Tag-free Combinators for Binding-Time Polymorphic Program Generation

Peter Thiemann[1] and Martin Sulzmann[2]

[1] Albert-Ludwigs-Universität Freiburg, Germany
thiemann@informatik.uni-freiburg.de
[2] Intaris martin.sulzmann@gmail.com

Abstract. Binding-time polymorphism enables a highly flexible binding-time analysis for offline partial evaluation. This work provides the tools to translate this flexibility into efficient program specialization in the context of a polymorphic language.
Following the cogen-combinator approach, a set of combinators is defined in Haskell that enables the straightforward transcription of a binding-time polymorphic annotated program into the corresponding program generator. The typing of the combinators mimics the constraints of the binding-time analysis. The resulting program generator is safe, tag-free, and it has no interpretive overhead.

1 Introduction

A polymorphic binding-time analysis empowers an offline partial evaluator to obtain specialization results on par with those of an online partial evaluator. However, implemented specializers for polymorphic binding-time analysis so far do not exploit the efficiency potential of offline partial evaluation. They are interpreter-based, they pass and interpret binding-time descriptions at specialization time, and they use tagging to distinguish ordinary *static* values from *dynamic* values (generated code).

For monomorphic binding-time analysis, there is a well-known approach to obtain compiled, tag-free program generators that perform offline partial evaluation. The *cogen approach* to partial evaluation [13] explains the direct construction of a program generator from a binding-time annotated program. For typed languages, this direct generation step is more efficient than going via the Futamura projections, which can lead to multiple levels of data encoding [11].

For example, the binding-time annotated `power` function

$$\mathit{power}\ x^D\ n^S = \texttt{if}\ n =^S 0\ \texttt{then}\ \mathit{lift}\ 1\ \texttt{else}\ x\ *^D\ \mathit{power}\ x\ (n -^S 1)$$

uses the superscripts ^{S} and ^{D} to indicate static and dynamic operations that happen at specialization time and at run time, respectively. The *lift* expression avoids a binding-time mismatch by converting the static constant 1 into the required dynamic code at that point. The translation to a program generator can be done in a compositional way, by specifying a translation for each annotated syntactic construct: The constructs annotated with ^{S} are translated to

themselves, the constructs annotated with ^{D} are translated to expressions that generate the respective expression tree, and *lift* maps to the appropriate syntax constructor. A translation to Haskell would look like this:

```
data Exp = Const Int | Mul Exp Exp -- and so on
power :: Exp -> Int -> Exp
power x n = if n==0 then Const 1 else Mul x (power x (n-1))
```

This simple example already demonstrates that the static data is neither encoded nor tagged and that, consequently, the static expressions execute efficiently.

The methods used so far for translating binding-time annotated programs to program generators are only suitable for monovariant annotation schemes [1, 2, 18, 20]. They do not cover annotations created by the more precise polyvariant binding-time analyses [5, 6, 8–10]. A polyvariant binding-time analysis enables abstraction over concrete binding times. To continue the example, the *power* function would receive three additional binding-time parameters that express the binding times of the arguments x and n and of the result of *power*, which must be more dynamic than either argument:

$$(power :: \texttt{Int} \to \texttt{Int} \to \texttt{Int}) : \forall\beta\gamma\delta.(\beta \le \delta, \gamma \le \delta) \Rightarrow \beta \overset{S}{\mapsto} \gamma \overset{S}{\mapsto} \delta$$
$$power\ \beta\ \gamma\ \delta\ x\ n = \texttt{if}\ n =^{\gamma} lift^{S,\gamma}\ 0\ \texttt{then}\ lift^{S,\delta}\ 1$$
$$\texttt{else}\ lift^{\beta,\delta}\ x\ *^{\delta}\ power\ \beta\ \gamma\ x\ (n -^{\gamma} lift^{S,\gamma}\ 1)$$

Evidently, the *lift* expression must be generalized to $lift^{\beta,\delta}$ which converts a base-type value of binding time β to binding time δ. This conversion requires $\beta \le \delta$ where the ordering is the least partial order such that $S \le D$. The other constraint, $\gamma \le \delta$, arises from the conditional. The binding time γ of the condition is a lower bound of the binding time δ of the result.

The translation of this annotated program to a satisfactory program generator becomes more tricky. Fig. 1 shows the naive approach, which is hardly satisfactory. First, binding times have to be passed and tested explicitly in the generator. Second, the generator relies on run-time tags to identify static and dynamic values in the `Value` datatype as evident from the implementations of `pIf` and `pOp2`: An untagged generator could omit stripping off (and reapplying) the `Bool` and `Exp` tags. Indeed, the `BT` argument would not be needed for the `pIf` combinator.[3] Third, the generator is not self-checking. Its type does not incorporate the constraints from the binding-time annotation, thus it can give rise to run-time errors because of binding-time mismatches. For example, an invocation `(powergen D D S)` can result in a run-time error when attempting to execute `(pLift D S x)`.

This particular generator has further shortcomings not addressed in this work. For example, `(powergen D D D x n)` does not terminate, for any `x` and

[3] Use of the combinator still requires a preceding binding-time analysis because it expects its `e1` and `e2` arguments to carry the `Exp` tag if the condition carries an `Exp` tag. Unlike the staged interpreters of Carette and coworkers [3], this `pIf` combinator would not be suitable for online partial evaluation because the dynamic version of the conditional does not convert static values in the branches to dynamic values.

```
data Value = Int Int | Bool Bool | Exp Exp
data BT = S | D
powergen :: BT -> BT -> BT -> Value -> Value -> Value
powergen b g d x n =
  pIf g (pOp2 g opEqu n (pLift S g (Int 0)))
        (pLift S d (Int 1))
        (pOp2 d opMul (pLift b d x)
                      (powergen b g d x (pOp2 g opSub n (pLift S g 1))))
-- lifting values of base type
pLift :: BT -> BT -> Value -> Value
pLift S S (Int i) = Int i
pLift S D (Int i) = Exp (Const i)
pLift D D (Exp e) = Exp e
-- conditional
pIf  :: BT -> Value -> Value -> Value -> Value
pIf S (Bool x) v1 v2 = if x v1 else v2
pIf D (Exp e) (Exp e1) (Exp e2) = Exp (If e e1 e2)
-- binary operator
pOp2 :: BT -> Op (Int -> Int -> Int) -> Value -> Value -> Value
pOp2 S op (Int x) (Int y) = Int (opvalue op x y)
pOp2 D op (Exp x) (Exp y) = Exp (opctor  op [x, y])
-- operators
data Op t = Op { opvalue :: t, opctor :: [Exp] -> Exp }
opMul = Op (*)  (\[x,y] -> Mul x y) :: Op (Int -> Int -> Int)
opSub = Op (-)  (\[x,y] -> Sub x y) :: Op (Int -> Int -> Int)
opEqu = Op (==) (\[x,y] -> Equ x y) :: Op (Int -> Int -> Bool)
```

Fig. 1. Naive generator with binding-time polymorphism.

