Principles of Programming Languages

Peter Thiemann

July 3, 2015

1 Introduction

Contents of the course

- concepts used in programming languages
- prerequisite: **semantics**
  - vocabulary for talking about programming languages
  - tools for describing the meaning of programs
  - techniques for reasoning about programs

Why?

- Understand programs
- Verify program transformations
- Verify compilers
- Verify static analyses

Task of semantics: formally assign meaning to a program text
Two facets of semantics

Static semantics: defines well-formedness of a program (beyond mere syntactical correctness)

Dynamic semantics: describes execution or evaluation of a program

Requires formalization of

- syntax (program text)
- semantics (well-formedness conditions, evaluation)

1.1 Syntax

Traditionally, the syntax of a programming language is described in two steps, lexical syntax and context-free syntax.

1.1.1 Lexical Syntax

The lexical syntax defines the "atoms" of the language in terms of regular languages. The *lexical analysis* (or *scanner* or *lexer*) of a compiler partitions a program into *lexemes* and maps them into *tokens*. (Lexemes are sequences of input characters, tokens are symbolic values.)

Example: Typical lexeme classes are identifiers, numeric literals, opening and closing parentheses, and keywords

class	regexp	example	token
identifier	`[A-Za-z][A-Za-z0-9]*`	`Catch22`	*ident(*`Catch22`*)*
numeric lit	`[-+]?[0-9]+`	`-42`	`num (42)`
opening par	`(`	`(`	*openingPar*
closing par	`)`	`)`	*closingPar*
keyword	`while`	`while`	*kwWhile*

The scanner typically ignores *whitespace*, that is sequences of spaces, tabulators, line feeds, and so on. The scanner also removes comments.

Formally, a scanner is a partial surjective function

$$\text{scan} : \texttt{Char}^* \to \texttt{Token}^*$$

so that there exists a function

$$\text{unscan} : \texttt{Token}^* \to \texttt{Char}^*$$

satisfying

$$\text{scan} = \text{scan} \circ \text{unscan} \circ \text{scan}$$

1.1.2 Context-free Syntax

The context-free syntax of the language is given by a context-free grammar $\mathcal{G}$ in terms of the tokens determined by the lexical analysis. The *parser* for $\mathcal{G}$ maps a token sequence to a derivation tree of $\mathcal{G}$ or fails if the token sequence is not in the language.

Formally, a parser is a partial, surjective function

$$\text{parse} : \texttt{Token}^* \to \texttt{tree}(\mathcal{G})$$

so that there exists a function

$$\text{unparse} : \texttt{tree}(\mathcal{G}) \to \texttt{Token}^*$$

satisfying

$$\text{parse} = \text{parse} \circ \text{unparse} \circ \text{parse}$$

The grammar is used for parsing is not the most natural grammar for the language. It is written so that it is not ambiguous, so that linear-time parsing is possible, and so that properties like operator precedences can be encoded.

For example, the following grammar is popular for parsing infix expressions.

$$\begin{array}{lcl} \langle \mathit{expr} \rangle & \rightarrow & \langle \mathit{factor} \rangle \\ \langle \mathit{expr} \rangle & \rightarrow & \langle \mathit{expr} \rangle - \langle \mathit{factor} \rangle \\ \langle \mathit{factor} \rangle & \rightarrow & \langle \mathit{atom} \rangle \\ \langle \mathit{factor} \rangle & \rightarrow & \langle \mathit{factor} \rangle / \langle \mathit{atom} \rangle \\ \langle \mathit{atom} \rangle & \rightarrow & \mathtt{a} \\ \langle \mathit{atom} \rangle & \rightarrow & (\langle \mathit{expr} \rangle) \end{array}$$

It reflects the convention that `/` binds tighter than `-` and that both associate to the left.

Exercise: Draw a derivation tree for `a / a - (a - a / a)`.

1.1.3 Abstract Syntax

Much of the structure of the derivation tree of a grammar suitable for parsing is irrelevant for the meaing of an expression. For that task, a much simpler structure is sufficient, the *abstract syntax*:

$$e \quad ::= \quad e - e \mid e / e \mid \mathtt{a}$$

The point of this grammar is *not* the set of strings derivable from it, but rather the set of its derivation trees, the *abstract syntax trees.*

Exercise: (not that easy) suppose that `tree` ($\mathcal{G}$) describes concrete syntax trees and `tree` ($\mathcal{A}$) describes abstract syntax trees. Develop a specification for `mkAST` : `tree`($\mathcal{G}$) $\rightarrow$ `tree`($\mathcal{A}$) which makes sure that the set `tree` ($\mathcal{A}$) all interesting information but is also sufficiently abstract. (So the second part of the specification has to justify the choice of $\mathcal{A}$, somehow.)

Technically, an abstract syntax is a set of terms given by a signature of operation symbols (the above grammar is a common, but sloppy way of writing that signature). Alternatively, the non-terminals of the grammar are considered as types and an explicit signature specifies the restrictions.

$$\begin{array}{lcccl} - & : & \langle \mathit{expr} \rangle \times \langle \mathit{expr} \rangle & \rightarrow & \langle \mathit{expr} \rangle \\ / & : & \langle \mathit{expr} \rangle \times \langle \mathit{expr} \rangle & \rightarrow & \langle \mathit{expr} \rangle \\ \mathtt{a} & : & & \rightarrow & \langle \mathit{expr} \rangle \end{array}$$

Some programming languages have builtin support for defining signatures of term forming operators (constructors). For example, Objective Caml allows the following definition for the type `expr`:

```
type expr = subExpr of expr * expr
          | divExpr of expr * expr
          | conExpr
```

(The lexemes `/` and `-` cannot be used because they are predefined by the Objective Caml language.)

Exercise: Take a fragment of JavaScript's grammar and define a set of abstract syntax trees for that fragment. Implement your abstract syntax in a programming language of your choice.

1.2 Semantics

There are a number of techniques for specifying the semantics of a programming language. All of them start with a mapping from abstract syntax into a mathematical model. The three main approaches are

- denotational
- operational
- axiomatic.

Denotational semantics models the meaning of a program phrase by a mathematical object. It is just concerned with the result, not with the individual computation steps. Nevertheless, a denotational semantics often looks like an interpreter and is sometimes implementable as such. One key idea in denotational semantics is *compositionality*: the meaning of a program phrase is a function of the meanings of its direct subphrases.

Operational semantics describes the individual computation steps and how they must be put together to obtain the result of a program run. It comes in two flavors, *natural semantics* and *structural operational semantics.*

- Natural semantics (also called *big-step* or *evaluation-style* semantics) specifies the relation between a program phrase, auxiliary information, and the result of evaluating the phrase.
- Structural operational semantics (also called *small-step* semantics) specifies an evaluation state and *individual computation steps* that relate successive states.

A further possibility is a *reduction semantics* which just specifies the possible computation steps but leaves some latitude where the steps might be applied.

Axiomatic semantics specifies pre- and postconditions for each kind of phrase and relies on a logic to put these together to statements about entire programs. It is well-suited to proving partial correctness of a program.

The focus of this course will be on denotational and operational techniques.

The example of an expression semantics illustrates both styles. The abstract syntax is slightly modified to avoid problems with modeling the partiality of division.

$$\mathsf{Expr} \ni e \quad ::= \quad e{+}e \mid e{*}e \mid \mathsf{a}$$

In each style, the example neither exercises the full power of the style nor does it touch upon its limitations. The style is informal to provide a first idea of what it is about.

1.2.1 Example: Denotational Semantics

Every denotational semantics starts with defining the mathematical domains used for modeling the syntactic phrases. In the example, there is just one domain involved, the *domain of denoted values.* Denoted values are all values that are potential evaluation results.

$$\mathsf{Val} \quad = \quad \mathbf{N}$$

The denotational semantics is a function which is defined by induction on the abstract syntax. It is customary to enclose syntactical domains in double brackets (also called "semantic brackets" or "Strachey bracket" in honor of Christopher Strachey, one of the inventors of denotational semantics).

$$\begin{array}{rcl} \mathcal{E} & : & \mathsf{Expr} \to \mathsf{Val} \\ \mathcal{C} & : & \mathsf{Const} \to \mathsf{Val} \end{array}$$

$$\begin{array}{lcl} \mathcal{E}[\![\mathtt{a}]\!] & = & \mathcal{C}[\![\mathtt{a}]\!] \\ \mathcal{E}[\![e_1 \mathtt{+} e_2]\!] & = & \mathcal{E}[\![e_1]\!] + \mathcal{E}[\![e_2]\!] \\ \mathcal{E}[\![e_1 \mathtt{*} e_2]\!] & = & \mathcal{E}[\![e_1]\!] \cdot \mathcal{E}[\![e_2]\!] \end{array}$$

where $\mathcal{C}$, the interpretation function for constants, is left unspecified.

Assuming that $\mathcal{C}$ maps the decimal representation of a number to its value, the semantics of an expression is its usual value.

$$\begin{array}{lcl} \mathcal{E}[\![(\mathsf{2}+\mathsf{3})*(\mathsf{5}+\mathsf{2})]\!] & = & 35 \\ \mathcal{E}[\![\mathsf{0}]\!] & = & 0 \end{array}$$

Exercise: Define the interpretation function for constants in binary representation based on the following abstract syntax (ε stands for the empty string):

$$c \quad ::= \quad \varepsilon \mid \mathtt{0}c \mid \mathtt{1}c$$

Exercise: Define an abstract syntax for binary fractions and give their interpretation function.

1.2.2 Example: Small-Step Operational Semantics

A small-step semantics is often specified via a term rewriting of abstract syntax trees. A term rewriting system is defined through a set of rules that denote the replacement of a subterm by another term.

In the present case, the following rewriting rules capture the behavior of `+` and `*`.

$$\begin{array}{lcl} \mathtt{a}_1 \mathtt{+} \mathtt{a}_2 & \longrightarrow & \mathtt{a}_3 \\ \mathtt{b}_1 \mathtt{*} \mathtt{b}_2 & \longrightarrow & \mathtt{b}_3 \end{array}$$

if $\mathcal{C}[\![\mathtt{a}_1]\!] + \mathcal{C}[\![\mathtt{a}_2]\!] = \mathcal{C}[\![\mathtt{a}_3]\!]$ and $\mathcal{C}[\![\mathtt{b}_1]\!] \cdot \mathcal{C}[\![\mathtt{b}_2]\!] = \mathcal{C}[\![\mathtt{b}_3]\!]$. Application of a rewrite rule to a term consists in finding a subterm that looks like the left side of the rule and replacing it with the right side.

These two (families of) rules would already suffice to define a *reduction semantics.* Such a semantics allows the application of a rule whereever it is possible. It is inherently non-deterministic.

$$\begin{array}{ll} & (2+3)*(5+2) \\ \longrightarrow & (2+3)*7 \\ \longrightarrow & 5*7 \\ \longrightarrow & 35 \end{array}$$

A *structural operational semantics* consists of reduction rules and a strategy of how to apply the rules. This strategy is usually deterministic, but non-deterministic systems are used for defining concurrent language features.

The above reduction sequence corresponds to a right-to-left evaluation strategy. In practice, left-to-right evaluation is more common.

$$\begin{array}{ll} & (2+3)*(5+2) \\ \longrightarrow & 5*(5+2) \\ \longrightarrow & 5*7 \\ \longrightarrow & 35 \end{array}$$

How does such a semantics now when it is finished? A reduction sequence is finished when the final term is a member of a predefined set of terms, the set of *values*. In the example, the set of values is the subset of expressions specified by

$$v \quad ::= \quad \mathtt{a}$$

Exercise: Add division `/` to the abstract syntax. Define the rule family for division and discuss what happens with terms like `5+(3/0)`.

1.2.3 Example: Big-Step Operational Semantics

A big-step operational semantics relates an expression directly to its value. Expressions and values are as before (but this is coincidental):

$$\begin{array}{lll} e & ::= & e{+}e \mid e{*}e \mid \mathtt{a} \\ v & ::= & \mathtt{a} \end{array}$$

The relation $\hookrightarrow \subseteq \mathsf{Exp} \times \mathbf{Z}$ is defined through a deduction system by induction on the abstract syntax of the expression. The evaluation of constants just relates a constant to its meaning.

$$\mathtt{a} \hookrightarrow \mathcal{C}[\![\mathtt{a}]\!]$$

For addition as well as for multiplication, first the two subexpressions need to be evaluated. Then, the operation is performed and the resulting value returned.

$$\frac{e_1 \hookrightarrow v_1 \qquad e_2 \hookrightarrow v_2}{e_1{+}e_2 \hookrightarrow v_1 + v_2} \qquad \frac{e_1 \hookrightarrow v_1 \qquad e_2 \hookrightarrow v_2}{e_1{*}e_2 \hookrightarrow v_1 \cdot v_2}$$

To compute the value of an expression, the deduction rules are pasted together to a proof tree with the expression at the root.

$$\frac{\dfrac{2 \hookrightarrow 2 \quad 3 \hookrightarrow 3}{2+3 \hookrightarrow 5} \qquad \dfrac{5 \hookrightarrow 5 \quad 2 \hookrightarrow 2}{5+2 \hookrightarrow 7}}{(2+3)*(5+2) \hookrightarrow 35}$$

2 Terms, Substitutions, and Identities

Terms play a central role in the theory of programming languages. They serve to define the abstract syntax, they define computation states in operational semantics, and they serve as domain of computation in logic programming.

This section is derived from Chapter 3 of the book *Term Rewriting and All That* by Franz Baader and Tobias Nipkow, Cambridge University Press, 1998. Universal algebra investigates terms and interpretations of terms in more detail.

Definition 1 A *signature* Σ is a set of operation symbols with a function $a : \Sigma \to \mathbf{N}$ indicating the *arity* of each symbol. Write $\Sigma^{(n)}$ for $\{f \in \Sigma \mid a(f) = n\}$. Call the elements of $\Sigma^{(0)}$ *constant symbols.*

Example: $\Sigma_{expr} = \{\mathtt{a}^{(0)}, +^{(2)}, *^{(2)}\}$ with superscripts indicating the arity.

In many cases, we want to distinguish between different sorts (types) of arguments and results, drawn from a set $\mathcal{S}$. In this case, the arity generalizes from a natural number to a pair (w, s) where $w \in \mathcal{S}^*$ and $s \in \mathcal{S}$. Such a signature is called a *many-sorted signature* or *heterogeneous signature*, whereas the above definition is *one-sorted* or *homogeneous.* Since all results and definitions generalize easily to the many-sorted case, we'll stick with the one-sorted signatures to avoid clutter.

Definition 2 Let Σ be a signature and X be a set of *variables* with $\Sigma \cap X = \emptyset$. The set $T(\Sigma, X)$ of Σ-*Terms over* X is inductively defined by

- $X \subseteq T(\Sigma, X)$
- for each $n \in \mathbf{N}$, if $t_1, \dots, t_n \in T(\Sigma, X)$ and $f \in \Sigma^{(n)}$, then $f(t_1, \dots, t_n) \in T(\Sigma, X)$.

Examples: Using Σ_{expr} from above and $X = \{x, y, z\}$ the following are terms:

$$\begin{array}{l} x \\ \mathtt{a}() \\ +(\mathtt{a}, y) \\ *(+(z, \mathtt{a}), y) \end{array}$$

For convenience, write $\mathtt{a}$ instead of $\mathtt{a}()$ and use infix notation if that's unambiguous, *e.g.*, $(\mathtt{a}+y)$.

Terms may be considered as ordered trees, which gives a nice graphical rendering.

The tree intuition naturally gives rise to the concept of a position in a term.

Definition 3 Let Σ be a signature, X be a set of variables with $\Sigma \cap X = \emptyset$, and $s, t \in T(\Sigma, X)$.

1. The set of *positions* of a term is defined by a function $Pos(\cdot) : T(\Sigma, X) \to \mathbf{N}^*$

- for all $x \in X$,
  $Pos(x) = \{\varepsilon\}$
- for all $f(t_1, \dots, t_n) \in T(\Sigma, X)$,
  $Pos(f(t_1, \dots, t_n)) = \{\varepsilon\} \cup \bigcup_{i=1}^{n} i.Pos(t_i)$

Position ε is the *root position* and the symbol at that that position is the *root symbol.*

2. The *size* of a term is its number of positions: $|s| = |Pos(s)|$.

3. For $p \in Pos(s)$, the *subterm of s at position p*, $s|_p$, is defined by

$$\begin{array}{lcl} s|_\varepsilon & = & s \\ f(t_1, \dots, t_n)|_{i.p} & = & t_i|_p \end{array}$$

4. For $p \in Pos(s)$, $s[t]_p$ denotes the term obtained from s by replacing the subterm at position p by t.

$$\begin{array}{lcl} s[t]_\varepsilon & = & t \\ f(t_1, \dots, t_n)[t]_{i.p} & = & f(t_1, \dots, t_i[t]_p, \dots, t_n) \end{array}$$

5. The set of *variables occurring in s*, $Var(s)$ is defined by

$$\begin{array}{lcl} Var(x) & = & \{x\} \\ Var(f(t_1, \dots, t_n)) & = & Var(t_1) \cup \dots \cup Var(t_n) \end{array}$$

Example (continued):

t	$Pos(t)$	$Var(t)$	$\lvert t\rvert$
x	$\{\varepsilon\}$	$\{x\}$	1
$\mathtt{a}()$	$\{\varepsilon\}$	$\emptyset$	1
$+(\mathtt{a}, y)$	$\{\varepsilon, 1, 2\}$	$\{y\}$	3
$*(+(z, \mathtt{a}), y)$	$\{\varepsilon, 1, 1.1, 1.2, 2\}$	$\{y, z\}$	5

A term $t \in T(\Sigma, x)$ is a *ground term* if $Var(t) = \emptyset$.

Exercise: A term may also be considered as a function from a set of positions ($\subseteq \mathbf{N}^*$) to symbols (variables or operation symbols). Rewrite the above definitions from that point of view.

Exercise: Lemma 3.1.4 in *Term Rewriting* contains four simplification rules about subterms and subterm replacement. Prove them by induction.

Definition 4 Let Σ be a signature and X be a countably infinite set of variables with $\Sigma \cap X = \emptyset$. A $T(\Sigma, X)$*-substitution* is a function $\sigma : X \to T(\Sigma, X)$ such that the set

$$Dom(\sigma) = \{x \in X \mid \sigma(x) \neq x\}$$

(the *domain of σ*) is finite.

The *range of* σ is the set of terms $Ran(\sigma) = \{\sigma(x) \mid x \in Dom(\sigma)\}$ and the *variable range of* σ is $VRan(\sigma) = \bigcup_{t \in Ran(\sigma)} Var(t)$.

Any $T(\Sigma, X)$-substitution σ can be extended to a function $\hat{\sigma} : T(\Sigma, X) \to T(\Sigma, X)$ by

$$\begin{array}{lcl} \hat{\sigma}(x) & = & \sigma(x) \\ \hat{\sigma}(f(t_1, \dots, t_n)) & = & f(\widehat{sigma}(t_1), \dots, \hat{\sigma}(t_n)) \end{array}$$

Notation:

- if $Dom(\sigma) = \{x_1, \dots, x_n\}$, then write $\sigma = \{x_1 \mapsto \sigma(x_1), \dots x_n \mapsto \sigma(x_n)\}$.
- the distinction between σ and $\hat{\sigma}$ is dropped most of the time.

Example: Let $t = *(+(z, \mathsf{a}), y)$ and $\sigma = \{y \mapsto \mathsf{a}, z \mapsto +(z, \mathsf{a})\}$.

Definition 5 Let σ, τ be $T(\Sigma, X)$-substitutions. Their *composition* $\sigma\tau$ is defined by $\sigma\tau(x) = \hat{\sigma}(\tau(x))$.

Exercise: Show that composition is associative and that $\widehat{\sigma\tau} = \hat{\sigma}\hat{\tau}$.

Definition 6 Let Σ be a signature and X be a countably infinite set of variables with $\Sigma \cap X = \emptyset$.

A Σ-*identity* is a pair $(s, t) \in T(\Sigma, X) \times T(\Sigma, X)$, written as $s \approx t$.

s is the *left-hand side* and t is the *right-hand side* of $s \approx t$.

Example: $+(+(x, y), z) \approx +(x, +(y, z))$ may be interpreted as saying that $\mathsf{+}$ associative.

