
class list
{
public:
    list();
    void insert(const T& x);
    T first();
    void sort();
    ...
};
```

```
int main()
{
    ...
    list<int> li;  // instantiation
    li.insert(1928);
}
```

This `list` template has one type parameter, called `T`, referring to the future type whose objects the list will contain. In order to use a list with some specific type, an *instantiation* is required. This process can be initiated either implicitly by the compiler when a list with a new type argument is referred, or explicitly by the programmer. During instantiation the template parameters are substituted with the concrete arguments, and the generated new code is compiled. Therefore a separate instance of code has to be generated for all type arguments of a template.

C++ template mechanism enables the definition of *partial* and *full specializations*. Let us suppose that for some type (in our example `bool`) we would like to create a more efficient type-specific implementation of the `list` template. We may define the following specialization:

```
template<>
class list<bool>
{
    // a completely different implementation may appear here
};
```

The specialization and the original template only share the *name*. A specialization does not need to provide the same functionality, interface, or implementation as the original.

With a partial specialization we can record one or more argument's types (like the `bool` in the full specialization) or their properties (like being pointer types):

```
template<class T, class U>
class A { ... };

template <class U>
class A<bool,U> { ... };
```

This partial specialization will be selected by the compiler if `A` is instantiated with its first argument being `bool`.

Another important unique property of C++ templates is that not only types but also integers, floating point numbers, even other templates may be template parameters:

```
template <class T>
class Q { ... };

template<template <class T1> class Expr>
class A { ... };

template<template <class T1> class Expr>
class A<Q<T1> > { ... };
```

In this example in the general case when the `template A` is used, its first version is instantiated. On the other hand when the `template Q` is passed to `A`, the second partial specialization is used.

The C++ template mechanism of instantiation is different from the implementation of *generics* in Java and C# languages. Java and C# work with *type erasure*; the argument of the generic is always converted to one common supertype (`Object` in Java) and all the generic data structures and methods are served by that single instance of code [4]. Thus the code blow problem is successfully avoided, but it is impossible to write specializations.

2.3 Metaprograms

In case the compiler deduces that in a certain expression a concrete instance of a template is needed, an implicit instantiation is carried out. Let us consider the following codes demonstrating programs computing the factorial of some integer number by invoking a recursion:

```
// compile-time recursion
template <int N>
struct Factorial
{
    enum { value = N *
```

```
// runtime recursion
int Factorial(int N)
{
    if (N==1) return 1;
    return N*Factorial(N-1);
```

```
      Factorial <N-1>::value };   }
};
template<>
struct Factorial<1>
{
    enum { value = 1 };
};
int main()                          int main()
{                                   {
    int r=Factorial<5>::value;          int r=Factorial(5);
}                                   }
```

As the expression `Factorial<5>::value` must be evaluated in order to initialize `r` with a value, the `Factorial` template is instantiated with the argument `5`. Thus in the template the parameter `N` is substituted with `5` resulting in the expression `4 * Factorial<4>::value`. Note that `Factorial<5>`'s instantiation cannot be finished until `Factorial<4>` is instantiated, etc. This chain is called an *instantiation chain*. When `Factorial<1>::value` is accessed, instead of the original template, the full specialization is chosen by the compiler so the chain is stopped, and the instantiation of all types can be finished. This is a *template metaprogram*, a program run in compile-time, calculating the factorial of 5.
A strong analogue exists between compile-time and runtime entities:

Metaprogram	Runtime program
(template) class	subprogram (function, procedure)
static const and enum class members	data (constant, literal)
symbolic names (typenames, typedefs)	variable
recursive templates, typelist	abstract data structures
static const initialization enum definition type inference	initialization *(but no assignment!)*

Table 1. Comparison of runtime and metaprograms

Conditional statements, like the stopping of recursions, are implemented with the help of specializations. Templates can be overloaded and the compiler has to choose the narrowest applicable template to instantiate. Subprograms in ordinary C++ programs can be used as data via function pointers or functor classes. Metaprograms are first class citizens in template metaprograms, as they can be passed as parameters for other metaprograms [7].

Data is expressed in runtime programs as constant values or literals. In metaprograms we use `static const` and enumeration values to store quantita-

tive information. Results of computations during the execution of a metaprogram are stored either in new constants or enumerations.

Complex data structures are also available for metaprograms. Recursive templates are able to store information in various forms, most frequently as tree structures, or sequences. Tree structures are the favorite implementation forms of expression templates [28]. The canonical examples for sequential data structures are `typelist` [2] and the elements of the `boost::mpl` library [33, 10].

2.4 Application of C++ template metaprograms

We have seen with the `Factorial` metaprogram that the whole operation happens in compile-time instead of runtime, thus this metaprogram may significantly slow the compilation process. On the other hand, this operation calculating a number's factorial results in a `O(n)` complexity in runtime. By replacing the calculation to compile-time, it will cause only a `O(1)` complexity in runtime. An important application of metaprograms is transferring calculations to compile-time, thus speeding up the execution of the program. Among other important applications of metaprograms are the implementation of *concept checking* [32] (testing for certain properties of types in compile-time), data structures containing types in compile-time (e.g. *typelist* [2]), *active libraries* [6], and others. By enabling the compile-time code adaptation, C++ template metaprograms (TMP) is a style within the *generative programming* paradigm [7]. Template metaprogramming is *Turing-complete* [29], in theory its expressive power is equivalent to that of a Turing machine (and thus most programming languages).

Despite all of its advantages TMP is not yet widely used in the software industry due to the lack of coding standards, and software tools. A common problem with TMP is the tedious syntax, and long code. Libraries like `boost::mpl` help the programmers by hiding implementation details of certain algorithms and containers, but still a big part of coding is left to the user. Due to the lack of a standardized interface for TMP, naming and coding conventions vary from programmer to programmer.

In the following we describe the similarities between functional programming and TMP, and describe a functional interface aimed to help overcome these problems.

3 Metaprogramming and Functional Programming

Template metaprogramming is many times regarded as a pure functional language. The common properties include referential transparency (metaprograms have no side-effects) and the lack of variables, loops, and assignments.

One of the most important functional property of TMP is that if a certain entity (the aforementioned constants, enumeration values, types) has been defined, it will be immutable. There is no way to change its value or meaning. A metaprogram does not contain assignments. That is the reason why we use recursion and specialization to implement loops: we are not able to change the

value of any loop variable. Immutability – as in functional languages – has a positive effect too: unwanted side effects do not occur.

In our opinion, the similarities require a more thorough examination, as the metaprogramming realm could benefit from the introduction and library implementation of more functional techniques.

Two methods are possible for integrating a functional interface into C++: modifying the compiler to extend the language itself, or creating a library-level solution and using a preprocessor or macros. The first approach is probably quicker, easier, and more flexible, but at the same time a language extension is undesirable in the case of a standardized, widely used language like C++.

Our approach is to re-implement the graph-rewriting engine of the Clean language as a compile-time metaprogram library using only ANSI standard compliant C++ language elements. Thus our solution has numerous advantages:

- Separating the user written embedded code from the graph-rewriting engine, the embedded Clean code fragments can be translated into C++ template metaprograms independently.
- Since the engine follows the graph-rewriting rules of the Clean language as it is defined in [5], the semantic of the translated code is as close to the programmers intension as possible.
- As our solution uses only standard C++ elements, the library is higly portable.

4 Lazy Evaluation

As lazy evaluation is one of the most characteristic features of the Clean language, our research centers around lazy evaluation and its application in C++ template metaprograms. In order to better understand the functional programing – TMP connection, and the possibilities in lazyness, we modeled the purely functional lazy language *Clean* in TMP.

Clean programs are represented by an expression graph in the compiler. This graph is *rewritten* automatically in several phases in runtime. The main function expression on the right side of the *Start* symbol is evaluated.

One of the basic data structures in *Clean* is the list. The list in *Clean* can be regarded as a linked list[11], but in the following we will describe lists as if they were represented with a head element and the "rest", or "tail". The constructor handling these two parts of the list is called `Cons`. For example the list `[2,3,4]` can be written as `Cons 2 Cons 3 Cons 4 Nil`, with `Nil` representing the end of the list.

A *lazy* evaluation strategy means that *"a redex is only evaluated when it is needed to compute the final result"*[16]. This lazyness enables us to specify represented lists that contain an infinite number of elements, e.g. the list of natural numbers: [1..]. A classic example for the usage of lazy lists is the *Eratosthenes sieve* algorithm producing the first arbitrarily many primes.

For future use we define a constructor `EnumFrom` for an infinite list starting at a certain number. The list `[2..]` can thus be written as `EnumFrom 2` (or `[2..]`).

Of course to acquire the head element of a list we need to rewrite this expression to `Cons 2 EnumFrom 3` (or `[2,EnumFrom 3]`), i.e. this is a list containing `2` and `[3..]`.

In the following we present a simple *Clean* program calculating the first 10 primes. (The symbols `R1`..`R6` are line numberings)

```
(R1)   take 0 xs = []
(R2)   take n [x,xs] = [x, take n-1 xs]
(R3)   sieve [prime:rest] = [prime : sieve (filter prime rest)]
(R4)   filter p [h:tl] | h rem p == 0  = filter p tl
                                       = [h : filter p tl]
(R5)   filter p [] = []
(R6)   Start = take 10 (sieve ([2..]))
```

Clean follows the *left-right outermost* rewriting strategy. The first examined expression is the Start expression. *Clean* first tries to apply one of the rewriting rules to the examined expression by substituting the current parameters with the rule's parameters. If it does not succeed, every expression's arguments (subexpressions) are examined recursively starting from left to right. If any of the subexpressions can be rewritten, the evaluation returns to the outermost expression. The evaluation process terminates, when none of the rules is applicable to any of the subexpressions.

In our example the first examined expression is (F1). Since none of the rules' left sides takes the form of (F1), *Clean* examines the first argument in (F1). It is `10`, for which we have no rewriting rule, and this expression has no subexpressions either. The other argument `sieve (EnumFrom 2)` cannot be rewritten either. On the other hand the argument `EnumFrom 2` is `[2, EnumFrom 3]`, and this is the substitution that takes place.

As a rewriting rule has been applied, we return to the outermost expression which is now (F2). Again, neither this whole expression, nor its first argument can be rewritten, thus the second argument `sieve [2, EnumFrom 3]` is examined. (R3) can be applied here with `prime=2` and `rest=EnumFrom 3`.

The outermost expression now takes the form of (F4), which is the expression we return to. Now (R2) can be directly applied with `n=10`, `x=2`, and `xs=sieve (filter 2 EnumFrom 3)`.

```
(F1)   take 10 (sieve [2..])
(F2)   take 10 (sieve [2, [3..]])
(F3)   take 10 ([2, sieve (filter 2 [3..])])
(F4)   [2, take 9 (sieve (filter 2 [3..]))]
(F5)   [2, take 9 (sieve [3, filter 2 [4..])]
(F6)   [2, take 9 [3, sieve (filter 3 (filter 2 [4..]))]]
(F7)   [2, 3, take 8 (sieve (filter 3 (filter 2 [4..])))]
...
```

In order to simulate a lazy language's inner workings, this rewriting algorithm was implemented using TMP.

5 The Implementation of the Graph-rewriting Engine

In the following we present thru examples the method to transform a *Clean* program into C++ templates, and also the engine that carries out the actual rewriting and evaluation process in compile-time.

5.1 The sieve program

In Section 2.3 we have described the various language constructs available in metaprogramming. We now use `typedefs`, and types created from templates to represent the expressions. In this approach our example Start expression has the form `take<10,sieve<EnumFrom<2> > >`. Here `take`, `sieve`, and `EnumFrom` are all `struct templates` having the corresponding number and type of parameters.

We have described the C++ templates and specializations in Section 2.2. The graph rewriting process can be modeled with the C++ compiler's instantiation process. When a template with certain arguments has to be instantiated, the C++ compiler chooses the narrowest matching template of that name from the specializations.

Therefore the rules can be implemented with template partial specializations. Each partial specialization has an inner `typedef` called `right` which represents the right side of a pattern matching rule. At the same time the template's name and parameter list represent the left side of a pattern matching rule, and the compiler will choose the most suitable of the specializations of the same name. Following is the example for a `sieve` rule.

```
template <int prime, class ys>
struct sieve<Cons<prime,ys> >
{
    typedef Cons<prime, sieve<filter<prime, ys> > > right;
};
```

The `sieve` template has to parameters, an integer `prime` and a list `ys`. This template describes the workings of (R3) in our *Clean* example. In case a subexpression has the form `sieve<Cons<N,T> >` where `N` is an integer, and `T` is an arbitrary type, the previously defined `sieve` specialization will be chosen by the compiler as a substitute for the subexpression.

In the general case, however, a rule is not applicable. For example in (F1) during the evaluation process the previous `sieve` rule will be considered as applicable when rewriting the subexpression `sieve [2..]`. The problem is that the argument `[2..]` (`EnumFrom 2`) does not match the `sieve` partial specialization parameter list which is expecting an expression in the form `Cons<N,T>` with an integer N, and a type T. During the compilation the C++ compiler will instantiate the type `sieve<EnumFrom<2> >`. However this is a pattern matching failure which has to be detected. Therefore each function must implement a partial specialization for the general case, when none of the rules with the same name can be applied. The symbol `NoMatch` is introduced, which signs that even

though this template has been instantiated with some parameter `xs`, there is no applicable rule for this argument. `NoMatch` is a simple empty `class`.

```
template <class xs>
struct sieve
{
    typedef NoMatch right;
};
```

The previously introduced `filter` function's case distinction is used to determine in compile-time whether `x` is divisible by `p`, and depending on that decision either of the two alternatives is chosen as the substitution. The C++ transformation of `filter` utilizes `boost::mpl::if_c` for making a compile-time decision:

```
template <int p, int x, class xs>
struct filter<p,Cons<x,xs> >
{
    typedef typename
        if_c< x%p==0,
            filter<p,xs>,
            Cons<x,filter<p,xs> >
        >::type right;
};
```

The working of the transformed `EnumFrom`is similar to the one in *Clean*: if a rewriting is needed with `EnumFrom`, a new list is created consisting of the list's head number, and an `EnumFrom` and the next number.

```
template <int r>
struct EnumFrom
{
    typedef Cons<r, EnumFrom<r+1> > right;
};
```

All other functions in Section 4 can be translated into templates using analogies with the previous examples. In the following we present the metaprogramming engine carrying out the evaluation process using the previous templates.

5.2 The graph-rewriting engine

Until now we have translated the *Clean* rewriting rules into C++ templates, by defining their names, parameter lists (the rule's partial specialization), and their right sides. These templates will be used to create types representing expressions thus storing information in compile-time. This is the first abstraction layer. In the following we present the next abstraction level, that uses this stored information. This is done by the library's core, the `Eval struct template`'s partial specializations which evaluate a given expression.

Since the specialization's parameter is a template itself (representing an expression), its own parameter list has to defined too. Because of this constraint separate implementations are needed for the evaluation of expressions with different arities. In the following we present one version of `Eval` that evaluates expressions with exactly one parameter:

```
 1 template <class T1, template <class> class Expr>
 2 struct Eval<Expr<T1> >
 3 {
 4     typedef typename
 5         if_c<is_same<typename Expr<T1>::right,
 6                      NoMatch>::value,
 7         typename
 8             if_c<!Eval<T1>::second,
 9                 Expr<T1>,
10                 Expr<typename Eval<T1>::result>
11             >::type,
12             typename Expr<T1>::right
13         >::type result;
14
15     static const bool second =
16         !(is_same<typename Expr<T1>::right,NoMatch>::value &&
17          !Eval<T1>::second);
18 };
```

`Eval`'s working is as follows. `Eval` takes one argument, an expression `Expr` with one parameter `T1`. The type variable `T1` can be any type, e.g. `int`, a list of `ints`, or a further subexpression. This way `Eval` handles other templates. The return type `result` defined in line 13 contains the new rewritten subexpression, or the same input expression if no rule can be applied to the expression and its parameters.

When the template `Expr` has no partial specialization for the parameter `T1`, the compiler chooses the general template as described in Section 5.1. The compile-time `if_c` in line 5 is used to determine if this is the case, and the `Expr<T1>::right` is equal to `NoMatch`.

- If this is the case, another `if_c` is invoked. In line 8 `T1`, the first (and only) argument is evaluated, with a recursive call to `Eval`. The boolean `second` determines whether `T1` or any of its parameters could be rewritten. If no rewriting has been done among these children, `Eval`'s return type will be the original input expression. Otherwise the return type is the input expression with its `T1` argument substituted with `Eval<T1>::result`, which means that either `T1` itself, or one of its parameters has been rewritten. This mechanism is similar to type inference.
- On the other hand, if a match has been found (the `if_c` conditional statement returned with a false value), the whole expression is rewritten, and `Eval` returns with this new expression (line 12).

The aforementioned boolean value `second` is defined by each `Eval` specialization (line 15). It is the logical value signaling whether the expression itself, or one of its subexpressions has been rewritten.

The implementation of `Eval` for more parameters is very similar to the previous example, the difference being that these parameters also have to be recursively checked for rewriting.

As our expressions are stored as types, during the transformation process the expression's changes are represented by the introduction of new types. This transformation is the very same as with the *Clean* example, as the following types are created as `right` typedefs:

```
take<10,sieve<EnumFrom<2> > >
take<10,sieve<Cons<2,EnumFrom<3> > > >
take<10,Cons<2,sieve<filter<2,EnumFrom<3> > > > >
Cons<2,take<9,sieve<filter<2>,EnumFrom<3> > > >
Cons<2,take<9,sieve<3,filter<2,EnumFrom<4> > > > >
Cons<2,take<9,Cons<3,sieve<filter<3,EnumFrom<4> > > > > >
Cons<2,3,take<8,filter<3,filter<2,EnumFrom<4> > > > >
...
```

We have demonstrated the evaluation engine's implementation, and its working.

6 Related Work and Future Plans

Functional language-like behavior in C++ has already been studied. *Functional C++* (FC++) [24] is a library introducing functional programming tools to C++, including currying, higher-order functions, and lazy data types. FC++, however, is a runtime library, and our aim was to utilize functional programming techniques in compile-time.

The *Boost::MPL* library is a mature library for C++ template metaprogramming. Boost::MPL containes a number of compile-time data structures, algorithms, and functional-style features, like *Partial Metafunction Application* and *Higher-order metafunctions*. However, Boost::MPL were designed mainly to follow the interface of the C++ Standard Template Library. There is no explicit support for lazy infinite data structures either.

The programming presented in Section 5.1 still does not hide the implementation details from the user, and only resembles *Clean* syntax. One of the possible solutions to this problem is introducing macros for hiding these details. A probably better approach is the creation of a preprocessor which could translate the embedded *Clean*-code into templates, and the following compilation process would run the metaprogram.

The speed of the compilation also needs to be improved.

7 Conclusion

In this paper we introduced the Meta<Fun> project which enhance the syntactical expressivity of C++ template metaprograms. The general-purpose functional programming language Clean is used as an embedded language to write metaprogram code in a C++ host environment. The graph-rewriting system of the Clean language has been implemented as a template metaprogram library. Clean code fragments are translated into classical C++ template metaprograms as clients of the rewriting library. Lazy evaluation of infinite data structures is implemented to demonstrate the feasibility of the approach. Since the graph-rewriting library uses only standard C++ language features, our solution requires no language extension and is highly portable.

References

1. David Abrahams, Aleksey Gurtovoy: C++ template metaprogramming, Concepts, Tools, and Techniques from Boost and Beyond. Addison-Wesley, Boston, 2004.
2. Andrei Alexandrescu: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley (2001)
3. ANSI/ISO C++ Committee. Programming Languages – C++. ISO/IEC 14882:1998(E). American National Standards Institute, 1998.
4. Bracha, G., Odersky, M., Stoutamire, D., Wadler, P.: Making the Future Safe for Past: Adding Genericity to the Java Programming Language.
   ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), pp. 183-200. 1998. Vancouver.
5. T. H. Brus, C. J. D. van Eekelen, M. O. van Leer, M. J. Plasmeijer: CLEAN: A language for functional graph rewriting.
   Proc. of a conference on Functional programming languages and computer architecture, pp.364-384, Springer-Verlag, 1987
6. Krzysztof Czarnecki, Ulrich W. Eisenecker, Robert Glck, David Vandevoorde, Todd L. Veldhuizen: Generative Programming and Active Libraries, Springer-Verlag, 2000
7. Krzysztof Czarnecki, Ulrich W. Eisenecker: Generative Programming: Methods, Tools and Applications. Addison-Wesley (2000)
8. Ulrich W. Eisenecker, Frank Blinn and Krzysztof Czarnecki: A Solution to the Constructor-Problem of Mixin-Based Programming in C++. In First C++ Template Programming Workshop, October 2000, Erfurt.
9. Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, Jeremiah Willcock: A Comparative Study of Language Support for Generic Programming. Proceedings of the 18th ACM SIGPLAN OOPSLA 2003, pp. 115-134.
10. Björn Karlsson: Beyond the C++ Standard Library, A Introduction to Boost. Addison-Wesley, 2005.
11. P. Koopman, R. Plasmeijer, M. van Eeekelen, S. Smetsers: Functional programming in Clean, 2002
12. David R. Musser and Alexander A. Stepanov: Algorithm-oriented Generic Libraries. Software-practice and experience, 27(7) July 1994, pp. 623-642.
13. David R. Musser and Alexander A. Stepanov: The Ada Generic Library: Linear List Processing Packages. Springer Verlag, New York, 1989.

14. Brian McNamara, Yannis Smaragdakis: Static interfaces in C++. In First C++ Template Programming Workshop, October 2000, Erfurt.
15. Odersky, M., Wadler, P.: Pizza into Java: Translating theory into practice In Symposium on Principles of Programming Languages, pp.146-159. ACM, 1997.
16. R. Plasmeijer, M. van Eeekelen: Clean Language Report, 2001
17. Jeremy Siek and Andrew Lumsdaine: Concept checking: Binding parametric polymorphism in C++. In First C++ Template Programming Workshop, October 2000, Erfurt.
18. Jeremy Siek and Andrew Lumsdaine: Essential Language Support for Generic Programming. Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation, New York, NY, USA, pp 73-84.
19. Jeremy Siek: A Language for Generic Programming. PhD thesis, Indiana University, August 2005.
20. Ádám Sipos: Effective Metaprogramming. M.Sc. Thesis. Budapest, 2006.
21. Bjarne Stroustrup: The C++ Programming Language Special Edition. Addison-Wesley (2000)
22. Bjarne Stroustrup: The Design and Evolution of C++. Addison-Wesley (1994)
23. Gabriel Dos Reis, Bjarne Stroustrup: Specifying C++ concepts. Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2006: pp. 295-308.
24. Brian McNamara, Yannis Smaragdakis: Functional programming in C++. Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, pp.118-129, 2000
25. Erwin Unruh: Prime number computation. ANSI X3J16-94-0075/ISO WG21-462.
26. David Vandevoorde, Nicolai M. Josuttis: C++ Templates: The Complete Guide. Addison-Wesley (2003)
27. Todd Veldhuizen: Using C++ Template Metaprograms. C++ Report vol. 7, no. 4, 1995, pp. 36-43.
28. Todd Veldhuizen: Expression Templates. C++ Report vol. 7, no. 5, 1995, pp. 26-31.
29. Todd Veldhuizen: C++ Templates are Turing Complete
30. István Zólyomi, Zoltán Porkoláb, Tamás Kozsik: An extension to the subtype relationship in C++. GPCE 2003, LNCS 2830 (2003), pp. 209 - 227.
31. István Zólyomi, Zoltán Porkoláb: Towards a template introspection library. LNCS Vol.3286 pp.266-282 2004.
32. Boost Concept checking. `http://www.boost.org/libs/concept_check/concept_check.htm`
33. Boost Metaprogramming library. `http://www.boost.org/libs/mpl/doc/index.html`
34. Boost Preprocessor library. `http://www.boost.org/libs/preprocessor/doc/index.html`
35. Boost Static assertion. `http://www.boost.org/regression-logs/cs-win32_metacomm/doc/html/boost_staticassert.html`

Speculative Inlining of Predefined Procedures in an R5RS Scheme to C Compiler

Marc Feeley

Dépt. d'informatique et de r.o., Université de Montréal, Canada

Abstract. The semantics of some dynamic programming languages, including Python, JavaScript, and R5RS Scheme, make it hard for a compiler to inline predefined procedures without compromising the semantics of the language. In the case of Scheme, many existing compilers can only achieve good execution speed by assuming that variables bound to predefined procedures are never mutated. This paper presents a *speculative inlining* approach which is portable and achieves good performance while fully conforming to the semantics of Scheme. It has been implemented in a mature Scheme to C compiler and we report on its performance on a large benchmark suite, both in execution speed and code size.

1 Introduction

Functional abstraction is useful for designing modular programs but the procedure call mechanism which implements the abstraction barrier has a run time cost for setting up parameters, directing the control flow to and from the procedure, and returning any results to the caller. The compiler can reduce the cost by *inlining* the procedure at the call site in the caller. This eliminates the need for the call/return control flow instructions, and it uncovers additional opportunities for optimization because the copy of the procedure body placed at the call site can be specialized for the actual parameters of that call.

We distinguish two kinds of inlinable procedures: *predefined procedures* provided by the language (e.g. `sqrt` and `map`), and *user procedures* whose definition must be given explicitly in the program. This classification covers language support operations, such as arithmetic, I/O, memory allocation and method dispatch, by treating them as inlinable predefined procedures. This paper addresses the problem of inlining predefined procedures in the R5RS Scheme language [12].

Some aspects of Scheme make inlining predefined procedures tricky. According to the semantics of Scheme the evaluation of `(+ x y)` decomposes into these steps: get the values of the variables `+`, `x`, and `y`, respectively t_1, t_2, and t_3, then check that t_1 is a procedure, and then call t_1 with the parameters t_2 and t_3. This usually adds `x` and `y` because the global variable `+` is initially bound to the addition procedure. During program execution it is possible to bind the variable `+` to a non-procedure value or to a different procedure, for example `(set! + list)`. After this assignment, the expression `(+ x y)` will in fact call the predefined procedure `list` and thus construct a two element list. This form of late

binding may be surprising, but it is sometimes useful as explained in Section 2. This problem is not specific to Scheme; indeed Python [15], JavaScript [2] and other scripting languages implement this form of late binding.

Predefined global variables may be mutated at run time in any part of the program, at the read-eval-print loop, in modules loaded dynamically using the predefined `load` procedure, and in S-expressions constructed and evaluated at run time by the predefined `eval` procedure. To achieve a better static analysis of programs, a Scheme compiler could adopt a static linking model by forbidding dynamic loading and `eval`, and not offering an interactive read-eval-print loop. This would allow a whole program analysis to prove that a given predefined variable is never mutated. Although this simplifies procedure inlining, it reduces the system's flexibility and it deviates from the Scheme semantics.

A popular alternative approach is the use of command-line flags and non-standard program annotations to force the compiler to assume the predefined variables contain their initial bindings. For example, the `(standard-bindings)` annotation of Gambit-C [3], the `--prim` flag of PLT Scheme's compiler [6], and the "benchmark mode" of Scheme48 [11]. This assumption is so common that it is the default compilation mode of CHICKEN [16], and the only compilation mode of Bigloo [14] which are both Scheme to C compilers.

Another aspect of Scheme which hinders the inlining of predefined procedures is the generic nature of fundamental operations such as `+` and `equal?`. Scheme supports a rich set of numerical types (infinite precision integers, rational, real, and complex) and the notion of exactness. Consequently most predefined procedures for performing arithmetic operations have non-trivial definitions which dispatch on the type and exactness of its arguments, type check the arguments, check for overflows, raise exceptions when appropriate, perform memory allocation, and so on. For space reasons it is unreasonable to fully inline arithmetic procedures. This is problematic because many programs need to do simple arithmetic on small exact integers for counting or indexing vectors.

To alleviate this problem many systems extend Scheme with a *fixnum* numerical type, which is a fixed-width exact integer type typically a few bits less than the natural word size of the machine, and procedures to operate on fixnums, such as `fx+` to add two fixnums, `fx<` to compare two fixnums, etc. Fixnum operations usually have a simple definition and they are fast because they do not require boxing and unboxing, and they have few special cases (overflow checking is typically the only special case). Some systems also provide a *flonum* numerical type, which is a fixed-precision inexact real type typically represented as a boxed machine floating point number. It is reasonable to inline fixnum and flonum operations because they take roughly the same space as a general procedure call.

The handling of fixnums and flonums varies considerably between systems, most notably in the width of fixnums and the handling of overflows (some systems detect fixnum overflows and signal an error, while others silently wraparound). For this reason it is difficult to write portable and fast programs even across systems that support fixnums and flonums. Moreover, fixnums cannot be used in applications where the computations result in bignum integers that exceed

the fixnum range even though most of the time the computations are within the fixnum range (financial calculations, number theoretic algorithms, etc).

A final concern in Scheme to C compilers, as considered here, is the high cost for implementing procedure calls. In order to correctly implement tail-calls the C code generated cannot directly translate Scheme calls into C calls. Moreover some compilation techniques, such as runtime code generation, cannot be used.

To address these problems we have designed a speculative inlining algorithm which fully obeys the semantics of R5RS Scheme and does not rely on any program annotations, although it can take advantage of annotations to further improve performance. This algorithm has been integrated into the Gambit-C Scheme to C compiler. With no annotations, some programs approach the speed of hand tuned code with annotations and fixnum/flonum specific operations.

This paper describes the speculative inlining algorithm, how it integrates into the Gambit-C Scheme compiler and its performance. Section 2 explains situations where mutation of predefined global variables is useful. Section 3 discusses aspects of Gambit-C's compiler which interact with the speculative inlining algorithm which is described in detail in Section 4. Finally Section 5 reports on the performance both in execution speed and code space.

2 Mutation of Predefined Global Variables

The ability to mutate predefined global variables is consistent with Scheme's minimalistic philosophy. Using a single namespace for procedures and values (a "Lisp$_1$" [8]) is conceptually simpler than using two namespaces (a "Lisp$_2$"). Disallowing mutation of predefined global variables would increase the language's complexity by adding a special class of global variables. Moreover, mutation of predefined global variables is useful, as shown in the following examples.

Debugging – Scheme programs are often structured as a set of procedures defined at top-level. These procedures are bound to global variables and each call references the appropriate variable to get the procedure to call. Tracing calls to these procedures can be done by setting the variable to a new procedure which calls the old one and also displays the arguments and result of the call. This can be achieved by defining and using a `trace` macro as shown in Figure 1 (a).

Defining new types – There are no constructs in R5RS Scheme to define new types. Portable programs must represent new types using a predefined type, usually vectors. New types defined this way are not distinct because they cannot be distinguished from other new types and the vector used for their representation. A common solution is to use a unique tag at the head of the vector to identify the type unambiguously from other new types. Moreover, the `vector?` type predicate must be redefined to distinguish plain vectors from vectors representing the new types. Figure 1 (b) shows how a 2D point type can be defined.

Overloading – Overloading of predefined procedures can be achieved easily with mutation. For example, `append` can be extended to allow concatenation of strings by setting the `append` variable to a procedure which either calls the `append` or `string-append` procedures depending on the type of the arguments.

```
> (define-syntax trace
    (syntax-rules ()
      ((trace var)
       (set! var (wrap 'var var)))))
> (define (wrap var proc)
    (lambda args
      (let ((r (apply proc args)))
        (write (cons var args))
        (display " ==> ")
        (write r)
        (newline)
        r)))
> (define (f n) (* (+ n 1) (+ n 2)))
> (trace f)
> (trace +)
> (f 10)
(+ 10 1) ==> 11
(+ 10 2) ==> 12
(f 10) ==> 132
132
```

(a) Debugging

```
(define old-vector? vector?)

(define (instance? obj tag)
  (and (old-vector? obj)
       (>= (vector-length obj) 1)
       (eq? (vector-ref obj 0) tag)))

(define pt (list 'pt)) ; unique tag
(define (make-pt x y) (vector pt x y))
(define (pt? obj) (instance? obj pt))
(define (pt-x p) (vector-ref p 1))
(define (pt-y p) (vector-ref p 2))

(set! vector?
      (lambda (obj)
        (and (old-vector? obj)
             (not (pt? obj)))))

(pt? (make-pt 11 22)) ⇒ #t
(vector? (make-pt 11 22)) ⇒ #f
```

(b) Defining new types

Fig. 1. Predefined global variable mutation examples

3 The Gambit-C System

3.1 System Architecture

The Gambit-C system [3] is an implementation of R5RS Scheme designed to be very portable while achieving good execution speed when programs are compiled.

A large part of the runtime system (roughly 50 kLOC), including the interpreter, debugger, bignum library, and all predefined procedures, is written in Scheme. The compiler is also written in Scheme (roughly 25 kLOC). The rest of the system (roughly 40 kLOC), including the garbage collector, operating system interface, and foreign function interface is written in portable C.

Many macros which abstract away from the specifics of the platform are defined in the `gambit.h` header file: the use of the `gcc` C compiler and its extensions, the machine's natural word size and endianness, the width of each numeric type, the definition of virtual machine instructions, the representation of objects, etc. This header file plays a key role in the compilation process. The C files produced by the Gambit compiler are entirely composed of calls to macros defined in `gambit.h`. This allows a very late binding of the behavior of the generated code. Indeed, a C file produced by the compiler on a given machine does not have to be changed when it is compiled on a machine with a different C compiler, a different operating system, or different endianness and word size (for practical reasons the word size is currently limited to 32 and 64 bits). Porting to an unconventional C compiler typically only requires small changes to `gambit.h`.

Gambit-C supports separate compilation. In particular the runtime system's Scheme code is contained in 9 modules which are separately compiled. The modules of the runtime system and of the user can be statically linked to form an

executable program. A running program can also load user modules dynamically and possibly more than once (to simplify debugging). Moreover, because the interpreter is written in Scheme [4], interpreted code and compiled code can freely call each other without compromising the Scheme semantics.

To implement tail-calls and continuations, Scheme calls cannot be translated directly into C calls. The runtime system manages a stack of Scheme continuation frames explicitly and independently from the C stack. The C code generated by the compiler is partitionned into a number of *host C procedures*. Depending on system build options, there is either a single host C procedure per Scheme module (*single host* mode) or one host C procedure per top-level Scheme procedure in the module (*multiple host* mode). Each host procedure contains a number of control points, which can either be procedure entry points or continuation return points. Trampolines are used to allow arbitrary jumps to a destination control point without C stack growth. Host procedures are only called from a dedicated *dispatcher* procedure. To jump to control point P, the current host returns to the dispatcher which then calls the host procedure containing P (i.e. the depth of the C call chain is never more than two). Upon entry to the host, a `switch` statement (or a computed `goto` if `gcc` is used), jumps to P in the host. There are a few optimizations to this basic approach which exploit locality (when P is in the same host). Regardless of the compilation mode and optimizations, calls to predefined procedures in the runtime are expensive because they necessarily require a non-local jump from the user program. The high cost of calling predefined procedures makes speculative inlining particularly attractive.

3.2 Extensions to Scheme

Gambit-C supports several extensions to R5RS Scheme. Some of the notable extensions are preemptive multithreading and a foreign function interface.

Low-Level Procedures Many low-level procedures meant primarily for the implementation of the runtime system are provided. These procedures typically perform a simple operation and are unsafe because they do not validate their arguments. For this reason, they are given an easily recognized name that is outside the standard Scheme identifier syntax (i.e. they cannot be found in an R5RS conformant program). These low-level procedures have names that begin with two hash signs. Here are a few examples:

- `##fx+` is the procedure which performs addition of fixnums. It does not check that the arguments are fixnums and whether there is a fixnum overflow.
- `##fx+?` is the procedure which performs addition of fixnums and checks for fixnum overflow. False (`#f`) is returned on overflow, otherwise the sum (a fixnum) is returned. It does not check that the arguments are fixnums.
- `##car` is a procedure which extracts a pair's `car` field. It does not check that the argument is a pair.
- `##cons` is the low-level procedure constructing pairs.

The duplication that occurs for `##cons` (which is identical to `cons`) and other low-level procedures is motivated by the need to easily distinguish internal low-level procedures from the procedures normally accessed by the user. This is convenient for the binding annotations explained in the next section.

User Annotations User annotations allow the programmer to force the compiler to assume certain properties about the code. This is useful when the programmer has knowledge that can help the compiler optimize the program. Annotations are specified inside the `declare` form. It is the programmer's responsibility to ensure that these annotations are correct; the compiler does not verify them. The `declare` form can appear anywhere a definition can appear. A `declare` at top-level has a lexical scope that extends until the end of the file. For a local `declare`, the lexical scope extends to the end of the enclosing binding form. Here is a typical use of user annotations:

```
(declare
  (standard-bindings) (extended-bindings) (block) (fixnum) (not safe))
(define z 0)
(define (iota n) (if (= n z) '() (##cons n (iota (- n 1)))))
```

The `(standard-bindings)` annotation asserts that a reference to a global variable predefined in R5RS will result in the corresponding predefined procedure. In other words in `iota` the calls to `=` and `-` will call the R5RS predefined procedures with those names. The `(extended-bindings)` annotation is similar but applies to Gambit-C extended procedures, such as `##fx+`, `##cons`, etc.

The `(block)` annotation asserts that all global variables defined in this file are only mutated in this file. Any global variable defined in a file and not mutated in that file can thus be treated like a constant. This enables constant propagation of global variables (e.g. replacing `z` in `iota` with 0), jump destination determination and inlining of user procedures defined at top-level (e.g. determining that the call to `iota` is a self-recursion).

The `(fixnum)` annotation asserts that all arguments and results of arithmetic procedures are fixnums. The `(not safe)` annotation tells the compiler that it is acceptable to generate unsafe code that could crash the program if some type checks fail. These two annotations in conjunction with the `(standard-bindings)` annotation allow the compiler to replace in `iota` the calls to `=` and `-` with unsafe calls to the fixnum specific procedures `##fx=` and `##fx-` respectively.

With carefully chosen annotations, programs can be made to run very fast, but at the price of safety. This is unacceptable in many situations. Moreover, annotations are brittle and are high maintenance. A small change in the program or in the dataset may invalidate the current set of annotations, but it is tedious and error-prone for the programmer to determine which ones.

In designing the speculative inlining algorithm, our goal was to improve the speed of execution without requiring that the programmer resort to annotations that are unsafe or otherwise change the semantics of the language (which includes all annotations described in this section).

3.3 Compiler Architecture

The compiler follows a conventional architecture. The source code is parsed and macros are expanded to produce an abstract-syntax tree (AST), which is transformed and annotated by subsequent passes. The AST is then traversed to generate the code for the Gambit Virtual Machine [5]. Low-level optimizations are then performed on this intermediate representation (dead code and common code elimination, instruction reordering, jump cascade removal, etc.) and finally it is expanded into C code in the form of calls to macros defined in `gambit.h`. The AST after all transformations is optionally pretty-printed as an S-expression.

The AST can represent expressions with no source code equivalent. Specifically, there is an AST node type representing procedure constants. These nodes are generated to refer to the procedure objects that exist at run time. Both predefined procedures and user procedures (but not closures) can be denoted. For example, the AST corresponding to this source code:

```
(let () (declare (standard-bindings)) (cons 11 22))
```

is transformed into an AST representing a call to a procedure constant denoting the predefined procedure `cons` (i.e. there is no longer a reference to the global variable `cons`). We use a box to denote procedure constants in the new AST:

```
(let () (declare (standard-bindings)) (cons 11 22)) → ('cons 11 22)
```

Note that $X \rightarrow Y$ will be used to mean "AST X is transformed into AST Y", where X and Y are external representations of ASTs possibly containing procedure constants. The different passes which transform the AST are briefly explained below. They are executed in the order of presentation.

Assignment Conversion This pass introduces cells for local variables (including parameters) that are mutated. An assignment to a local variable is replaced with a mutation of the corresponding cell. This simplifies the implementation of closures and continuations which share mutated local variables. The remaining passes can assume that local variables are never mutated.

Beta Reduction This pass performs simple beta reductions of the code. The following transformations are done.

- **Constant and copy propagation**: When it is known that a variable V is never mutated and V is bound to X which is either a constant or a variable that is never mutated, references to V are replaced with X. For example:
  `(let ((x 5)) (let ((y x)) (+ y y)))` → `(+ 5 5)`
  `(let () (declare (standard-bindings)) +)` → `'+`
- **Constant folding**: When a constant predefined procedure is called, and all the arguments are constants of the correct type, and the procedure does not have side-effects, the call is replaced by a constant equal to the compile-time application of the procedure to the arguments. For example:
  `('+ 5 5)` → `10`
  There are subtle semantic issues which hinder constant folding. Calls to predefined procedures which allocate their result (e.g. `list` and `append`) are

not constant folded because this would not preserve the uniqueness of the result (in the sense of `eq?`). Specifically, in Scheme:

`(eq? (list 5) (list 5))` $\neq$ `(eq? '(5) '(5))`

Because the target platform is not known at the time of the Scheme compilation, constant folding is tricky for procedures whose meaning is dependent on the target platform. This is specifically a problem for the fixnum operations because the width of a fixnum depends on the target machine's word size (30 bit fixnums on 32 bit machines, and 62 bit fixnums on 64 bit machines). An exact integer that does not fit in a 30 bit fixnum and that fits in a 62 bit fixnum is a bignum on 32 bit machines and a fixnum on 64 bit machines. So the `##fixnum?` procedure, which tests if its argument is a fixnum, is only constant folded when its argument is small enough to fit in a 30 bit fixnum or larger than fits in a 62 bit fixnum:

`('##fixnum? 123)` $\rightarrow$ `#t`
`('##fixnum? 1000000000000)` *is not constant folded*
`('##fixnum? 2305843009213693952)` $\rightarrow$ `#f`

Constant folding is also performed on conditional expressions, that is the `if`, `and`, and `or` special forms. For example:

`(if (and #f (f 2)) 123 (g 3))` $\rightarrow$ `(g 3)`

– **Inlining of user procedures**: When it is known that a given variable V is never mutated and V is bound to a lambda-expression, calls to V are replaced by calls to a copy of the lambda-expression. As an additional condition, the size of the new call (measured in number of nodes in the AST) must not be larger that a certain factor F of the size of the original call in the source code. By default F is 3, but the programmer can modify this with the `(inlining-limit` F`)` user annotation. For example:

`(let ((f (lambda (x) (+ x x)))) (f 5))` $\rightarrow$ `(+ 5 5)`

To improve the effectiveness of these beta reductions, processing generally starts at the leaves of the AST and progresses towards the root. For binding forms, the bound values are processed before the body. Finally, the top-level procedures are processed in the reverse order of their dependencies. If procedure `f` contains a call to procedure `g`, which contains a call to procedure `h`, then `h` is processed first, then `g`, and then `f`.

Lambda Lifting This pass transforms local user procedures using the lambda lifting transformation [10]. This eliminates the creation of closures for lambda-expressions bound to local variables when these local variables are only referenced in the operator position of calls. The lambda-expressions are modified so that they take their free variables as explicit parameters. All calls to these variables are also modified to pass the value of the free variables.

4 Speculative Inlining

Speculative inlining of predefined procedures is performed as an AST transformation pass just before assignment conversion.

4.1 Basic Approach

Our approach capitalizes on the high likelihood that predefined global variables contain the corresponding predefined procedure. When a predefined procedure is speculatively inlined, the inlined code must be guarded by a *run time binding test* to verify that the variable is indeed bound to the expected predefined procedure.

If the binding test fails, the inlined code is not appropriate and a normal procedure call using the global variable must be performed. For correct handling of tail-calls, this call must be a tail call with respect to the original call.

If the binding test succeeds, the inlined code is executed. In the ideal case this code will perform the work required of the predefined procedure and return the required result. It is possible however that the inlined code encounters an exceptional case, such as an argument of the wrong type, or a complex case that would be too space inefficient to handle inline (such as a fixnum arithmetic operation overflowing into the bignum range). We will call these conditions the *inlining conditions* of the procedure. When the inlining conditions do not hold the execution can fall back to a normal procedure call. We require that the inlined code only perform side-effects after verifying the inlining conditions. As an example, here is the speculative inlining of `car`:

```
(f (car (g 5))) → (f (let ((x (g 5)))
                       (if (and ('##eq? car 'car)
                                ('##pair? x))
                           ('##car x)
                           (car x))))
```

Falling back to a normal procedure call is not only correct, it ensures that the behavior of a call to a predefined procedure is the same, except for execution speed, whether the procedure is inlined or not: the same exceptions are raised, the same continuation is used, etc. The inlining is purely a compiler optimization that is transparent to the programmer.

4.2 Inlining Scheme's Numeric Procedures

The inlining of Scheme's numeric procedures is problematic because most numeric operations are generic, they can accept several numeric types, and can accept mixed types. In Gambit-C, there are five representations for numbers: exact integers are represented with fixnums and bignums, exact rationals are represented as pairs of exact integers, inexact reals are represented as flonums (64 bit IEEE 754 floating point number), and complex numbers are represented as pairs of reals. Except for fixnums, these representations are memory allocated.

If we take addition as an example, the algorithm for adding two numbers depends on the representation of the numbers to add. It is necessary to dispatch on the type of both arguments to determine how to proceed. In the Gambit-C runtime all 25 cases are laid out to avoid needless representation conversions.

It is unreasonable to inline this much code routinely. Instead, the most likely case must be handled inline, with the less likely cases handled by the fall back.

But what constitutes a likely case depends on the nature of the computation. There is a large class of algorithms which process small exact integers (e.g. counting and indexing vectors). On the other hand, scientific applications usually perform the bulk of their computations with inexact reals. The other numeric types (exact rationals and complex numbers) are less useful to handle inline in Gambit-C because the algorithms operating on them are complex and often require procedure calls (e.g. computing the GCD for normalizing rational results).

Consequently, there are two cases that are interesting to handle inline: when all the arguments are fixnums, and when all the arguments are flonums. A set of 5 user annotations is provided to allow the programmer to specify which case is most likely, and which cases to inline:

- `(mostly-fixnum)`: The fixnum case is more likely and is inlined.
- `(mostly-flonum)`: The flonum case is more likely and is inlined.
- `(mostly-fixnum-flonum)`: The fixnum case is more likely than the flonum case, but both are likely and are inlined. The fixnum case is checked first.
- `(mostly-flonum-fixnum)`: The flonum case is more likely than the fixnum case, but both are likely and are inlined. The flonum case is checked first.
- `(mostly-generic)`: The numeric procedures are not inlined.

These annotations are purely advisory; they do not compromise the Scheme semantics. Only the performance of the code is affected. The following example shows the speculative inlining of `+` when the user annotation `(mostly-fixnum-flonum)` is in effect:

```
(let () (declare (mostly-fixnum-flonum)) (f (+ (g 2) (h 3))))
→
(f (let ((x (g 2)) (y (h 3)))
     (if (’##eq? + ’+)
         (if (and (’##fixnum? y) (’##fixnum? x))
             (or (’##fx+? x y) (+ x y))
             (if (and (’##flonum? y) (’##flonum? x))
                 (’##fl+ x y)
                 (+ x y)))
         (+ x y))))
```

In the resulting AST, the `##fx+?` predefined procedure is used to perform the fixnum addition and overflow check. If the global variable `+` does not have its standard binding, or a fixnum overflow is detected, or the arguments are not both fixnum or both flonums, then a normal call to `+` is performed. The common code elimination optimization of the compiler will generate compact code by merging all three calls to `+`.

4.3 Inlining Recursive Procedures

The following recursive predefined procedures on lists are speculatively inlined: `assq`, `memq`, `map`, and `for-each`. Both `assq` and `memq` are worth inlining because many Scheme programs rely on them, their definitions are short, and they do

not need to call procedures that are not easily inlined (`assv`, `assoc`, `memv`, and `member` are not inlined because they call `eqv?` and `equal?`).

To avoid too much code expansion, the higher-order procedures `map` and `for-each` are only inlined when they are passed two arguments: the procedure argument and list. They are worth inlining not only because many Scheme programs rely on them but because the inlined code exposes optimization opportunities at the call to the procedure argument which can often avoid an expensive general call. If the procedure argument is a user procedure in the same file then a direct jump to the procedure can be performed (and without a parameter count if it does not take a rest parameter). The procedure argument is also a candidate for inlining, whether it is a user procedure or predefined procedure.

4.4 Interaction with Beta Reduction Pass

Implementing speculative inlining as an AST transformation has the advantage that subsequent transformations can further optimize the inlined code. In particular, the beta reduction pass may simplify the inlined code through constant propagation and constant folding. Consider a slight variation on the previous example, where the second argument to `+` is the constant 1. The speculative inlining of `+`, followed by constant propagation will give:

```
(let () (declare (mostly-fixnum-flonum)) (f (+ (g 2) 1)))
→
(f (let ((x (g 2)))
     (if ('##eq? + '+)
         (if (and ('##fixnum? 1) ('##fixnum? x))
             (or ('##fx+? x 1) (+ x 1))
             (if (and ('##flonum? 1) ('##flonum? x))
                 ('##fl+ x 1)
                 (+ x 1)))
         (+ x 1))))
```

The calls `('##fixnum? 1)` and `('##flonum? 1)` will then be constant folded to `#t` and `#f` respectively, allowing both `and`s and the `if` guarding the flonum case to be constant folded:

```
(f (let ((x (g 2)))
     (if ('##eq? + '+)
         (if ('##fixnum? x)
             (or ('##fx+? x 1) (+ x 1))
             (+ x 1))
         (+ x 1))))
```

The constant propagation transformation can also make use of user annotations to further improve the code. If we add the annotation `(standard-bindings)` to the previous example, the code at the end of the AST transformations will be:

```
(f (let ((x (g 2)))
     (if ('##fixnum? x)
         (or ('##fx+? x 1) (+ x 1))
         (+ x 1))))
```

5 Experimental Results

To evaluate the effectiveness of the speculative inlining approach, we compiled several Scheme benchmarks using the Gambit-C compiler with various user annotations. We are interested in measuring the impact of our approach on the execution speed and also on the code size. We used Gambit-C version 4.0 beta 20 with `gcc` 4.0.2 on a 64 bit, 2.2 GHz AMD Athlon running Linux. To measure the code size we measured the size of the machine code generated by the C compiler and substracted the machine code size for an empty program (note that we did not measure the size of the program's data because it could not easily be isolated from the data of the Gambit-C runtime). We also ran the benchmarks with Bigloo 2.8c to compare the execution time with a high-performance Scheme compiler. The Bigloo compilation mode assumed that all predefined global variables were not mutated and did not check for arithmetic overflow.

The benchmarks contain the Gabriel suite [7] and other programs representative of typical Scheme applications. There are 36 benchmarks in all. The largest are: `scheme` (Scheme interpreter in Scheme, 1 kLOC), `slatex` (Scheme to LaTeX formatter, 2.3 kLOC), and `nucleic` (scientific application [9], 3.5 kLOC).

Given our goal of achieving the best execution speed without compromising the Scheme semantics, one set of trials avoided the `(standard-bindings)` annotation, but we tried each of the numeric user annotations: `(mostly-fixnum)`, `(mostly-flonum)`, etc. The `(mostly-fixnum-flonum)` case is used as the baseline because it corresponds to the default when the programmer does not supply any user annotations. Another set of trials was done with those user annotations combined with `(standard-bindings)`. This is useful to evaluate the cost of the run time binding test.

We also tried the benchmarks with the set of user annotations that achieve the best speed. That is the `(not safe)`, `(block)`, and `(standard-bindings)` user annotations were used, in addition to benchmark specific annotations for arithmetic (either `(fixnum)` or `(flonum)` as appropriate for the benchmark).

In addition, a trial was done with the speculative inlining transformation disabled. This situation approximates the Gambit-C compiler before the speculative inlining transformation was added. This trial and those using speculative inlining are the only ones which do not violate the Scheme semantics.

The results are given in Table 1. For each combination of benchmark and compilation mode, the execution time and the code size are given (the code size is underlined). Lower values are better. To simplify comparison, all measurements are relative to the baseline (i.e. with `(mostly-fixnum-flonum)` but without `(standard-bindings)`). A value of 1 means the same time or space as the baseline. For space reasons the columns for the baseline are omitted since they contain 1 everywhere for time and space. Moreover we omit the columns for `(mostly-flonum-fixnum)` because the time and space were within a few percent of the columns for `(mostly-fixnum-flonum)`.

By examining the "Inlining disabled" column we see that the benchmarks always execute faster with speculative inlining than without. On average it is 5.18 times faster, but in several cases it is more than 10 times faster. The code

size is on average 2.7 times larger when speculative inlining is used, and up to 8 times larger. We have noticed that a significant part of the code expansion is due to the user procedure inlining which is more aggressive when predefined procedures have been inlined. Overall we view these results positively since among the compilation modes which do not violate the R5RS semantics, speculative inlining is consistently faster while the code size is typically not unreasonably large.

If we now compare the `(mostly-fixnum)` and `(mostly-flonum)` modes with speculative inlining we see that the execution time is better for the `(mostly-fixnum)` case in general but worse on benchmarks which are floating point intensive (`fibfp`, `mbrot`, `nucleic`, `pnpoly`, `sumfp` and `ray`). The ratio can be up to 8 times in favor of `(mostly-flonum)` for `sumfp` and up to 23 times in favor of `(mostly-fixnum)` for `sum` (the same computation as `sumfp` but performed using small integers). In terms of code size the `(mostly-flonum)` case is normally better, by about 9% on average. This is probably due to the absence of an overflow check when operating on flonums.

Interestingly, the `fft` benchmark, which uses a mix of operations on fixnums and flonums is about the same speed with `(mostly-fixnum)` and `(mostly-flonum)`. This is explained by the fact that there is an equal number of fixnum and flonum operations, so the same number of non-local jumps result whether the fixnum case or the flonum case is inlined. The execution speed improves by a factor of over 4 when `(mostly-fixnum-flonum)` or `(mostly-flonum-fixnum)` are used. Considering all benchmarks these compilation modes give the best execution speed; 1.39 times faster than with `(mostly-fixnum)` on average and 2.32 times faster than with `(mostly-flonum)` on average. The `(mostly-fixnum-flonum)` mode gives marginally better performance which is why Gambit-C uses it as the default compilation mode. The code size is consistently bigger than with `(mostly-fixnum)` and `(mostly-flonum)`, but by only 20-30% on average.

The cost of the run time binding tests can be evaluated by looking at the column for the `(standard-bindings)` + `(mostly-fixnum-flonum)` compilation mode. This mode generally yields code that is faster than the baseline, by about 15% on average. This mode also generally yields more compact code than the baseline, by about 50% on average. Our view is that this is an acceptable cost for the run time binding tests which are required for conformance to the Scheme semantics.

The "Unsafe mode" column indicates that with hand tuned user annotations and unsafe code, programs can run considerably faster, in some cases 8 times faster than the baseline, but closer to a factor of 2 on average. Moreover the code is almost 5 times more compact because there remains very few procedure calls in the code (and consequently fewer return points, continuation frame allocations and setup, stack overflow checks, etc. which all contribute to the total code). This shows in our view that speculative inlining does not completely eliminate the need for unsafe user annotations when very high performance and compact code are required. However, speculative inlining does contribute to lessen the

urgency to resort to user annotations and promote a more maintainable coding style.

Finally, we can see that the performance of Gambit-C with speculative inlining is comparable to Bigloo's performance; about 17% faster than Bigloo on average and about 5% slower if we ignore the `call/cc` intensive benchmarks `ctak` and `fibc`.

6 Related Work and Conclusion

Inlining has been used in other dynamically-typed programming languages to improve performance. Most notable is the compiler for SELF [1], an object-oriented dynamically-typed programming language, which uses *message inlining* to speed up message sends by reducing the frequency of method lookups. On the first execution of the messsage send, a normal method lookup is performed to find the correct method to call based on the type of the receiving object. The message send is then *backpatched* to jump directly to this method and the type of the receiving object is saved. On subsequent message sends the method will be called if the type of the new receiving object is the same, otherwise the system reverts to a new method lookup and backpatch. *Selective recompilation* of the program is used when method definitions are changed. All of this requires a complex system architecture, the presence of the compiler in the runtime system, and runtime code generation. More recently the Java HotSpot VM [13] has used a similar inlining-with-recompilation approach and a complex runtime architecture.

Our approach is in comparison much simpler and can be applied in situations where runtime code generation is not an option (such as in compilers which generate C code, in memory constrained systems, and embedded systems where the code must be stored in read-only memory). With an extensive experimental evaluation using a mature Scheme system, we have shown that our approach can be used to correctly implement the R5RS semantics while achieving execution speeds comparable to other Scheme compilers which attain high-performance by violating the R5RS semantics.

Acknowledgements

This work was supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada.

References

1. Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* PhD thesis, Stanford, 1992.
2. ECMA. Ecma-262: Ecmascript language specification, 1999.
3. Marc Feeley. Gambit-C version 4. `http://www.iro.umontreal.ca/~gambit`.

4. Marc Feeley and Guy Lapalme. Using closures for code generation. *Computer Languages*, 12(1):47–66, 1987.
5. Marc Feeley and James S. Miller. A Parallel Virtual Machine for Efficient Scheme Compilation. In *Proc. of the ACM Symposium on LISP and Functional Programming*, pages 119–130, 1990.
6. Matthew Flatt. PLT MzScheme: Language manual. Technical Report PLT-TR05-1-v300, 2005. `http://www.plt-scheme.org/techreports/`.
7. Richard P. Gabriel. *Performance and Evaluation of Lisp Systems*. Series in Computer Science. MIT Press, Cambridge, Massachusetts, 1985.
8. Richard P. Gabriel and Kent M. Pitman. Technical issues of separation in function cells and value cells. *Lisp and Symbolic Computation*, 1(1):81–101, June 1988.
9. P. H. Hartel, M. Feeley, M. Alt, L. Augustsson, P. Baumann, M. Beemster, E. Chailloux, C. H. Flood, W. Grieskamp, J. H. G. Van Groningen, K. Hammond, B. Hausman, M. Y. Ivory, R. E. Jones, J. Kamperman, P. Lee, X. Leroy, R. D. Lins, S. Loosemore, N. Röjemo, M. Serrano, J.-P. Talpin, J. Thackray, S. Thomas, P. Walters, P. Weis, and P. Wentworth. Benchmarking implementations of functional languages with "Pseudoknot", a float-intensive benchmark. *Journal of Functional Programming*, 6(4):621–655, 1996.
10. Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proc. of the conference on Functional Programming and Computer Architecture*, pages 190–203, New York, NY, USA, 1985.
11. R. Kelsey and J. Rees. The Incomplete Scheme 48 Reference Manual, 1999.
12. Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.
13. Michael Paleczny, Christopher A. Vick, and Cliff Click. The Java HotSpot Server Compiler. In *Java Virtual Machine Research and Technology Symposium*. USENIX, 2001.
14. M. Serrano and P. Weis. Bigloo: a portable and optimizing compiler for strict functional languages. In *Proc. of the Static Analysis Symposium*, pages 366–381, 1995.
15. Guido Van Rossum. *The Python Language Reference Manual*. Network Theory Ltd., September 2003.
16. Felix L. Winkelmann. *CHICKEN - A practical and portable Scheme system*, 2005.

Table 1. Relative execution time and relative code size (underlined) for various compilation modes (baseline is speculative inlining and `(mostly-fixnum-flonum)`)

Program	Speculative inl. + fix		Speculative inl. + flo		`(standard-bindings)` + fix		`(standard-bindings)` + flo		`(standard-bindings)` + fix-flo		Inlining disabled		Unsafe mode		Big-loo
`boyer`	1.00	1.00	1.01	.99	.72	.55	.70	.54	.74	.56	8.57	.44	.37	.32	.30
`browse`	1.01	.91	1.03	.87	.84	.51	.83	.46	.86	.59	4.31	.32	.70	.17	.49
`cpstak`	1.01	.94	4.53	.83	.96	.77	4.43	.68	.95	.76	4.35	.58	.85	.42	2.77
`ctak`	1.01	.98	1.57	.94	.95	.81	1.53	.78	.95	.84	1.51	.74	.90	.95	60.72
`dderiv`	1.02	1.00	1.10	1.00	.91	.58	1.01	.58	.91	.58	2.99	.42	.74	.29	1.19
`deriv`	1.01	1.04	1.13	1.04	.88	.61	.96	.60	.87	.60	2.02	.38	.78	.28	1.41
`destruc`	1.03	.82	6.60	.59	.86	.47	6.38	.28	.84	.48	9.61	.21	.68	.09	.74
`diviter`	1.02	1.00	1.07	.87	.92	.61	.95	.50	.92	.61	4.99	.39	.80	.17	1.44
`divrec`	1.00	1.01	1.04	.81	.86	.62	.94	.45	.98	.63	3.75	.46	.62	.26	.99
`puzzle`	.96	.73	8.43	.63	.81	.52	8.20	.45	.84	.63	12.20	.27	.35	.10	1.76
`takl`	1.00	1.01	.99	.94	.55	.71	.58	.67	.65	.73	3.80	.60	.23	.33	.18
`triangl`	1.02	.87	7.28	.73	.87	.57	6.64	.46	.88	.54	10.50	.27	.53	.14	.75
`fft`	4.65	.65	4.30	.45	4.26	.48	4.18	.32	.92	.61	10.53	.13	.29	.06	1.42
`fib`	.98	.90	9.60	.68	.89	.73	9.39	.55	.93	.81	9.24	.43	.35	.28	.98
`fibfp`	6.02	.60	.98	.94	5.94	.56	.90	.87	.93	.92	5.93	.50	.66	.80	1.33
`mbrot`	6.29	.56	2.01	.50	5.97	.40	1.94	.34	.96	.60	7.05	.20	.56	.13	1.64
`nucleic`	2.57	.83	.92	.81	2.45	.65	.77	.66	.88	.76	3.29	.47	.16	.22	.69
`pnpoly`	6.86	.73	3.68	.55	6.62	.51	3.59	.42	.93	.63	12.44	.28	.22	.09	3.77
`sum`	.88	.85	20.61	.64	.68	.68	19.91	.51	.80	.86	19.97	.36	.12	.11	1.91
`sumfp`	8.29	.47	1.00	.95	8.12	.38	.96	.88	.96	.95	8.11	.35	.83	.19	1.79
`tak`	1.03	.96	7.19	.90	.85	.73	6.61	.65	.88	.75	6.55	.55	.43	.40	.74
`conform`	1.00	1.00	1.01	1.00	.86	.68	.87	.68	.87	.68	2.13	.54	.50	.42	.32
`earley`	.99	.84	2.98	.63	.89	.55	2.80	.38	.92	.57	4.94	.22	.74	.11	1.05
`fibc`	.99	.96	2.60	.96	.94	.83	2.52	.83	.94	.87	2.52	.70	.77	.63	21.91
`graphs`	.99	.92	2.32	.75	.88	.60	2.17	.56	.87	.64	4.71	.38	.52	.25	1.36
`lattice`	1.00	1.00	1.00	1.00	.81	.66	.80	.67	.81	.67	4.62	.51	.66	.49	1.19
`matrix`	1.02	.82	2.54	.71	.90	.54	2.38	.48	.89	.63	4.70	.33	.70	.25	.96
`maze`	.98	.82	3.02	.68	.93	.60	2.96	.51	.93	.66	4.25	.37	.25	.18	.41
`mazefun`	1.02	.97	5.76	.91	.85	.67	5.44	.66	.87	.71	8.05	.49	.48	.34	1.06
`nqueens`	.92	.76	6.91	.57	.65	.34	5.79	.35	.67	.38	8.68	.22	.44	.10	.92
`paraffins`	1.00	.51	1.03	.40	.91	.55	.93	.32	.91	.64	2.10	.12	.92	.06	1.28
`peval`	1.00	.94	1.06	.96	.74	.54	.77	.54	.74	.54	3.40	.38	.60	.26	.44
`primes`	1.00	.99	3.05	1.21	.92	.93	2.92	.75	.92	.90	4.96	.50	.75	.27	1.11
`ray`	3.35	.69	1.00	.73	3.38	.52	.93	.59	.94	.73	4.07	.37	.23	.23	.99
`scheme`	1.01	1.00	1.30	.97	.89	.73	1.18	.69	.89	.73	2.96	.60	.80	.39	.61
`simplex`	2.03	.70	3.56	.52	1.84	.48	3.43	.37	.84	.55	6.59	.24	.27	.09	.75
geom. mean	1.39	.84	2.32	.77	1.21	.59	2.09	.53	.87	.66	5.18	.37	.49	.22	1.17

Circuit Parallelism in Haskell Programs (Extended Abstract)

Andreas Koltes and John O'Donnell

University of Glasgow

Abstract. Digital circuit design offers several opportunities for speeding up programs. In some cases the results can be significantly faster than task parallelism. FPGA technology provides a cost effective way to incorporate parallel circuits into Haskell programs. We describe a flexible interface allowing straightforward FPGA programming for Haskell programs. Two different kinds of application are described: the use of functional units to provide fast function applications, and the use of data parallelism to provide active data structures.

1 Introduction

Task parallelism is a common programming model, where a computation is partitioned into smaller tasks that can be computed simultaneously, so that the overall execution time is reduced. Task parallelism in Haskell corresponds to the parallel evaluation of expressions, and is supported by a variety of mechanisms such as *par* and *seq*, strategies [7], and transactional memory [2]. Task parallelism is widely used because it fits well with multiprocessor architectures, including multicore processors with shared memory.

Data parallelism is an alternative programming model, where the main computation may be a single sequential thread, but parallelism is used to provide fast operations on data structures. Data parallelism has often been used with SIMD architectures, which have the ability to perform the same arithmetic operation on each element of a vector.

Most applications of data parallelism, especially on SIMD architectures, have been limited to simple iterations across dense arrays. It is often assumed that this is all that the model offers. However, it was shown as long ago as 1981 [3] that SIMD data parallelism can perform computations on irregular data structures, by representing nested structures in a flat sequence of cells. The essential techniques to make this work include flexible tree sweeps (alternatively, parallel scans). It is unfortunate that commercial SIMD architectures supported only the simplest kind of data parallel operations, for example by providing only first order operations like $(scanl((+) :: Int \rightarrow Int \rightarrow Int))$, $(scanl((+) :: Float \rightarrow Float \rightarrow Float)$, $(scanl((||) :: Bool \rightarrow Bool \rightarrow Bool))$, and so on, instead of providing the higher order *scanl*. The interesting data parallel algorithms require general polymorphic scans and sweeps—indeed, they fit perfectly with Haskell but not with Fortran.

Although task and data parallelism seem quite different, it is possible to implement either model using the other. Data parallelism is straightforward to implement on top of a system with parallel tasks, by spawning threads to operate on independent parts of an aggregate data structure. This is the approach taken in Data Parallel Haskell [1]. It is also possible to embed something like parallel tasks within data parallelism; this was already demonstrated in 1988 by a data parallel quicksort [4], and parallel combinator reduction, and the technique can be generalised [5].

Even more flexible than data parallelism is *circuit parallelism*, which makes use of the extraordinary degree of parallelism among logic gates. In some cases it is just as easy to design a circuit to solve a problem as to program a general purpose machine. For example, some applications require repeated rapid evaluation of large Boolean expressions. The specification of such an expression can be translated automatically into a circuit that will evaluate it much faster even than a data parallel machine.

Ideally, the Haskell programmer would have all of these forms of parallelism to choose from, and furthermore could decide whether to embed data parallelism within tasks (as in Data Parallel Haskell) or to run it directly on hardware (as in SIMD machines). Although SIMD architectures are no longer popular, one can now do even better: in many interesting cases it is possible to transform a data parallel algorithm into a digital circuit, and then to realise this by programming it onto an FPGA. The FPGA program can be obtained through a Haskell circuit specification [6]. In effect, we bypass the general programmable SIMD architectures, and go from the data parallel algorithm directly to circuit parallelism.

This paper describes a system that makes circuit parallelism available to Haskell programmers. In Section 2 we describe the software and hardware techniques needed to provide a flexible connection between a Haskell program and a digital circuit programmed into an FPGA. There are several ways to use this system for practical applications; Section 4 describes a simple set of functional units that provide fast implementations of pure functions that are made available to a Haskell program, while 5 shows how a simple data parallel algorithm can be implemented. The concluding section describes the status of the project.

2 Architecture of FPGA Interface

Figure 1 shows the overall architecture of the system. There are two main subsystems: a conventional processor running a Haskell program, and an FPGA that runs intensive computations using circuit parallelism.

In our prototype implementation, the FPGA is a small Altera chip on a board that has serial interface connector. In order to use this, we need software drivers for the serial interface. In the processor subsystem, this consists of a small C program that is called directly by a Haskell program through the foreign function interface; the C program controls the serial interface through the Windows API.

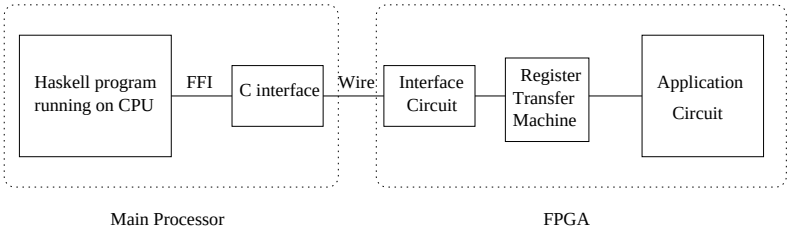


Fig. 1. Architecture of the system. A Haskell program running on a conventional CPU communicates with a parallel application circuit running on the FPGA. Interface software on the CPU and programmed into the FPGA complete the interaction.

In the FPGA subsystem, a UART circuit converts signals on the serial interface into data. This UART is programmed into the FPGA.

Beyond just a UART, the FPGA needs to provide a usable interface to the application circuit. Many issues need to be addressed in order to get a user-designed digital circuit to work with a processor. Our approach to this is to provide a programmable RISC-style register transfer machine which handles the gathering and transmission of data. This is a pure load/store architecture. It provides the idiom required by a processor—sending and receiving messages—as well as the idiom required by a digital circuit—a register file, with combinational inputs and outputs.

The register transfer machine consists of an *interface*, a *main pipeline*, and a *register file*, and it connects with the application circuit, viewing this as a set of *functional units*. Figure 2 shows an outline of the on-chip organisation of components.

The register transfer machine is designed to be flexible: the number and size of registers and their interaction with the circuit can all be configured. All core components of the platform are independent of the specific data types used in the application, as well as from the hardware platform. Although the prototype has a slow UART interface, the architecture is designed to work efficiently with a high-speed interconnection fabric, such as a HyperTransport channel.

The register transfer machine offers a general interface to an arbitrary application circuits. It can dispatch one operation to the application circuit in each clock cycle, and it can end one off-loaded operation at least every second clock cycle. During a clock cycle, the register transfer machine can supply to the application circuit up to three input values and one input flag vector.

The register file is configurable; it can have between 8 and 256 registers, with a data record size consisting of 1 to 256 32-bit words. In addition to the general register file, there is a configurable flag register file allowing for multiple streams of flag sensitive operations to be carried out in parallel.

The application circuit can have any architecture; its design is entirely up to the programmer. For example, it could be organised as a set of functional units, as a data parallel processor, or as a specialised dedicated circuit.

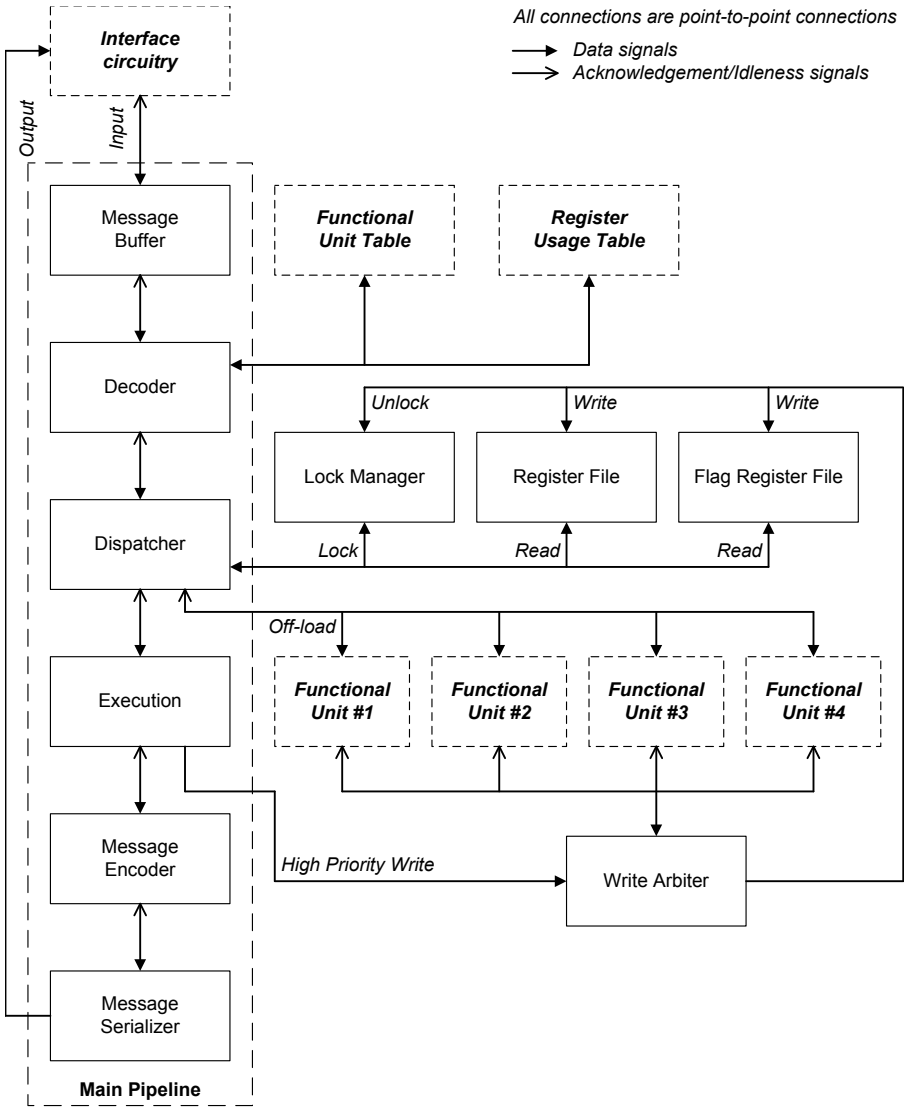


Fig. 2. Architecture of interface circuit programmed onto the FPGA

A common situation is an architecture consisting of a set of independent functional units. These could perform unrelated operations, as in a superscalar architecture, or they could be distinct portions of a data parallel architecture. The register transfer machine supports out-of-order execution within the functional units, along with reordering of received commands. It has a highly pipelined design allowing the FPGA hardware to run at high clock frequencies.

The framework is designed to operate together with high-performance interconnection fabrics like HyperTransport channels. For testing purposes, the use of slower interface options is possible, too. However, using these will limit the reachable performance of the system. A reference testbench using a serial interface based on an UART will be available. For test cases not requiring host interaction, it is also feasible to cache coprocessor commands in on-chip or on-board memory to simulate a high-speed data channel between FPGA and main CPU. Currently, the input and output modules of the framework process one byte each clock cycle. For operation with high-performance communication channels, these should be adjusted accordingly.

The core of the coprocessor consists of a six stage pipeline split into *message buffer*, *decoder*, *dispatcher*, *execution stage*, *message encoder* and *message serializer*. The decoder transforms the received command and data words into a signal vector and an input data record providing information for the dispatcher. The dispatcher performs all register file reads, enforces register locking and offloads user operations to functional units. The decoder stage uses the *functional unit table* and the *register usage table* to discover suitable functional units and required register slots.

'The only parts of the architecture that are actually aware of the data types held by the register file and the semantics of the flags in the flag register file are the functional units. This fact makes it easy to plug in specialised functional units handling application specific data types and operations.

Since the framework only handles locking and dependency checking of the register file a wide variety of data types ranging from integral values over vectors to complex types are possible. The flag register can be used to indicate and handle application specific conditions which are the result of an executed operation.

The interface provides a generic way to access registers as well as flags and also has limited support for primitive conditional operations to increase the operation throughput. However, since there is no support for branching operations, complex conditional behaviour still has to be controlled by the connected general purpose processor.

2.1 Portability

All components of the framework are designed to run across a wide variety of FPGA platforms. The only special feature required by the architecture is the availability of on-chip true dual port SRAM blocks which are accessible within one clock cycle. The pipelining of the architecture is aimed at allowing high

clock frequencies, in many cases up to the maximum frequency possible for the on-chip memory blocks.

Despite not being required for the core framework, implementations of the architecture benefit from available dedicated multiplexors included into the FPGA cells like they are available in many high-performance FPGA platforms.

2.2 Instruction Set

All commands sent from the main program to the coprocessor (FPGA) consist of a single 64-bit command word which is optionally followed by one or more 32-bit data words, depending on the requested operation. Command words are always expected to be sent to the coprocessor in big endian byte order (MSB first). The byte order of data words depends on the requested operation.

The highest bit of a command word specifies whether the requested operation is an I/O or core operation provided by the framework itself or whether it is a user operation being dispatched to a functional unit.

Communication between the FPGA and main processor is performed by load and store operations. The register transfer machine provides a variety of instructions that give general access to the register file, including to parts of registers. The design has provisions to make it easier to cope with compatibility issues, particularly with big-endian vs. little-endian data representations.

The partial register I/O instructions are useful to extract partial values from a register or to send small data amounts to the coprocessor. This is useful if the registers hold vectors or complex data types requiring only updates of single components. These instructions might also be useful to initialise registers with small values. Endianess is handled analogous to the full register I/O instructions. This means that the data word embedded within the command word can be sent in arbitrary byte order as long as its endianess is flagged correctly. There are instructions to operate on the flag registers, which can be copied, modified according to bit masks, as well as RISC style instructions that move register values.

2.3 User operations

The architecture allows up to 256 different types of functional units to be incorporated. It is possible to include multiple functional units of the same type to reach a higher degree of parallelism.

Each functional unit can support up to three input values, up to two output values as well as an input and an output flag vector allowing for a maximum degree of flexibility and parallelism. Further details about the implementation of functional units is given in the next chapter.

There are four different modes available for encoding commands to the functional units. Besides the register indexes, the commands contain the function code of the requested functional unit as well as a variety code. The function code is handled by the dispatcher whereas the variety code is sent to the functional unit as is. Please note that the maximum number of encodable functional

unit types having three input parameters is 8 and that there are restrictions regarding the number of supported variety codes. All variety codes whose encoding is shorter than 8 bits get zero-extended to a length of 8 bits before they are sent to the functional unit for processing.

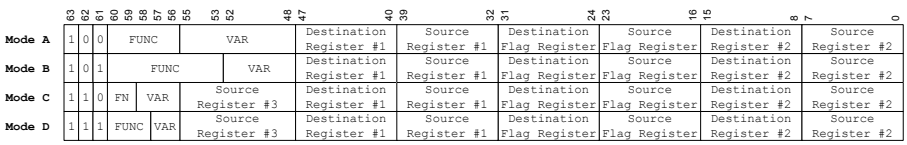


Fig. 3. Encoding of user instructions

3 Programming

From the perspective of the application, the circuit running in the FPGA is a coprocessor, analogous to a dedicated floating point or vector processing unit. Operations running on the coprocessor may in parallel with operations on the general purpose CPU, but data dependencies may force the main CPU wait for results from the coprocessor.

Despite its out-of-order execution capabilities, the coprocessor is guaranteed to behave as if all commands sent to it were executed strictly in the order they were sent to the unit without any parallelism. This is a standard correctness condition for superscalar architectures. As long as the outward sizes of the data types used are similar, it is possible to include functional units handling different or even polymorphic data types within the same application circuit.

A possible extension to the coprocessor, which may give significant performance improvements for some applications, is support for multiple separated command input and data output streams. This would be especially beneficial for multi-threaded or otherwise parallel applications in which multiple execution streams are operating based on the same data types. Using multiple command streams, the register files of the coprocessor could implicitly be partitioned into disjunctive register groups and each of the command streams could operate on one of the register groups. This way operations belonging to different execution flows on the main CPU could be spread over the available functional units and being processed in parallel. It would also be possible to further increase parallelism by including a write arbiter capable of writing multiple output data records to the register file within a single clock cycle.

4 Accelerating Pure Functions with Functional Units

In hardware, a functional unit is a circuit that calculates a function of one or more inputs, producing a result. An ALU does this as well, but ALUs normally

compute simple functions (e.g. binary addition) that can be performed within one clock cycle. Functional units are sequential circuits that take several clock cycles to produce a result.

A standard application of FPGAs is to provide functional units that perform calculations that would be slow in software. Most computers have hardware functional units for floating point, but some other operations (e.g. arithmetic for graphics) can be performed faster by hardware than software.

As an illustration of the technique, we have implemented a set of functional units that perform calculations on integers of a fixed but large number of bits. This is intended mainly intended as a reference implementation for demonstration and research purposes. It provides functional units acting as wrappers around library components provided by Altera. The functional units support generic integer lengths. However, since this package is developed to be used on comparatively small devices not having dedicated hardware multipliers, the performance of these functional units compared to a high-performance implementation is comparatively poor.

The package contains functional units to perform additions, subtractions, and comparisons, using either binary or two's complement representations, as well as a variety of basic bitwise logic operations. It allows operations on long integers through an externally provided carry bit read from the input carry flag.

The operations provided by this package require state to be retained over multiple clock cycles. In general, however, the state can be discarded at the end of an operation, so the functional units are calculating pure functions of their inputs. This can be used, in turn, to provide pure functions that are callable by the main Haskell program. However, it may be better to retain some state, such as carry flags, across multiple operations.

Functions implemented by FPGA programming will have significant overhead in the prototype implementation, mainly because of the extremely slow serial interface. However, our architecture is designed to work with high speed systems, where the FPGA would be able to communicate with the main processor at speeds comparable to memory accesses. Furthermore, the flexibility of FPGA programming would allow us to compute full arithmetic expressions that arise in a program, not just individual arithmetic operations. In such an environment, this system could provide a very useful and cost-effective benefit to performance.

5 Data Parallelism

A data parallel circuit has a state that holds an aggregate data structure, and it provides a set of operations that take inputs, update the state, and return results. (This differs from ordinary functional units, which may have internal state but which implement pure functions.) The development of a program using a data parallel circuit embedded in an FPGA proceeds as follows:

– The programmer begins by defining the data parallel state and a suitable set of operations.

– A digital circuit that implements the operations. is designed, with the specification written in the functional hardware description language Hydra [6]. Most programmers never think about designing circuits, but it is surprisingly easy, given suitable language tools. This step requires some knowledge of hardware and some thought, but it is essentially similar to ordinary functional programming.
– A netlist is extracted from the Hydra specification. A netlist is a precise description of the components in the circuit and how they are connected by wires. This step is automatic, and Hydra accomplishes it using an automated program transformation that maintains referential transparency. As a result, the netlist is guaranteed to correspond to the circuit specified by the programmer.
– The netlist is converted into the input language required by the FPGA. In our prototype, this is VHDL. Also, the parameters for the interface circuit are used to generate the VHDL for the interface. (These steps are currently manual, but work is in progress to automate them.)

Several nontrivial data parallel algorithms are described in the references. To illustrate how the implementation works, consider a minimal data parallel algorithm. The state is a vector of four registers, a, b, c, and d, each holding a binary number n bits wide. These are organised as a shift register, so that a shift right causes a data input x to be loaded into a, a to be loaded into b, and so on (these are parallel loads, so the data is shifted, and x is not replicated). There are four operations, selected by an operation code op: shift right, shift left, read d, and no operation. A circuit is easily defined:

```
shifter n op x = y
  where
    a = latch n (mux2 op x b a a)
    b = latch n (mux2 op a c b b)
    c = latch n (mux2 op b d c c)
    d = latch n (mux2 op c x d d)
```

This Hydra specification is now translated into VHDL, and the handshaking with the interface circuit is incorporated. Although VHDL is said to be a very high level language, the specification becomes much lengthier. Here is a small extract:

```
-- Registers
  process (clock, reset, newa, newb, newc, newd, newstate)
  begin
    if rising_edge (clock) then
      if reset='1' then
  -- handle reset, initialise registers
      else
        regidle <= newidle;
        CurrentState <= newstate;
```

```
  -- update the registers with their combinational inputs
        rega <= newa;
        regb <= newb;
         ...
      end if;
    end if;
  end process;

-- Combinational logic
   ...
  begin
    case CurrentState is
      when Operating =>
        if dispatch='1' then -- the inputs are valid
          case variety_code is
            when "00000010" => -- Shift right
              newstate <= Sending;
              newa <= data_input_1;  -- x
              newb <= rega;
              newc <= regb;
              newd <= regc;
              newidle  <= '0';
                ...
            when "00000100" => -- Shift left
              newstate <= Sending;
              newa <= regb;
                ...
            when "00001000" => -- Send regd value
              newstate <= Sending;
              newa <= rega;
                ...
              newdataoutreg <= regd;
                ...

            when others => -- includes No Operation, 00000001
                ...

          end case;
          else  -- inputs not ready
            newstate <= Operating;
            newa <= rega;
                ...
          end if;
      when Sending =>
      if data_acknowledge='1' then ...
```