`n`, because the recursive call to `powergen` is implicitly static, that is, it is always performed at specialization time.

The present work is the first to address the construction of efficient program generators with polymorphic binding times. Because it applies to languages with ML-style polymorphism, it paves the way for efficient program specialization for Haskell. It addresses all shortcomings of the naive generator.

1. **No interpretive overhead**. Binding-time descriptions are passed at run time but they are never tested. Due to laziness they have virtually no cost.
2. **Tag-free**. The generator requires no tagging, neither type tags nor tags to distinguish static from dynamic values.
3. **Safety**. The typing of the generator ensures that binding-time inconsistencies in the input of the generator are caught by the type checker before starting the specialization.

The main contribution is a set of combinators that enables the construction of tag-free program generators via a simple type-directed translation from a polymorphic binding-time type derivation to a Haskell program using these combinators. The starting point is the polyvariant binding-time analysis for ML-style polymorphic languages by Glynn and coworkers [6].

```
powergen :: (...) => (forall a. a -> dx a) -> (forall a. a -> dn a)
                  -> (forall a. a -> dz a)
         -> R Int (dx Int) -> R Int (dn Int) -> R Int (dz Int)
powergen dx dn dz x n =
    cIf (dn bool) (dz int)
        (cOp2 sEqInt oEqInt n (cSub (St int) (dn int) (R 0)))
        (cSub (St int) (dz int) (R 1))
        (cOp2 sMult oMult
              (cSub (dx int) (dz int) x)
              (powergen dx dn dz
               x (cOp2 sMinus oMinus n (cSub (St int) (dn int) (R 1)))))
 where { sEqInt = bop2 dn int int bool; sMinus = bop2 dn int int int;
         sMult = bop2 dz int int int }
```

Fig. 2. Tagfree generator for specializations of `power`. The type signature is truncated to save space.

The implementation is in Haskell [15] with various extensions (e.g., type functions [17], multi parameter type classes, rank-2 types [16], GADTs) as implemented by GHC. For lack of space, we assume familiarity with the language and the extensions.

2 Tagfree Polymorphic Program Generation

Figure 2 contains the tag free variant of the polymorphic generator for the `power` function shown in Fig. 1. Before delving into a detailed explanation of the combinators, let's introduce some preliminaries and run the generator on examples.

Like the previous generator, the new generator receives three binding-time parameters and two value parameters. Binding times are represented by polymorphic functions that construct binding-time descriptions (bt descriptions), which are passed to the combinators. A bt description has the same structure as the underlying type but alternates binding times with regular type constructors. Binding times are represented by two data types, `St` and `Dy`.

```
newtype St a = St a  -- static annotation
newtype Dy a = Dy a  -- dynamic annotation
```

For example, `(St int :: St Int)` describes a static integer and `(St (St Int -> Dy Bool))` is the type of a description of a static function with static input and dynamic output. *Descriptions are reified type arguments, which are never evaluated.*

Depending on the instantiation of the binding time parameters, `powergen` exhibits dramatically different behaviors as shown and labeled in Fig. 3. The nontermination of the third example is the expected behavior because the recursion in `powergen` is always static. The error message for the last example accurately reflects the failing constraints of the binding-time analysis (§2.1, §2.3).

The computation of the generator happens in terms of a representation type `R t btd`, which depends on the underlying type `t` and its bt description `btd`.

```
> -- an all static run computes the power function
> unR (powergen St St St (R 2) (R 5))
32
> -- a run with dynamic basis performs specialization
> toString $ unR $ powergen Dy St Dy (R (EVar "x")) (R 5)
"EOp2 (*) (EVar x) (EOp2 (*) (EVar x) (EOp2 (*) (EVar x)
      (EOp2 (*) (EVar x) (EOp2 (*) (EVar x) (EInt 1)))))"
> -- nonterminating specialization
> toString $ unR $ powergen Dy Dy Dy (R (EVar"x")) (R (EVar"n"))
"EIf (EOp2 (==) (EVar n) (EInt 0)) (EInt 1) (EOp2 (*) (EVar x)
(EIf (EOp2 (==) (EOp2 (-) (EVar n) (EInt 1)) (EInt 0)) (EInt 1) (EOp2 (*) (EVar x)
(EIf (EOp2 (==) Interrupted.
> -- binding-time mismatch
> powergen Dy Dy St
    No instances for (CIF Dy St, CSUB (Dy Int) (St Int) Int)
```

Fig. 3. Running `powergen`.

Any value that is passed into (out of) the generator must first be wrapped (unwrapped). As `R` is an isomorphism, its use does *not* amount to tagging.[4]

```
-- representation type
newtype R t btd = R { unR :: ImpT t btd }
-- implementation type
type family ImpT t btd
--
type instance ImpT Int      (St Int)        = Int
type instance ImpT Bool     (St Bool)       = Bool
type instance ImpT [a]      (St [ba])       = [ImpT a ba]
type instance ImpT (a -> b) (St (ba -> bb)) = ImpT a ba -> ImpT b bb
--
type instance ImpT a        (Dy aa)         = Exp a
```

The argument to the `R` constructor must have the *implementation type*, computed by the type function `ImpT`. For type constructors with static bt descriptions, `ImpT` rebuilds the type constructors and translates components of the type recursively. This strategy implies that static computations are implemented by themselves. If the translation hits a dynamic annotation, then well-formedness dictates that further components of the type carry a dynamic annotation, too. Hence, any value of dynamic type `a` is implemented as an expression of type `Exp a`. The latter type is a GADT with the usual definition (see appendix).

2.1 Basic Combinators

Continuing the analysis of the code in Fig. 2, the `cIf` combinator takes two bt descriptions, one `(dn bool)` describing the binding time of the condition and one

[4] The reader may wonder why `R t btd` is needed as it is isomorphic to `ImpT t btd`. However, when type inference equates `R t1 b1 = R t2 b2` it can deduce that `t1 = t2` and `b1 = b2`. It cannot deduce these equalities from `ImpT t1 b1 = ImpT t2 b2`.