Definition 7 Let $\mathcal{E}$ be a set of Σ-identities. The *reduction relation* $\to_{\mathcal{E}} \subseteq T(\Sigma, X) \times T(\Sigma, X)$ is defined by

$s \to_{\mathcal{E}} t$ iff

$$(\exists (l, r) \in \mathcal{E})\ (\exists p \in Pos(s))\ (\exists \sigma)\ s|_p = \sigma(l) \land t = s[\sigma(r)]_p$$

Definition 8 Let $\mathcal{E}$ be a set of Σ-identities and $\to_{\mathcal{E}}$ its associated reduction relation.

1. The *reflexive transitive closure* $\overset{*}{\to}_{\mathcal{E}}$ of $\to_{\mathcal{E}}$ is defined via a set of inference rules

$$\frac{}{s \overset{*}{\to}_{\mathcal{E}} s} \qquad \frac{s \to_{\mathcal{E}} s'}{s \overset{*}{\to}_{\mathcal{E}} s'} \qquad \frac{s \overset{*}{\to}_{\mathcal{E}} s' \quad s' \overset{*}{\to}_{\mathcal{E}} s''}{s \overset{*}{\to}_{\mathcal{E}} s''}$$

2. The *reflexive transitive symmetric closure* $\overset{*}{\leftrightarrow}_{\mathcal{E}}$ of $\to_{\mathcal{E}}$ is defined by

$$\frac{}{s \overset{*}{\to}_{\mathcal{E}} s} \qquad \frac{s \to_{\mathcal{E}} s'}{s \overset{*}{\to}_{\mathcal{E}} s'} \qquad \frac{s' \to_{\mathcal{E}} s}{s \overset{*}{\to}_{\mathcal{E}} s'} \qquad \frac{s \overset{*}{\to}_{\mathcal{E}} s' \quad s' \overset{*}{\to}_{\mathcal{E}} s''}{s \overset{*}{\to}_{\mathcal{E}} s''}$$

A set of rules such as above forms a *deduction system* and the following definition makes precise how such a system works.

Definition 9 A *deduction system* consists of a signature Γ for forming *judgements* and a set of *inference rules*.

A *judgement* is a term $\mathcal{J} \in T(\Gamma, V)$ where V is a countably infinite set of variables disjoint from Γ.

An *inference rule* is a pair $(J_1 \dots J_n, J_0)$ of a sequence of judgements $J_1 \dots J_n$ and a judgement J_0, written as

$$\frac{J_1 \dots J_n}{J_0}$$

A *proof tree* validating judgement J is defined inductively by:

- Let $\dfrac{J_1 \dots J_n}{J_0}$ be an inference rule and σ a substitution so that $J = \sigma(J_0)$. Let $\mathcal{J}_1, \dots, \mathcal{J}_n$ be proof trees validating $\sigma(J_1), \dots, \sigma(J_n)$, respectively. Then $\mathcal{J} = \dfrac{\mathcal{J}_1 \ \dots \ \mathcal{J}_n}{J}$ is a *proof tree validating judgement J*.

Example: given the identity $+(+(x,y),z) \approx +(x,+(y,z))$ for associativity of **+** establish that

$$+(+(+(x,y),z),w) \overset{*}{\leftrightarrow}_{\mathcal{E}} +(x,(+(y,+(z,w))))$$

$$\dfrac{\dfrac{+(+(+(x,y),z),w) \to_{\mathcal{E}} +(+(x,y),+(z,w))}{+(+(+(x,y),z),w) \overset{*}{\leftrightarrow}_{\mathcal{E}} +(+(x,y),+(z,w))} \qquad \dfrac{+(+(x,y),+(z,w)) \to_{\mathcal{E}} +(x,(+(y,+(z,w))))}{+(+(x,y),+(z,w)) \overset{*}{\leftrightarrow}_{\mathcal{E}} +(x,(+(y,+(z,w))))}}{+(+(+(x,y),z),w) \overset{*}{\leftrightarrow}_{\mathcal{E}} +(x,(+(y,+(z,w))))}$$

Example: big-step evaluation in previous section.

3 Algebras

The central idea of denotational semantics is to map an abstract syntax tree to a value in a mathematical structure that *denotes* the meaning of the tree. In addition, this mapping must be *compositional*, that is, the meaning of a phrase is a function of the meanings of its subphrases.

If we consider abstract synax trees as terms, then the above statement means that denotational semantics requires a mechanism to define a mapping from a set of terms to a set of meanings (the *denotations* of terms). Furthermore, compositionality means that this mapping must be defined by the interpretations of the term constructors (the operation and constant symbols). This matches exactly the concept of an *algebra over a signature*. (Once again, the presentation sticks to the one-sorted case; the transliteration to the many-sorted case is straightforward but tedious).

This section is derived from Chapter 3 of the book *Term Rewriting and All That* by Franz Baader and Tobias Nipkow, Cambridge University Press, 1998.

Definition 10 Let Σ be a signature. A Σ-*algebra* $\mathcal{A} = (A, \alpha)$ consists of

- a *carrier set* A and
- a mapping α so that, for all $n \in \mathbf{N}$, $f \in \Sigma^{(n)}$, $\alpha(f) : A^n \to A$.

Sometimes the mapping α is left implicit by writing $f^{\mathcal{A}}$ instead of $\alpha(f)$.

Example: The "expected" algebra for $\Sigma_{expr} = \{\mathtt{a}^{(0)}, +^{(2)}, *^{(2)}\}$ is $\mathcal{Z} = (A, \alpha)$ with $A = \mathbf{Z}$ and α defined by

$$\begin{array}{lcl} \alpha(\mathtt{a}) & = & \mathcal{C}[\![\mathtt{a}]\!] \\ \alpha(+)(v_1, v_2) & = & v_1 + v_2 \\ \alpha(*)(v_1, v_2) & = & v_1 \cdot v_2 \end{array}$$

Definition 11 Let Σ be a signature and $\mathcal{A}$, $\mathcal{B}$ be Σ-algebras.

1. $\mathcal{B}$ is a *subalgebra* of $\mathcal{A}$ if $B \subseteq A$ and, for all $n \in \mathbf{N}$, $f \in \Sigma^{(n)}$, $b_1, \dots, b_n \in B$, it holds that $f^{\mathcal{A}}(b_1, \dots, b_n) = f^{\mathcal{B}}(b_1, \dots, b_n) \in B$.
2. The subalgebra $\mathcal{B}$ is *generated by a set* X if $\mathcal{B}$ is the smallest algebra such that $X \subseteq B$.

Example: Consider again Σ_{expr}. The algebra (B, α) with $B = \{2x \mid x \in \mathbf{Z}\}$ is a subalgebra of $\mathcal{Z}$ above. It is generated by $\{2\}$ or $\{-2\}$.

Exercise: Let $\mathcal{A}$ be a Σ-algebra.

1. Show that the intersection of two subalgebras of $\mathcal{A}$ is also a subalgebra of $\mathcal{A}$.
2. Show that the subalgebra generated by X is the intersection of all subalgebras whose carrier contains X.

Definition 12 Let $\mathcal{A}$ and $\mathcal{B}$ be Σ-algebras with carrier sets A and B. A Σ-*homomorphism* $h : \mathcal{A} \to \mathcal{B}$ is a mapping from A to B such that for all $n \in \mathbf{N}$, $f \in \Sigma^{(n)}$, and $a_1, \dots, a_n \in A$, it holds that

$$h(f^{\mathcal{A}}(a_1, \dots, a_n)) = f^{\mathcal{B}}(h(a_1), \dots, h(a_n))$$

The usual notions of epimorphism, monomorphism, isomorphism carry over from the underlying mapping to homomorphisms.

Example:

1. Let $m \in \mathbf{N}$, $m > 0$. Then $\mathcal{Z}_m$ is a Σ_{expr}-algebra with carrier $\{0, 1, \dots, m-1\}$ and $f^{\mathcal{Z}_m}(\overline{z}) = f^{\mathcal{Z}}(\overline{z}) \pmod{m}$. The mapping $h(x) = x \pmod{m}$ is a Σ_{expr}-homomorphism from $\mathcal{Z}$ to $\mathcal{Z}_m$.
2. Is there a non-trivial Σ_{expr}-homomorphism from $\mathcal{Z}$ to itself?

Lemma 1 *Let $\mathcal{A}$, $\mathcal{B}$ be Σ-algebras with $\mathcal{A}$ generated by X. If $h, g : \mathcal{A} \to \mathcal{B}$ are homomorphisms and, for all $x \in X$, $h(x) = g(x)$, then $h = g$.*

Proof: Since $\mathcal{A}$ is generated by X, its carrier $A = \bigcup_{n \in \mathbf{N}} A_n$ where

$$\begin{array}{rcl} A_0 & = & X \\ A_{n+1} & = & A_n \cup \{f^{\mathcal{A}}(a_1, \dots, a_n) \mid n \geq n, a_1, \dots, a_n \in A_n, f \in \Sigma^{(n)}\} \end{array}$$

By definition, $h = g$ iff, for all $a \in A$, $h(a) = g(a)$. Since $A = \bigcup_{n \in \mathbf{N}} A_n$ it is sufficient to show that, for all $n \in \mathbf{N}$, $a \in A_n$, $h(a) = g(a)$. The proof is by induction on n:

- Case 0. For $A_0 = X$, this holds by the assumption.
- Case $n + 1$. Let $a \in A_{n+1}$. By definition of A_{n+1}, if $a \in A_n$, then the induction hypothesis yields $h(a) = g(a)$. Otherwise, $a \in \{f^{\mathcal{A}}(a_1, \dots, a_n) \mid n \geq n, a_1, \dots, a_n \in A_n, f \in \Sigma^{(n)}\}$, that is, there exists $n \in \mathbf{N}$, $f \in \Sigma^{(n)}$, $a_1, \dots, a_n \in A_n$, such that $a = f^{\mathcal{A}}(a_1, \dots, a_n)$. Since h and g are both homomorphisms, it holds that

$$\begin{array}{rcl} h(a) & = & h(f^{\mathcal{A}}(a_1, \dots, a_n)) \\ & = & f^{\mathcal{A}}(h(a_1), \dots, h(a_n)) \\ & = & f^{\mathcal{A}}(g(a_1), \dots, g(a_n)) \\ & = & g(f^{\mathcal{A}}(a_1, \dots, a_n)) \\ & = & g(a) \end{array}$$

Definition 13 Let Σ be a signature and X be a set of variables with $X \cap \Sigma = \emptyset$. The Σ-*term algebra* $\mathcal{T}(\Sigma, X)$ has carrier $T(\Sigma, X)$ and operations (with $f \in \Sigma^{(n)}$)

$$f^{\mathcal{T}(\Sigma,X)}(t_1, \dots, t_n) = f(t_1, \dots, t_n)$$

Theorem 1 *If $\mathcal{B}$ is a Σ-algebra and $\varphi : X \to B$ is variable assignment, then there is a unique homomorphism $\hat{\varphi} : \mathcal{T}(\Sigma, X) \to \mathcal{B}$.*

Proof: The crucial steps are

- $\mathcal{T}(\Sigma, X)$ is generated by X
- $\hat{\varphi}$ is a homomorphism defined by
  - $\hat{\varphi}(x) = \varphi(x)$, for all $x \in X$,
  - $\hat{\varphi}(f(t_1, \dots, t_n)) = f^{\mathcal{B}}(\hat{\varphi}(t_1), \dots, \hat{\varphi}(t_n))$, for all $n \in \mathbf{N}$, $f \in \Sigma^{(n)}$, $t_1, \dots, t_n \in T(\Sigma, X)$.
- uniqueness follows with Lemma 1

For the case $X = \emptyset$ the above means that indeed every algebra (A, α) defines a unique function from $T(\Sigma, \emptyset) \to A$. Hence, an algebra can be used to uniquely define the denotational semantics of an abstract syntax. The *semantic equations* typically employed serve as a notational device for defining such an algebra:

$$[\![f(t_1, \dots, t_n)]\!] = f^{\mathcal{A}}([\![t_1]\!], \dots, [\![t_n]\!])$$

4 Variables and Binding

Variables are placeholders for values. In a programming language, they serve to avoid repeated computations, capture meaningful intermediate results, and play an important role in parameter passing. The first round will ignore that last role and concentrate on modeling variables in the three styles of semantics under consideration.

The language in this section is L_{ev}. Its abstract syntax extends the expression language form the introduction with a syntactic category of variables and two new expression constructs, the *let binding* and the *variable reference*.

$$\begin{array}{rrcl} & v & \in & \mathsf{Var} \\ \mathsf{Exp} & e & ::= & v \mid \mathsf{a} \mid e\texttt{+}e \mid \texttt{let}\ v\ \texttt{=}\ e\ \texttt{in}\ e \end{array}$$

Var is a countably infinite set of variable names disjoint from expression constructors. The occurrence of v on the right-hand side is a shorthand for a term constructor that maps a variable name to an expression. Similarly, `let` is a term constructor that maps a variable name and two expressions to an expression.

Important: Do not confuse the elements $v \in \mathit{Var}$ with variables as they are used in terms. From the term perspective, the elements of *Var* are constant symbols. Hence, $\mathit{Var}(\texttt{let x=1 in x}) = \emptyset$!

4.1 Variables, Denotationally

Background reading: Peter D. Mosses, *Denotational Semantics*, Handbook of Theoretical Computer Science, Volume B.

The definition of a denotational semantics start with the definition of the *semantic domains*. At present, the semantic domains are sets (but this will be refined later).

The presence of variables requires a new distinction among the semantic domains.

- The values that can be bound to variables are *denoted values*.
- The values that arise as interpretations of expressions are *expressed values*.

A typical relation between the two kinds of values is that expressions denote computations that yield denoted values. However, other constellations are possible.

The domain Env of *environments* captures the binding of variable names to denoted values. An environment is a finite (hence partial) function from variables to (denoted) values. The set Val specifies the denoted values.

$$\begin{array}{rcl} \mathsf{Env} & = & \mathsf{Var} \hookrightarrow \mathsf{Val} \\ \mathsf{Val} & = & \mathbf{Z} \end{array}$$

The denotation of an expression is a function that maps some environments to a value:

$$\mathsf{Comp} = \mathsf{Env} \hookrightarrow \mathsf{Val}$$

The semantic equations specify a function that maps an expression to an expressed value (a computation, in this case). In the equations, $\rho \in \mathsf{Env}$.

$$\begin{array}{lcl}
\mathcal{E} : \mathsf{Exp} \to \mathsf{Comp} & & \\
\mathcal{E}[\![v]\!]\rho & = & \rho(v) \\
\mathcal{E}[\![\mathsf{a}]\!]\rho & = & \mathcal{C}[\![a]\!] \\
\mathcal{E}[\![e_1\texttt{+}e_2]\!]\rho & = & \mathcal{E}[\![e_1]\!]\rho + \mathcal{E}[\![e_2]\!]\rho \\
\mathcal{E}[\![\texttt{let } v \texttt{ = } e_1 \texttt{ in } e_2]\!]\rho & = & \mathcal{E}[\![e_2]\!]\rho[v \mapsto \mathcal{E}[\![e_1]\!]\rho]
\end{array}$$

where the environment update $\rho[v \mapsto z]$ is defined by

$$\rho[v \mapsto z](y) = \begin{cases} z & v = y \\ \rho(v) & v \neq y \end{cases}$$

Note:

1. Variable names are interpreted by the identity function (as themselves).
2. Constant expressions ignore the environment.
3. Operators pass the environment to the subterms.
4. The propagation of the environment and its overwriting in the `let` clause defines the *scope* or *static extent* of a variable binding. The scope of a variable v starts with the body e_2 of `let` v=e_1 `in` e_2 and extends up to the header expression e_1' of a subterm `let` v=e_1' `in` e_2' occuring in e_2 such that no intervening subterm of this form exists. If such a new binding `let` v=e_1' `in` e_2' exists, it *shadows* the previous binding of the variable v in its body e_2'.

Example: The semantics of `let x=17+4 in x+x`

$$\begin{array}{cl}
 & \mathcal{E}[\![\texttt{let x=17+4 in x+x}]\!]\{\ \} \\
= & \mathcal{E}[\![\texttt{x+x}]\!]\{\texttt{x} \mapsto \mathcal{E}[\![\texttt{17+4}]\!]\{\ \}\} \\
= & \mathcal{E}[\![\texttt{x+x}]\!]\{\texttt{x} \mapsto \mathcal{E}[\![\texttt{17}]\!]\{\ \} + \mathcal{E}[\![\texttt{4}]\!]\{\ \}\} \\
= & \mathcal{E}[\![\texttt{x+x}]\!]\{\texttt{x} \mapsto 17 + 4\} \\
= & \mathcal{E}[\![\texttt{x}]\!]\{\texttt{x} \mapsto 21\} + \mathcal{E}[\![\texttt{x}]\!]\{\texttt{x} \mapsto 21\} \\
= & 21 + 21 \\
= & 42
\end{array}$$

Exercise: What is the semantics of

1. `let x=1 in let x=x+x in let x=x+x in x`
2. `let x=42 in x+y`

Exercise: The scope of a let binding is a set of positions in an Exp term e. Define a function that, given a position in the abstract syntax tree e and a variable, determines if that variable is in scope and if so, the position of its binding let expression. A suggestion for the type of the function is

$$S : (e : \mathsf{Exp}) \to \mathit{Pos}(e) \to \mathsf{Var} \to \mathcal{P}(\mathit{Pos}(e))$$

A number of properties are provable from such a semantics. For example, if each variable occurrence in an expression has a corresponding binding position, then the denotational semantics of the expression is defined. Or: the renaming a bound variable does not change the denotation of an expression. Or: the correctness of substitution.

Stating these properties requires a few definitions about variables in expressions.

Definition 14 The set, $FV(e)$, of *free variables* of an expression is defined by

$$\begin{array}{lcl} FV(v) & = & \{v\} \\ FV(\mathtt{a}) & = & \emptyset \\ FV(e_1\mathtt{+}e_2) & = & FV(e_1) \cup FV(e_2) \\ FV(\mathtt{let}\ v\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2) & = & FV(e_1) \cup (FV(e_2) \setminus \{v\}) \end{array}$$

An expression e is *closed* if $FV(e) = \emptyset$; otherwise it is *open.*

Lemma 2 *If $FV(e) \subseteq dom(\rho)$, then there exists y such that $\mathcal{E}[\![e]\!]\rho = y$.*

Proof: By induction on e.

Definition 15 *Syntactic substitution* of a variable v' by expression e' in expression e, written $e[v' \mapsto e']$

$$\begin{array}{lcl} v[v' \mapsto e'] & = & \left\{ \begin{array}{ll} e' & v = v' \\ v & v \neq v' \end{array} \right. \\ \mathtt{a}[v' \mapsto e'] & = & \mathtt{a} \\ (e_1\mathtt{+}e_2)[v' \mapsto e'] & = & e_1[v' \mapsto e']\mathtt{+}e_2[v' \mapsto e'] \\ (\mathtt{let}\ v\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2)[v' \mapsto e'] & = & \mathtt{let}\ v''\ \mathtt{=}\ e_1[v' \mapsto e']\ \mathtt{in}\ e_2'[v \mapsto v''][v' \mapsto e'] \\ & & \text{where } v'' \neq v', v'' \notin FV(e') \cup FV(e_2) \end{array}$$

Syntactic substitution and term substitution are held apart by notation, the former is written $e[v \mapsto e']$ while the latter is $\{x \mapsto t'\}(t)$.

The side conditions on syntactic substitution are required to avoid *capturing of variables*, hence it is often called *capture avoiding substitution.* For example, in the expression

$$(\mathtt{let\ x=2\ in\ y})[\mathtt{y} \mapsto \mathtt{x}]$$

the `y` in the in the body of the let refers to a binding of `y` somewhere in the context of the expression. Simply replacing `y` by `x` would change the binding structure of the expression (by capturing `x`) and hence the expression's meaning. The correct result of the above substitution is

$$(\mathtt{let\ x=2\ in\ y})[\mathtt{y} \mapsto \mathtt{x}] = \mathtt{let\ x'=2\ in\ x}$$

Similarly, substitution must respect shadowing. As the example shows, a substitution for the binding variable proceeds in the header of the let but leaves the let's body alone:

$$(\mathtt{let\ x=x+x\ in\ x})[\mathtt{x} \mapsto \mathtt{4}] = \mathtt{let\ x'=4+4\ in\ x'}$$

Again, substitution must not destroy the binding structure of the expression.