```
        else ...
        end if;
    end case;
  end process;
```

The full paper will give more information about the interfacing, and if space permits a more substantive data parallel algorithm will be used.

6 Conclusion

We have implemented a prototype system along the lines described in Section 1. It consists of an interface between the Haskell program and the FPGA, a methodology for programming the FPGA, and a small set of examples. At the time of writing (August 2007) the pieces of the system have been developed, but some of the steps must be performed manually rather than automatically.

The FPGA hardware we are using is small and slow, so absolute performance is poor. The Altera chip has a limited number of logic blocks, and the interface circuit uses about 25% of them. The clock speed is less than 50MHz. However, the main limitation in the prototype is in the serial interface, which causes each transaction between the control processor and the FPGA to require on the order of a millisecond.

The approach we have demonstrated, however, should give excellent performance on modern production quality hardware. There are two suitable methods for connecting a processor to FPGA. The more flexible method uses separate chips that communicate at approximately the same speed as main memory accesses; this would also allow for various combinations of processor chips, memory chips, and FPGA chips. The faster method uses a processor and FPGA fabricated together on one chip, allowing for communication at cache speeds. These high performance FPGAs also offer very large arrays of logic elements, and high clock speeds.

However, the results do provide a proof of concept: data parallel programming via circuit design on an FPGA is a promising new technique for Haskell programmers seeking good performance on applications with intensive computational requirements.

References

1. Manuel Chakravarty, Roman Leschinskiy, Simon Peyton Jones, Gabrielle Keller, and Simon Marlow. Data parallel haskell: A status report.
2. Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions.
3. John O'Donnell. *A Systolic Associative LISP Computer Architecture with Incremental Parallel Storage Management.* PhD thesis, University of Iowa, Iowa City, 1981. Technical Report 81-5.

4. John O'Donnell. Functional microprogramming for a data parallel architecture. In *Proceedings of the 1988 Glasgow Workshop on Functional Programming*, pages 124–145. Computing Science Department, University of Glasgow, 1988.
5. John T. O'Donnell. Supporting tasks with adaptive groups in data parallel programming. *International Journal of Computational Science and Engineering (IJCSE)*, 1(2/3/4):86–98, 2005. Inderscience Publishers.
6. John T. O'Donnell. Overview of hydra: A concurrent language for synchronous digital circuit design. *International Journal of Information*, 9(2):249–264, March 2006.
7. Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.

On Implementing S-Net

A Typed Stream Processing Language — Draft —

Clemens Grelck[1,2] and Frank Penczek[1]

[1] University of Hertfordshire
Department of Computer Science
Hatfield, Herts, AL10 9AB, United Kingdom
{c.grelck,f.penczek}@herts.ac.uk
[2] University of Lübeck
Institute of Software Technology and Programming Languages
Ratzeburger Allee 160, 23538 Lübeck, Germany
grelck@isp.uni-luebeck.de

Abstract. S-Net is a declarative coordination language; it allows us to assemble asynchronous stream processing components into a stream processing network at a high level of abstraction. We sketch out the major design decisions in implementing S-Net on top of Pthreads for truly concurrent execution on contemporary shared memory multiprocessor and multicore architectures.

1 Introduction

The recent advent of multicore technology in processor designs [1] has introduced parallel computing power to the desktop. Unlike the increase in clock frequency characteristic for previous generations of processors, application programs do not automatically benefit from multiple cores, but require explicit parallelisation. This need brings parallel and distributed programming techniques from the niche of traditional supercomputing application areas into the main stream of computer science. However, the shift in application characteristics also demands new programming concepts, tools and infrastructure.

S-Net [2, 3] is a novel approach of a declarative coordination language based on the idea of stream processing. Asynchronously executing stream processing components constitute the basic building blocks of an S-Net. They are implemented in a separate compute language, which in principle can be any language suitable for the computational aspects of the problem to be solved that adheres to certain interfacing contracts. S-Net assembles these stream processing components into a stream processing network described by algebraic formulae and a type system that makes heavy use of structural subtyping of records.

2 S-Net

S-Net[3] describes the coordination behaviour of networks of asynchronous components and their orderly interconnection via typed streams. We deliberately restrict S-Net to coordination aspects and leave the specification of the concrete operational behaviour of basic components, named *boxes* in S-Net terminology, to a box implementation language. For the time being we focus on the functional array programming language SaC [4] as our primary box language as far as implementation issues are concerned. However, coordination and computation layer are sufficiently orthogonalised in our approach to support a range of box implementation languages, and the same S-Net may well contain boxes implemented in different box languages. This strict separation between computing and coordination layer facilitates the reuse of existing software and opens an avenue towards mixed language programming.

An S-Net box is connected to the outside world by two typed streams, a single input stream and a single output stream. Data on these streams is organised as non-recursive records, i.e. collections of label-value pairs. The operational behaviour of a box is characterised by a stream transformer function that maps a single record from the input stream to a (possibly empty) stream of records on the output stream. In order to facilitate dynamic reconfiguration of networks, a box has no internal state and any access to external state (e.g. file system, environment variables, etc.) is confined to using the streaming network. Boxes execute fully asynchronously: as soon as a record is available on the input stream, a box starts computing and, potentially, producing records on the output stream. The restriction to a single input stream avoids any confusion as whether to implicitly synchronise or how to respond to partial availability of data.

S-Net features two built-in components: Filter boxes take care of various kinds of housekeeping tasks that do not require the full power of a box language, e.g. deletion, duplication or renaming of record fields. Synchronisation boxes recombine multiple records on the input stream into a single record on the output stream based on structural pattern matching.

The construction of streaming networks based on instances of defined and built-in asynchronous components is a distinctive feature of S-Net: Thanks to the restriction to a single-input/single-output stream component interface we can describe entire networks through algebraic formulae. S-Net features four network combinators that take either one or two operand components and construct a network that again has a single input stream and a single output stream. As such a network again is a component, construction of streaming networks becomes an inductive process. We have identified a total of four network combinators that prove sufficient to construct a large number of network prototypes: static serial and parallel composition of heterogeneous components as well as dynamic serial and parallel replication of homogeneous components.

Structural subtyping on records greatly facilitates adaptation of individual components to varying contexts. More precisely, components only need to be

[3] http://snet.feis.herts.ac.uk/

specific about record fields that are actually needed for the associated computation or that are (at least potentially) created by that computation. In excess to these required fields, however, an input record to some component may have an arbitrary number of further fields. These additional fields bypass the component and are added to any outgoing record through an automatic coercion mechanism, named *flow inheritance*.

Acknowledgements

The development of S-Net is funded by the European Union through the Framework VI Integrated Project Æther[4], *Self-adaptive Embedded Technologies for Pervasive Computing Architectures.*

References

1. Sutter, H.: The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobb's Journal **30** (2005)
2. Grelck, C., Scholz, S., Shafarenko, A.: S-Net: A Typed Stream Processing Language. In Horváth, Z., Zsók, V., eds.: Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages (IFL'06), Budapest, Hungary. Technical Report 2006-S01, Eötvös Loránd University, Faculty of Informatics, Budapest, Hungary (2006) 81–97
3. Grelck, C., Shafarenko, A.: Report on S-Net: A Typed Stream Processing Language, Part I: Foundations, Record Types and Networks. Technical report, University of Hertfordshire, Department of Computer Science, Compiler Technology and Computer Architecture Group, Hatfield, England, United Kingdom (2006)
4. Grelck, C., Scholz, S.B.: SAC: A functional array language for efficient multithreaded execution. International Journal of Parallel Programming **34** (2006) 383–427

[4] http://www.aether-ist.org/

From Contracts Towards Dependent Types: Proofs by Partial Evaluation

— Draft —

Stephan Herhut[1], Sven-Bodo Scholz[1], Robert Bernecky[2], Clemens Grelck[1,3], and Kai Trojahner[3]

[1] University of Hertfordshire, U.K.
{s.a.herhut,s.scholz,c.grelck}@herts.ac.uk
[2] University of Toronto, Canada
bernecky@acm.org
[3] University of Lübeck, Germany
{grelck,trojahner}@isp.uni-luebeck.de

Abstract. The specification and resolution of non-trivial domain constraints has become a well-recognised measure for improving the stability of large software systems. In this paper we propose an approach based on partial evaluation which tries to prove such constraints statically as far as possible and inserts efficient dynamic checks otherwise.

1 Introduction

Resolving domain constraints for operations on arrays is known to be a challenging task. The central challenge is that one of the most frequently used operations, array selection, has value constraints which are undecidable in general. In the context of array languages such as APL [1] or SaC [2] which aim at generic operations on n-dimensional arrays the challenge becomes even harder as in these languages the rank and shape of arrays, at least conceptually, are part of the value as well.

In APL, all these consistency checks are purely dynamic. This design decision has a considerable runtime impact as noted in [3]. In that article, Bernecky supposes to introduce further array information named "array predicates" which, despite the overhead for housekeeping this information, at least to some extent ameliorates the runtime cost for dynamic checking.

In order to avoid the overhead due to dynamic checks, several other approaches have been developed that try to resolve these requirements statically. However, the undecidable nature of the problem forces these approaches to restrict the expressiveness of the language in one way or the other. Some approaches are based on restricted forms of dependent types such as the *indexed types* proposed by Zenger in [4] or the type system of DML [5]. Other approaches rely on a strict separation of arrays and indices and force all indices to be defined in a rather restricted manner only. This enables languages such as Zpl [6] or Chapel [7] to avoid runtime checks.

In this paper, we propose a hybrid approach. Rather than restricting either the language or the compiler to programs whose constraints can be statically resolved, we try to infer as many constraints as possible and check the unresolved ones at runtime. For many straight-forward programs this yields the same static safety as strongly typed systems would. Only for programs that rely on more complex index computations dynamic checks remain.

The central idea is to use partial evaluation as constraint resolution mechanism. In a first step, all domain constraints are inserted into the program explicitly. At that stage programs are very similar to programs that contain contracts as first proposed in the context of Eiffel [8]. In fact, the proposed approach caters for a seamless integration of arbitrary contracts as found in several modern languages from the object-oriented domain.

Subsequently, partial evaluation is applied, which is geared towards eliminating the dynamic checks. A detailed analysis of remaining checks allows the programmer to decide whether the level of static guarantees is sufficient for the application given. In case not, further partial evaluation can be applied or the program can be re-written in a way so that static resolution becomes feasible. As a nice side-effect, those checks that remain until runtime have been minimised wrt. the actual checks being performed.

We demonstrate this approach in the context of the functional array language SaC. Since the existing compiler for SaC already supports powerful mechanisms for partial evaluation as part of its type system and as part of its optimisation cycle, an implementation of the proposed approach comes for a moderate implementation effort.

The paper is structured as follows. Section 2 gives a brief introduction in SaC$_\lambda$, a strip-down version of core SaC better suited for formal reasoning. A formal definition of several of the built-in operations of SaC$_\lambda$ builds the basis for Section 3 which explains how the inherent constraints of these operations translate into explicit contracts. Section 4 discusses implementation issues that result from the intention to reuse the existing mechanisms for partial evaluation without any alteration. A detailed discussion of how the partial evaluation is being targeted towards resolving the contracts is given in Section 5 before Section 6 concludes.

2 SaC$_\lambda$

SaC$_\lambda$ is a stripped-down version of SaC, comprising only the bare essentials of the language: its syntax has been modified to a λ-calculus style, in order to ease comprehension by a functional-programming audience.

Figure 1 shows the syntax of SaC$_\lambda$. A program consists of a set of mutually recursive function definitions and a designated main expression. Essentially, expressions are either constants, variables or function applications. Since SaC does not, at present, support higher-order functions nor nameless functions, all abstractions (function definitions) are explicitly user-defined. Function applications are written in C-style, *i.e.*, with parentheses around arguments, rather than

$$
\begin{array}{lll}
\mathit{Program} & \Rightarrow & \left[\mathit{FunId} = \lambda \mathit{Id} \left[\ , \mathit{Id} \right]^* . \mathit{Expr} \ ; \ \right]^* \\
& & \quad \textbf{main} = \mathit{Expr} \ ; \\
\\
\mathit{Expr} & \Rightarrow & \mathit{Val} \\
& | & \mathit{FunId} \ (\ \mathit{Val} \left[\ , \mathit{Val} \right]^* \ ) \\
& | & \mathit{Prf} \ (\ \mathit{Val} \left[\ , \mathit{Val} \right]^* \ ) \\
& | & \textbf{if} \ \mathit{Val} \ \textbf{then} \ \mathit{Expr} \ \textbf{else} \ \mathit{Expr} \\
& | & \textbf{let} \ \mathit{Id} \left[\ , \mathit{Id} \right] = \mathit{Expr} \ \textbf{in} \ \mathit{Expr} \\
\\
\mathit{Val} & \Rightarrow & \mathit{Const} \\
& | & [\ \left[\mathit{Val} \left[\ , \mathit{Val} \right]^* \right]] \\
& | & \mathit{Id} \\
\\
\mathit{Prf} & \Rightarrow & \textbf{shape} \\
& | & \textbf{dim} \\
& | & \textbf{sel} \\
& | & \textbf{modarray} \\
& | & \textbf{add\_SxS} \ | \ \textbf{add\_SxA} \ | \ \textbf{add\_AxS} \ | \ \textbf{add\_AxA} \\
& | & \textbf{eq\_SxS} \ | \ \textbf{eq\_SxA} \ | \ \textbf{eq\_AxS} \ | \ \textbf{eq\_AxA} \\
& | & \ldots
\end{array}
$$

Fig. 1. The syntax of SAC$_\lambda$

around entire applications of functions. Constants are either scalars or vectors of expressions enclosed by square brackets.

SAC$_\lambda$ provides a few built-in array operators, referred to as primitive functions (Prf). Among these are `shape` and `dim` for computing an array's shape and dimensionality (rank), respectively. A selection operation, `sel`, is also provided; it takes two arguments: an index vector, specifying the element to be selected, and an array from which to select. As its dual, SAC$_\lambda$ provides a `modarray` operation which computes a new array from an existing one by altering a single element only; it takes three arguments: a template array, the index position at which the result array is supposed to be different from the template array and the value to which the referenced element of the array is to be set to. These basic array operations are complemented by element-wise extensions of arithmetic and relational operations, such as *addition* (`add`) and *equality* (`eq`), respectively. We distinguish between four different versions of these binary operations:

SxS suffixes operations on two scalar values. The result is the application of the operation to both values.

SxA denotes operations that expect a scalar value as first argument and an array as second argument. Here the result is an array of the same shape as the second argument, computed by applying the operation to the first argument and the corresponding element of the second argument.

AxS is used for operations that expect an array as first argument and a scalar value as second argument. They are computed analogous to the `SxA` operations.

AxA represents operations on array arguments. These, given that the shape of both arguments is identical, compute a result array by applying the operation element-wise to the corresponding elements of their two arguments.

We can formalize the semantics of SaC$_\lambda$ by a standard big-step operational semantics for λ-calculus-based applicative languages as defined in several textbooks, *e.g.*, [9]. The core relations, *i.e.*, those for conditionals, abstractions, and function applications can be used in their standard form. Hence, only those relations pertaining to the array specific features of SaC$_\lambda$ are shown in Figure 2.

As a unified representation for n-dimensional arrays we use pairs of vectors $< [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >$ where the vector $[\ s_1, \ldots, s_n]$ denotes the shape of the array, *i.e.*, its extent with respect to the n individual axes, and the vector $[\ d_1, \ldots, d_m]$ contains all elements of the array in a linearized form. Since the number of elements within an array equals the product of the number of elements per individual axis, we have $m = \prod_{i=1}^{n} s_i$. The linearization we choose is row-major, *i.e.*, elements that correspond to variations in the rightmost index only are consecutive in the vector of elements.

The first two evaluation rules of Figure 2 show how scalars as well as vectors are transformed into the internal representation. The rule VECT requires that all elements need to be of the same shape, thereby ensuring shape consistency in the overall result.

The next three rules formalize the semantics of the main primitive operations on arrays: `dim`, `shape`, `sel` and `modarray`. There are two aspects of the SEL rule to be observed: Firstly, we require the selection index to be of the same length as the shape of the array to be selected from. This ensures scalar values as results. Secondly, the selection index needs to be within the bounds of the array argument, *i.e.*, each element i_j of the index vector needs to be non-negative and less than the corresponding element s_j of the shape vector of the array argument. Finally, the selection requires a transformation of the index vector into a scalar offset l into the linearized form of the array. The sum of products used here reflects the row-major linearization we have chosen.

Element-wise extensions of standard operations such as the arithmetic and relational operations are demonstrated by the example of the rules for addition (`add_SxS`, `add_AxS` and `add_AxV`). We have left out the rules for the `SxA` variants, as they a symmetrical to their `AxS` counterparts.

Whereas `add_SxS` and `add_AxS` can be applied to any pair of scalar values or an array of arbitrary shape as first argument und any scalar value as second argument, respectively, we require the arguments of `add_AxA` to be of the same shape.

$$\textsc{const} : \frac{}{n \rightarrow < [], [n] >}$$

$$\textsc{vect} : \frac{\forall i \in \{1, \ldots, n\} : e_i \rightarrow < [\ s_1, \ldots, s_m], [\ d_1^i, \ldots, d_p^i] >}{[\ e_1, \ldots, e_n] \rightarrow < [\ \mathrm{n},\ s_1, \ldots, s_m], [\ d_1^1, \ldots, d_p^1, \ldots, d_1^n, \ldots, d_p^n] >}$$

$$\textsc{dim} : \frac{e \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >}{\texttt{dim(}\ e\texttt{)} \rightarrow < [], [n] >}$$

$$\textsc{shape} : \frac{e \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >}{\texttt{shape(}\ e\texttt{)} \rightarrow < [\ n], [\ s_1, \ldots, s_n] >}$$

$$\textsc{sel} : \frac{\begin{array}{c} iv \rightarrow < [\ n], [\ i_1, \ldots, i_n] > \\ e \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] > \end{array}}{\texttt{sel(}\ iv\texttt{,}\ e\texttt{)} \rightarrow < [], [\ d_{l+1}] >}$$
$$\text{where } l = \sum_{j=1}^{n} (i_j * \prod_{k=j+1}^{n} s_k)$$
$$\iff \forall k \in \{1, \ldots, n\} : 0 \leq i_k < s_k$$

$$\textsc{modarray} : \frac{\begin{array}{c} iv \rightarrow < [\ n], [\ i_1, \ldots, i_n] > \\ e_d \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] > \\ e_v \rightarrow < [], v > \end{array}}{\texttt{modarray(}\ iv\texttt{,}\ e_d\texttt{,}\ e_v\texttt{)} \rightarrow < [\ s_1, \ldots, s_n], [\ d'_1, \ldots, d'_m] >}$$
$$\text{where } d'_i = \begin{cases} v & \text{if } i = \sum_{j=1}^{n} (i_j * \prod_{k=j+1}^{n} s_k) + 1, \\ d_i & \text{otherwise.} \end{cases}$$
$$\iff \forall k \in \{1, \ldots, n\} : 0 \leq i_k < s_k$$

$$\textsc{add\_SxS} : \frac{\begin{array}{c} e_1 \rightarrow < [], d_1 > \\ e_2 \rightarrow < [], d_2 > \end{array}}{\texttt{add\_SxS(}\ e_1\texttt{,}\ e_2\texttt{)} \rightarrow < [], [\ d_1 + d_2] >}$$

$$\textsc{add\_AxS} : \frac{\begin{array}{c} e_1 \rightarrow < [\ s_1, \ldots, s_n], [\ d_1^1, \ldots, d_m^1] > \\ e_2 \rightarrow < [], d > \end{array}}{\texttt{add\_AxS(}\ e_1\texttt{,}\ e_2\texttt{)} \rightarrow < [\ s_1, \ldots, s_n], [\ d_1^1 + d, \ldots, d_m^1 + d] >}$$

$$\textsc{add\_AxA} : \frac{\begin{array}{c} e_1 \rightarrow < [\ s_1, \ldots, s_n], [\ d_1^1, \ldots, d_m^1] > \\ e_2 \rightarrow < [\ s_1, \ldots, s_n], [\ d_1^2, \ldots, d_m^2] > \end{array}}{\texttt{add\_AxA(}\ e_1\texttt{,}\ e_2\texttt{)} \rightarrow < [\ s_1, \ldots, s_n], [\ d_1^1 + d_1^2, \ldots, d_m^1 + d_m^2] >}$$

Fig. 2. An operational semantics for SaC$_\lambda$.

3 Contracts for Built-In Functions

Given the semantic rules presented in the previous section, we can now extract contracts, *i.e.*, pre- and post-conditions, for the built-in functions of SaC$_\lambda$. As

a first step, we have to find suitable SAC$_\lambda$-representations for the preconditions of each built-in function. As an example, consider the built-in function `add_SxS`. The rule ADD_SxS in Figure 2 gives two preconditions; it requires each argument to evaluate to a scalar value. Thus, a contract for `add_SxS` needs to contain code that ensures that both arguments at runtime are scalar values. Furthermore, we need a way to express a runtime error explicitly. We do this by introducing an explicit termination symbol $\perp$. Given an application of `add_SxS` of the following form