`(dz int)` fixing the binding time of the result of the conditional. The remaining arguments stand for the condition, the true-branch, and the false-branch, where the two branches have the same representation type. `cIf` is overloaded such that there are instances for either static `dn` and arbitrary `dz` or for dynamic `dn` and `dz`. The type checker rejects any other combination of binding times (via unresolved instance), thus enforcing the constraints of the binding-time analysis. The definition shows that the bt descriptions are not touched.

```
class CIF bb bt where
  cIf :: bb Bool -> bt shp
      -> R Bool (bb Bool) -> R t (bt shp) -> R t (bt shp) -> R t (bt shp)
instance CIF St bt where
  cIf _ _ b x y = if (unR b) then x else y
instance CIF Dy Dy where
  cIf _ _ b x y = R (EIf (unR b) (unR x) (unR y))
```

The combinator `cOp2` for binary primitive operations takes a bt description for the type of the operation, the operation itself, and its two arguments. It is implemented in terms of a more general operator `cConst`, which injects constants into a generator, and function application `cApp`. Again, the overloading of these combinators enables the dual use of static and dynamic operations.

```
data Op a = Op { opname :: String, opvalue :: a }
cOp2 btd op x y = cApp undefined (cApp btd (cConst btd op) x) y
```

Instead of examining the unwieldy type of `cOp2`, it is simpler and more general to look at the `cConst` operator (but see Appendix C). To safely embed a constant of arbitrary type in a generator requires that the constant's bt description is *uniform*, that is, it is either completely static or completely dynamic [19]. This requirement is stronger than the usual well-formedness (see §3.4), which can be enforced locally. Uniformity is asserted with a two-parameter type class `Uniform`.

```
class Uniform t btd => CONST t btd where
  cConst :: btd -> Op t -> R t btd
instance (AllStatic t aa) => CONST t (St aa) where
  cConst btd op = R (toImpT btd (opvalue op))
instance (AllDynamic aa) => CONST t (Dy aa) where
  cConst _ op = R (EConst op)
```

The dynamic case is straightforward, but the static case has a slight complication. Because of the recursive definition of `ImpT` for static bt descriptions, the type checker needs a proof that `t` is equal to `ImpT t btd` if `btd` is fully static. The class `Uniform` defines identities providing this proof in the usual way.[5]

```
class Uniform t btd where
  toImpT :: btd -> t -> ImpT t btd
  fromImpT :: btd -> ImpT t btd -> t
```

[5] See the appendix for the full definitions of `Uniform`, `AllStatic`, and `AllDynamic`.

2.2 Function Combinators

The encoding of functions follows the ideas of higher-order abstract syntax as in previous work [3, 18, 20]. Thus, the generator represents bound variables by metavariables so that the `cLam` combinator for abstraction takes a function from representation type to representation type as an argument.

```
class CLAM bf bs bt where
  cLam :: bf (bs as -> bt at)
       -> (R s (bs as) -> R t (bt at)) -> R (s -> t) (bf (bs as -> bt at))
instance CLAM St bs bt where
  cLam _ f = R $ (unR . f . R)
instance CLAM Dy Dy Dy where
  cLam _ f = R $ ELam (unR . f . R)
```

The two instances for the overloaded `cLam` combinator reflect exactly the binding-time constraints: a static function does not restrict the binding time of its argument and result, whereas a dynamic function requires dynamic argument and result. The function `ELam` is the constructor for the typed (higher-order) abstract syntax (see Appendix A).

The definitions of the combinators `cApp` for function application and `cFix` for the fixpoint follow a similar scheme and are thus relegated to Appendix D.

2.3 Subtyping

Subtyping is the final important ingredient of binding-time analysis. This subtyping does not take part on the value level, but on the level of bt descriptions and expresses conversions between binding times. For example, a static integer can be converted into one of unknown binding time by the coercion `(cSub (St int) (dz int) (R 1))` from Fig. 2.

In general, a coercion `(cSub bfrom bto v)` takes two bt descriptions and converts value `v` from binding time `bfrom` to binding time `bto`. The function `cSub` is defined in type class `CSUB`.

```
class CSUB b1 b2 t where
  cSub :: b1 -> b2 -> R t b1 -> R t b2
```

The instances of this class follow the inductive definition of the subtyping relation in the binding-time analysis (see §3.4). For base types, it corresponds to the well-known lifting operation.

```
-- reflexivity
instance CSUB a a t where
  cSub _ _ = id
-- base
instance CSUB (St Int) (Dy Int) Int where
  cSub _ _ = R . EInt . unR
instance CSUB (St Bool) (Dy Bool) Bool where
  cSub _ _ = R . EBool . unR
```

For function types, the code for the instances also follows the inductive definition, but it requires extra type annotations for technical reasons. Appendix E contains the full definitions.

2.4 List Operations

Using the same principles as for functions, it is straightforward to develop combinators that support partially static operations on recursive data types. The signature for list-processing combinators serves as an example. Appendix F contains their implementations.

```
class L d da where
  -- d : bt of list, da : bt of elements
  cNil  :: d [da s] -> R [a] (d [da s])
  cCons :: d [da s] -> R a (da s) -> R [a] (d [da s]) -> R [a] (d [da s])
  cHead :: d [da s] -> R [a] (d [da s]) -> R a [da s]
  cTail :: d [da s] -> R [a] (d [da s]) -> R [a] [da s]
  cNull :: d [da s] -> R [a] (d [da s]) -> R Bool (d Bool)
```

3 From Binding-Time Analysis to Tagfree Program Generators

This section defines a translation that maps a polymorphic binding-time type derivation generated by the polymorphic binding-time analysis of Glynn and coworkers [6] into a valid Haskell program that uses the combinators from §2. Precluding a formal correctness argument, we argue informally that the Haskell types express the binding-time constraints and, thus, that Haskell's type soundness guarantees specialization soundness. Furthermore, our automatic translation scheme relieves the programmer from the cumbersome task of writing tagfree program generators by hand.

Before we formalize the type-directed translation scheme, we recapitulate the essentials of Glynn's and coworkers polymorphic binding-time analysis and establish connections to our set of combinators.

3.1 Underlying Type System

We consider the translation of an ML-style let-polymorphic typed language with base types **Int** and **Bool**. For brevity, the formalization omits structured data types, but the implementation supports them (§2.4).

$$\begin{array}{ll} \textbf{Types} & t ::= \alpha \mid \textbf{Int} \mid \textbf{Bool} \mid t \to t \\ \textbf{Type Schemes} & \sigma ::= t \mid \forall \bar{\alpha}.t \\ \textbf{Expressions} & e ::= x \mid \lambda x.e \mid e\, e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \end{array}$$

The vector notation $\bar{\alpha}$ represents a sequence $\alpha_1, \ldots \alpha_n$ of type variables. Constructors for numerals and Boolean values are recorded in some initial type environment.

The treatment of Haskell's advanced language feature such as type classes is possible but postponed to future work. For instance, Glynn's and coworkers polymorphic binding-time analysis is performed on GHC's internal System F style type language CORE where type classes have already been 'removed' via the dictionary-passing translation. Hence, we would require combinators operating on GHC's CORE language directly to properly deal with type classes.

$$\Delta \vdash b : \mathbf{Int} \qquad \Delta \vdash b : \mathbf{Bool} \qquad \frac{(\beta : \alpha) \in \Delta}{\Delta \vdash \beta : \alpha} \qquad \frac{\Delta_1 \vdash \tau_1 : t_1 \quad \Delta_2 \vdash \tau_2 : t_2}{\Delta_1 \cup \Delta_2 \vdash \tau_1 \overset{b}{\mapsto} \tau_2 : t_1 \to t_2}$$

Fig. 4. Shape Rules

3.2 Binding-Time Descriptions

On top of the underlying type structure we impose a binding-time (type) description structure which reflects the structure of the underlying type system. For instance, $S \overset{S}{\mapsto} D$ describes a static function that takes a static value of base type as an argument and returns a dynamic value of base type.