Lemma 3 (Renaming) *Suppose that $v' \notin FV(e_2)$. Then, for all $\rho \in \mathsf{Env}$, $\mathcal{E}[\![\texttt{let}\ v\ \texttt{=}\ e_1\ \texttt{in}\ e_2]\!]\rho = \mathcal{E}[\![\texttt{let}\ v'\ \texttt{=}\ e_1\ \texttt{in}\ e_2[v \mapsto v']]\!]\rho$.*

Proof: Induction on e_2.

Renaming of bound variables is often called *α-renaming* or *α-conversion.*

Lemma 4 (Substitutivity) *Suppose that $\mathcal{E}[\![e]\!]\rho[v \mapsto z] = y$ and $\mathcal{E}[\![e']\!]\rho = z$. Then $\mathcal{E}[\![e[v \mapsto e']]\!]\rho = y$.*

Proof: Induction on e.

4.2 Variables, Big-Step Style

To accommodate variables in a big-step operational semantics, environments are used as in a denotational semantics. The evaluation relation receives such an environment as an additional component:

$$\rho \vdash e \hookrightarrow y$$

where $\rho \in \mathsf{Env}$, $e \in \mathsf{Exp}$, and $y \in \mathbf{Z}$.

As before, the relation is specified by decomposing the expression.

$$\textit{(const)}\ \frac{}{\rho \vdash \mathsf{a} \hookrightarrow \mathcal{C}[\![\mathsf{a}]\!]}$$

$$\textit{(add)}\ \frac{\rho \vdash e_1 \hookrightarrow y_1 \qquad \rho \vdash e_2 \hookrightarrow y_2}{\rho \vdash e_1\texttt{+}e_2 \hookrightarrow y_1 + y_2}$$

$$\textit{(var)}\ \frac{\rho(v) = y}{\rho \vdash v \hookrightarrow y}$$

$$\textit{(let)}\ \frac{\rho \vdash e_1 \hookrightarrow y_1 \qquad \rho[v \mapsto y_1] \vdash e_2 \hookrightarrow y_2}{\rho \vdash \texttt{let}\ v\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \hookrightarrow y_2}$$

In this particular case, properties of evaluation can be proved analogously to the denotational semantics.

Exercise: State and prove the properties Renaming and Substitutivity in the big-step semantics framework.

Exercise: Model the environment ρ as a list of pairs, *i.e.*, as terms generated by the signature

$$\begin{array}{lcl} \texttt{empty-env} & : & \mathsf{Env} \\ \texttt{extend-env} & : & \mathsf{Var} \times \mathbf{Z} \times \mathsf{Env} \to \mathsf{Env} \end{array}$$

and redefine the big-step operational semantics using this representation of the environment. Hint: only rules *(var)* and *(let)* change; introduce an auxiliary judgement for $\rho(v) = y$.

4.3 Variables, Small-Step Style

The small-step semantics does not require environments. Instead, it computes the value of the header of a let expression and then substitutes it into the body. This works because the set of values is a subset of the set of expressions, in our case the set of constants:

$$\mathsf{Val} \ni y \quad ::= \quad \mathtt{a}$$

The small-step reduction rules are thus

$$\begin{gathered}\mathtt{a}_1\mathtt{+}\mathtt{a}_2 \longrightarrow \mathtt{a}_3 \quad \textit{where} \quad \mathcal{C}[\![\mathtt{a}_1]\!] + \mathcal{C}[\![\mathtt{a}_2]\!] = \mathcal{C}[\![\mathtt{a}_3]\!] \\ \mathtt{let}\ v\ \mathtt{=}\ y\ \mathtt{in}\ e \longrightarrow e[v \mapsto y]\end{gathered} \tag{1}$$

In particular, the let rule is only applicable is the header is already a value.

Terminology: An expression that matches the left-hand side of a rule (where the rule is applicable at position ε) is a *redex*. The result of applying the rule to the redex is the *reductum*.

In a *reduction semantics*, these rules could be applied anywhere in an expression. A more accurate reflection of what happens during evaluation requires a strategy that specifies which rule to apply at which position in the next step. One way of defining that strategy is the following:

$$\begin{gathered}\frac{e_1 \longrightarrow e_1'}{e_1\mathtt{+}e_2 \longrightarrow e_1'\mathtt{+}e_2} \\[2ex] \frac{e_2 \longrightarrow e_2'}{y_1\mathtt{+}e_2 \longrightarrow y_1\mathtt{+}e_2'} \\[2ex] \frac{e_1 \longrightarrow e_1'}{\mathtt{let}\ v\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 \longrightarrow \mathtt{let}\ v\ \mathtt{=}\ e_1'\ \mathtt{in}\ e_2}\end{gathered} \tag{2}$$

These three rules define the strategy of searching for redexes:

1. In an expression of the form e_1+e_2, look for redexes in the left subexpression, e_1, first.
2. If an expression has the form y_1+e_2 (that is, its left subexpression is already reduced to a value), then look for redexes in the right subexpression, e_2.
3. If an expression has the form `let` v `=` e_1 `in` e_2, then look for redexes in the header, e_1, first.

Taken together, the rules specify a *leftmost evaluation strategy*.

Example: Reduce `let x=17 in let y=x+4 in y+y` to a value.

For a big-step semantics description it is fairly easy to see that its evaluation relation is deterministic. For a small-step semantics, this can be more involved and requires a proof.

Lemma 5 *1. For each expression e there is at most one e' such that $e \longrightarrow e'$.*

2. *If e is closed then either e is a value or there is exactly one e' such that $e \longrightarrow e'$.*

3. *If e is closed, then there is exactly one* $\mathtt{a}$ *such that $e \xrightarrow{*} \mathtt{a}$.*

Proof: Items 1 and 2 are proven by induction on e.
Item 3 is by induction on the proof tree of $e \xrightarrow{*} \mathtt{a}$.

Hence it makes sense to define an evaluation function in terms of small-step reduction.

Definition 16 The *evaluation function eval* maps any closed expression to a value. It is defined by

$$\mathit{eval}\,(e) = \mathtt{a} \qquad \text{iff} \qquad e \xrightarrow{*} \mathtt{a}$$

An alternative way of specifying the evaluation strategy employs contexts.

Definition 17 Let Σ be a signature and X a set of variables with $x \in X$, $\Sigma \cap X = \emptyset$.
A Σ-*context* C is a pair of a term $t_C \in T(\Sigma, X)$ and a position $p_C \in \mathit{Pos}(t_C)$.
The position p is called the *hole of* C.
The operation *hole filling* that sticks a term t in a context C is defined by $C[t] = t_C[t]_{p_C}$.
The *empty context* (x, ε) is written $[\,]$.
If $n \in \mathbf{N}$, $n \geq 0$, $f \in \Sigma^{(n)}$, $t_1, \ldots, t_n \in T(\Sigma, X)$, $1 \leq i \leq n$, and $C = (t_C, p_C)$ a context then $f(t_1, \ldots, C', \ldots, t_n) := (f(t_1, \ldots, t_C, \ldots, t_n), i.p_C)$ is also a context.

Intuitively, a context is a term with a hole.
Example: The set of contexts for the abstract syntax of arithmetic expressions with let is generated by the following grammar.

$$C \quad ::= \quad [\,] \mid C\texttt{+}e \mid e\texttt{+}C \mid \texttt{let}\ v\ \texttt{=}\ C\ \texttt{in}\ e \mid \texttt{let}\ v\ \texttt{=}\ e\ \texttt{in}\ C$$

The grammar of the above example generates all possible contexts. That is, the relation $e \rightarrow e'$ defined by $e = C[r]$, $r \longrightarrow r'$ according to equation (1), and $e' = C[r']$ specifies the reduction semantics: a reduction rule may be applied anywhere in an expression.

Defining an evaluation strategy like the above leftmost evaluation strategy requires the set of contexts to be suitable restricted. This kind of context is an *evaluation context.*

$$E \quad ::= \quad [\,] \mid E\texttt{+}e \mid y\texttt{+}E \mid \texttt{let}\ v\ \texttt{=}\ E\ \texttt{in}\ e$$

Exercise: Show that the relation $e \rightarrow e'$ defined by $e = E[r]$, $r \longrightarrow r'$ according to ruleset (1), and $e' = E[r']$ specifies the leftmost reduction relation obtained by taking ruleset (1) and (2) together.

5 Functions, Conditionals, and Recursion

In this section, the language consists of expressions and recursive function definitions. Conditionals are added so that interesting recursive functions may be defined.

The abstract syntax has three categories, programs $p \in \mathsf{Prog}$, function definitions $d \in \mathsf{Def}$, and expressions $e \in \mathsf{Exp}$. For simplicity, functions have just one parameter.

$$\begin{array}{lcll} & v & \in & \mathsf{Var} \\ & f & \in & \mathsf{Fun} \\ \mathsf{Exp} & e & ::= & v \mid \mathtt{a} \mid e\texttt{+}e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid f\texttt{(}e\texttt{)} \\ \mathsf{Def} & d & ::= & f\texttt{(}v\texttt{)}\ \texttt{=}\ e \\ \mathsf{Prog} & p & ::= & d^*\ e \end{array}$$

The notation d^* stands for a list of ds. Var and Fun are disjoint, unspecified sets of variable and function symbols. Expressions are extended with a conditional and with a function call.

Both styles of operational semantics extend smoothly to this extended language. However, the denotational semantics requires some extra effort.

5.1 Functions and Conditionals, Big-Step Style

For modeling recursive functions, the big-step evaluation relation takes the list of function definitions as an extra component:

$$d^*, \rho \vdash e \hookrightarrow y$$

where $d^* \in \mathsf{Def}^*$, $\rho \in \mathsf{Env} = \mathsf{Var} \hookrightarrow \mathbf{Z}$, $e \in \mathsf{Exp}$, and $y \in \mathbf{Z}$.

As before, the evaluation relation is defined by decomposition of the expression. However, due to the presence of function calls, the evaluation trees (proof trees) no longer correspond directly to the structure of an expression.

$$\textit{(const)}\ \frac{}{d^*, \rho \vdash \mathtt{a} \hookrightarrow \mathcal{C}[\![\mathtt{a}]\!]} \qquad \textit{(var)}\ \frac{\rho(v) = y}{d^*, \rho \vdash v \hookrightarrow y}$$

$$\textit{(add)}\ \frac{d^*, \rho \vdash e_1 \hookrightarrow y_1 \qquad d^*, \rho \vdash e_2 \hookrightarrow y_2}{d^*, \rho \vdash e_1\texttt{+}e_2 \hookrightarrow y_1 + y_2}$$

$$\textit{(iftrue)}\ \frac{d^*, \rho \vdash e_1 \hookrightarrow y \qquad y \neq 0 \qquad d^*, \rho \vdash e_2 \hookrightarrow y_2}{d^*, \rho \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \hookrightarrow y_2}$$

$$\textit{(iffalse)}\ \frac{d^*, \rho \vdash e_1 \hookrightarrow 0 \qquad d^*, \rho \vdash e_3 \hookrightarrow y_3}{d^*, \rho \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \hookrightarrow y_3}$$

$$\textit{(funcall)}\ \frac{d^*, \rho \vdash e \hookrightarrow y \qquad f\texttt{(}v\texttt{)}\ \texttt{=}\ e' \in d^* \qquad d^*, [v \mapsto y] \vdash e' \hookrightarrow y'}{d^*, \rho \vdash f\texttt{(}e\texttt{)} \hookrightarrow y'}$$

1. There are two rules for conditionals, one for the case where the condition evaluates to a non-zero value and one for condition zero. Each rule selects the appropriate branch of the conditional and ignores the other.

2. The rule for a function call first evaluates the argument of the function. Next, it binds the variable to the argument value $[v \mapsto y]$, extracts the right-hand side of f's function definition (the body of f) from the list of function definitions, and starts evaluating that body. The value of the function call is the value of the function's body in the environment specified by the parameter's value.

Example: Build the evaluation tree for

```
fac (n) = if n=0 then 1 else n*fac (n-1)
fac (2)
```

Due to the presence of (recursive) function calls, the evaluation of an expression may not terminate. The big-step operational semantics only yields a result for terminating evaluations. Intuitively, a non-terminating evaluation builds an infinite evaluation tree that continues forever to develop new branches.

For this reason, conditionals require special treatment with the two evaluation rules above. In particular, a conditional *cannot* simply be considered as a ternary function because one branch may contain a non-terminating function call (consider evaluation of `fac (0)` which requires evaluation of `if n=0 then 1 else n*fac (n-1)` for $[\mathtt{n} \mapsto 0]$; calling `fac` with argument -1 does not terminate).

Exercise: Show that `fac (n)` terminates for $n \in \mathbf{N}$.

5.2 Functions and Conditionals, Small-Step Style

For the small-step semantics, new reduction rules need to be added for evaluating $p = d^*\ e$. There are two reduction rules for the conditional and one reduction rule for each function definition in the program.

$$\mathtt{if\ 0\ then}\ e_2\ \mathtt{else}\ e_3 \longrightarrow e_3$$

$$\mathtt{if\ a\ then}\ e_2\ \mathtt{else}\ e_3 \longrightarrow e_2 \qquad \mathtt{a} \neq \mathtt{0}$$

$$f\mathtt{(}y\mathtt{)} \longrightarrow e[v \mapsto y] \qquad f\mathtt{(}v\mathtt{)}\ \mathtt{=}\ e \in d^*$$

The conditional reductions require that the condition is already evaluated. The function call reductions require that the argument expression is evaluated before calling the function.

The remainder of the evaluation strategy is defined by evaluation contexts.

$$E \quad ::= \quad [\,] \mid E\mathtt{+}e \mid y\mathtt{+}E \mid \mathtt{if}\ E\ \mathtt{then}\ e\ \mathtt{else}\ e \mid f\mathtt{(}E\mathtt{)}$$

That is, beyond the leftmost evaluation of the addition, (only) the condition of a conditional is evaluated (to proceed further, one of the reduction rules for conditionals must be applied) and the argument of a function call must be evaluated before invoking the function.

The set of values does not change.

$$\mathsf{Val} \ni y \quad ::= \quad \mathsf{a}$$

As before, the small-step evaluation relation $\to$ is defined by $e \to e'$ if $e = E[r]$, $r \longrightarrow r'$ (r is a redex), and $e' = E[r']$. By similar reasoning as before, the following result can be established.

Lemma 6 *Let $e \in \mathsf{Exp}$ an expression. Exactly one of the following cases is true.*

1. *$e \in \mathsf{Val}$ is a value,*
2. *e has the form $E[v]$, for some evaluation context E and variable v,*
3. *e has the form $E[r]$ where r is a redex.*

Furthermore, the choice of an evaluation context is deterministic.

Lemma 7 *Let w, w' range over variables v and redexes r.*
If $e = E[w]$ and $e = E'[w']$, then $E = E'$ and $w = w'$.

And hence, evaluation of a closed expression always yields a unique value in the end.

Lemma 8 *Let $e \in \mathsf{Exp}$ be a closed expression.*
Then there exists exactly one $\mathsf{a} \in \mathsf{Val}$ such that $e \overset{*}{\to} \mathsf{a}$.

5.3 A Denotational Attempt

A denotational semantics for a language with recursive functions and conditionals requires a fundamental change in the choice of the underlying domains. A concrete example will provide the intuition.

5.3.1 Recursion

Question: Given the definition

```
fac (n) = if n=0 then 1 else n*fac (n-1)
```

what is the meaning of `fac` in a function call like `fac (2)`? (Recall that, in a denotational semantics, the meaning of $\mathcal{E}[\![\texttt{fac (2)}]\!] = A(\mathcal{E}[\![\texttt{fac}]\!])\ (\mathcal{E}[\![2]\!])$ is a function of the meanings of the direct subexpressions.)

Since the value of `n` is drawn from $\mathbf{Z}$, `fac` must be interpreted by some function. From the big-step semantics, we know that `fac (-1)` does not terminate, so that a partial function $g : \mathbf{Z} \hookrightarrow \mathbf{Z}$ seems appropriate for the meaning of `fac`.

How can we define this function? We partition $\mathbf{Z} = A_0 \cup A_1 \cup \dots \cup A_\infty$ such that

$$\begin{aligned} A_j &= \{y \in \mathbf{Z} \mid y - j = 0, (\forall 0 \leq i < j)\ y - i \neq 0\} & (3) \\ A_\infty &= \{y \in \mathbf{Z} \mid (\forall i \in \mathbf{N})\ y - i \neq 0\} & (4) \end{aligned}$$

With this definition

- A_j is the set of arguments such that the condition `n=0` becomes true in the $j + 1$st recursive call to `fac` and
- A_∞ is the set of arguments such that the condition `n=0` never becomes true.

Hence g can be defined by case distinction as

$$g(y) = \begin{cases} \Pi_{i=0}^{j-1}(y - i) & \text{if } y \in A_j \\ \text{undefined} & \text{if } y \in A_\infty \end{cases}$$

Since $A_j = \{j\}$ and $A_\infty = \{y \in \mathbf{Z} \mid y < 0\}$ this means that

$$g(y) = \begin{cases} \Pi_{i=1}^{y} i = y! & \text{if } y \geq 0 \\ \text{undefined} & \text{if } y < 0 \end{cases}$$

Unfortunately, this is an ad-hoc computation which needs to be redone for each particular recursive function definition. Hence, we look for a more general way of defining the function g from a recursive function definition like `fac`. To do so, we consider the recursive definition as a *functional equation* and g as a particular solution of that equation:

For all $y \in \mathbf{Z}$ it holds that

$$g(y) = \begin{cases} 1 & \text{if } y = 0 \\ y \cdot g(y - 1) & \text{if } y \neq 0 \end{cases} \tag{5}$$

Proof: by case analysis over $y \in A_j$ and $y \in A_\infty$.

Unfortunately, such functional equations have more than one solution, in general. In particular, equation (5) has infinitely many solutions: For $x \in \mathbf{Z}$ define g_x by

$$g_x(y) = \begin{cases} y! & \text{if } y \geq 0 \\ x & \text{if } y < 0 \end{cases}$$

It is easy to check that $(\forall x \in \mathbf{Z})$ g_x solves equation (5)!

However, considering the operational intuition, none of the solutions g_x make sense because each of them *invents* a value whereas g makes the least possible assumptions.

Hence, what we are looking for is a partial function that solves the functional equation but which is "as undefined as possible". That is, a sensible meaning for a recursive equation is the least partial function (with respect to inclusion of function graphs) that solves the equation.

5.3.2 Strictness

As another example for a program with recursively defined functions consider

```
g (n) = 3
h (n) = h (n)
```

What is the value of `g (h (1))`?

The above reasoning tells us that the meaning of `h` is the least function h such that $(\forall y \in \mathbf{Z})\ h(y) = h(y)$. Clearly, all partial functions solve this equation. But the smallest of them is $\emptyset$, the function with the empty graph (its domain and range are empty).

Similarly, the meaning of `g` is a function g such that $(\forall y \in \mathbf{Z})\ g(y) = 3$ so that g must be the constant function that always returns 3.

Coming back to the value of `g (h (1))`, we find that `h (1)` is undefined so that `g` is never consider and the value of the whole expression is undefined.

This way of reasoning corresponds to *call-by-value* parameter passing: first, we need to compute the value of `h (1)` and only if that value is defined, then we can pass it to `g` for further computation.

Alternatively, parameters could be passed using *call-by-name parameter passing.* Under that regime, a parameter is only evaluated if its value is needed to compute the result. Since g above is a constant function, its call-by-name evaluation never needs to look at its argument, so that `g (h (1))` would yield 3 in the end.

However, it is not possible to model this behavior of g using partial functions! What would be the graph of a function g that returns a value even if its argument is undefined (*i.e.*, not present)?

NB. Similar problems arise if we want to define the meaning of a conditional by a function:

```
f (n) = if 0 then h (n) else 1
h (n) = h (n)
```

Clearly, `f` denotes a constant function (even with call-by-value parameter passing) but the meaning of `f`'s right-hand side must be a function that maps the undefined value of `h (n)` to `1`.

Functions that require their arguments to be present (evaluated) are *strict*, otherwise they are *non-strict.*

These problems indicate that sets and set-theoretic functions are not adequate for defining denotational semantics for recursive functions and conditionals. Hence, we set out to find a suitable foundation for expression such semantics. There are two main requirements for this foundation

1. recursive functional equations need to have a least solution
2. non-strict functions should be expressible.