```
let
  R = add_SxS( A, B)
in ...
```

the required contracts can be made explicit by the following code:

```
let
  R = if eq_SxS( dim( A), 0) then
        if eq_SxS( dim( B), 0) then
           add_SxS( A, B)
        else
           ⊥
      else
         ⊥
in ...
```

The outer conditional checks whether argument `A` has dimensionality 0, *i.e.*, whether it is a scalar value. The same condition is checked by the inner conditional for argument `B`. In case any of the pre-conditions does not hold we return $\perp$. Thus, within the above code, we do not need to check the implicit preconditions when applying `add_SxS`, as the application is only evaluated if both pre-conditions are known to hold. Furthermore, we have made the implicit pre-conditions of `add_SxS` explicit in the code and thereby accessible to standard partial evaluation techniques.

Nonetheless, the above code still requires implicit checks: the newly introduced applications of `dim` and `eq` come with their own set of pre-conditions. Thus, in order to make all pre-conditions within a program explicit, we have to introduce explicit checks for these, as well, which, ultimately, leads to a non-terminating code transformation.

To circumvent this, we introduce special built-in functions designed to be used as predicates in contracts. They all go without constraining the arguments in the pre-conditions, as can be seen in Figure 3. This ensures termination of the code transformation. Additionally, they cater for a concise representation and allow us to easily differentiate between user-specified conditions and automatically inserted contracts. For the built-in function `add_SxS`, we add a built-in function `is_scalar` as follows:

is_scalar evaluates to `true` if its first argument is a scalar value. Otherwise it evaluates to `false`.

Using this built-in function, we can rewrite the contracts for `add_SxS`:

$$\text{IS\_SCALAR} : \frac{e \rightarrow < [\ s_1, \ldots, s_i], [\ d_1, \ldots, d_k] >}{\texttt{is\_scalar(}\ e\texttt{)} \rightarrow v}$$
$$\text{where } v = \begin{cases} < [], \texttt{true} > & \text{if } i = 0, \\ < [], \texttt{false} > & \text{otherwise.} \end{cases}$$

$$\text{SAME\_SHAPE} : \frac{\begin{array}{c} e_1 \rightarrow < [\ s_1^1, \ldots, s_i^1], [\ d_1^1, \ldots, d_k^1] > \\ e_2 \rightarrow < [\ s_1^2, \ldots, s_j^2], [\ d_1^2, \ldots, d_l^2] > \end{array}}{\texttt{same\_shape(}\ e_1\texttt{,}\ e_2\texttt{)} \rightarrow v}$$
$$\text{where } v = \begin{cases} < [], \texttt{true} > & \text{if } i = j \\ & \quad \wedge \forall m \in \{1, \ldots, i\} : s_m^1 = s_m^2, \\ < [], \texttt{false} > & \text{otherwise.} \end{cases}$$

$$\text{INDEX\_MATCHES\_DIM} : \frac{\begin{array}{c} iv \rightarrow < [\ s_1^{iv}, \ldots, s_i^{iv}], [\ d_1^{iv}, \ldots, d_k^{iv}] > \\ e \rightarrow < [\ s_1^e, \ldots, s_j^e], [\ d_1^e, \ldots, d_l^e] > \end{array}}{\texttt{index\_matches\_dim(}\ iv\texttt{,}\ e\texttt{)} \rightarrow v}$$
$$\text{where } v = \begin{cases} < [], \texttt{true} > & \text{if } i = 1 \wedge s_1^{iv} = j, \\ < [], \texttt{false} > & \text{otherwise.} \end{cases}$$

$$\text{NON\_NEG\_VAL} : \frac{e \rightarrow < [\ s_1, \ldots, s_i], [\ d_1, \ldots, d_j] >}{\texttt{non\_neg\_val(}\ e\texttt{)} \rightarrow v}$$
$$\text{where } v = \begin{cases} < [], \texttt{true} > & \text{if } \forall l \in \{1, \ldots, j\} : d_l \geq 0, \\ < [], \texttt{false} > & \text{otherwise.} \end{cases}$$

$$\text{VAL\_LT\_SHAPE} : \frac{\begin{array}{c} iv \rightarrow < [\ s_1^{iv}, \ldots, s_i^{iv}], [\ d_1^{iv}, \ldots, d_k^{iv}] > \\ e \rightarrow < [\ s_1^e, \ldots, s_j^e], [\ d_1^e, \ldots, d_l^e] > \end{array}}{\texttt{val\_lt\_shape(}\ iv\texttt{,}\ e\texttt{)} \rightarrow v}$$
$$\text{where } v = \begin{cases} < [], \texttt{true} > & \text{if } i = 1 \wedge s_1^{iv} = j \\ & \quad \wedge \forall m \in \{1, \ldots, j\} : d_m^{iv} < s_m^e, \\ < [], \texttt{false} > & \text{otherwise.} \end{cases}$$

Fig. 3. Semantic rules for additional contract predicate built-in functions.

```
let
  R = if is_scalar( A) then
        if is_scalar( B) then
           add_SxS( A, B)
        else
           ⊥
      else
        ⊥
in ...
```

Using `is_scalar`, evaluating the above code does not involve checking any implicit pre-conditions.

The constraints generated by applications of `add_SxS`, as shown above, only assert a certain fixed dimensionality. This kind of constraint, *i.e.*, asserting a statically known dimensionality or shape, can be handled by existing type systems, *e.g.*, the type-system for SaC presented in [2]. More sophisticated pre-conditions, *i.e.*, equality constraints and value dependent constraints, are more difficult to handle by static type-systems. As an example for equality constraints as pre-condition, consider the following application of `add_AxA`:

```
let
  R = add_AxA( A, B)
in ...
```

Here, the semantic rule ADD_AxA given in Figure 2 requires the shapes of both arguments to be identical. As in the previous example, we can directly express this in SaC$_\lambda$:

```
let
  R = if eq_AxA( shape( A), shape( B)) then
         add_AxA( A, B)
      else
         ⊥
in ...
```

Again, we have wrapped the call to the built-in function into a conditional. The predicate asserts that both arguments of `add_AxA` are of the same shape; it is expressed by means of the existing built-in functions `dim` and `eq`. As in our first example, using existing built-in functions to express automatically inserted contracts leads to a non-terminating code transformation. We therefore introduce the following new built-in function:

same_shape is used to express the equality of argument shapes. It expects two arrays as arguments and evaluates to `true` if both have the same shape. Otherwise it evaluates to `false`.

Using `same_shape`, we can rewrite the above example as follows:

```
let
  R = if same_shape( A, B) then
         add_AxA( A, B)
      else
         ⊥
in ...
```

This code is free of implicit pre-conditions: The application of `same_shape` does not involve any pre-conditions and the application of `add_AxA` is only evaluated if all of its pre-conditions hold.

Finally, as an example for value dependent constraints, consider the built-in function `sel`. Rule SEL in Figure 2 declares the following pre-conditions:

1. the length of the index vector needs to match the dimensionality of the array
2. the index vector needs to be non-negative for each of its elements

3. each element of the index vector needs to be less than the corresponding component of the array's shape

The first condition ensures that the result of the selection is a scalar value. Conditions 2 and 3 assert that the index vector is within the set of legal indices of the argument array. These conditions can directly be expressed in SaC$_\lambda$. However, as for the previous examples, we add a special built-in function for each constraint:

shape_matches_dim is used to express the first condition. It expects an index vector as first argument and an array as second argument and evaluates to `true` if the length of the index vector matches the number of dimensions of the array argument. Otherwise it evaluates to `false`.

non_neg_val checks whether its first argument does not contain negative values and evaluates to `true` if this condition holds and to `false` otherwise. It is used to express the second condition.

val_lt_shape evaluates to `true` if each element of the index vector given as its first argument is smaller than the corresponding component of the shape of the second argument. Otherwise it evaluates to `false`. We use this built-in operation to model the third condition.

Given these three additional primitives, we can now make the pre-conditions of `sel` explicit in the code. As an example consider the code fragment

```
let
  v = sel( iv, A)
in ...
```

which can now be transformed into

```
let
  v = if shape_matches_dim( iv, A) then
        if non_neg_val( iv) then
           if val_lt_shape( iv, A) then
              sel( iv, A)
           else
              ⊥
        else
           ⊥
      else
         ⊥
in ...
```

Again, the resulting code is free of implicit pre-conditions. All pre-conditions are explicitly expressed by contracts. We use three specific built-in functions for each pre-condition instead of one built-in function for all pre-conditions. Firstly, this makes the built-in functions more generally applicable. Secondly, it eases partial evaluation: In contexts where only one or two of the pre-conditions can be statically decided, our fine-grained approach allows us to evaluate corresponding contracts statically. In a coarse-grained approach with only one built-in function for all pre-conditions, this can not be easily done.

Apart from the transformations sketched out in the above examples, we need to define the contracts for the built-in function `modarray`. We omit an example here. In short, `modarray` demands the same pre-conditions as the built-in function `sel` and additionally requires its third argument to be a scalar. This is handled by emitting an additional contract to that effect.

4 Implementation Issues

The explicit contracts produced by the transformation scheme presented in the previous section, although viable in theory, are difficult to compile into efficient runtime code. As an example, consider the following code fragment:

```
let
  C = add_AxA( A, B)
in  let
      a = sel( iv, A)
    in let
         b = sel( iv, B)
       in ...
```

Applying the transformations described in the previous section yields the following code:

```
let
  C = if same_shape( A, B) then
        add_AxA( A, B)
      else
        ⊥
in let
     a = if shape_matches_dim( iv, A) then
           if non_neg_val( iv) then
             if val_lt_shape( iv, A) then
               sel( iv, A)
             else
               ⊥
           else
             ⊥
         else
           ⊥
   in let
        b = if shape_matches_dim( iv, B) then
              if non_neg_val( iv) then
                if val_lt_shape( iv, B) then
                  sel( iv, B)
                else
                  ⊥
              else
                ⊥
            else
              ⊥
      in ...
```

It contains the following contracts:

1. `A` and `B` are required to have the same shape (`same_shape`)
2. `iv` is required to be a valid index into `A` (`shape_matches_dim`, `non_neg_val`, `val_lt_shape`)
3. `iv` is required to be a valid index into `B` (`shape_matches_dim`, `non_neg_val`, `val_lt_shape`)

When looking at all three contracts in conjunction, the second and third contract are identical: given that `A` and `B` have the same shape, any valid index into `A` is a valid index into `B`, as well. For an efficient runtime implementation, it would therefore be desirable to exploit the outcome of previously evaluated contracts for eliminating redundant contracts in parts of the code that are executed later on. However, formally the information that `A` and `B` have the same shape is only valid within the then-branch of the conditional of the corresponding contract. Thus, to exploit that information, we would have to move the consecutive selections from after the conditional into its then-branch. Although possible in the above example, a generally applicable transformation scheme is complicated. We therefore chose a different route.

To more easily allow us to incorporate optimisations as the one sketched out above, we change the representation of contracts. Instead of using conditionals,

we weave the predicates of contracts into the dataflow: we modify each of the built-in functions defined in Figure 3 to either return those of their arguments, for which the asserted condition holds, or $\bot$ otherwise. Thereby, we introduce new identifiers that reference the same values, but give us a handle to the additional knowledge previously evaluated contracts have asserted. As an example, consider the transformation of the example given above. Using our new representation, we get the following code:

```
let
  A', B' = same_shape( A, B)
in  let
      C = add_AxA( A', B')
    in  let
          iv' = shape_matches_dim( iv, A')
        in  let
              iv'' = non_neg_val( iv')
            in  let
                  iv''' = val_lt_shape( iv'', A')
                in  let
                      a = sel( iv''', A')
                    in  let
                          iv⁴ = shape_matches_dim( iv''', A')
                        in  let
                              iv⁵ = non_neg_val( iv⁴)
                            in  let
                                  iv⁶ = val_lt_shape( iv⁵, B')
                                in  let
                                      a = sel( iv⁶, B')
                                    in  ...
```

The result of the application of `same_shape` is bound to two new identifiers `A'` and `B'`. If `A` and `B` have the same shape, `same_shape` is the identity function on its arguments, *i.e.*, `A'` and `B'` are just aliases of `A` and `B`, respectively. Otherwise, `same_shape` evaluates to $\bot$ for both result values. The consecutive application of `add_AxA` therefore either computes the element-wise sum of `A'` and `B'`, alias `A` and `B`, or evaluates to $\bot$. Thus, the above encoding is semantically equivalent to the encoding using conditionals introduced in Section 3. However, by introducing new identifiers for `A` and `B`, we can now exploit the additional knowledge that they have the same shape in the consecutive applications of `shape_matches_dim`. The first application of `shape_matches_dim` ensures that `iv'` matches the shape of `A'` (or is $\bot$), therefore `iv'` matches the shape of `B'`, as well. As all built-in predicate functions are either the identity or return $\bot$, it follows that `iv'''` matches the shape of `B'` (or is $\bot$) and that the second application of `shape_matches_dim` always is the identity. Thus, we can statically evaluate $\texttt{iv}^4$ to `iv'''`. Similarly, we can statically evaluate the second application of `non_neg_val` and `val_lt_shape`.

Using our second approach, we can transform all implicit pre-conditions into explicit contracts and, furthermore, optimise the resulting code. However, our

optimisations still need to be aware of the original implicit pre-conditions. As an example, consider the following code fragment:

```
let
  B = modarray( A, iv, b)
in  let
      c = sel( B, iv)
    in  ...
```

Given an array `A`, the above code creates a new array `B` by replacing the value at index `iv` with `b`. In a consecutive selection, the value at position `iv` in `B` is selected as `c`. On first sight, one is tempted to conclude that `c` equals `b`, as the element at index `iv` in `B` clearly is `b`. However, this is only true if all pre-conditions for `modarray` and `sel` hold. Otherwise `B` equals $\perp$, as does `c`.

Unfortunately, transforming the pre-conditions into explicit contracts does not help, either, as the following example shows:

```
let
  b' = is_scalar( b)
in  let
      iv' = shape_matches_dim( iv, A)
    in  let
          iv'' = non_neg_val( iv')
        in  let
              iv''' = val_lt_shape( iv'', A)
            in  let
                  B = modarray( A, iv''', b')
                in  let
                      iv4 = shape_matches_dim( iv''', A)
                    in  let
                          iv5 = non_neg_val( iv4)
                        in  let
                              iv6 = val_lt_shape( iv5, A)
                            in  let
                                  c = sel( B, iv6)
                                in  ...
```

In the above code, the selection uses a different index vector than the preceding `modarray`. Thus, our naïve optimisation is not applicable. This situation changes, if we remove redundant contracts, *i.e.*, the contracts of `sel` that assert the same properties as those of the preceding `modarray` operation:

```
let
  b' = is_scalar( b)
in  let
      iv' = shape_matches_dim( iv, A)
    in  let
          iv'' = non_neg_val( iv')
        in  let
              iv''' = val_lt_shape( iv'', A)
            in  let
                  B = modarray( A, iv''', b')
                in  let
                      c = sel( B, iv''')
                    in  ...
```

Now, our naïve optimisation can be applied, as both operations use the same index vector, yielding the following code:

```
let
  b' = is_scalar( b)
in  let
      iv' = shape_matches_dim( iv, A)
    in  let
          iv'' = non_neg_val( iv')
        in  let
              iv''' = val_lt_shape( iv'', A)
            in  let
                  c = b'
                in  ...
```

By removing the `sel` and `modarray` operations, we have decoupled the contracts from the dataflow as `iv` is not referenced anymore. Therefore, a standard optimisation like dead-code removal might strip these out, resulting in an illegal program transformation.

A straight forward solution would be to make all optimisations aware of the implicit pre-conditions of built-in functions, or to ensure that dead-code removal and similar optimisations do not touch contracts. This, however, increases the complexity of the specification and implementation of most optimisations. Instead, we alter our representation of contracts once more.

Conceptually, we introduce contracts to assert certain pre-conditions of applications of built-in functions prior to their evaluation. Therefore, we insert contracts before function applications and weave them into the dataflow to ensure that they are evaluated prior to the corresponding built-in function. In particular, we weave the contracts into the dataflow of the arguments of a function application. However, in the above example, function applications are evaluated from below by combining two operations into a single operation. As this potentially removes all references to some of the arguments of the original function applications, the introduced contracts might be stripped out. To prevent this, we additionally add a dependency between the contracts and the results of a

function application. The additional built-in function `after_guard` serves this purpose.

after_guard is the identity on its first argument, if all other arguments evaluate to `true`. Otherwise it evaluates to $\perp$.

Furthermore, we add an additional boolean return value to the previously introduced built-in operations used in the predicates of contracts. This allows us to encode a dataflow dependency between the return value of a function application and the contracts. This leads to the following code:

```
let
  b', p^1 = is_scalar( b)
in  let
      iv', p^2 = shape_matches_dim( iv, A)
    in  let
          iv'', p^3 = non_neg_val( iv')
        in  let
              iv''', p^4 = val_lt_shape( iv'', A)
            in  let
                  B = modarray( A, iv''', b')
                in  let
                      B' = after_guard( B, p^1, p^2, p^3, p^4)
                    in  let
                          c = sel( B', iv''')
                        in  ...
```

Note here, that although the `modarray` and `sel` operations use the same index vector `iv'''`, our naïve optimisation does not apply as the second argument of the selection differs from the result of the array modification: The inserted `after_guard` inhibits optimisation, unless it can be statically evaluated, *i.e.*, unless all contacts are statically known to be true.

Using this representation, we have replaced all implicit pre-conditions of built-in functions by explicit contracts and have made checking pre-conditions redundant both during program evaluation and program optimisation.

5 Constraint Resolution by Partial Evaluation

Using the explicit encoding of pre-conditions as contracts presented in the previous sections, we can now use existing partial evaluation techniques developed for SAC to (partially) prove contracts at compile-time. We can identify three different classes of contracts:

1. contracts that assert a certain dimensionality or shape of an argument, *e.g.*, `is_scalar`
2. contracts ensuring equality constraints on shapes, *i.e.*, `same_shape`
3. contracts that are used to enforce value-dependent constraints, *i.e.*, `shape_matches_dim` and `val_matches_shape`

The first class of contracts is handled by the type-system of SaC as described in [2]. A type in SaC consists of the type of the elements of an array and information about its shape. We use subtyping to represent different levels of static shape-knowledge: unknown dimensionality (no static knowledge), known dimensionality (partial static knowledge) and known shape (full static knowledge). Type inference allows us, given at least partial static knowledge of the argument shapes, to statically resolve applications of constraints like `is_scalar`.

To enable optimisations in cases where only limited static shape-knowledge is available, we have developed a mechanism to infer static knowledge of shape equalities [10]. This technique can be used to statically evaluate and thus prove contracts of the second kind. We are currently undertaking further research to extend this approach to value-dependent constraints, thereby tackling the third class of contracts.

6 Conclusions

This paper demonstrates how domain constraints can be resolved semi-statically by means of partial evaluation and other optimisation techniques. The appealing aspect of this approach lies in its versatility and its lean implementation. Not only implicit domain constraints can be resolved that way but also user specified contracts. This allows, in principle, to specify restrictions similar to those found in languages based on dependent types. The resolution itself comes almost for free by simply applying the existing optimisation techniques. The key techniques here seem to be the type inference system of SaC [2] as well as the static array attributes [10] which provide shape equalities throughout programs. However, since the constraint resolution has been reduced to standard optimisations there is a mutual benefit: whenever a new optimisation technique is being introduced, the static contract resolution may benefit from it as well as any other part of any given program.

We made similar observations in the context of several other optimisations: expressing certain aspects of any given program explicitly and then applying the existing optimisations often turned out to be a very powerful and relatively easy to implement technique. Examples are the static array attributes which utilise optimisations such as CSE (common subexpression elimination) in order to propagate equality constraints, or IVE (index vector elimination) [11] which utilises LIR (loop invariant removal) as well as CSE in order to avoid superfluous index computations.

By applying it to the resolution of constraints as they arise from built-in operations as well as user-specified contracts, we achieve a behaviour which comes close to that of more strongly typed systems based on various forms of dependent types: For many programs we can give static soundness guarantees wrt. certain domain requirements. In those cases where we cannot give these guarantees, we can clearly identify the program parts where unresolved constraints remain. Then it is up to the user to decide whether further program optimisation should be applied or the dynamic contract checks should remain.

It remains as future research to investigate whether such a general purpose optimisation mechanism is capable to resolve more complex constraints in an effective way. In particular, it would be interesting to compare its effectiveness with that obtained by dedicated resolution systems such as Epigram [12].

References

1. International Standards Organization: Programming Language APL, Extended. ISO N93.03, ISO (1993)
2. Scholz, S.B.: Single Assignment C — efficient support for high-level array operations in a functional setting. Journal of Functional Programming **13**(6) (2003) 1005–1059
3. Bernecky, R.: Reducing computational complexity with array predicates. In: APL '98: Proceedings of the APL98 conference on Array processing language, New York, NY, USA, ACM Press (1998) 39–43
4. Zenger, C.: Indexed types. Theor. Comput. Sci. **187**(1-2) (1997) 147–165
5. Xi, H., Pfenning, F.: Dependent Types in Practical Programming. In: POPL '99, ACM Press (1999) 214–227
6. Chamberlain, B., Choi, S.E., Lewis, E., Lin, C., Snyder, L., Weathersby, W.: ZPL: A Machine Independent Programming Language for Parallel Computers. IEEE Transactions on Software Engineering **26**(3) (2000) 197–211
7. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel programmability and the chapel language. In: International Journal of High Performance Computing Applications. (2007) 291–312
8. Meyer, B.: Eiffel: The Language. Prentice Hall (1990)
9. Pierce, B.C.: Types and programming languages. MIT Press, Cambridge, MA, USA (2002)
10. Trojahner, K., Grelck, C., Scholz, S.: On Optimising Shape-Generic Array Programs using Symbolic Structural Information. In Horváth, Z., Zsók, V., eds.: Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages (IFL'06), Budapest, Hungary. Technical Report 2006-S01, Eötvös Loránd University, Faculty of Informatics, Budapest, Hungary (2006) 13–27
11. Bernecky, R., Herhut, S., Scholz, S., Trojahner, K., Grelck, C., Shafarenko, A.: Index Vector Elimination: Making Index Vectors Affordable. In Horváth, Z., Zsók, V., eds.: Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages (IFL'06), Budapest, Hungary. Technical Report 2006-S01, Eötvös Loránd University, Faculty of Informatics, Budapest, Hungary (2006) 28–43
12. McBride, C., McKinna, J.: The view from the left. J. Funct. Program. **14**(1) (2004) 69–111

A Rational Simplifier for GHC

Laszlo Nemeth

Istanbul Bilgi University, Turkey

Abstract. We describe an attempt to replace the Glasgow Haskell Compiler's simplifier with one written in a multi-agent system (Jason) based on AgentSpeak and argue that the framework allows many aspects of optimisation to be expressed in a concise manner. This is very much work in progress and it remains to be seen whether the techniques developed so far scale to a complete and better replacement.

Submitted to IFL 2007.

Title:
    Amortizing the cost of commuting conversions
    when beta-reducing monadic normal forms and A-normal forms

Author:
    Olivier Danvy
    University of Aarhus
    <danvy@brics.dk>

Abstract:
    In "Compiling with Continuations, Continued" (ICFP 2007), Andrew
    Kennedy points out that commuting conversions increase the complexity
    of simplifying intermediate-language terms, and states that "it is
    far from clear how to amortize the cost of commuting conversions to
    obtain a linear number of reductions for A-normal forms", in contrast
    to CPS. We show how to achieve this amortization, as a corollary of
    a syntactic bijection between CPS terms and terms in A-normal form.

---

1. Background and introduction

1.1. Grammar of terms in direct style

```
Programs:
  p ::= e
Expressions:
  e ::= x | \x.e | e e | if e e e

  x in Ide -- source identifiers
```

1.1. Grammar of terms in CPS

The output grammar of the CBV CPS transformation:

```
Programs:
   p ::= \k.s
Serious terms:
   s ::= c t                     -- return
       | t t c                   -- call
       | if t s s                -- tail conditional expression
       | let k = \j.s in if t s s -- non-tail conditional expression
Trivial terms:
   t ::= x | i | j | \x.\k.s
Continuations:
   c ::= k | \i.s

   x in Ide  -- identifiers from the source term
   k in Cide -- continuation identifiers
   i in Vide -- parameters of continuations for non-tail calls
   j in Jide -- parameters of continuations for non-tail conditional expressions
```

```
1.2. Grammar of terms in CPS, after administrative reductions

The c production disappears:

Programs:
   p ::= \k.s
Serious terms:
   s ::= k t                      -- return
       | t t k                    -- tail call
       | t t \i.s                 -- non-tail call
       | if t s s                 -- tail conditional expression
       | let k = \j.s in if t s s -- non-tail conditional expression
Trivial terms:
   t ::= x | i | j | \x.\k.s

A CPS term is in administrative normal form if all its continuations are
beta-reduced.

-----

1.3. Grammar of terms in A-normal form

Programs:
   p ::= s
Serious terms:
   s ::= return t                 -- return point
       | t t                      -- tail call
       | let i = t t in s         -- non-tail call
       | if t s s                 -- tail conditional expression
       | let k = \j.s in if t s s -- non-tail conditional expression
       | join t k                 -- join point
Trivial terms:
   t ::= x | i | j | \x.s

-----

1.4. Bijective correspondence between CPS terms in administrative normal
     form and A-normal forms

Reference: Olivier Danvy, "Back to Direct Style", ESOP 1992 and Science
of Computer Programming 1994.

The following relation is defined by structural induction over the two
grammars above.  Each judgment is indexed by a non-terminal, and we use a
diacritical convention: accent grave (`) for terms in CPS, and accent
aigu (') for terms in ANF:

  |-P p_cps <-> p_anf

k |-S s_cps <-> s_anf

  |-T t_cps <-> t_anf

The inference rules:

k |-S    s` <-> s'
-----------------
  |-P \k.s` <-> s'
```

```
  |-T t‘ <-> t’
-----------------
k |-S k t‘ <-> t’

  |-T t0‘ <-> t0’   |-T t1‘ <-> t1’
-----------------------------------
k |-S t0‘ t1‘ k <-> t0’ t1’

  |-T t0‘ <-> t0’   |-T t1‘ <-> t1’   k |-S s‘ <-> s’
-----------------------------------------------------
k |-S t0‘ t1‘ \i.s‘ <-> let i = t0’ t1’ in s’

  |-T t‘ <-> t’   k |-T s1‘ <-> s1’   k |-T s2‘ <-> s2’
-------------------------------------------------------
k |-S if t‘ s1‘ s2‘ <-> if t’ s1’ s2’

k |-S s‘ <-> s’   |-T t‘ <-> t’   k1 |-S s1‘ <-> s1’   k1 |-S s2‘ <-> s2’
-----------------------------------------------------------------------
k |-S let k1 = \j.s‘ in if t‘ s1‘ s2‘ <-> let k1 = \j.s’ in if t’ s1’ s2’

  -----------
  |-T x <-> x

  -----------
  |-T i <-> i

  -----------
  |-T j <-> j

k |-S s‘ <-> s’
------------------------
  |-T \x.\k.s‘ <-> \x.s’
```

(The whole thing is of course much nicer once latexed; the author apologizes about this hasty presentation.)

The function that, given a CPS term, yields a term in A-normal form based on the judgments above computes what is called "un-CPS" in Flanagan et al.’s article at PLDI 1993.

---

2. Issues

2.1. ANFs are not closed under beta-reduction

For example,

```
    let i2 = (\x.let i1 = t2 t3 in s1) t1 in s2
    ->beta
    let i2 = let i1 = t2[t1/x] t3[t1/x] in s1[t1/x] in s2
    ->let.assoc
    let i1 = t2[t1/x] t3[t1/x] in let i2 = s1[t1/x] in s2
```

2.2. CPS is (almost) closed under beta-reduction

For (the corresponding) example,
    (\x.\k.t2 t3 \i1.s1) t1 \i2.s2
    ->beta
    t2[t1/x] t3[t1/x] \i.s1[t1/x, \i2.s2/k]

As one can note, the contractum is not in administrative normal form if k
is applied in s1 since the substitution of k yields a term where the
continuation \i2.s2 occurs in a beta-redex.

2.3. The problem

In "Compiling with Continuations, Continued", Andrew Kennedy states that
in contrast to CPS, "it is far from clear how to amortize the cost of
commuting conversions to obtain a linear number of reductions for
A-normal forms".

2.4. Our solution

Instead of throwing in the towel, one can instead exploit the syntactic
structure of A-normal forms and compose beta-reduction with a
renormalization phase:

    let i2 = (\x.let i1 = t2 t3 in s1) t1 in s2
    ->beta
    let i1 = t2[t1/x] t3[t1/x] in s1{\t.s2[t/i2]}[t1/x]

The definition of s1{\t.s2[t/i2]} is the topic of the next section.

----

3. Sub-linear renormalization

Here and just above, overlined "\" and "@" indicate static
lambda-abstraction and static application, as usual in the one-pass
CPS transformation (Danvy & Filinski, MSCS 1992).

The renormalization s{K} is defined by structural induction over s:

                  (return t){K} = K @ t

                     (t0 t1){K} = let i = t0 t1 in K @ i

        (let i = t0 t1 in s){K} = let i in t0 t1 in s{K}

                (if t s1 s2){K} = let k = \j.K @ j in if t s1 s2

(let k = \j.s in if t s1 s2){K} = let k = \j.s{K} in if t s1 s2

NB. The case "join t k" cannot happen since such a join term occurs in
conditional branches, which are not traversed by the renormalization.

Intuitively, the renormalization 'hooks' the (flat) body of the
lambda-abstraction just beta-reduced with its (flat) context,
and it does so in time proportional to the 'length' of this flat body.

For example,
```
    let i3 = (\x.let i1 = t1 t1' in let i2 = t2 t2' in t3 t3') t' in s4
    ->beta
    let i1 = (t1 t1')[t'/x] in                _
    let i2 = (t2 t2')[t'/x] in (t3 t3')[t'/x]{\t.s4[t/i3]}[t'/x]
    ->renormalization
    let i1 = (t1 t1')[t'/x] in
    let i2 = (t2 t2')[t'/x] in
    let i4 = (t3 t3')[t'/x] in s4[i4/i3][t'/x]
```

---

4. Conclusion

Because A-normal forms and CPS terms in administrative normal form are in
bijective correspondence, much of what can be done to terms in one style
can easily be done to terms in the other style. We believe that this is
the case too for renormalization after beta-reduction, which as shown
here can be achieved at sub-linear cost for terms in A-normal form.

That said, the renormalization is yet another ad-hoc bitty bit to emulate
a virtue of CPS in a non-CPS world. No wonder Olin Shivers prefaced his
PhD dissertation with Archilocus's quote:

> The fox knows many things,
> but the hedgehog knows one great thing.

References:

Olivier Danvy:
"Back to direct style"
Proceedings of ESOP, 1992 and Science of Computer Programming, 1994

Olivier Danvy and Andrzej Filinski:
"Representing control, a study of the CPS transformation"
Mathematical Structures in Computer Science, 1992

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen:
"The essence of compiling with continuations"
Proceedings of PLDI, 1993

Andrew Kennedy:
"Compiling with continuations, continued"
Proceedings of ICFP, 2007

Carl Nielsen:
"The fog is lifting"
Concerto for flute & harp, 1920

Olin Shivers:
"Control-flow analysis of higher-order languages"
PhD dissertation, Carnegie Mellon University, 1991