Annotations	$b ::= \delta \mid S \mid D$
Binding-Time Descriptions	$\tau ::= \beta \mid b \mid \tau \overset{b}{\mapsto} \tau$
Binding-Time Type Schemes	$\eta ::= \tau \mid \forall \bar{\beta}, \bar{\delta}.C \Rightarrow \tau$
Constraints	$C ::= (\tau \leq \tau) \mid \mathit{wft}(\tau) \mid C \wedge C$

The grammar distinguishes annotation variables δ, which may only be instantiated to S or D, from binding-time type variables β, which may be instantiated to any τ, including δ. Constraints are described in §3.4.

3.3 Shapes

The binding-time description of an expression must generally have the same 'shape' as its underlying type, in particular in the presence of polymorphism. For this purpose, a shape environment Δ maps a polymorphic binding-time description variable to its corresponding underlying polymorphic type variable. The judgment $\Delta \vdash \tau : t$ states that under shape environment Δ the binding-time description τ has shape t. A judgment $\Delta \vdash \eta : \sigma$ is valid if it can be derived by the shape rules in Figure 4. For brevity, we omit the straightforward rule for quantified types.

The combinator system in §2 detects ill-shaped types via unresolved instances. For example, the (ill-shaped) type `R (a -> b) (St Int)` yields the unresolved type function application `ImpT (a -> b) (St Int)`.

3.4 Binding-Time Constraints

A subtype constraint $(x \leq y)$ is read as "y is at least as dynamic as x". It comes in various flavors: an ordering on annotations $(\cdot \leq_a \cdot)$, a structural ordering $(\cdot \leq_s \cdot)$ on bt descriptions, and an auxiliary ordering $(\cdot \leq_f \cdot)$, which is used in combination with the 'well-formed' constraint $\mathit{wft}()$ to rule out 'ill-formed' constraints such as $S \overset{D}{\mapsto} S$. Figure 5 summarizes the constraint rules.

$$\text{(Sta)}\ C \vdash (S \leq_a b) \qquad \text{(Dyn)}\ C \vdash (b \leq_a D)$$

$$\text{(Hyp)}\ C_1, (b_1 \leq_a b_2), C_2 \vdash (b_1 \leq_a b_2)$$

$$\text{(Refl)}\ C \vdash (b \leq_a b) \qquad \text{(Trans)}\ \frac{C \vdash (b_1 \leq_a b_2) \quad C \vdash (b_2 \leq_a b_3)}{C \vdash (b_1 \leq_a b_3)}$$

$$\text{(Bas}_w\text{)}\ C \vdash \mathit{wft}(b) \qquad \text{(Arrow}_w\text{)}\ \frac{\begin{array}{c} C \vdash (b_3 \leq_f \tau_1) \quad C \vdash \mathit{wft}(\tau_1) \\ C \vdash (b_3 \leq_f \tau_2) \quad C \vdash \mathit{wft}(\tau_2) \end{array}}{C \vdash \mathit{wft}(\tau_1 \overset{b_3}{\mapsto} \tau_2)}$$

$$\text{(Bas}_f\text{)}\ \frac{C \vdash (b_1 \leq_a b_2)}{C \vdash (b_1 \leq_f b_2)} \qquad \text{(Arrow}_f\text{)}\ \frac{C \vdash (b_1 \leq_a b_2)}{C \vdash (b_1 \leq_f \tau_1 \overset{b_2}{\mapsto} \tau_3)}$$

$$\text{(Bas}_s\text{)}\ \frac{C \vdash (b_1 \leq_a b_2)}{C \vdash (b_1 \leq_s b_2)} \qquad \text{(Arrow}_s\text{)}\ \frac{\begin{array}{c} C \vdash (b_2 \leq_a b_5) \\ C \vdash (\tau_4 \leq_s \tau_1) \quad C \vdash (\tau_3 \leq_s \tau_6) \end{array}}{C \vdash (\tau_1 \overset{b_2}{\mapsto} \tau_3 \leq_s \tau_4 \overset{b_5}{\mapsto} \tau_6)}$$

Fig. 5. Binding-Time Constraint Rules

Our combinator system detects ill-formed constraints via unresolved instances. For example, the irreducible constraint `CLAM Dy St St` corresponds to the ill-formed description $S \overset{D}{\mapsto} S$. A binding-time description is only well-formed if a dynamic annotation at the top of a binding-time description implies that all its components are dynamic, too, because nothing can be known about them. Hence, the above constraint is ill-formed.

The remaining binding-time subtype relations are expressed via the type class `CSUB` and its instances (§2.3). The instance bodies construct the necessary coercions among binding-time values.

To be honest, the Haskell encoding leads to a slightly inferior system for the following reasons. First, the transitivity rule (Trans) cannot be easily expressed because the straightforward encoding in Haskell

```
instance (CSUB a b t, CSUB b c t) => CSUB a c t
```

requires guessing the intermediate type `b` during type class instance resolution. A second short-coming of the Haskell encoding is that out of `CSUB (St (a -> b)) (St (a -> c)) (Int -> Int)` we cannot extract the proof term (a.k.a. dictionary) connected to `CSUB b c Int`. The reverse direction is of course possible. Hence, if a program text requires `CSUB (St (a -> b)) (St (a -> c)) (Int -> Int)` but the surrounding context only provides `CSUB b c Int`, Haskell's type inference will fail. A simple workaround for both problems is to provide additional constraints which either mimic application of the transitivity rule or supply the necessary proof terms.

3.5 Type-Directed Translation from BTA to Program Generators

Now everything is in place to describe the automatic construction of program generators based on our combinators out of Glynn and coworkers binding-time analysis. The construction is achieved via a type-directed translation scheme and relies on judgments of the form $C, \Gamma \vdash (e :: t) : \tau \rightsquigarrow (e_H \mid C_H)$ where C is a binding-time constraint, Γ a binding-time environment, e an expression well-typed in the underlying system with type t, τ is a binding-time description, e_H is the Haskell expression derived from e instrumented with program generator combinators and C_H is a Haskell constraint which contains all the requested combinator instances including subtype (coercion) constraints.

Figure 6 contains the (non-syntax directed) translation rules. It is an easy exercise to make them syntax directed, following Glynn and coworkers [6].