6 Complete Partial Orders and Fixpoints

Domains which are suitable for denotational semantics are constructed from certain partial orderings. The special feature that makes it possible to determine the *least* solution of a functional equation in the set of partial functions is the fact that such functions can be ordered by definedness (graph inclusion).

Background reading:

- Carl A. Gunter and Dana S. Scott, *Semantic Domains*, Handbook of Theoretical Computer Science, Volume B.
- Rudolf Berghammer, Semantik von Programmiersprachen, Logos Verlag, Berlin, 2001. (Kapitel 2)
- Peter Thiemann, Grundlagen der funktionalen Programmierung, Teubner Verlag, 1994. (Kapitel 10)

6.1 Complete Partial Orderings, Continuous Functions, Fixpoints

Definition 18 $(A, \sqsubseteq)$ is a *partially ordered set* (poset) if $A \neq \emptyset$ is a set and $\sqsubseteq\ \subseteq A \times A$ is a reflexive, transitive, and antisymmetric relation.

Hence,

reflexive	$(\forall a \in A)$	$a \sqsubseteq a$
transitive	$(\forall a, b, c \in A)$	$a \sqsubseteq b \wedge b \sqsubseteq c \Rightarrow a \sqsubseteq c$
antisymmetric	$(\forall a, b \in A)$	$a \sqsubseteq b \wedge b \sqsubseteq a \Rightarrow a = b$

Examples:

1. For each set M, $(M, =)$ is the *discrete poset.*
2. $(\mathbf{N}, \leq)$ is a poset.
3. For each set M, $(\mathcal{P}(M), \subseteq)$ is a poset.
4. For each pair of sets M, N, the set of partial functions

   $$M \hookrightarrow N \subseteq \mathcal{P}(M \times N)$$

   is a poset where $f \sqsubseteq g$ iff

   $$(\forall (a, b) \in M \times N)\ (a, b) \in f \Rightarrow (a, b) \in g$$

Definition 19 Let $(A, \sqsubseteq)$ be a poset, $X \subseteq A$, and $a \in A$.

- a is an *upper bound of* X if $(\forall x \in X)\ x \sqsubseteq a$.
- a is a *least element of* X if $a \in X$ and $(\forall x \in X)\ a \sqsubseteq x$.

- If the set $\{x \in A \mid x \text{ is upper bound of } X\}$ has a least element, then $a = \bigsqcup X$ is the *least upper bound of* X or the *supremum of* X.

Notation: the supremum of $\{a, b\}$ (if exists) is written $a \sqcup b$.

There are dual definitions for *lower bound*, *greatest element*, and *infimum* denoted by the symbol $\sqcap$.

Lemma 9 *A poset $(A, \sqsubseteq)$ has at most one least (greatest) element.*

Proof: Suppose that a and b are both least elements of A. Since a is least, it must be $a \sqsubseteq b$, and since b is also least, it must be $b \sqsubseteq a$. Taken together with antisymmetry, it follows that $a = b$.

Examples:

- The discrete poset $(M, =)$ has neither least nor greatest elements if $|M| \geq 2$.
- $(\mathbf{Z}, \leq)$ is a poset without least (greatest) element.
- $(\mathcal{P}(M), \subseteq)$ has least element $\emptyset$ and greatest element M.
- $M \hookrightarrow N$ is a poset with least element $\emptyset$ (the empty function), but no greatest element.

Definition 20 Let $(A, \sqsubseteq)$ be a poset.

A *chain* is a subset $X \subset A$ such that for all $x, y \in X$ it holds that $x \sqsubseteq y$ or $y \sqsubseteq x$.

Examples:

- The empty set and each singleton set are chains in any poset.
- Any subset $X \subseteq \mathbf{Z}$ is a chain in $(\mathbf{Z}, \leq)$.
- Consider the poset $(\{1, 2, 3\}, \subseteq)$.

  The set $X_1 = \{\emptyset, \{1\}, \{1, 2\}, \{1, 2, 3\}\}$ is a chain.

  The set $X_2 = \{\emptyset, \{1\}, \{1, 2\}, \{1, 3\}\}$ is *not* a chain because neither $\{1, 2\} \subseteq \{1, 3\}$ nor $\{1, 3\} \subseteq \{1, 2\}$.
- Consider the poset $\{\mathbf{Z} \hookrightarrow \mathbf{Z}, \subseteq\}$.

  The set $\{f, g\}$ with $f(x) = x$ and $g(x) = 2x$ is not a chain.

  The set $\{g_i \mid i \in \mathbf{N}\}$ is a chain if

$$g_i(y) = \begin{cases} y! & \text{if } 0 \leq y < i \\ \text{undefined} & \text{otherwise} \end{cases}$$

Definition 21 A *chain-complete partial ordering (CCPO)* is a poset $(A, \sqsubseteq)$ such that

1. A has a least element $\bot$ (*bottom*) and
2. Each chain $X \subseteq A$ has a supremum $\bigsqcup X \in A$.

Examples:

- The discrete poset $(A, =)$ is not a CCPO if $|A| > 1$ (because it does not have a least element).
- $(\mathbf{N}, \leq)$ is not a CCPO. It has a least element but the chain $\mathbf{N}$ does not have a supremum in $\mathbf{N}$.
- For any set M, the poset $(\mathcal{P}(M), \subseteq)$ is a CCPO. The least element is $\emptyset$ and for any subset $X \subseteq \mathcal{P}(M)$ it holds that $\bigsqcup X = \bigcup X \in \mathcal{P}(M)$, hence it also holds for arbitrary chains.
- $M \hookrightarrow N$ is a CCPO with least element $\emptyset$. The proof that every chain in $M \hookrightarrow N$ has a supremum in $M \hookrightarrow N$ is left as an exercise.

Definition 22 A poset $(A, \sqsubseteq)$ is a *flat ordering* if it has a least element $\bot \in A$ such that, for all $x, y \in A$,

$$x \sqsubseteq y \qquad \text{iff} \qquad x = \bot \vee x = y$$

Lemma 10 *Every flat ordering is a CCPO.*

Example: Consider $\mathbf{N}_\bot = (\mathbf{N} \cup \{\bot\}, \sqsubseteq)$ where $\sqsubseteq$ is the flat ordering relation.

Definition 23 Let $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$ be CCPOs. A function $f \in A \to B$ is

1. *monotone* if, for all $x, y \in A$, $x \sqsubseteq_A y$ implies $f(x) \sqsubseteq_B f(y)$,
2. *continuous* if f is monotone and, for all chains $X \subseteq A$, $f(\bigsqcup_A X) = \bigsqcup_B f(X)$,
3. *strict* if $f(\bot_A) = \bot_B$.

Examples: Define the successor function $succ_\bot : \mathbf{N}_\bot \to \mathbf{N}_\bot$ by

$$succ_\bot(x) = \begin{cases} \bot & x = \bot \\ x + 1 & x \neq \bot \end{cases}$$

This function is

1. monotone: suppose that $x \sqsubseteq y$. By definition of $\sqsubseteq$, $x = \bot \vee x = y$. If $x = \bot$, then $succ_\bot(x) = succ_\bot(\bot) = \bot \sqsubseteq succ_\bot(y)$. If $x = y$, then also $succ_\bot(x) = succ_\bot(y)$. Taken together, it follows that $f(x) \sqsubseteq f(y)$.
2. continuous: we already know that $succ_\bot$ is monotone. If X is a chain in $\mathbf{N}_\bot$, then either $X = \{x\}$ or $X = \{\bot, y\}$ with $y \neq \bot$. In either case, $succ_\bot(\bigsqcup X) = \bigsqcup succ_\bot(X)$ is immediate.

3. strict: immediate by definition.

A more interesting function is the conditional $ite : \mathbf{N}_\bot \times \mathbf{N}_\bot \times \mathbf{N}_\bot \to \mathbf{N}_\bot$ defined by

$$ite(b,t,e) = \begin{cases} \bot & b = \bot \\ t & b \neq \bot \wedge b > 0 \\ e & b \neq \bot \wedge b = 0 \end{cases}$$

Exercise: *ite* is monotone and continuous in each argument and strict in the first argument.

Recall the definition of function composition $(f \circ g)(x) = (f(g(x)))$.

Lemma 11 *Let $(A, \sqsubseteq_A)$, $(B, \sqsubseteq_B)$, and $(C, \sqsubseteq_C)$ be CCPOs and $f \in A \to B$, $g \in B \to C$.*

1. *If $(B, \sqsubseteq_B)$ is a flat ordering and f monotone, then f is either strict or constant.*
2. *If $(A, \sqsubseteq_A)$ is a flat ordering and f strict, then f is monotone.*
3. *If every chain in $(A, \sqsubseteq_A)$ is finite and f is monotone, then f is continuous.*
4. *If f and g are both monotone (continuous, strict), then their composition $g \circ f \in A \to C$ is monotone (continuous, strict).*
5. *If f is monotone and $X \subseteq A$ is a chain, then $\bigsqcup_B f(X) \sqsubseteq_B f(\bigsqcup_A X)$.*

In the previous section, we found that the factorial function $n \mapsto n!$ is a solution of the equation

$$g(y) = \begin{cases} 1 & \text{if } y = 0 \\ y \cdot g(y-1) & \text{if } y \neq 0 \end{cases} \tag{6}$$

This fact can be reinterpreted by considering the equation as defining a function τ that constructs a new function from any given $g : \mathbf{Z} \hookrightarrow \mathbf{Z}$.

$$\tau(g)(y) = \begin{cases} 1 & \text{if } y = 0 \\ y \cdot g(y-1) & \text{if } y \neq 0 \end{cases} \tag{7}$$

So $\tau : (\mathbf{Z} \hookrightarrow \mathbf{Z}) \to (\mathbf{Z} \hookrightarrow \mathbf{Z})$.

This new point of view has two remarkable consequences.

1. τ's definition is *not* recursive.
2. The factorial function is a *fixpoint* of τ:

   Let $f : \mathbf{Z} \hookrightarrow \mathbf{Z}$ defined by $f(n) = n!$ if $n \geq 0$ and undefined otherwise. Then

$$\tau(f)(y) = \begin{cases} 1 & \text{if } y = 0 \\ y \cdot f(y-1) & \text{if } y \neq 0 \end{cases} = \begin{cases} 1 & \text{if } y = 0 \\ y \cdot (y-1)! & \text{if } y - 1 \geq 0 \\ \text{undefined} & \text{if } y < 0 \end{cases} = \begin{cases} y! & \text{if } y \geq 0 \\ \text{undefined} & \text{if } y < 0 \end{cases}$$

   Hence f is a fixpoint of τ.

So, instead of finding the least solution of a functional equation, it is sufficient to find the least fixpoint of a function like τ (like solutions, fixpoints are not unique). It turns out that CCPOs are the right framework for constructing functions that have such (least) fixpoints.

Definition 24 Let $(A, \sqsubseteq)$ be a CCPO and $f : A \to A$ a function.
$x \in A$ is a *fixpoint of f* if $f(x) = x$.
If the set $\{x \mid x \text{ fixpoint of } f\}$ has a least element x_0, then x_0 is the *least fixpoint of f*.

Notation: μf or *fix* f denote the least fixpoint of f.

Lemma 12 *Let $(A, \sqsubseteq)$ be a CCPO and $f : A \to A$ monotone.*
If x_0 is a fixpoint of f, then $\bigsqcup_{i \in \mathbf{N}} f^{(i)}(\bot) \sqsubseteq x_0$.

Proof: Two things need to be established:

1. $X = \{f^{(i)}(\bot) \mid i \in \mathbf{N}\}$ is a chain.
2. $(\forall i \in \mathbf{N})\ f^{(i)}(\bot) \sqsubseteq x_0$, *i.e.*, x_0 is an upper bound.

The chain $\{f^{(i)}(\bot) \mid i \in \mathbf{N}\}$ is often called *fixpoint iteration for f*. It always yields a lower approximation to the fixpoints of f. If additionally f is continuous, then the supremum of the fixpoint iteration *is* the least fixpoint. This is stated in the Knaster-Tarski-Kleene fixpoint theorem.

Theorem 2 *Let $(A, \sqsubseteq)$ be a CCPO and $f : A \to A$ continuous.*
Then μf exists and $\mu f = \bigsqcup_{i \in \mathbf{N}} f^{(i)}(\bot)$.

Proof: From Lemma 12, we know that $x_0 = \bigsqcup_{i \in \mathbf{N}} f^{(i)}(\bot) \sqsubseteq \mu f$ (if the latter exists). It remains to show that x_0 is a fixpoint.

Example: To apply the fixpoint theorem to the factorial function, we need to establish that the function τ is indeed continuous.

τ is monotone: Let $g \sqsubseteq h$ be functions in $\mathbf{Z} \hookrightarrow \mathbf{Z}$. If $(y, z) \in \tau(g)$, then there are two cases (by definition of τ).

- $y = 0$ and $z = 1$. Clearly, $(y, z) = (0, 1) \in \tau(h)$.
- $y \neq 0$ and $(\exists z' \in \mathbf{Z})\ (y - 1, z') \in g$ and $z = y \cdot z'$. Since $g \sqsubseteq h$ is assumed, $(y - 1, z') \in h$ and hence $(y, y \cdot z') \in \tau(h)$, too.

In each case, $(y, z) \in \tau(h)$ so that τ is monotone.

τ is continuous: Let G be a chain in $\mathbf{Z} \hookrightarrow \mathbf{Z}$ with supremum $g = \bigsqcup G$. Show that $\tau(G)$ has supremum $\tau(g)$.

$\bigsqcup \tau(G) \sqsubseteq \tau(g)$ because of Lemma 11.5.

It remains to see that $\tau(g) \sqsubseteq \bigsqcup \tau(G)$. Suppose that $(y, z) \in \tau(g)$. There are two cases (by definition of τ).

- $y = 0$ and $z = 1$. Since $(0,1) \in \tau(g')$ for all $g' \in G$, it must be that $(0,1) \in \bigsqcup \tau(G)$.
- $y \neq 0$ and $(\exists z' \in \mathbf{Z})\ (y-1, z') \in g$ and $z = y \cdot z'$. Since $g = \bigsqcup G$, there must be some $g' \in G$ with $(y-1, z') \in g'$. Hence, $(y, y \cdot z') \in \tau(g') \subseteq \bigsqcup \tau(G)$.

In each case, $(y,z) \in \bigsqcup \tau(G)$.

Hence, the Knaster-Tarski-Kleene theorem is applicable to τ and its least fixpoint can be determined from the fixpoint iteration. The starting point, $\perp$, is the empty function $\emptyset \in \mathbf{Z} \hookrightarrow \mathbf{Z}$.

$$\begin{aligned}
\tau^{(0)}(\emptyset)(y) &= \text{undefined} \\
\tau^{(1)}(\emptyset)(y) &= \begin{cases} 1 & \text{if } y = 0 \\ y \cdot \tau^{(0)}(\emptyset)(y-1) & \text{if } y \neq 0 \end{cases} = \begin{cases} 1 & \text{if } y = 0 \\ \text{undefined} & \text{if } y \neq 0 \end{cases} \\
\tau^{(2)}(\emptyset)(y) &= \begin{cases} 1 & \text{if } y = 0 \\ y \cdot \tau^{(1)}(\emptyset)(y-1) & \text{if } y \neq 0 \end{cases} = \begin{cases} 1 & \text{if } y = 0 \\ 1 & \text{if } y = 1 \\ \text{undefined} & \text{if } y < 0 \vee y > 1 \end{cases} \\
\tau^{(3)}(\emptyset)(y) &= \begin{cases} 1 & \text{if } y = 0 \\ y \cdot \tau^{(2)}(\emptyset)(y-1) & \text{if } y \neq 0 \end{cases} = \begin{cases} 1 & \text{if } y = 0 \\ 1 & \text{if } y = 1 \\ 2 & \text{if } y = 2 \\ \text{undefined} & \text{if } y < 0 \vee y > 2 \end{cases} \\
\tau^{(4)}(\emptyset)(y) &= \begin{cases} 1 & \text{if } y = 0 \\ y \cdot \tau^{(3)}(\emptyset)(y-1) & \text{if } y \neq 0 \end{cases} = \begin{cases} 1 & \text{if } y = 0 \\ 1 & \text{if } y = 1 \\ 2 & \text{if } y = 2 \\ 6 & \text{if } y = 3 \\ \text{undefined} & \text{if } y < 0 \vee y > 3 \end{cases} \\
&\vdots
\end{aligned}$$

By induction on i we can show that

$$\tau^{(i)}(\emptyset)(y) = \begin{cases} y! & \text{if } 0 \leq y < i \\ \text{undefined} & \text{if } y < 0 \vee y \geq i \end{cases}$$

and from that we obtain that the supremum is the factorial function:

$$(\bigsqcup_{i \in \mathbf{N}} \tau^{(i)}(\emptyset))(y) = \begin{cases} y! & \text{if } 0 \leq y \\ \text{undefined} & \text{if } y < 0 \end{cases}$$

6.2 Admissible Predicates

Having fixed the semantics of recursive definitions as the least fixpoint of an associated function, the next question is how to establish properties of such a recursive function. The obvious idea would be to express properties as predicates and define the validity of predicates by fixpoint iteration, too. Unfortunately, this approach fails because some interesting predicates do not behave continuously. For example, the predicate

$$P(g) = \qquad (\exists x \in \mathbf{N})\ g(x) = \perp$$

is true for all iterates $\tau^{(i)}(\bot)$ in the previous example, that is, for all elements of the chain $\{\tau^{(i)}(\bot) \mid i \in \mathbf{N}\}$. However, the predicate is false for the least fixpoint of τ, since $\bigsqcup_{i \in \mathbf{N}} \tau^{(i)}(\emptyset)$ is defined for all $x \in \mathbf{N}$.

Let $\mathbf{B}$ be the flat partial order with elements $\{\mathit{true}, \mathit{false}, \bot\}$.

Definition 25 Let $(A, \sqsubseteq)$ be a CCPO and $P : A \to \mathbf{B}$ a predicate.
P is *admissible* if, for each chain $X \subseteq A$,

$$(\forall x \in X)\ P(x) \Rightarrow P(\bigsqcup X)$$

Clearly, an admissible predicate on A is a continuous function $A \to \mathbf{B}$.

Admissible predicates can be proven using the fixpoint induction principle:

Theorem 3 *Let $(A, \sqsubseteq)$ be a CCPO, $f \in A \to A$ continuous, and $P : A \to \mathbf{B}$ an admissible predicate. Then*

$$\frac{P(\bot) = \mathit{true} \qquad (\forall x \in A)\ P(x) = \mathit{true} \Rightarrow P(f(x)) = \mathit{true}}{P(\mathit{fix}\ f) = \mathit{true}}$$

Proof: Let $X = \{f^{(i)}(\bot) \mid i \in \mathbf{N}\}$. Since f is monotone, X is a chain.

Assuming the premises of the inference rule, an induction on $i \in \mathbf{N}$ yields that $P(x) = \mathit{true}$, for all $x \in X$. Since P is admissible, it holds that $P(\bigsqcup X) = \mathit{true}$, too, and the fixpoint theorem yields that $\bigsqcup X = \mathit{fix}\ f$. Hence, the consequence of the rule.

The following two lemmas provide important tools to build admissible predicates.

Lemma 13 *A predicate $P : A \to \mathbf{B}$ is admissible if there is a CCPO $(B, \sqsubseteq)$ and continuous functions $f, g : A \to B$ such that*

$$(\forall x \in A)\ P(x) = \mathit{true} \qquad \Leftrightarrow \qquad f(x) \sqsubseteq g(x)$$

Lemma 14 *Let $P, Q : A \to \mathbf{B}$ be admissible predicates.*
Then $(P \wedge Q) : A \to \mathbf{B}$ is also admissible with

$$(P \wedge Q)(x) = \begin{cases} \bot & P(x) = \bot \text{ or } Q(x) = \bot \\ P(x) \wedge Q(x) & P(x) \neq \bot \text{ and } Q(x) \neq \bot \end{cases}$$

Exercise: Show that, for continuous functions $f, g : A \to B$, the predicate $P(x) = (f(x) = g(x))$ is admissble.

Lemma 15 *Let $(A, \sqsubseteq)$ be a CCPO, $x_0 \in A$, and $f : A \to A$ continuous. If $f(x_0) \sqsubseteq x_0$, then $\mathit{fix}\ f \sqsubseteq x_0$.*

Proof: Establish that $P(x) = x \sqsubseteq x_0$ is admissble and apply fixpoint induction.

6.3 Domain Constructions

To construct denotational semantics, requires an arsenal of CCPOs analogous to the well-known constructions on sets (product, sum, functions). This subsection establishes the corresponding constructions on CCPOs.

Starting with this section, if A is a poset, then its carrier set is $|A|$ and its ordering is $\sqsubseteq_A$ with the subscript omitted if it is clear from the context.

Definition 26 Let A and B be posets.

Define the *cartesian product* $A \times B$ by $|A \times B| = |A| \times |B|$ and

$$(x_1, y_1) \sqsubseteq (x_2, y_2) \qquad \Leftrightarrow \qquad x_1 \sqsubseteq_A x_2 \wedge y_1 \sqsubseteq_B y_2$$

Lemma 16 *1. If A, B are CCPOs, then so is $A \times B$.*

2. The projection functions $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ are continuous.

This product is sometimes too general because it allows its components to be $\perp$ independently. In many programming languages, the components of a pair cannot exist independently. If the computation of one component fails, then the pair (record, object) is not constructed. This intuition is reflected in the following variant of the product, the *smash product*.

Definition 27 Let A and B be posets with least element.

Define the *smash product* $A \otimes B$ by

$$|A \otimes B| = ((|A| \setminus \{\perp_A\}) \times (|B| \setminus \{\perp_B\})) \cup \{\perp\}$$

and the ordering by

$$x \sqsubseteq y \qquad \Leftrightarrow \qquad x = \perp \vee x = (x_1, x_2) \wedge y = (y_1, y_2) \wedge x_1 \sqsubseteq_A y_1 \wedge x_2 \sqsubseteq_B y_2$$

Define further $\mathit{smash} : A \times B \to A \otimes B$ by

$$\mathit{smash}\ (x, y) = \begin{cases} \perp & x = \perp \vee y = \perp \\ (x, y) & x \neq \perp \wedge y \neq \perp \end{cases}$$

Lemma 17 *1. If A, B are CCPOs, then $A \otimes B$ is a CCPO.*

2. The projection functions $\pi_1^{\otimes} : A \otimes B \to A$ and $\pi_2^{\otimes} : A \otimes B \to B$ are continuous (with $\pi_i^{\otimes}(\perp) = \perp$).

3. The construction function $\mathit{smash} : A \times B \to A \otimes B$ is continuous.

Similar to the product construction, there are two variants for constructing sums.

Definition 28 Let A and B be posets.
The *separated sum* $A + B$ is defined by

$$|A + B| = \{(x, 0) \mid x \in A\} \cup \{(y, 1) \mid y \in B\} \cup \{\bot\}$$

with ordering

$$\begin{array}{lclcl} x \sqsubseteq y & \Leftrightarrow & & & x = \bot \\ & & \vee & & x = (x_1, 0) \wedge y = (y_1, 0) \wedge x_1 \sqsubseteq_A y_1 \\ & & \vee & & x = (x_2, 1) \wedge y = (y_2, 1) \wedge x_2 \sqsubseteq_B y_2 \end{array}$$

Lemma 18 *1. If A, B are CCPOs, then $A + B$ is a CCPO.*

2. The injection functions $Inl : A \to A + B$ and $Inr : B \to A + B$ defined by $Inl\ (x) = (x, 0)$ and $Inr\ (y) = (y, 1)$ are continuous.

3. The elimination function $case : A + B \times (A \to C) \times (B \to C) \to C$ defined by

$$\begin{array}{rcl} case\ ((x, 0), f, g) & = & f(x) \\ case\ ((y, 1), f, g) & = & g(y) \\ case\ (\bot, f, g) & = & \bot \end{array}$$

is continuous.

Definition 29 Let A and B be posets with least element.
The *coalesced sum* $A \oplus B$ is defined by

$$|A \oplus B| = \{(x, 0) \mid x \in A \setminus \{\bot_A\}\} \cup \{(y, 1) \mid y \in B \setminus \{\bot_B\}\} \cup \{\bot\}$$

with ordering

$$\begin{array}{lclcl} x \sqsubseteq y & \Leftrightarrow & & & x = \bot \\ & & \vee & & x = (x_1, 0) \wedge y = (y_1, 0) \wedge x_1 \sqsubseteq_A y_1 \\ & & \vee & & x = (x_2, 1) \wedge y = (y_2, 1) \wedge x_2 \sqsubseteq_B y_2 \end{array}$$

Lemma 19 *1. If A, B are CCPOs, then $A \oplus B$ is a CCPO.*

2. The injection functions $Inl^{\oplus} : A \to A \oplus B$ and $Inr^{\oplus} : B \to A \oplus B$ defined by $Inl^{\oplus}(x) = \bot$ if $x = \bot_A$ and $(x, 0)$, otherwise, and $Inr^{\oplus}(y) = \bot$ if $y = \bot_B$ and $(y, 1)$, otherwise, are continuous.

3. The elimination function $case^{\oplus} : A \oplus B \times (A \to C) \times (B \to C) \to C$ defined by

$$\begin{array}{rcl} case^{\oplus}((x, 0), f, g) & = & f(x) \\ case^{\oplus}((y, 1), f, g) & = & g(y) \\ case^{\oplus}(\bot, f, g) & = & \bot \end{array}$$

is continuous.

Definition 30 Let A be a set and B be a poset.
Define $[A \to B]$ by

$$|[A \to B]| = |B|^{|A|}$$

with the pointwise ordering

$$f \sqsubseteq g \qquad \Leftrightarrow \qquad (\forall x \in A)\ f(x) \sqsubseteq_B g(x)$$

Lemma 20 *1. If A is a set and B a CCPO, then $[A \to B]$ is a CCPO.*

2. If A is also CCPO, then eval $: [A \to B] \times A \to B$ defined by eval $(f, x) = f(x)$ is continuous.

Definition 31 Let A be a poset.
Define $A_\perp$ by $|A_\perp| = \{(x, 0) \mid x \in |A|\} \cup \{\perp\}$ with ordering

$$x \sqsubseteq y \qquad \Leftrightarrow \qquad x = \perp \quad \vee \quad x = (x_1, 0) \wedge y = (y_1, 0) \wedge x_1 \sqsubseteq_A y_1$$

Lemma 21 *1. If M is a discrete poset, then $M_\perp$ is a CCPO.*

2. If A is a CCPO, then $A_\perp$ is a CCPO.

3. The functions up $: A \to A_\perp$ and down $: A_\perp \to A$ defined by up $(x) = (x, 0)$ and down $(x, 0) = x$ and down $(\perp) = \perp$ are continuous.

7 Functions and Recursion, continued

Now that the fundamentals of domain theory are explained, a denotational semantics for the language with functions can be defined. Furthermore, we will consider variations on evaluation strategy (call-by-value vs. call-by-name), nesting of function definitions, and static vs. dynamic scope.

7.1 Functions and Recursion, Denotationally

Instead of sticking with sets and using the CCPO $\mathbf{Z} \hookrightarrow \mathbf{Z}$ directly, we'll commit to CCPOs entirely.

Exercise: for all sets M and N, there is a continuous embedding from $M \hookrightarrow N$ into $[M_\perp \to N_\perp]$.

For brevity, we write $A \to B$ instead of $[A \to B]$ and assume that $\to$ associates to the right. That is, $A \to B \to C$ stands for $[A \to [B \to C]]$.

The semantic domains are the following CCPOs.

$$\begin{array}{rcl} \mathsf{Val} & = & \mathbf{Z}_\perp \\ \mathsf{Env} & = & \mathsf{Var} \to \mathsf{Val} \\ \mathsf{FVal} & = & \mathsf{Val} \to \mathsf{Val} \\ \mathsf{FEnv} & = & \mathsf{Fun} \to \mathsf{FVal} \end{array}$$

The expressed values are still taken from Val. However, there are two kinds of denoted values. The elements of Val can be bound to variables, whereas the elements of FVal can be bound to function symbols.

Hence, the semantic equations for expressions are parameterized with two kinds of environments, the value environment Env and the function environment FEnv, and the semantic equation for programs needs to build a function environment.

$$\begin{array}{rcl} \mathcal{E} & : & \mathsf{Exp} \to \mathsf{FEnv} \to \mathsf{Env} \to \mathsf{Val} \\ \mathcal{P} & : & \mathsf{Prog} \to \mathsf{Val} \end{array}$$

In the equations, $\rho \in \mathsf{Env}$ and $\varphi \in \mathsf{FEnv}$.

$$\begin{array}{lcl} \mathcal{E}[\![v]\!]\varphi\rho & = & \rho(v) \\ \mathcal{E}[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]\varphi\rho & = & \mathit{ite}(\mathcal{E}[\![e_1]\!]\varphi\rho, \mathcal{E}[\![e_2]\!]\varphi\rho, \mathcal{E}[\![e_3]\!]\varphi\rho) \\ \mathcal{E}[\![f(e)]\!]\varphi\rho & = & \varphi(f)(\mathcal{E}[\![e]\!]\varphi\rho) \end{array}$$

$$\begin{array}{rcl} \mathcal{P}[\![\Big(f_i(v_i) = e_i\Big)_{i=1}^{n}\ \ e_0]\!] & = & \mathcal{E}[\![e_0]\!](\mathit{fix}\ g)\perp \\ \text{where } g(\varphi)(f_i)(y) & = & \mathcal{E}[\![e_i]\!]\varphi\{v_i \mapsto y\} \end{array}$$

The remaining equations for constants and primitives (addition) carry over mutatis mutandis.

Question: Does this definition really match our intentions? That is, is $\mathcal{P}[\![p]\!]$ really the value of p obtained by a call-by-value evaluation of p (according to the previously stated big-step or small-step semantics)?

Let's check with the previous example

```
f (x) = 1
h (x) = h (x)
f (h (0))
```

This program does not terminate according to the operational semantics specified above.

The function g from the definition of $\mathcal{P}$ is defined by

$$\begin{array}{rcl} g(\varphi)(\mathtt{f})(y) & = & 1 \\ g(\varphi)(\mathtt{h})(y) & = & \varphi(\mathtt{h})(y) \end{array}$$

Fixpoint iteration yields that $\varphi_0 = \mathit{fix}\ g$ is defined by

$$\begin{array}{rcl} \varphi_0(\mathtt{f})(y) & = & 1 \\ \varphi_0(\mathtt{h})(y) & = & \perp \end{array}$$

Hence, the body of the program yields

$$\begin{array}{rl} & \mathcal{E}[\![\mathtt{f(h(0))}]\!]\varphi_0\perp \\ = & \varphi_0(\mathtt{f})(\mathcal{E}[\![\mathtt{h(0)}]\!]\varphi_0\perp) \\ = & \varphi_0(\mathtt{f})(\varphi_0(\mathtt{h})(\mathcal{E}[\![\mathtt{0}]\!]\varphi_0\perp)) \\ = & \varphi_0(\mathtt{f})(\varphi_0(\mathtt{h})(0)) \\ = & \varphi_0(\mathtt{f})(\perp) \\ = & 1 \end{array}$$

Oops, we have accidentally specified call-by-name evaluation!

To obtain call-by-value evaluation, we change the definition of g in the semantic equation for programs by insisting that all arguments to a function must be evaluated, *i.e.*, they must be $\neq \perp$.

$$\begin{array}{rcl} \mathcal{P}_v[\![\Big(f_i(v_i) = e_i\Big)_{i=1}^{n}\quad e_0]\!] & = & \mathcal{E}[\![e_0]\!](\mathit{fix}\ g')\perp \\ \text{where } g'(\varphi)(f_i)(y) & = & \begin{cases} \perp & y = \perp \\ \mathcal{E}[\![e_i]\!]\varphi\{v_i \mapsto y\} & y \neq \perp \end{cases} \end{array}$$

With this setting, $\varphi_0' = \mathit{fix}\ g'$ is defined by

$$\begin{array}{rcl} \varphi_0(\mathtt{f})(y) & = & \begin{cases} \perp & y = \perp \\ 1 & y \neq \perp \end{cases} \\ \varphi_0(\mathtt{h})(y) & = & \perp \end{array}$$

so that

$$\mathcal{E}[\![\mathtt{f(h(0))}]\!]\varphi_0\perp = \perp$$

as expected.

In short, to obtain a call-by-value semantics, all functions must be strict and variables in the environment must not be bound to the value $\bot$.

The following connection may be established between the call-by-value and the call-by-name semantics.

Lemma 22 *For all programs p, $\mathcal{P}_v[\![p]\!] \sqsubseteq \mathcal{P}[\![p]\!]$.*

That is, whenever call-by-value evaluation computes a value ($\neq \bot$), this value is correct with respect to call-by-name evaluation. It can also be shown that call-by-name evaluation yields a value whenever *any* evaluation strategy would yield a value.

7.2 More on Call-by-name

Of course, the call-by-name evaluation strategy can also be imposed in an operational semantics. One simplification is that CCPOs are **not required** for an operational semantics, sets are entirely sufficient.

7.2.1 Small-Step

In the small-step semantics, we need to revise the reduction rules for function calls and the evaluation contexts, slightly. The reduction rule for function calls no longer requires that its parameter expression be reduced to a value.

$$f\texttt{(}e'\texttt{)} \longrightarrow e[v \mapsto e'] \qquad f\texttt{(}v\texttt{)} \texttt{ = } e \in d^*$$

Correspondingly, the evaluation contexts do not descend into the parameter position, anymore.

$$E_n \quad ::= \quad [\,] \mid E_n\texttt{+}e \mid y\texttt{+}E_n \mid \texttt{if } E_n \texttt{ then } e \texttt{ else } e$$

The set of values remains unchanged. Also, the remaining definitions and statements carry over without change.

7.2.2 Big-Step

The big-step semantics turns out to be more tricky to adjust. It forces the evaluation of function arguments by having an environment $\rho \in \mathsf{Var} \hookrightarrow \mathbf{Z}$. Now that we want to bind unevaluated expressions to variables, the semantics needs to transport the information required to perform the evaluation to the place where the variable is used. That is, the environment for a call-by-name evaluation must map a variable name to a pair (ρ, e) of an environment and an expression! Such a pair is called a *closure* or a *suspension*.

$$\mathsf{Env}_n \ni \rho ::= \emptyset \mid (v, (\rho, e))\rho$$

The revised evaluation judgement has type $\mathsf{Env}_n \times \mathsf{Exp} \times \mathbf{Z}$ and its rules for variables and function calls are

$$\frac{\rho(v) = (\rho', e') \qquad d^*, \rho' \vdash e' \hookrightarrow y}{d^*, \rho \vdash v \hookrightarrow y}$$

$$\frac{d^*, (v, (\rho', e'))\emptyset \vdash e \hookrightarrow y \qquad f(v) = e \in d^*}{d^*, \rho' \vdash f(e') \hookrightarrow y}$$

7.2.3 Call-By-Need

The call-by-need evaluation strategy is an optimization of call-by-name evaluation.

strategy	function arguments are evaluated ...
call-by-value	exactly once
call-by-name	zero-arbitrary often (each use of an argument is reevaluated)
call-by-need	at most once (result of first evaluation is cached)

Hence, call-by-value may perform unnecessary work and call-by-name may redo work arbitray often. Both can lead to inefficiency. Call-By-Need can avoid unnecessary work and repetition of work.

Denotationally, call-by-need behaves exactly like call-by-name.

Operationally, the big-step semantics for call-by-name is not hard to modify to model call-by-need evaluation. However, a small-step semantics for call-by-need is non-trivial to construct. See Ariola, Felleisen, Maraist, Odersky, and Wadler, *A Call-By-Need Lambda Calculus*, 25th ACM Symposium on Principles of Programming Languages, San Francisco, USA, 1995.

7.3 Nested Scopes

Many languages allow for nested function definitions (*e.g.*, Pascal, Modula-2, JavaScript, ML, Scheme) or similar constructs (*e.g.*, nested classes in Java). The idea is that the nested functions are locally available only inside the body of the defining function.

Since each nested function binds its own formal parameters, confusing effects can be achieved due to shadowing of surrounding variable bindings.

For example, the following is a legal JavaScript program.

```
function f (x,y) {
  function h (x) {
    return x+y;
  }
  function g (y) {
    return h (x+y);
  }
  return g (5);
}
```

The syntactic combination of a list of definitions and a function body is traditionally called a *block* (this name is due to Algol68). Hence, a suitable abstract syntax would be

$$\begin{array}{lcl} e & ::= & \cdots \mid f(e) \\ d & ::= & f(v) = b \\ b & ::= & d^*\ e \end{array}$$

The corresponding big-step semantics defines two judgements

1. $d^*, \rho \vdash b \hookrightarrow y$
2. $d^*, \rho \vdash e \hookrightarrow y$

The expression judgement is defined as before. The interesting part is the rule for evaluating a block.

$$\frac{d_2^*; d_1^*, \rho \vdash e \hookrightarrow y}{d_1^*, \rho \vdash (d_2^*\ e) \hookrightarrow y}$$

This rule relies on the convention that the condition $f(v) = b \in d^*$ obtains the leftmost definition for f in d^*. Alternatively, d could be represented by a definition environment, *i.e.*, a function from function symbols to function definitions, and then the block rule would overwrite the previous function definitions.

Building a small-step semantics requires to α-rename all function symbols to make them globally unique and then lift all definitions to the toplevel by adding free variables as parameters. We omit the tedious construction. It is a special case of the *lambda lifting* transformation, known from the implementation of functional programming languages.

For a denotational semantics, the main step is to define the semantics of a block. In comparison to the language without nested functions, each block requires the construction of a fixpoint environment.

$$\begin{array}{rcl} \mathcal{B}[\![\Big(f_i(v_i) = e_i\Big)_{i=1}^{n}\ e]\!]\varphi\rho & = & \mathcal{E}[\![e]\!](\varphi[f_i \mapsto \bot|_{i=1}^{n}] \sqcup \mathit{fix}\ g)\rho \\ \text{where } g(\varphi')(f_j)(y) & = & \mathcal{E}[\![e_j]\!](\varphi[f_i \mapsto \bot|_{i=1}^{n}] \sqcup \varphi')\{v_j \mapsto y\} \end{array}$$

As before, this defines a call-by-name semantics. To obtain a call-by-value version, the argument of f_j must be tested against $\bot$ before entering the body of the function.

7.4 Static Scope vs. Dynamic Scope

Once nested function definitions are present in a language, there is a further choice of interpreting variable binding. The standard choice is that variable occurrences always refer to their next, lexically enclosing definition. That is, given the definition

```
function f (x) {
  function g (y) {
    return x;
  }
  function h (x) {
    return g (x);
  }
  return h (10);
}
```

the function call `f (0)` yields `0`. This is because the `x` in the definition of `g` refers to the next lexically enclosing binding of `x` as a formal parametere of `f`.

This way of resolving a binding is called *static scope*. An equally sensible way of resolution is *dynamic scope*. Under the regime of dynamic scope, a variable occurrence picks up the last *dynamically preceding* binding of the variable, *i.e.*, the last executed binding for the variable.

Executing the above definition with dynamic scope yields `10` because the last executed binding of `x` before entering `g` was the parameter passing of `10` to function `h`.

Most often, dynamic scope is *not* the expected behavior. However, there are a few useful applications for it, for example, redefining standard output ports by rebinding them before calling the output procedure. A few languages use dynamic scope by default, among them TEX and Emacs Lisp.

Exercise: Modify a semantics style of your choice to evaluate function calls with dynamically scoped variables.

8 First Class Functions

The next step is to make functions *first class*, that is, to promote them to being expressed values. In the syntax, the main novelty is to have an expression to construct a function (function abstraction) and an expression to deconstruct a function (function application). (We'll drop named functions for now, they can be added in the same way as before.)

$$\begin{array}{rcll}
\mathsf{Exp} \ni e & ::= & \dots & \\
& | & v & \text{variables} \\
& | & \lambda v \,.\, e & \text{function abstraction} \\
& | & e\ e & \text{function application}
\end{array}$$

For brevity, we stick to a mathematical notation for abstraction and application. In actual programming languages, the notation varies widely.

```
// JavaScript
var f = function (x) { return x+1; };
f (0);
// Scheme
(define f (lambda (x) (+ x 1))
(f 0)
// ML
val f = fn x => x + 1
f 0
// Haskell
f = \x -> x + 1
f 0
// Java
interface Function {
  int apply (x : int);
}
Function f = new Function { int apply (x : int) { return x+1; }};
f.apply (0);
```

It turns out that first-class functions are reasonably straightforward to model in a big-step semantics. However, the original framework for first-class function, the lambda calculus, has a small-step (reduction) semantics. Also, the denotational model of a language with first-class functions requires further investigation of domain theory.