In rule (Sub), $C \rightsquigarrow C_H$ denotes the translation of binding-time subtype constraints ($\tau_1 \leq \tau_2$) to Haskell type class constraints `CSUB` τ_1 τ_2 t for some appropriate t.[6] Ill-formed binding-time descriptions are caught via unresolved instances. Hence, the translation simply drops the well-formed constraint $\mathit{wft}(\tau)$. The translation of the judgment $C \vdash (\tau_2 \leq_s \tau_1)$ to the Haskell setting may not hold any more, unless C contains redundant constraints as discussed in §3.4. Hence, we assume from now on that such redundant constraints are present in C. In the resulting (Haskell) program text, the expression e_H is coerced to the expected bt description τ_1 by inserting the combinator call `cSub` τ_2 τ_1. Descriptions such as τ_1 occurring in expressions are short-hands for `undefined` :: τ_1 where variables appearing in τ_1 are bound by lexically scoped type annotations.[7] Recall that binding-time descriptions passed at run-time are never inspected. Thanks to laziness they have virtually no cost.

Rule (Abs) and (App) are straightforward and do not contain any surprises. The rule (Let) additionally abstract over the binding-time descriptions $\bar{\beta}$ and $\bar{\delta}$ which then will be supplied with arguments at the instantiation site (see rule (∀E)). The function *inst* computes the corresponding binding-time description instances for each underlying type instance. Let Δ be a shape environment, $\bar{t}$ a sequence of underlying types, and $\bar{\alpha}$ a sequence of underlying type variables. Let $\mathit{inst}(\Delta, \bar{t}, \bar{\alpha}) = \bar{\tau}$ where $\bar{\tau}$ are fresh binding-time types of appropriate shape: Each element of $\bar{\tau}$ is related to the corresponding element of $\bar{\beta}$ by the shape environment Δ. That is, $\Delta, t_i \vdash \tau_{ij}$ where $\Delta \vdash \beta_{ij} : \alpha_i$.

The last rule (Fix) deals with polymorphic recursion (in the binding-time descriptions). A fixpoint iteration is required to compute the set of combinator instances `CIF` etc. The constraints resulting from $(e :: t)$ are split into those, which are not connected to $\bar{\delta}$ (C_{1H}), and those which constrain $\bar{\delta}$ (C_{2H}). The fixpoint operator $\mathcal{F}$ starts with C_{2H} plus the Haskell equivalent C'_{2H} of the subtype constraints in C_2 and iterates until a fixpoint C_{3H} is found. The exact

[6] In a syntax-directed inference system the program text determines the type t.

[7] The alternative is to build an explicit term of type τ_1 as in Fig. 2.

definition of $\mathcal{F}$ is as follows:

$$\begin{array}{ll} \mathcal{F}(\Gamma \cup \{x : \forall\bar{\delta}.C_{2H} \Rightarrow \tau\}, e :: t) & \\ = \mathcal{F}(\Gamma \cup \{x : \forall\bar{\delta}.C_{2H} \wedge C_{3H} \Rightarrow \tau\}, e :: t) & \text{if } C_{2H} \neq_{set} C_{3H} \\ = C_{2H} & \text{otherwise} \end{array}$$

where $\Gamma \cup \{x : \forall\bar{\delta}.C_{2H} \Rightarrow \tau\} \vdash (e :: t) : \tau \rightsquigarrow (e_H \mid C_{3H})$.

The following example serves to illustrate the fixpoint iteration.

```
f x y = if x == 0 then 1 else f y (x-1)
```

Function `f`'s most general binding time description is

$$\forall b_x, b_y, b.(b_x \leq b) \wedge (b_y \leq b) \Rightarrow b_x \overset{S}{\mapsto} b_y \overset{S}{\mapsto} b$$

Binding-time polymorphism is required in the subexpression `f y (x-1)` for building the instance $b_y \overset{S}{\mapsto} b_x \overset{S}{\mapsto} b$.

The first step of the translation yields the following (Haskell) constraints from the program text:

```
SUB bx b Int, SUB by b Int,
CIF bx b,
CLAM St by b, CLAM St bx (by->b),
CAPP St b x b, CAPP St by (bx->b)
```

These constraints are not sufficient for the resulting program to type check. For example, at the instantiation site `f y (x-1)` the constraint `CIF by b` is needed but there is only `CIF bx b`. Another iteration starting with the above constraints leads to the fixpoint:[8]

```
SUB bx b Int, SUB by b Int,
CIF bx b, CIF by b,
CLAM St by b, CLAM St bx (by->b), CLAM St bx b, CLAM St by (bx->b),
CAPP St bx b, CAPP St by (bx->b), CAPP St by b, CAPP St bx (by->b)
```

The fixpoint iteration must terminate because it only iterates over annotations whose shape is fixed/bound by the underlying type. Hence, the number of instances arising is finite.

An alternative translation scheme could employ the `cFix` combinator also provided by the library. It corresponds to a monomorphic (Fix) rule, which requires no fixpoint iteration.

In summary, the type-directed translation scheme builds a tight correspondence between the typing of the combinators and the typing rules of the binding-time analysis. It might be stated as a slogan in the following way.

Proposition 1. *Let* $True, \emptyset \vdash (e :: t) : \tau \rightsquigarrow (e_H \mid C_H)$. *Then, the resulting expression* e_H *is well-typed in Haskell with type* $R\ t\ \tau$ *under constraints* C_H.

[8] The fixpoint iteration requires a variant of the (∀E) rule which also infers the required instantiation constraints, rather than simply checking if the provided constraints imply the instantiation constraints. For reasons of space, we omit the straightforward details.

We have no proof for this proposition, although it is easy in many cases to match the typing of a single combinator with its corresponding typing rule. An attempt to prove it would have to overcome the shortcomings discussed in the preceding text and it would have to draw on a formalization of a large subset of Haskell's type system. Both tasks are out of scope of the present work.

4 Related Work and Conclusion

Among the large body of related work on partial evaluation (see the respective overviews [7, 11]), there are only few works which consider offline partial evaluation based on a polymorphic binding-time analysis for polymorphic languages [6, 8, 9]. None of them consider the direct construction of program generators. Only Heldal and Hughes [8] deal with the pragmatics of constructing the specializer. Other works that consider polymorphism concentrate either on polymorphic binding-time analysis for monomorphic languages [5, 10] or monomorphic analysis for polymorphic languages [4, 12, 14].

Closely related are previous constructions of combinators that perform specialization by the first author [18, 20] as well as combinators by Carette and coworkers [3] that can be statically configured (either via overloading or via the ML module language) to perform evaluation, compilation, or (online) partial evaluation. Two main differences to the latter work are (1) that our combinators are geared towards offline partial evaluation and require a preceding binding-time analysis and (2) that our combinators are dynamically configured by type passing.

The present work complements the earlier work of Glynn and coworkers [6] and puts it into practice. Our combinators solve the open question of obtaining safe and efficient (tag-free) program generators for ML-style languages based on a polymorphic binding-time analysis. Our proof-of-concept implementation relies on GHC's advanced (source) typing features and allows us to experiment with smaller examples.

There are many opportunities for future work. We doubt that there an analogous set of combinators that can be implemented in ML, but it is an interesting question to consider. We believe that the approach is extensible to typing features of Haskell beyond ML. We further believe that the approach can be extended to cater for typical partial evaluation features like program point specialization, multi-level specialization, or continuation-based specialization.