Hence, the rest of this section contains a big-step description of a language with first-class functions and the following sections will establish the foundations for the other semantic frameworks.

The evaluation judgement still has the form $\rho \vdash e \hookrightarrow y$ but the definitions of the components change.

$$\begin{array}{rcl}
y \in & \mathsf{Val} & = \mathbf{Z} + \mathsf{Closure} \\
\langle \rho, v, e \rangle \in & \mathsf{Closure} & = \mathsf{Env} \times \mathsf{Var} \times \mathsf{Exp} \\
\rho \in & \mathsf{Env} & = \mathsf{Var} \hookrightarrow \mathsf{Val}
\end{array}$$

There are three main changes.

1. Each element $\langle \rho, v, e \rangle$ of $\mathsf{Closure}$ models a *function closure* where v is the formal variable, e is the function's body, and ρ is an environment that determines the values of the free variables in e.

2. Since functions are now expressed values, a value may be a closure.

3. Consequently, the definitions of Val, $\mathsf{Closure}$, and Env are mutually recursive.

$$\textit{(var)}\ \frac{\rho(v) = y}{\rho \vdash v \hookrightarrow y}$$

$$\textit{(lam)}\ \frac{}{\rho \vdash \lambda v \,.\, e \hookrightarrow \langle \rho, v, e \rangle}$$

$$\textit{(abs)}\ \frac{\rho \vdash e_1 \hookrightarrow \langle \rho', v', e' \rangle \qquad \rho \vdash e_2 \hookrightarrow y_2 \qquad \rho'[v' \mapsto y_2] \vdash e' \hookrightarrow y}{\rho \vdash e_1\ e_2 \hookrightarrow y}$$

The rule *(var)* for variables is as before.

The rule *(lam)* for lambda abstraction grabs the environment (where the function is defined!) and wraps it with the name of the formal parameter and the function's body in a closure.

The rule *(app)* for function application first attempts to evaluate the function part (sometimes called *rator*), e_1, of the expression to a closure. Then it evaluates the argument part (sometimes called *rand*), e_2 of the expression. Finally, it evaluates the body of the closure in the environment *where the closure was defined* extended with the bindung of the actual parameter y_2 to the formal parameter v'. The resulting value is the value of the function application.

The rules model call-by-value evaluation and static scoping because the free variables in a function always refer to the value in the environment where the function was created. This is implemented by the closure.

Example:

$$\frac{\dfrac{}{\emptyset \vdash \lambda \mathtt{x} \,.\, \mathtt{x} + \mathtt{4} \hookrightarrow \langle \emptyset, \mathtt{x}, \mathtt{x} + \mathtt{4} \rangle} \qquad \dfrac{}{\emptyset \vdash \mathtt{17} \hookrightarrow \mathtt{17}} \qquad \dfrac{\dots}{\{\mathtt{x} \mapsto \mathtt{17}\} \vdash \mathtt{x} + \mathtt{4} \hookrightarrow \mathtt{21}}}{\emptyset \vdash (\lambda \mathtt{x} \,.\, \mathtt{x} + \mathtt{4})\ \mathtt{17} \hookrightarrow \mathtt{21}}$$

Exercise:

1. Change the big-step semantics to model call-by-name (by introducing suspensions).

2. Change the big-step semantics to model dynamic scope.

9 The Lambda Calculus

The Lambda Calculus will serve us (at least) two purposes.

- It provides the small-step semantics for first-class functions.
- It provides notation for the metalanguage of denotational semantics.

9.1 Syntax and reduction semantics

The lambda calculus is a logical reduction calculus: It consists of a language for terms and a set of reduction rules which describe how to transform terms into other terms.

Definition 32 (Syntax of the lambda calculus) Let Var be a countable set of *variables*. The following grammar defines the set Exp of *lambda terms*.

$$e \quad ::= \quad v \mid \lambda v\,.\,e \mid e\ e$$

Terms of the form $e_0\ e_1$ are *applications*, terms of the form $\lambda v\,.\,e$ are *abstractions* with *body* e.

To save on parentheses, the following conventions apply to the representation of lambda calculus terms:

- Applications are left-associative.
- The body of an abstraction reaches as far to the right as possible.
- $\lambda xy\,.\,e$ stands for $\lambda x\,.\,\lambda y\,.\,e$ (analogously for more arguments).

Intuitively, the objects of the lambda calculus are functions: An abstraction denotes a function, an application an—application. However, there are different methods for evaluating terms containing functions: first inner terms, then outer, or vice versa, left-to-right, or right-to-left. The lambda calculus, in its original form, has only conversion rules that define a notion of equality between terms. Imposing a direction on the conversion rules turns them into reduction rules and suitable restrictions on where a reduction rule applies will describe such strategies. To properly understand the implications of committing to a particular strategy, it is necessary to first examine the general theory, however.

A description of the meaning of lambda terms requires some auxiliary definitions.

Definition 33 (Free and bound variables) The functions $FV(), BV() : \mathsf{Exp} \to$

$\mathcal{P}(\mathsf{Var})$ return the set of *free* or *bound* variables of a lambda term, respectively.

$$\begin{array}{rcl} FV(x) & := & \{x\} \\ FV(e_0\ e_1) & := & FV(e_0) \cup FV(e_1) \\ FV(\lambda x\,.\,e) & := & FV(e) \setminus \{x\} \end{array}$$

$$\begin{array}{rcl} BV(x) & := & \emptyset \\ BV(e_0\ e_1) & := & BV(e_0) \cup BV(e_1) \\ BV(\lambda x\,.\,e) & := & BV(e) \cup \{x\} \end{array}$$

Furthermore, $Var(e) := FV(e) \cup BV(e)$ is the *set of variables* of e. A lambda term e is *closed* (e is a *combinator*) iff $FV(e) = \emptyset$.

Definition 34 (Syntactic Substitution) For $e, f \in E$, $e[x' \mapsto f]$ is inductively defined by:

$$\begin{array}{ll} x[x' \mapsto f] & = \begin{cases} f & \text{if } x = x' \\ x & \text{if } x \neq x' \end{cases} \\ (\lambda x\,.\,e)[x' \mapsto f] & = \begin{cases} \lambda x\,.\,e & \text{if } x = x' \\ \lambda x''\,.\,(e[x \mapsto x''][x' \mapsto f]) & \text{if } x \neq x', x'' \notin FV(e) \cup FV(f) \cup \{x'\} \end{cases} \\ (e_0\ e_1)[x' \mapsto f] & = (e_0[x' \mapsto f])\ (e_1[x' \mapsto f]) \end{array}$$

Definition 35 (Reduction rules) There are three different notions of reduction for the lambda calculus: α reduction, β reduction, and η reduction. Each is a binary relation on lambda terms.

$$\begin{array}{lll} \lambda x.e & \rightarrow_\alpha \lambda y.e[x \mapsto y] & y \notin FV(e) \\ (\lambda x.e)\ f & \rightarrow_\beta e[x \mapsto f] & \\ (\lambda x.e\ x) & \rightarrow_\eta e & x \notin FV(e) \end{array}$$

Every term matching the left side of a reduction rule is a *redex*.

Each notion of reduction is *compatibly extended* to work in all contexts. That is,

$$\frac{e \rightarrow_x e'}{\lambda y.e \rightarrow_x \lambda y.e'} \qquad \frac{e_0 \rightarrow_x e_0'}{(e_0\ e_1) \rightarrow_x (e_0'\ e_1)} \qquad \frac{e_1 \rightarrow_x e_1'}{(e_0\ e_1) \rightarrow_x (e_0\ e_1')}$$

For $x \in \{\alpha, \beta, \gamma\}$, $\overset{*}{\rightarrow}_x$ is the reflexive-transitive closure, and $\leftrightarrow_x$ is its symmetric closure, and $\overset{*}{\leftrightarrow}_x$ is its reflexive-transitive-symmetric closure.

A β-reduction step corresponds closely to the intuitive notion of function application.

Lambda terms will be considered equivalent if only the names of their bound variables differ (i.e., if they are α-convertible). If variable names matter, $e \equiv e'$ indicates that e and e' are identical, including the names of bound variables.

Definition 36 (Normal form) Let e be a lambda term. A lambda term e' is a *normal form* of e iff $e \overset{*}{\rightarrow}_\beta e'$ and if there is no e'' with $e' \rightarrow_\beta e''$.

Lambda terms with equivalent (equal modulo α reduction) normal forms exhibit the same behavior. The reverse is not always true. Also, some lambda terms do not have a normal form:

$$(\lambda x.x\ x)(\lambda x.x\ x) \rightarrow_\beta (\lambda x.x\ x)(\lambda x.x\ x)$$

Theorem 4 (Church-Rosser) *The β reduction has the* Church-Rosser property*:*

$$\begin{tikzcd} e_1 \arrow[rr, leftrightarrow, "\beta^*"] \arrow[dr, "\beta^*"'] & & e_2 \arrow[dl, "\beta^*"] \\ & e' & \end{tikzcd}$$

In words: For all lambda terms e_1, e_2 with $e_1 \overset{*}{\leftrightarrow}_\beta e_2$, there is a lambda term e' with $e_1 \overset{*}{\rightarrow}_\beta e'$ and $e_2 \overset{*}{\rightarrow}_\beta e'$.

Corollary 1 (Uniqueness of Normal Form) *A lambda term e has at most one normal form modulo α reduction.*

9.2 Programming in the lambda calculus

The lambda calculus may at first seem a fairly silly way to go about describing functions: In the world of the lambda calculus, there is nothing *but* functions, and the scarcity of its language seems to allow for only the most primitive computations (if any). Nevertheless, the lambda calculus has the same computational power as any programming language. (The theoreticians say it is “Turing-equivalent.”)

Adding conventional programming language constructs to the lambda calculus is somewhat tedious. As such, these constructions are not readily usable for programming language implementations. However, it is good to have a working knowledge of the necessary mechanisms if only to get some practice dealing with the calculus. In practice, the “pure” lambda calculus gives way to an “applied” lambda calculus which has the necessary built-in data types and primitive operations on them to directly perform useful computations.

What constructs are necessary for useful computations? The lambda calculus at first glance seems to lack the following fundamental ingredients:

- some sort of conditional and booleans,
- numbers, and
- recursion.

These are (almost) the elements of the theory of recursive functions. An applied lambda calculus typically has all of these, but it is possible to model all of them (and more) in the pure one. This yields an informal proof that every recursive function on natural numbers can be encoded in a lambda term. Consequently, every Turing machine can be simulated by a lambda term. To prove Turing-equivalence, as hinted above, it is now sufficient to implement β-reduction on a Turing machine.

9.2.1 Booleans and conditionals

Conditionals have the form `if` e `then` e_1 `else` e_2: Depending on the (boolean) result of evaluating e, the conditional "selects" either e_1 or e_2. The way to go in the lambda calculus is to give booleans themselves an "active" interpretation that *performs* the selection by itself. Thus, *true* is a lambda term that selects the first of two arguments, and *false* is one that selects the second:

$$\begin{aligned} \mathit{true} &= \lambda xy.x \\ \mathit{false} &= \lambda xy.y \end{aligned}$$

Consequently, the conditional degenerates to an identity function:

$$\mathit{ite} = \lambda txy.t\ x\ y$$

It is straightforward to verify that *if* actually adheres to the intuition. For a true test, the beta reduction goes like this:

$$\begin{aligned} \mathit{if}\ \mathit{true}\ e_1\ e_2 &= (\lambda txy.t\ x\ y)\ \mathit{true}\ e_1\ e_2 \\ &\to_\beta (\lambda xy.\mathit{true}\ x\ y)\ e_1\ e_2 \\ &\to_\beta^2 \mathit{true}\ e_1\ e_2 \\ &= (\lambda xy.x)\ e_1\ e_2 \\ &\to_\beta (\lambda y.e_1)\ e_2 \\ &\to_\beta e_1 \end{aligned}$$

For *false*, the proof goes analogously.

9.2.2 Numbers

Numbers can be represented in several different ways by lambda terms. One is to use *Church numerals.* The Church numeral $\lceil n \rceil$ of some natural number n is a function which takes two parameters, a function f and some x, and applies f n-times to x. (Hence, $\lceil 0 \rceil$ is the identity.)

$$\lceil n \rceil = \lambda f \lambda x.f^{(n)}(x)$$

where

$$f^{(n)}(e) = \begin{cases} e & \text{if } n = 0 \\ f(f^{(n-1)}(e)) & \text{otherwise} \end{cases}$$

The successor function adds an application:

$$succ = \lambda n.\lambda f \lambda x.n\ f\ (f\ x)$$

The predecessor is somewhat more complicated:

$$pred = \lambda x.\lambda y.\lambda z.x\ (\lambda p.\lambda q.q\ (p\ y))\ ((\lambda x.\lambda y.x)\ z)\ (\lambda x.x)$$

(A proof that it actually does subtract one from a Church numeral is a worthwhile exercise.)

Also, a test for zero is possible:

$$zero? = \lambda n.n\ (\lambda x.false)\ true$$

Again, a simple test case serves as an example:

$$\begin{aligned} zero?\ \lceil 0 \rceil &= (\lambda n.n\ (\lambda x.false)\ true)\ \lceil 0 \rceil \\ &\rightarrow_\beta \lceil 0 \rceil\ (\lambda x.false)\ true \\ &= (\lambda f.\lambda x.x)\ (\lambda x.false)\ true \\ &\rightarrow_\beta (\lambda x.x)\ true \\ &\rightarrow_\beta true \end{aligned}$$

9.2.3 Recursion

The only thing missing now is recursion. Since a recursive function needs to refer to itself, it needs to receive a name which is passed to it by a magical term called a *fixpoint combinator.* The magic is sufficient to warrant a theorem:

Theorem 5 (Fixpoint theorem) *Every lambda term has a fixpoint.*

That is, for every lambda term f there is a lambda term e with $f\ e \overset{*}{\leftrightarrow}_\beta e$.

Proof:

Choose $e := Y\ f$ with

$$Y := \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)).$$

Then:

$$\begin{aligned} Y\ F &= (\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))\ F \\ &\rightarrow_\beta (\lambda x.F\ (x\ x))\ (\lambda x.F\ (x\ x)) \\ &\rightarrow_\beta F\ ((\lambda x.F\ (x\ x))\ (\lambda x.F\ (x\ x))) \\ &\leftarrow_\beta F\ ((\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)))\ F) \\ &= F\ (Y\ F) \end{aligned}$$

A fixpoint combinator suitable for multiple recursion does not involve new principles, but is tedious to formulate.

As an example, consider expressing the recursive definition of the factorial function

$$fac\ n = if\ (zero?\ n)\ \lceil 1 \rceil\ times\ n\ (fac\ (pred\ n))$$

where *times* and *pred* are multiplication and predecessor functions. An equivalent non-recursive definition can be found using the fixpoint combinator.

$$\mathit{fac}' = Y\ (\lambda f\ n.\mathit{if}\ (\mathit{zero?}\ n)\ \lceil 1 \rceil\ \mathit{times}\ n\ (f\ (\mathit{pred}\ n)))$$

It is straightforward to show that, for all $n \in \mathbf{N}$, $\mathit{fac}\ \lceil n \rceil \overset{*}{\leftrightarrow}_\beta \mathit{fac}'\ \lceil n \rceil$.

9.2.4 Pairs

Other data structures are readily implementable, too. For example, a pair can be encoded as a function that takes a projection function and applies it to the components of the pair. Hence, the selectors take a pair and apply it to the appropriate projection function.

$$\begin{array}{ll} \mathit{pair} & = \lambda xyt.t\ x\ y \\ \mathit{fst} & = \lambda p.p\ \lambda xy.x \\ \mathit{snd} & = \lambda p.p\ \lambda xy.y \end{array}$$

9.3 Evaluation strategies

Since vanilla β reduction applies to arbitrary subterms, having normal forms is of limited value: It is not clear how to compute them because success is highly dependent on the order in which subterms are subject to reduction. In the practice of programming, full normal forms are rarely important. Instead, it is usually sufficient to evaluate lambda terms to the point where they are simple values or abstractions; it is not necessary to evaluate anything "inside the lambda." This leads to the notion of *weak head-normal forms*:

Definition 37 (Weak head-normal form) A lambda term which is an abstraction is called a *value* or a *weak head-normal form.* All other lambda terms are called *expression juxtapositions.*

Next, it is desirable to formulate deterministic strategies that prescribe how to evaluate a λ term to its weak head-normal form, so-called *evaluation strategies.* A succinct formalism for describing such strategies are evaluation contexts as introduced before.

As usual, for each notion of evaluation, a suitable set of lambda terms must be identified as answers or values, which qualify as results of an evaluation. To make sense, values should not evaluate further (they should be in WHNF). In addition, evaluation is only defined for closed lambda terms. Evaluation contexts must be defined such that every lambda term e falls in one of the following categories.

- e is a value;
- e can be uniquely written as $E[r]$ where E is an evaluation context and r is a redex;
- e is stuck (it cannot be reduced further).

The last case does not occur in the pure lambda calculus. In an applied calculus with built-in constants, it corresponds to a type mismatch.

Hence, we start with defining a notion of value.

$$\mathsf{Val} \ni y ::= \lambda x.e$$

It turns out that weak head-normal forms are suitable as *values* for all common evaluation strategies. All other terms (of the form $e_0\ e_1$) are *non-values.*

The two important strategies for computing weak head-normal forms are called *call-by-name* and *call-by-value*:

Definition 38 (Call-by-name lambda calculus) Call-by-name evaluation contexts are defined by

$$E_n ::= [\,] \mid E_n\ e.$$

The *call-by-name one-step evaluation relation* $\rightarrow_{\beta n}$ is defined by:

$$E_n[(\lambda x.e)\ f] \rightarrow_{\beta n} E_n[e[x \mapsto f]].$$

The call-by-name evaluation function is a partial function $\mathit{eval}_n : \mathsf{Exp} \rightarrow \mathsf{Exp}$ where $\mathit{eval}_n(e) = y$ iff there exists a value y such that $e \rightarrow^*_{\beta n} y$.

Unfortunately, the call-by-name lambda calculus is of limited value for programming language implementation—it may evaluate a subexpression multiple times. Real-world programming languages either use a refinement of the call-by-name strategy to avoid multiple evaluation (lazy evaluation) or use a call-by-value strategy which evaluates arguments to lambda abstractions before β-reducing them.

Definition 39 (Call-by-value lambda calculus) Call-by-value evaluation contexts are defined by

$$E_v ::= [\,] \mid (E_v\ e) \mid (y\ E_v).$$

The *call-by-value one-step evaluation relation* $\rightarrow_{\beta v}$ on lambda terms is defined by

$$E_v[(\lambda x.e)\ y] \rightarrow_{\beta v} E_v[e[x \mapsto y]].$$

The call-by-value evaluation function is a partial function $\mathit{eval}_v : \mathsf{Exp} \rightarrow \mathsf{Exp}$ where $\mathit{eval}_v(e) = y$ iff there exists a value y such that $e \rightarrow^*_{\beta v} y$.

In this definition, β-reduction is restricted to a β_v-reduction where the argument position in the redex is already a value.

None of the above strategies is guaranteed to compute a normal form for the original unrestricted notion of β-reduction. Therefore, theoretician consider less restrictive strategies, normal-order reduction and applicative-order reduction, which have different properties.

Normal-order reduction always finds the normal form of a lambda term if it has one. Intuitively, it corresponds to always choosing the leftmost-outermost β-redex. To specify it, the final result of such an evaluation must be described.

Hence, answers are now normal forms of lambda terms. The following grammar generates the set Nf of normal forms. They are ranged over by n.

$$\begin{array}{lcl} n & ::= & m \mid \lambda v\,.\,n \\ m & ::= & v \mid m\ n \end{array}$$

Exercise: show that Nf is exactly the set of normal forms in the pure lambda calculus.

Definition 40 (Normal-order reduction) Leftmost-outermost evaluation contexts are defined by

$$\begin{array}{lcl} E_o & ::= & [\,] \mid E'_o\ e \mid \lambda x.E_o \\ E'_o & ::= & [\,] \mid E'_o\ e \end{array}$$

The *normal-order reduction relation* $\rightarrow_{\beta o}$ (also called *leftmost-outermost reduction* or *standard reduction*) is defined by:

$$E_o[(\lambda x.e)\ f] \rightarrow_{\beta o} E_o[e[x \mapsto f]].$$

Standard reduction is an important construction because it finds a normal form whenever one exists.