References

1. L. Birkedal and M. Welinder. Hand-writing program generator generators. In M. V. Hermenegildo and J. Penjam, editors, *Intl. Symp. Programming Languages, Implementations, Logics and Programs (PLILP '94)*, volume 844 of *LNCS*, pages 198–214, Madrid, Spain, Sept. 1994. Springer.
2. A. Bondorf and D. Dussart. Improving CPS-based partial evaluation: Writing cogen by hand. In P. Sestoft and H. Søndergaard, editors, *Proc. 1994 ACM Workshop Partial Evaluation and Semantics-Based Program Manipulation*, pages 1–10,

Orlando, Fla., June 1994. University of Melbourne, Australia. Technical Report 94/9, Department of Computer Science.
3. J. Carette, O. Kiselyov, and C. chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009.
4. A. De Niel, E. Bevers, and K. De Vlaminck. Partial evaluation of polymorphically typed functional languages: the representation problem. In *JTASPEFT/WSA'91*, pages 90–97, 1991.
5. D. Dussart, F. Henglein, and C. Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In A. Mycroft, editor, *Proc. 1995 International Static Analysis Symposium*, volume 983 of *LNCS*, pages 118–136, Glasgow, Scotland, Sept. 1995. Springer.
6. K. Glynn, P. Stuckey, M. Sulzmann, and H. Søndergaard. Boolean constraints for binding-time analysis. In *Programs as Data Objects II*, volume 2053 of *LNCS*, pages 39–62, Aarhus, Denmark, May 2001. Springer.
7. J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, editors. *Partial Evaluation—Practice and Theory. Proceedings of the 1998 DIKU International Summerschool*, volume 1706 of *LNCS*, Copenhagen, Denmark, 1999. Springer.
8. R. Heldal and J. Hughes. Binding-time analysis for polymorphic types. In *PSI-01: Andrei Ershov Fourth International Conference, Perspectives of System Informatics*, volume 2244 of *LNCS*, pages 191–204, Novosibirsk, Russia, July 2001. Springer.
9. S. Helsen and P. Thiemann. Polymorphic specialization for ML. *ACM TOPLAS*, 26(4):1–50, July 2004.
10. F. Henglein and C. Mossin. Polymorphic binding-time analysis. In D. Sannella, editor, *Proc. 5th ESOP*, volume 788 of *LNCS*, pages 287–301, Edinburgh, UK, Apr. 1994. Springer.
11. N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
12. J. Launchbury. A strongly-typed self-applicable partial evaluator. In J. Hughes, editor, *Proc. FPCA 1991*, volume 523 of *LNCS*, pages 145–164, Cambridge, MA, USA, 1991. Springer.
13. J. Launchbury and C. K. Holst. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming*, pages 210–218, Skye, Scotland, 1991. Glasgow University.
14. T. Æ. Mogensen. Binding time analysis for polymorphically typed higher order languages. In J. Díaz and F. Orejas, editors, *TAPSOFT '89*, volume 351,352 of *LNCS*, pages II, 298–312, Barcelona, Spain, Mar. 1989. Springer.
15. S. Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.
16. S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17(1):1–82, 2007.
17. T. Schrijvers, S. L. Peyton Jones, M. M. T. Chakravarty, and M. Sulzmann. Type checking with open type functions. In P. Thiemann, editor, *Proc. ICFP 2008*, pages 51–62, Victoria, BC, Canada, Oct. 2008. ACM Press, New York.
18. P. Thiemann. Cogen in six lines. In K. Dybvig, editor, *Proc. 1996 ICFP*, pages 180–189, Philadelphia, PA, May 1996. ACM Press, New York.
19. P. Thiemann. Aspects of the PGG system: Specialization for standard Scheme. In Hatcliff et al. [7], pages 412–432.
20. P. Thiemann. Combinators for program generation. *J. Funct. Program.*, 9(5):483–525, Sept. 1999.

$$\text{(Var)}\quad \frac{(x:\eta)\in\Gamma \quad \Delta \vdash \eta:\sigma \quad C \rightsquigarrow C_H}{C,\Gamma \vdash (x :: \sigma) : \eta \rightsquigarrow (x \mid C_H)}$$

$$\text{(Sub)}\quad \frac{C,\Gamma \vdash (e :: t) : \tau_2 \rightsquigarrow (e_H \mid C_H) \quad C \vdash (\tau_2 \leq_s \tau_1) \quad C \vdash \mathit{wft}(\tau_1)}{C,\Gamma \vdash (e :: t) : \tau_1 \rightsquigarrow (\texttt{cSub}\ \tau_2\ \tau_1\ e_H \mid C_H \wedge \texttt{CSUB}\ \tau_2\ \tau_1\ t)}$$

$$\text{(Abs)}\quad \frac{\begin{array}{c} C,\Gamma \cup \{x:\tau_1\} \vdash (e :: t_2) : \tau_2 \rightsquigarrow (e_H \mid C_H) \\ C \vdash \mathit{wft}(\tau_1) \quad \Delta \vdash \tau_1 : t_1 \end{array}}{\begin{array}{c} C,\Gamma \vdash (\lambda x.e :: t_1 \to t_2) : \tau_1 \overset{S}{\mapsto} \tau_2 \\ \rightsquigarrow \\ (\texttt{cLam}\ \texttt{ST}(\tau_1 \mapsto \tau_2)\ (\lambda x.e_H) \mid C_H \wedge \texttt{CLAM}\ \texttt{ST}\ \tau_1\ \tau_2\}) \end{array}}$$

$$\text{(App)}\quad \frac{\begin{array}{c} C_1,\Gamma \vdash (e_1 :: t_1 \to t_2) : (\tau_1 \overset{b}{\mapsto} \tau_2) \rightsquigarrow (e_{1H} \mid C_{1H}) \\ C_2,\Gamma \vdash (e_2 :: t_1) : \tau_1 \rightsquigarrow (e_{2H} \mid C_{2H}) \end{array}}{\begin{array}{c} C_1 \wedge C_2,\Gamma \vdash (e_1\ e_2 :: t_2) : \tau_2 \\ \rightsquigarrow \\ (\texttt{cApp}\ (b\ (\tau_1 \mapsto \tau_2))\ e_{1H}\ e_{2H} \mid C_{1H} \wedge C_{2H} \wedge \texttt{CAPP}\ b\ \tau_1\ \tau_2) \end{array}}$$