Theorem 6 (Standardization) *If e has normal form n, then $e \rightarrow^*_{\beta o} n$.*

There is a similar generalization for call-by-value evaluation.

Definition 41 (Applicative-order reduction) Leftmost-innermost evaluation contexts are defined by

$$E_i ::= [\,] \mid (E_i\ e) \mid (n\ E_i) \mid \lambda x.E_i.$$

The *applicative-order reduction relation* $\rightarrow_{\beta i}$ (*leftmost-innermost reduction*) on lambda terms is defined by

$$E_i[(\lambda x.n)\ n'] \rightarrow_{\beta i} E_i[n[x \mapsto n']].$$

9.4 Applied Lambda Calculus

Neither Church numerals nor the fixpoint combinator are particularly efficient ways of implementing realistic programs. Therefore, real programming languages incorporate the lambda calculus in some applied form which already contains essential primitive data types and their operations as well as recursion. In the simplest form, the values and operations of these data types are supplied as constants with special reduction rules, called δ reductions.

Usually, each constant $c^{(a)}$ has an arity, a, and its δ reduction is defined by a partial function $\delta_c : \mathsf{Val}^a \rightarrow \mathsf{Val}$ which maps an a-tuple of terms in WHNF to a term in WHNF. The generic definition of δ reduction is thus

$$c^{(a)}\ v_1 \dots v_a \rightarrow_\delta v \quad \text{if} \quad ((v_1, \dots, v_a), v) \in \delta_c$$

For this to make sense, the definition of WHNF (Val, respectively) needs to extended:

$$\mathsf{Val} \ni y ::= \lambda v \,.\, e \mid c^{(a)}\ y_1 \dots y_k \text{ where } 1 \leq k < a$$

For instance, an applied lambda calculus with integers and addition would have a nullary constant $\lceil n \rceil$ for each integer n as well as a constant $+^{(2)}$ with δ_+ defined by:

$$\delta_+(\lceil m \rceil, \lceil n \rceil) = \lceil m + n \rceil$$

Hence, the set of values is extended as follows:

$$y \quad ::= \quad \cdots \mid \lceil n \rceil \mid + \mid +\ y.$$

9.5 Execution Errors and Typing Disciplines

In an applied lambda calculus, there are usually terms which cannot be evaluated further although they are not in weak head-normal form. These terms are called *stuck terms.* They are regarded as execution errors because they amount to misinterpretation of data. Here are some examples.

$\lceil 5 \rceil\ v$	number used as a function, arity mismatch
$+\ (\lambda x.e)\ v$	operand out of domain
$\mathit{if}\ (\lambda x.e)\ \mathit{then}\ e_1\ \mathit{else}\ e_2$	type mismatch
$\mathit{if}\ \lceil 42 \rceil\ \mathit{then}\ e_1\ \mathit{else}\ e_2$	type mismatch

Programming languages take one of two stances. Either they expect the compiler to generate code that tests all operands before it executes an operation. Or they impose a typing discipline that rules out some or all programs that may lead to execution errors. The first case is often called *dynamic typing* or *dynamic checking* and it requires that every value is equipped with sufficient type information at runtime. The other case amounts to *static typing* or *static checking* and it imposes on the compiler writer the burden of implementing a type checker to enforce the typing discipline. Depending on the discipline, this task can range from straightforward through demanding to impossible (there are some typing disciplines with undecidable type checking).

Another distinction with a similar flavor is the one between *strong typing* and *weak typing.* In a strongly typed language, each value has one designated type and only operations for this particular type apply to the value. Weakly typed languages usually a notion of conversion (or *coercion*) that silently converts unsuitable operands into suitable arguments for an operation.

Both concepts are independent of each other. A language can be strongly typed with a dynamic typing discipline (*e.g.*, Scheme) or it can be weakly typed with a static typing discipline (old versions of the C language, PL/1). In many cases, however, the combinations are either strong, static typing (Haskell, ML) or weak, dynamic typing (JavaScript). Java is a special case because a strong, static type discipline is meant to imply that no type mismatches can occur at runtime. However, this is not true in Java due to the presence (and wide use) of type casts in the language.

9.6 Denotational Semantics

Now is the time to look at a denotational semantics for an applied lambda calculus. Our constants will be the familiar ones, integer constants and binary addition, and we'll include the conditional, too.

Due to the presence of first-class functions, we are once again in the nice situation that the denoted values coincide with the expressed values. However, the corresponding domain turns out to be nontrivial to construct.

Clearly, the domain must comprise $\mathbf{Z}_\perp$ for modeling numbers and $\mathbf{B}_\perp$ for modeling truth values. In addition, the semantics should be able to tell these two subsets apart, hence it must contain a sum of $\mathbf{Z}_\perp$ and $\mathbf{B}_\perp$. But which one? The separated sum would enable us to distinguish an undefined number from an undefined boolean, whereas the coalesced sum would only allow this distinction on defined values. Currently, the calculus does not contain any operators to make this distinction, but we would like the semantics to report errors by returning by returning a dynamic "type error message" (instead of mapping them to $\perp$). This motivates the use of the separated sum for the call-by-name case whereas call-by-value semantics require the use of the coalesced sum. Hence:

$$\begin{array}{rcl} \mathsf{Val}_n &=& \mathbf{Z}_\perp + \mathbf{B}_\perp + \ldots \\ \mathsf{Val}_v &=& \mathbf{Z}_\perp \oplus \mathbf{B}_\perp \oplus \ldots \end{array}$$

In both cases, the missing part is a summand for modeling functions. Clearly, we would like to model functions by a function space $[V \to V]$, but this has two tricky twists. The first one is that it is possible in a call-by-value language to defined a context that distinguishes between $\perp \in [V \to V]$ and $\lambda x . \perp \in [V \to V]$: the context

$$(\lambda xy . y)\ [\,]$$

becomes undefined when $\perp$ is plugged into the hole and $\lambda y . y$, otherwise. To make this distinction denotationally, require lifting the function space: $[V \to V]_\perp$. To be fully precise, we could further restrict the function space to strict functions in the call-by-value case.

The second twist is the question, what is V in this definition? Since V should range over the domain of expressed values, it should be Val_n or Val_v, respectively. That is:

$$\begin{array}{rcl} \mathsf{Val}_n &=& \mathbf{Z}_\perp + \mathbf{B}_\perp + [\mathsf{Val}_n \to \mathsf{Val}_n]_\perp + \ldots \\ \mathsf{Val}_v &=& \mathbf{Z}_\perp \oplus \mathbf{B}_\perp \oplus [\mathsf{Val}_v \to \mathsf{Val}_v]_\perp \oplus \ldots \end{array}$$

These definitions contain a novelty, namely that a semantic domain is defined recursively, it is specified by a *recursive domain equation.* It turns out that domains can be specified as solutions of such equations but it is non-trivial to establish the existence of these solutions. We'll take them as granted for the moment and leave that topic to later investigation. In fact, all equations that start from flat domains and use the domain constructions introduced so far have solutions. One particular way of demostrating that is to show that there exists a *universal domain*, that is, a CCPO that contains all flat CCPOs as sub-CCPOs, where the class of sub-CCPOs is closed under the domain constructions, and where certain functions on the class of sub-CCPOs have fixpoints.

One last ingredient is missing in the domain cocktail, the treatment of errors. For that, we'll construct a one-point domain $\mathbf{W}$ with $\mathbf{W} = \{\mathit{wrong}\}$ and add it to each domain equation.

$$\begin{array}{lcl} \mathsf{Val}_n & = & \mathbf{Z}_\bot + \mathbf{B}_\bot + [\mathsf{Val}_n \to \mathsf{Val}_n]_\bot + \mathbf{W} \\ \mathsf{Val}_v & = & \mathbf{Z}_\bot \oplus \mathbf{B}_\bot \oplus [\mathsf{Val}_v \to \mathsf{Val}_v]_\bot \oplus \mathbf{W} \end{array}$$

For better readability, instead of writing In_1, In_2, etc for the injection functions into Val_x, we use symbolic tags InZ, InB, InF, and InW with the obvious meaning.

The type of the semantic function for expressions is now, for $x \in \{n, v\}$,

$$\begin{array}{lcl} \mathcal{E} & : & \mathsf{Exp} \to \mathsf{Env}_x \to \mathsf{Val}_x \\ \mathsf{Env}_x & = & \mathsf{Var} \to \mathsf{Val}_x \end{array}$$

Since the machinery for modeling execution errors is now available, the empty environment is the function $\rho_0 = \lambda v\,.\,\mathit{InW}(\mathit{wrong})$ which maps every variable to an error.

Here are the semantic equations for the call-by-name case.

$$\begin{array}{lcl}
\mathcal{E}[\![n]\!] & = & \lambda\rho\,.\,\mathit{InZ}(\mathcal{C}[\![n]\!]) \\
\mathcal{E}[\![e_1 + e_2]\!] & = & \lambda\rho\,.\ \mathit{case}\ (\mathcal{E}[\![e_1]\!]\rho)\ \mathit{of} \\
& & \qquad \mathit{InZ}\ y_1 \to \quad \mathit{case}\ (\mathcal{E}[\![e_2]\!]\rho)\ \mathit{of} \\
& & \qquad\qquad\qquad \mathit{InZ}\ y_2 \to \mathit{InZ}(y_1 + y_2) \\
& & \qquad\qquad\qquad \_ \to \mathit{InW}(\mathit{wrong}) \\
& & \qquad \_ \to \mathit{InW}(\mathit{wrong}) \\
\mathcal{E}[\![\mathtt{if}\ e_1\ e_2\ e_3]\!] & = & \lambda\rho\,.\ \mathit{case}\ (\mathcal{E}[\![e_1]\!]\rho)\ \mathit{of} \\
& & \qquad \mathit{InB}\ y \to \mathit{ite}(y, \mathcal{E}[\![e_2]\!]\rho, \mathcal{E}[\![e_3]\!]\rho) \\
& & \qquad \_ \to \mathit{InW}(\mathit{wrong}) \\
\mathcal{E}[\![v]\!] & = & \lambda\rho\,.\,\rho(v) \\
\mathcal{E}[\![\lambda v\,.\,e]\!] & = & \lambda\rho\,.\,\mathit{InF}(\lambda y\,.\,\mathcal{E}[\![e]\!]\rho[v \mapsto y]) \\
\mathcal{E}[\![e_1\ e_2]\!] & = & \lambda\rho\,.\ \mathit{case}\ (\mathcal{E}[\![e_1]\!]\rho)\ \mathit{of} \\
& & \qquad \mathit{InF}(y) \to y(\mathcal{E}[\![e_2]\!]\rho) \\
& & \qquad \_ \to \mathit{InW}(\mathit{wrong})
\end{array}$$

For obtaining a call-by-value semantics, all the injections and cases need to be annotated with $^\oplus$, but only one line needs to change:

$$\mathcal{E}[\![\lambda v\,.\,e]\!] = \lambda\rho\,.\,\mathit{InF}(\lambda y\,.\,\mathit{ite}(y = \bot, \bot, \mathcal{E}[\![e]\!]\rho[v \mapsto y]))$$

10 Simple Types

The applied lambda calculus shares a feature with many programming languages. It has a built-in concept of a type and refuses to evaluate programs with a type conflict (as demonstrated in Section 9.5).

This problem stipulates the design of an analysis that infers from a given program whether it can possibly lead to a type conflict. Such an analysis relies on a formal system, a *type system.* The present section introduces the simplest of these systems, the simply typed lambda calculus. There are two different flavors to that calculus. In Church-style, each binding occurrence of a variable is labeled with the intended type of the variable. In Curry-style, typing is completely separate from the terms. Our presentation adheres to Curry-style.

Background reading: John C. Mitchell, chapter 4 of *Foundations for Programming Languages*, MIT-Press, Cambridge, MA, 1996.

Type systems are customarily defined using deduction systems. The judgment of the system, the *typing judgment*, relates a typing environment, Γ, and an expression, e, to a type, τ:

$$\Gamma \vdash e : \tau$$

The types of the simply typed lambda calculus are generated by the grammar

$$\mathsf{Type} \ni \tau \quad ::= \quad \alpha \mid \mathtt{Int} \mid \tau \to \tau$$

where α is a type variable, drawn from a set TVar, $\mathtt{Int}$ is a type constant, and $\tau' \to \tau''$ is the type of functions that map values of type τ' to values of type τ''. (For the pure lambda calculus, type variables and function types suffice.) Remember: all this is only syntax!

A typing environment acts like a finite map from Var to Type. It is generated by

$$\Gamma \quad ::= \quad \emptyset \mid \Gamma, v : \tau$$

Often, the case $\Gamma, v : \tau$ is restricted so that variable v must not occur in Γ. The following definition enables us to use a typing environment as a finite map and to write $dom(\Gamma)$ for the set of variables defined in Γ.

$$(\Gamma, v : \tau)(v') = \begin{cases} \tau & v = v' \\ \Gamma(v') & v \neq v' \end{cases}$$

The typing judgment is defined by the following inference rules.

$$\frac{}{\Gamma \vdash n : \texttt{Int}}$$

$$\frac{\Gamma \vdash e : \texttt{Int} \qquad \Gamma \vdash e' : \texttt{Int}}{\Gamma \vdash e{+}e' : \texttt{Int}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Int} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ e_2\ e_3 : \tau}$$

$$\frac{\Gamma(v) = \tau}{\Gamma \vdash v : \tau}$$

$$\frac{\Gamma, v : \tau' \vdash e : \tau''}{\Gamma \vdash \lambda v\,.\,e : \tau' \to \tau''}$$

$$\frac{\Gamma \vdash e : \tau' \to \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\ e' : \tau}$$

Example: type derivation for $y : \tau \vdash (\lambda x\,.\,x)\ y : \tau$.

10.1 Static Properties

Type derivations are quite well-behaved and enjoy a number of properties. First of all, the typing environment must contain typings for all free variables.

Lemma 23 *If $\Gamma \vdash e : \tau$, then $FV(e) \subseteq dom(\Gamma)$.*

Next, type assumptions for variables not appearing in the expression may be omitted. This is called *weakening.*

Lemma 24 *If $\Gamma, v : \tau \vdash e : \tau'$ and $v \notin FV(e)$, then $\Gamma \vdash e : \tau'$.*

Renaming extends in the obvious way to typing environments. Consistent renaming is also compatible with typing.

Lemma 25 *If $\Gamma \vdash e : \tau$ and $y \notin dom(\Gamma)$, then $\Gamma[x \mapsto y] \vdash e[x \mapsto y] : \tau$.*

An important property is the substitution lemma for expressions. It holds (with slight variations) for virtually all type systems.

Lemma 26 (Expression Substitution) *If $\Gamma, x' : \tau' \vdash e : \tau$ and $\Gamma' \vdash e' : \tau'$ and $\Gamma \cup \Gamma'$ is a well-formed typing environment, then $\Gamma \cup \Gamma' \vdash e[x' \mapsto e'] : \tau$.*

10.2 Type Inference

Quite often, we are not interested in writing down all typings but rather in having an algorithm that infers a valid typing for a term. Ideally, the algorithm infers the best possible typing, whatever that may be. For the simply-typed lambda calculus, it turns out that each term has a "best possible" type, which is called a principal type. This has been shown independently by Hindley and Milner.

To determine a principal type requires an ordering relation on types. This relation is defined in terms of substitution.

Definition 42 A term t' is an *instance* of term t, if there is a substitution σ'' such that $t' = \sigma''(t)$. Notation: $t \sqsubseteq t'$.

The relation $\sqsubseteq$ is a preorder (reflexive and transitive). It can be lifted to a preorder on substitutions. (Exercise: prove this and give an example that $\sqsubseteq$ is not a partial ordering.)

Definition 43 Let σ, σ' be substitutions. Define $\sigma \sqsubseteq \sigma'$ if there exists a substitution σ'' such that $\sigma' = \sigma'' \circ \sigma$.

The relations on types and substitutions need to be further extended, once to pairs of a type and a substitution on type variables, (τ, σ), and to entire typing judgements.

Definition 44 $(\tau, \sigma) \sqsubseteq (\tau', \sigma')$ iff exists σ'' such that $\tau' = \sigma''(\tau)$ and $\sigma' = \sigma'' \circ \sigma$.

Definition 45 $\Gamma \vdash e : \tau \sqsubseteq \Gamma' \vdash e : \tau'$ iff $\mathit{dom}(\Gamma) \subseteq \mathit{dom}(\Gamma')$ and there exists σ'' such that $(\forall v \in \mathit{dom}(\Gamma))\ \Gamma'(v) = \sigma''(\Gamma(v))$ and $\tau' = \sigma''(\tau)$.

Given a type derivation for judgement $\mathcal{J}$, it turns out to be trivial to generate arbitrary many derivable judgements $\mathcal{J}'$ with $\mathcal{J} \sqsubseteq \mathcal{J}'$.

Lemma 27 (Type Substitution) *Suppose that $\Gamma \vdash e : \tau$ and let σ be an arbitrary substitution. Then $\sigma(\Gamma) \vdash e : \sigma(\tau)$.*

Definition 46 Suppose that Γ is a type environment and e is a term (with $FV(e) \subseteq \mathit{dom}(\Gamma)$).

A *solution* for the type inference problem for Γ and e is a pair (σ, τ), where σ is a type substitution (from type variables to types) and $\sigma(\Gamma) \vdash e : \tau$.

(σ, τ) is a *principal solution* if, for all solutions (σ', τ'), $(\sigma, \tau) \sqsubseteq (\sigma', \tau')$. In this case, τ is a *principal type* for e.

Another related notion also considers the type environment as an unknown.

Definition 47 The typing judgment $\mathcal{J} = \Gamma \vdash e : \tau$ is a *principal typing for e*, if for each typing judgment $\mathcal{J}' = \Gamma' \vdash e : \tau'$ it holds that $\mathcal{J} \sqsubseteq \mathcal{J}'$.

$$
\begin{array}{lcl}
\mathit{unify}(\emptyset) & = & \{\,\} \\
\mathit{unify}(\mathcal{E} \cup \{\alpha = \tau\}) & = & \mathit{if}\ \alpha \in \mathit{Var}(\tau) \wedge \tau \neq \alpha\ \mathit{then}\ \mathit{fail} \\
 & & \mathit{else}\ \mathit{unify}(\{\alpha \mapsto \tau\}(\mathcal{E})) \circ \{\alpha \mapsto \tau\} \\
\mathit{unify}(\mathcal{E} \cup \{\alpha = \alpha\}) & = & \mathit{unify}(\mathcal{E}) \\
\mathit{unify}(\mathcal{E} \cup \{\mathtt{Int} = \mathtt{Int}\}) & = & \mathit{unify}(\mathcal{E}) \\
\mathit{unify}(\mathcal{E} \cup \{\tau_1 \to \tau_2 = \tau_1' \to \tau_2'\}) & = & \mathit{unify}(\mathcal{E} \cup \{\tau_1 = \tau_1', \tau_2 = \tau_2'\}) \\
\mathit{unify}(\mathcal{E} \cup \{\tau_1 \to \tau_2 = \mathtt{Int}\}) & = & \mathit{fail}
\end{array}
$$

In the algorithm, we consider $=$ as symmetric.

Figure 1: Unification algorithm for types

The construction of a type inference algorithm requires the types computed for the subterms of, say, an application $e_1\ e_2$, have a certain shape. For instance, if the algorithm has found type τ_1 for e_1 and τ_2 for e_2, then τ_1 must have the form $\tau_2 \to \tau_3$, for some τ_3, for the typing to succeed. Phrased differently, the algorithm needs to find a substitution S, such that $S(\tau_1) = S(\tau_2 \to \alpha)$ where α is a new type variable. Fortunately, there is an algorithm that finds "the best" such substitution or fails if none exists.

Definition 48 Let $\mathcal{E}$ be a set of identities between terms.
A substitution σ is a *unifier* of $\mathcal{E}$ if, for all identities $(t, t') \in \mathcal{E}$, $\sigma(t) = \sigma(t')$.

Lemma 28 (Robinson 1965) *Let $\mathcal{E}$ be any set of identities between terms. There is an algorithm unify such that*

- *if σ' is a unifier for $\mathcal{E}$, then $\sigma = \mathit{unify}(\mathcal{E})$ and $\sigma \sqsubseteq \sigma'$,*
- *if no unifier exists for $\mathcal{E}$, then $\mathit{unify}(\mathcal{E})$ fails.*

Figure 1 defines the unification algorithm.