$$\text{(Let)}\quad \frac{\begin{array}{c} C_1,\Gamma \vdash (e_1 :: t_1) : \tau_1 \rightsquigarrow (e_{1H} \mid C_{1H}) \\ \Delta \vdash \tau_1 : t_1 \quad \bar{\beta},\bar{\delta} \subseteq \mathit{fv}(C_1,\tau_1) \backslash \mathit{fv}(\Gamma) \\ \text{where} \quad \Delta \vdash \beta_{ij} : \alpha_i \\ C_2,\Gamma \cup \{x : \forall\bar{\beta}\bar{\delta}.C_1 \Rightarrow \tau_1\} \vdash (e_2 :: t_2) : \tau_2 \rightsquigarrow (e_{2H} \mid C_{2H}) \end{array}}{\begin{array}{c} C_2,\Gamma \vdash (\mathsf{let}\ x = (e_1 :: \forall\bar{\alpha}.t_1)\ \mathsf{in}\ e_2 :: t_2) : \tau_2 \\ \rightsquigarrow \\ (\mathsf{let}\ x = (\lambda\bar{\beta}.\lambda\bar{\delta}.e_{1H} :: \forall\bar{\alpha}\bar{\beta}\bar{\delta}.C_{1H} \Rightarrow R\ t_1\ \tau_1)\ \mathsf{in}\ e_{2H} \mid C_{2H}) \end{array}}$$

$$\text{(}\forall\text{E)}\quad \frac{\begin{array}{c} C,\Gamma \vdash (e :: \forall\bar{\alpha}.t) : \forall\bar{\beta},\bar{\delta}.D \Rightarrow \tau \rightsquigarrow (e_H \mid C_H) \\ \Delta \vdash \tau : t \quad \mathit{inst}(\Delta,\bar{t},\bar{\alpha}) = \bar{\tau} \\ C \vdash [\bar{\tau}/\bar{\beta},\bar{b}/\bar{\delta}]D \end{array}}{C,\Gamma \vdash (e :: [\bar{t}/\bar{\alpha}]t) : [\bar{\tau}/\bar{\beta},\bar{b}/\bar{\delta}]\tau \rightsquigarrow (e_H\ \bar{\tau}\ \bar{b} \mid C_H)}$$

$$\text{(Fix)}\quad \frac{\begin{array}{c} \eta = \forall\bar{\delta}.C_2 \Rightarrow \tau \quad C_2 \rightsquigarrow C_2'_H \\ C_1 \wedge C_2,\Gamma \cup \{x:\eta\} \vdash (e :: t) : \tau \rightsquigarrow (e_H \mid C_{1H} \wedge C_{2H}) \\ C_1 \wedge C_2 \vdash \mathit{wft}(\tau) \quad \Delta \vdash \eta : t \\ \mathit{fv}(C_{1H}) \cap \bar{\delta} = \emptyset \quad \mathit{fv}(C_{2H}) \subseteq \bar{\delta} \\ \mathcal{F}(\Gamma \cup \{x : \forall\bar{\delta}.C_2'_H \wedge C_{2H} \Rightarrow \tau\}, e :: t) = C_{3H} \end{array}}{\begin{array}{c} C_1,\Gamma \vdash ((\mathsf{fix}\ x :: t\ \mathsf{in}\ e) :: t) : \eta \\ \rightsquigarrow \\ (\mathsf{let}\ x = (\lambda\bar{\delta}.e_H :: \forall\bar{\delta}.C_{3H} \Rightarrow R\ t\ \tau)\ \mathsf{in}\ x \mid C_{1H}) \end{array}}$$

Fig. 6. Type-direction translation rules

This appendix contains additional material for scrutiny by the interested reviewer. The source code of the combinators is available online at http://proglang.informatik.uni-freiburg.de/projects/polyspec/

A Expression Datatype

```
data Exp t where
    EVar :: String -> Exp t
    EBool :: Bool -> Exp Bool
    EInt :: Int -> Exp Int
    EIf :: Exp Bool -> Exp t -> Exp t -> Exp t
    EConst :: Op t -> Exp t
    EOp1 :: Op (a -> t) -> Exp a -> Exp t
    EOp2 :: Op (a -> b -> t) -> Exp a -> Exp b -> Exp t
    ELam :: (Exp a -> Exp b) -> Exp (a -> b)
    EApp :: Exp (a -> b) -> Exp a -> Exp b
    EFix :: Exp (a -> a) -> Exp a
    ENil :: Exp [a]
    ECons :: Exp a -> Exp [a] -> Exp [a]
    EHead :: Exp [a] -> Exp a
    ETail :: Exp [a] -> Exp [a]
    ENull :: Exp [a] -> Exp Bool
```

B Uniform Binding-Time Descriptions

```
class Uniform t sh_t where
  toImpT :: sh_t -> t -> ImpT t sh_t
  fromImpT :: sh_t -> ImpT t sh_t -> t

  toImpT = undefined
  fromImpT = undefined

instance AllStatic t aa => Uniform t (St aa) where
  toImpT ~(St sh) v = toImpT' sh v
  fromImpT ~(St sh) v = fromImpT' sh v
instance AllDynamic aa => Uniform t (Dy aa)

class AllStatic t sh where
  toImpT' :: sh -> t -> ImpT t (St sh)
  toImpT' = error "toImp': AllStatic instance missing"
  fromImpT' :: sh -> ImpT t (St sh) -> t
  fromImpT' = error "fromImpT': AllStatic instance missing"
instance AllStatic Int Int where
  toImpT' _ v = v
  fromImpT' _ v = v
instance AllStatic Bool Bool where
  toImpT' _ v = v
  fromImpT' _ v = v

instance AllStatic' t aa => AllStatic [t] (Con1 aa) where
  toImpT' ~(Con1 sh) vs = map (toImpT'' sh) vs
  fromImpT' ~(Con1 sh) vs = map (fromImpT'' sh) vs
instance (AllStatic' s aa, AllStatic' t ab)
      => AllStatic (s -> t) (Con2 aa ab) where
  toImpT' ~(Con2 sh_s sh_t) f = toImpT'' sh_t . f . fromImpT'' sh_s
  fromImpT' ~(Con2 sh_s sh_t) f = fromImpT'' sh_t . f . toImpT'' sh_s

class AllStatic' t sh where
  toImpT'' :: sh -> t -> ImpT t sh
  fromImpT'' :: sh -> ImpT t sh -> t
instance AllStatic t aa => AllStatic' t (St aa)  where
  toImpT'' ~(St sh) v = toImpT' sh v
  fromImpT'' ~(St sh) v = fromImpT' sh v

class AllDynamic aa
instance AllDynamic Int
instance AllDynamic Bool

instance AllDynamic' aa => AllDynamic [aa]
instance (AllDynamic' aa, AllDynamic' ab) => AllDynamic (aa -> ab)

class AllDynamic' aa
instance AllDynamic aa => AllDynamic' (Dy aa)
```

C Primitive Operators

```
-- construct bt description for function types
fun :: a -> b -> (a -> b)
fun = undefined

-- binary operators
-- bt description for uniformly annotated types of the form a -> b -> c
type BOP2 bt aa ab ac = bt ((bt aa) -> (bt ((bt ab) -> (bt ac))))
bop2 :: (forall a. a -> bt a) -> aa -> ab -> ac -> BOP2 bt aa ab ac
bop2 bt aa ab ac =  bt (fun (bt aa) (bt (fun (bt ab) (bt ac))))

cOp2
  :: (CONST
        (ta -> tb -> tc)
        (bt ((bt aa) -> (bt ((bt ab) -> (bt ac))))),
      CAPP bt bt bt) =>
     BOP2 bt aa ab ac
     -> Op (ta -> tb -> tc)
     -> R ta (bt aa)
     -> R tb (bt ab)
     -> R tc (bt ac)
cOp2 sh op x y =
    cApp undefined (cApp sh (cConst sh op) x) y

-- unary operators
-- bt description for uniformly annotated types of the form a -> b
type BOP1 bt aa ab = bt ((bt aa) -> (bt ab))
bop1 :: (forall a. a -> bt a) -> a1 -> a2 -> BOP1 bt a1 a2
bop1 bt a1 a2 = bt (fun (bt a1) (bt a2))

cOp1 :: (CAPP bt bt bt, CONST (a -> b) (bt ((bt aa) -> (bt bb))))
     => BOP1 bt aa bb -> Op (a -> b) -> R a (bt aa) -> R b (bt bb)
cOp1 sh op x =
    cApp sh (cConst sh op) x
```

D Function Application and Fixpoint

```
class CAPP bt ba bb where
  cApp :: (LE bt ba, LE bt bb)
       => bt (Con2 (ba aa) (bb ab))
       -> R (ta -> tb) (bt (Con2 (ba aa) (bb ab)))
       -> (R ta (ba aa) -> R tb (bb ab))

instance CAPP St ba bb where
  cApp _ f x = R $ unR f (unR x)

instance CAPP Dy Dy Dy where
  cApp _ f x = R $ EApp (unR f) (unR x)

--

class CFIX bt ba where
  cFix :: (LE bt ba)
       => (bt (Con2 (ba aa) (ba aa)))
       -> R (ta -> ta) (bt (Con2 (ba aa) (ba aa)))
       -> R ta (ba aa)

instance CFIX St ba where
  cFix _ f = R $ fix (unR f)

instance CFIX Dy Dy where
  cFix _ f = R $ EFix (unR f)

fix :: (a -> a) -> a
fix f = f (fix f)
```

E Subtyping for Functions

E.1 Static Functions

```
instance (CSUB b3 b1 t1, CSUB b2 b4 t2)
      => CSUB (St (Con2 b1 b2)) (St (Con2 b3 b4)) (t1->t2) where
  cSub _ _ f = subStArrowContra (subStArrowCo f)

-- subStArrowCo casts result type
-- subStArrowContra casts argument type
-- we could of course merge both functions

-- we require lexically scoped type annotations to resolve ambiguities
subStArrowCo :: forall t1 t2 b1 b2 b4.
         CSUB b2 b4 t2
       => R (t1->t2) (St (Con2 b1 b2)) -> R (t1->t2) (St (Con2 b1 b4))
subStArrowCo f = R $ (\x ->
                  let r1 :: ImpT (t1->t2) (St (Con2 b1 b2))
                      r1 = unR f
                      r2 :: ImpT t2 b2
                      r2 = r1 x
                      r3 :: R t2 b2
                      r3 = R r2
                      r4 :: R t2 b4
                      r4 = cSub undefined undefined r3
                      r5 :: ImpT t2 b4
                      r5 = unR r4
                  in r5)
--subStArrowCo f = R $ (\x -> unR (cSub (R ((unR f) x))))


subStArrowContra :: forall t1 t2 b1 b2 b3.
             CSUB b3 b1 t1
          => R (t1->t2) (St (Con2 b1 b2)) -> R (t1->t2) (St (Con2 b3 b2))
subStArrowContra f = R $ (\x ->
                  let r1 :: R t1 b3
                      r1 = R x
                      r2 :: R t1 b1
                      r2 = cSub undefined undefined r1
                      r3 :: ImpT t1 b1
                      r3 = unR r2
                      r4 :: ImpT (t1->t2) (St (Con2 b1 b2))
                      r4 = unR f
                      r5 :: ImpT t2 b2
                      r5 = r4 r3
                  in r5)
-- subStArrowContra f = R $ (\x -> (unR f) (unR (cSub (R x))))
```

E.2 Dynamic Functions

```
-- CSUB (St (b1' -> b2)) (Dy (b3' -> b4')) (t1->t2)
-- implies due to wft that b3' and b4' dynamic
-- CSUB b3' b1' implies b1' dynamic, hence, we obtain
-- the following
instance (CSUB (Dy b3) (Dy b1) t1, CSUB b2 (Dy b4) t2)
      => CSUB (St (Con2 (Dy b1) b2)) (Dy (Con2 (Dy b3) (Dy b4))) (t1->t2) where
  cSub _ _ f = subDyArrowContra (subDyArrowCo f)


subDyArrowCo :: forall t1 t2 b1 b2 b4.
         CSUB b2 (Dy b4) t2
       => R (t1->t2) (St (Con2 (Dy b1) b2)) -> R (t1->t2) (St (Con2 (Dy b1) (Dy b4)))
subDyArrowCo f = R $ (\x ->
                  let r1 :: ImpT (t1->t2) (St (Con2 (Dy b1) b2))
                      r1 = unR f
                      r2 :: ImpT t2 b2
                      r2 = r1 x
                      r3 :: R t2 b2
                      r3 = R r2
                      r4 :: R t2 (Dy b4)
                      r4 = cSub undefined undefined r3
                      r5 :: ImpT t2 (Dy b4)
                      r5 = unR r4
                  in r5)
--subDyArrowCo f = R $ ELam $ (\x -> unR (cSub (R ((unR f) x))))


subDyArrowContra :: forall t1 t2 b1 b2 b3.
             CSUB (Dy b3) (Dy b1) t1
          => R (t1->t2) (St (Con2 (Dy b1) (Dy b2)))
          -> R (t1->t2) (Dy (Con2 (Dy b3) (Dy b2)))
subDyArrowContra f = R $ ELam $ (\x ->
                  let r1 :: R t1 (Dy b3)
                      r1 = R x
                      r2 :: R t1 (Dy b1)
                      r2 = cSub undefined undefined r1
                      r3 :: ImpT t1 (Dy b1)
                      r3 = unR r2
                      r4 :: ImpT (t1->t2) (St (Con2 (Dy b1) (Dy b2)))
                      r4 = unR f
                      r5 :: ImpT t2 (Dy b2)
                      r5 = r4 r3
                  in r5)
-- subStArrowContra f = R $ Elam $ (\x -> (unR f) (unR (cSub (R x))))
```

F List Processing

```
instance L St da where
  cNil  _ = R []
  cCons _ = \(R x) (R y) -> R ( x:y )
  cHead _ = R . head . unR
  cTail _ = R . tail . unR
  cNull _ = R . null . unR

instance L Dy Dy where
  cNil  _ = R ENil
  cCons _ (R x) (R y) = R $ ECons x y
  cHead _ (R x) = R $ EHead x
  cTail _ (R x) = R $ ETail x
  cNull _ = R . ENull . unR
```