The algorithm in Figure 2 computes a typing for a term of the simply-typed lambda calculus. It maps a lambda term to a typing judgement for the term.

Each type inference algorithm comes with two technical results.

- The correctness property states that type inference only produces derivable types.
- The completeness produces states that, whenever a type derivation exists for the term, then the algorithm will find a corresponding, more general judgement. In other words, the algorithm finds a principal type (or typing).

Theorem 7 (Correctness of Type Inference) *Suppose that* $\mathrm{PT}(e) = \Gamma \vdash e' : \tau$. *Then $e = e'$ and $\Gamma \vdash e : \tau$ is derivable.*

```
$PT(v)$          $=$  $v : \alpha \vdash v : \alpha$      $\alpha$ fresh
$PT(\lambda v\,.\,e)$   $=$  let     $\Gamma \vdash e' : \tau \;=\; PT(e)$
                     in if   $v \notin dom(\Gamma)$
                     then    $\Gamma \vdash \lambda v\,.\,e' : \alpha \to \tau$      $\alpha$ fresh
                     else    $\Gamma \setminus \{v\} \vdash \lambda v\,.\,e' : \Gamma(v) \to \tau$
$PT(e_1\ e_2)$      $=$  let   $\Gamma_1 \vdash e_1' : \tau_1 \;=\; PT(e_1)$
                           $\Gamma_2 \vdash e_2' : \tau_2 \;=\; PT(e_2)$
                           $Var(\Gamma_i, \tau_i)$ pairwise disjoint
                           $\sigma = unify(\Gamma_1, \Gamma_2; \{\tau_1 = \tau_2 \to \alpha\})$      $\alpha$ fresh
                     in    $\sigma(\Gamma_1) \cup \sigma(\Gamma_2) \vdash e_1'\ e_2' : \sigma(\alpha)$
$PT(n)$          $=$  $\emptyset \vdash n : \mathtt{Int}$
$PT(e_1{+}e_2)$     $=$  let   $\Gamma_1 \vdash e_1' : \tau_1 \;=\; PT(e_1)$
                           $\Gamma_2 \vdash e_2' : \tau_2 \;=\; PT(e_2)$
                           $Var(\Gamma_i, \tau_i)$ pairwise disjoint
                           $\sigma = unify(\Gamma_1, \Gamma_2; \{\tau_1 = \mathtt{Int}, \tau_2 = \mathtt{Int}\})$
                     in    $\sigma(\Gamma_1) \cup \sigma(\Gamma_2) \vdash e_1'{+}e_2' : \mathtt{Int}$
$PT(\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3)$
                 $=$  let   $\Gamma_1 \vdash e_1' : \tau_1 \;=\; PT(e_1)$
                           $\Gamma_2 \vdash e_2' : \tau_2 \;=\; PT(e_2)$
                           $\Gamma_3 \vdash e_3' : \tau_3 \;=\; PT(e_3)$
                           $Var(\Gamma_i, \tau_i)$ pairwise disjoint
                           $\sigma = unify(\Gamma_1, \Gamma_2, \Gamma_3; \{\tau_1 = \mathtt{Int}, \tau_2 = \tau_3\})$
                     in    $\sigma(\Gamma_1) \cup \sigma(\Gamma_2) \cup \sigma(\Gamma_3) \vdash \mathtt{if}\ e_1'\ \mathtt{then}\ e_2'\ \mathtt{else}\ e_3' : \sigma(\tau_2)$
```

In the algorithm, $unify(\Gamma_1, \Gamma_2)$ is a shorthand for $unify(\{\Gamma_1(x) = \Gamma_2(x) \mid x \in dom(\Gamma_1) \cap dom(\Gamma_2)\})$, and similarly $unify(\Gamma_1, \Gamma_2, \Gamma_3)$ collects identities for all pairs of type environments.

Figure 2: Algorithm for principal typing

Theorem 8 (Completeness of Type Inference) *Suppose that* $\mathcal{J}' = \Gamma' \vdash e : \tau'$ *is derivable, for some* Γ' *and* τ'*.*

Then there exist Γ *and* τ *such that* $\mathrm{PT}(e) = \Gamma \vdash e : \tau \sqsubseteq \mathcal{J}'$*.*

Hence, the simply-typed lambda calculus possesses principal typings and the *PT* algorithm computes them. This is a an important, non-trivial property.

10.3 Dynamic Properties

The dynamic properties relate the typing judgement to one of the semantics. Since the type system is supposed to predict to some extent the outcome of a computation, each computation step should be compatible with the typing relation. In the context of the reduction semantics of the lambda calculus, the required property is called *subject reduction*: whenever a typed term can make a computation step, then the resulting term is typable with the same type.

Lemma 29 (Subject Reduction) *If* $\Gamma \vdash e : \tau$ *and* $e \to e'$*, then* $\Gamma \vdash e' : \tau$*.*

In this lemma, the relation $\to$ is one-step $\beta\eta$ reduction plus the delta rules for addition and the conditional, all permitted in an arbitrary context.

This is easily generalized to sequences of computations steps.

Lemma 30 *If* $\Gamma \vdash e : \tau$ *and* $e \overset{*}{\to} e'$*, then* $\Gamma \vdash e' : \tau$*.*

Subject reduction seems to be closely connected to type correctness since it implies that computation does not change the type of a term. However, it does not make guarantees that terms do not get stuck, eventually, before being reduced to values. This requires another statement which is customarily called *progress.*

Lemma 31 (Progress) *If* $\Gamma \vdash e : \tau$*, then exactly one of the following holds.*

1. *e is a value.*
2. *There is some* e' *such that* $e \to e'$*.*

A suitable set of values for our simply typed lambda calculus is

$$y \quad ::= \quad n \mid \lambda v\,.\,e$$

Proving the progress lemma requires another straightforward auxiliary result.

Lemma 32 (Classification) *Suppose that* $\Gamma \vdash y : \tau$*.*

- *If* $\tau = \texttt{Int}$*, then* $y = n$*, for some* n*.*
- *If* $\tau = \tau' \to \tau''$*, then* $y = \lambda v\,.\,e$*, for some* v *and* e*.*

Subject reduction and progress taken together yield a *type soundness* result.

Theorem 9 (Type Soundness) *If $\emptyset \vdash e : \tau$, then exactly one of the following is true.*

- *There exists a value y such that $e \overset{*}{\to} y$ and $\emptyset \vdash y : \tau$.*
- *For each e' such that $e \overset{*}{\to} e'$ there exists e'' such that $e \to e''$.*

When starting from an evaluation-step relation such as βn in the previous section, the requirement of "subject reduction" can be weakened to "type preservation". This is just a restatment of the above, but in terms of the deterministic evaluation relation.

Lemma 33 (Type Preservation) *If $\Gamma \vdash e : \tau$ and $e \to_{\beta n} e'$, then $\Gamma \vdash e' : \tau$.*

It would also be sufficient to only state and prove type preservation for closed terms, since evaluation always starts with a closed term and this property is preserved by each evaluation step.

Exercise: Prove that closed terms remain closed under reduction.

At last, the simply-typed lambda calculus has the surprising property that every simply-typed lambda term has a normal form!

Theorem 10 (Strong Normalization) *Suppose that $\Gamma \vdash e : \tau$. Then there exists a term e' with $e \overset{*}{\to}_{\beta\eta} e'$ such that e' is in normal form.*

The symbol $\to_{\beta\eta}$ denotes reduction in arbitrary context.

10.4 Type Soundness, Denotationally

Type soundness can also be expressed in a denotational setting. To that end, recall the domain of expressed values for the call-by-name variant of the applied lambda calculus:

$$\mathsf{Val}_n \quad = \quad \mathbf{Z}_\perp + \mathbf{B}_\perp + [\mathsf{Val}_n \to \mathsf{Val}_n]_\perp + \mathbf{W}$$

where $\mathbf{W} = \{\mathit{wrong}\}$.

An easy way of stating type soundness is to postulate that

$$\emptyset \vdash e : \tau \qquad \Rightarrow \qquad \mathcal{E}[\![e]\!](\lambda y \,.\, \mathit{InW}(\mathit{wrong})) \neq \mathit{InW}(\mathit{wrong})$$

However, it turns out that this statement needs to be rephrased in a positive way to be amenable to proof. A prerequisite is to fix what exactly we mean with a type. We use an development due to MacQueen, Plotkin, and Sethi (*An Ideal Model for Recursive Polymorphic Types*, Information and Computation 71:92–130, 1986) which is richer than strictly required at this point, but which extends easily to later additions to the type system.

Definition 49 Let A be a CCPO and $I \subseteq |A|$.

I is a *(weak) ideal* of A if

1. (non-empty) $I \neq \emptyset$,
2. (downward closed) for all $x \in I$, $y \in |A|$, if $y \sqsubseteq x$ then $y \in I$,
3. (chain complete) if $X \subseteq I$ is a chain in A, then $\bigsqcup X \in I$.

Let $\mathcal{I}(A)$ be the set of ideals of A.

We aim at modeling types as particular ideals in Val_n that do not contain *wrong*. Let $\mathcal{K}(\mathsf{Val}_n)$ be the set of ideals of Val_n that do not contain *wrong*. It is straightforward to verify that the injections of $\mathbf{Z}_\perp$ and $\mathbf{B}_\perp$ in Val_n yield ideals. Functions require an auxiliary definition.

Definition 50 Let A be a CCPO and $X, Y \subseteq |A|$.
Define $X \boxed{\to} Y = \{f \in [A \to A] \mid x \in X \Rightarrow f(x) \in Y\}$.

This construction turns out to be an operation on ideals.

Lemma 34 *Let A be a CCPO and $I, J \in \mathcal{I}(A)$. Then $I \boxed{\to} J \in \mathcal{I}(A \to A)$.*

Proof: We verify the items of the definition in turn.

1. $\perp \in I \boxed{\to} J \neq \emptyset$.
2. Let $f \in I \boxed{\to} J$ and $g \in [A \to A]$ with $g \sqsubseteq f$. If $x \in I$, then $g(x) \sqsubseteq f(x)$. Since $f(x) \in J$ and J is downward closed, it holds that $g(x) \in J$, too.
3. Let $F \subseteq I \boxed{\to} J$ be a chain in $[A \to A]$. Since $[A \to A]$ is a CCPO, $\bigsqcup F \in [A \to A]$. For all $x \in I$, the set $Y_x = \{f(x) \mid f \in F\}$ is a chain in J so that $\bigsqcup Y_x \in J$. Since $(\bigsqcup F)(x) = \bigsqcup Y_x \in J$, the result follows.

These definitions enable us to define the semantics of a type as follows.

Definition 51 The function $\mathcal{T}[\![\cdot]\!] : \mathsf{Type} \to \mathcal{P}(|\mathsf{Val}_n|)$ is defined by

$$\begin{array}{lcl} \mathcal{T}[\![\mathtt{Int}]\!] & = & \{\perp\} \cup \{\mathit{InZ}(z) \mid z \in \mathbf{Z}_\perp\} \\ \mathcal{T}[\![\mathtt{Bool}]\!] & = & \{\perp\} \cup \{\mathit{InB}(b) \mid b \in \mathbf{B}_\perp\} \\ \mathcal{T}[\![\tau \to \tau']\!] & = & \{\perp\} \cup \{\mathit{InF}(f) \mid f \in \mathcal{T}[\![\tau]\!] \boxed{\to} \mathcal{T}[\![\tau']\!]\} \end{array}$$

Lemma 35 *For each $\tau \in \mathsf{Type}$, $\mathcal{T}[\![\tau]\!] \in \mathcal{K}(\mathsf{Val}_n)$.*

A value environment ρ conforms to a typing environment if all variables in ρ have values of the corresponding type.

Definition 52 Let $\rho \in \mathsf{Var} \to \mathsf{Val}_n$ and Γ a typing environment.
$\Gamma \models \rho$ holds by definition if, for all $v \in \mathit{dom}(\Gamma)$, $\rho(v) \in \mathcal{T}[\![\Gamma(v)]\!]$.

Theorem 11 (Semantic Type Soundness)
If $\Gamma \vdash e : \tau$ and $\Gamma \models \rho$, then $\mathcal{E}[\![e]\!]\rho \in \mathcal{T}[\![\tau]\!]$.

10.5 Extensions

On top of the minimal feature set considered for the simply-typed lambda calculus, a programming language typically has recursion, as well as records (products), enumerated types (sums), and perhaps recursive types.

10.5.1 Recursion

In the untyped lambda calculus, the fixpoint combinator Y provides a means to encode recursion. In the simply-typed lambda calculus, the term Y does not have a type. This is not an accident of our choice of Y (perhaps we could choose another fixpoint combinator), but rather caused by the structure of simple types. Any fixpoint combinator can easily be used to construct terms without normal form, for example, $Y(\lambda x\,.\,x)$. Hence, if Y had a type in simply-typed lambda calculus, this would contradict the strong normalization result from Lemma 10.

However, a fixpoint operator can be added to the calculus without breaking the results about types (clearly, strong normalization is lost).

$$e \quad ::= \quad \cdots \mid \texttt{fix}\, e$$

A suitable notion of reduction for the fixpoint operator is

$$\texttt{fix}\ \ e \to e\ (\texttt{fix}\ \ e)$$

(This fixpoint operator is not suitable for call-by-value evaluation, since a call-by-value reduction sequence starting with $\texttt{fix}\ \ e$ does not terminate. We leave the consideration of a call-by-value fixpoint operator as an exercise.) The typing rule for this operator is

$$\frac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash \texttt{fix}\, e : \tau}$$

Equivalently, the `fix` operator could be regarded as a constant of type $(\tau \to \tau) \to \tau$, for each τ.

The denotational semantics of $\texttt{fix}\, e$ is just the least fixpoint of the semantics of e.

$$\begin{aligned}\mathcal{E}[\![\texttt{fix}\ e]\!] = \lambda\rho.\ \ & \textit{case}\ (\mathcal{E}[\![e]\!]\rho)\ \textit{of} \\ & \textit{InF}(y) \to \textit{fix}\ y \\ & \_ \to \textit{InW}(\textit{wrong})\end{aligned}$$

10.5.2 Product and Record Types

Product types arise from adding a data type for pairs (or more generally tuples) to the language. Each component of a pair has a type of its own. Records are a simple variation of tuples where the components are named.

Here is a typical syntax for pairs.

$$e \quad ::= \quad \cdots \mid (e, e) \mid \pi_1(e) \mid \pi_2(e)$$

The new constructs may be considered as constants in the applied lambda calculus. $(,)$ is a binary constant *without reduction rules* (hence, it serves as a data constructor), whereas π_1 and π_2 are unary constants with reduction rules

$$\pi_1(e_1, e_2) \to e_1 \qquad \pi_2(e_1, e_2) \to e_2$$

In a call-by-value language, e_1 and e_2 are restricted to be values in thes reduction rules. In a call-by-name language, (e_1, e_2) *is* a value (a weak head normal form) even if neither e_1 nor e_2 is.

The type of pairs with component types τ_1 and τ_2 is written $\tau_1 \times \tau_2$. The corresponding typing rules are the obvious ones.

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_1(e) : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_2(e) : \tau_2}$$

Abstractly, an n-tuple can be considered as a mapping from $\{1, \ldots, n\}$ to the respective set of values for the ith component. A record generalizes this mapping by using names (*field labels*) instead of numbers. Apart from that difference, the reduction rules and typing rules are exactly like the ones for products. The label names are drawn from an unspecified, denumerable set Label, which is disjoint to all other syntactic categories.

$$\begin{array}{lcl} l & \in & \mathsf{Label} \\ e & ::= & \cdots \mid [l_1 = e_1, \ldots, l_n = e_n] \mid e.l \end{array}$$

The two operations are record construction and field selection. Sometimes, further record operations are considered, for example, field update, record concatenation, record restriction, etc. These require more sophisticated type systems and are left for future consideration. They play an important role in modeling object orientation.

Operationally, the reductions are similar to the ones for pairs. Record construction is a value that can be decomposed using field selection.

$$[l_1 = e_1, \ldots, l_n = e_n].l_i \to e_i$$

Again, call-by-value requires $e_1 \ldots e_n$ to be values, whereas call-by-name allows for records with unevaluated fields.

The type of a record is just the label-indexed collection of the field types as in $\tau ::= \cdots \mid [l_1 : \tau_1, \ldots, l_n : \tau_n]$. Such a type is only well-formed if the labels $l_1, \ldots, l_n$ are all distinct. The typing rules are also straighforward adaptations of the ones for products.

$$\frac{(\forall 1 \leq i \leq n)\ \Gamma \vdash e_i : \tau_i}{\Gamma \vdash [l_1 = e_1, \ldots, l_n = e_n] : [l_1 : \tau_1, \ldots, l_n : \tau_n]} \qquad \frac{\Gamma \vdash e : [l_1 : \tau_1, \ldots, l_n : \tau_n]}{\Gamma \vdash e.l_i : \tau_i}$$

Record types have a degenerate case, the nullary record type where $n = 0$. This type has just one element, the empty record $[\ ]$ of type $[\ ]$. It is often considered separately as the *unit type*.

10.5.3 Sum and Variant Types

Dually to the product construction, the sum construction accumulates a set of alternatives indexed by numbers. A variant is the named counterpart of a sum, just like a record is the named counterpart of a product.

Here is the typical syntax for sums.

$$e \quad ::= \quad \texttt{Inl}\ e \mid \texttt{Inr}\ e \mid \texttt{case}\ e\ \texttt{of}\ \texttt{Inl}\ x \to e; \texttt{Inr}\ x \to e$$

The operators `Inl` and `Inr` serve as constructors that inject values into the left or right summand. The `case` eliminates a sum by determining which summand the value belongs to and extracting the respective component value. Hence, there are no reduction rules for `Inl` and `Inr`, but two for `case` that generalize the reduction for the conditional.

$$\begin{array}{lcl}\texttt{case}\ (\texttt{Inl}\ e)\ \texttt{of}\ \texttt{Inl}\ x_1 \to e_1; \texttt{Inr}\ x_2 \to e_2 & \to & e_1[x_1 \mapsto e] \\ \texttt{case}\ (\texttt{Inr}\ e)\ \texttt{of}\ \texttt{Inl}\ x_1 \to e_1; \texttt{Inr}\ x_2 \to e_2 & \to & e_2[x_2 \mapsto e]\end{array}$$

For call-by-value, again, e is restricted to a value. In call-by-name, for example, `Inl` e is considered a value even if e is not.

The type of the sum of τ_1 and τ_2 is written $\tau_1 + \tau_2$. The typing rules are dual to the rules for products.

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{Inl}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \texttt{Inr}\ e : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \qquad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \texttt{Inl}\ x_1 \to e_1; \texttt{Inr}\ x_2 \to e_2 : \tau}$$

Variants behave very much the same way as sums. The syntax varies widely between languages.

$$e \quad ::= \quad \cdots \mid \texttt{In}_l(e) \mid \texttt{case}\ e\ \texttt{of}\ \texttt{In}_l(x) \to e; \ldots$$

A variant type is written as $[l_1 : \tau_1 \mid \cdots \mid l_n : \tau_n]$ where all labels are distinct. (There is not really much difference between a one-field record and a variant with one alternative). The typing rules are the obvious generalizations of the ones for sums.

$$\frac{\Gamma \vdash e : \tau_i}{\Gamma \vdash \texttt{In}l_i(e) : [l_1 : \tau_1 \mid \cdots \mid l_n : \tau_n]}$$

$$\frac{\Gamma \vdash e : [l_1 : \tau_1 \mid \cdots \mid l_n : \tau_n] \qquad (\forall 1 \leq i \leq n)\ \Gamma, x_i : \tau_i \vdash e_i : \tau}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \texttt{In}_{l_1}(x_1) \to e_1; \ldots; \texttt{In}_{l_n}(x_n) \to e_n : \tau}$$

10.5.4 Recursive Types

The defining characteristic of a recursive type is that it may be defined in terms of itself. Recursive types are used to model recursive data structures like lists and trees without introducing concepts like memory or pointers.

11 State

The key feature of *imperative programming languages* is state and the notion of a *statement.* In contrast to an expression that primarily serves to compute a value, a statement does not compute a value but is executed for its effect on the state. Also the mental model of a variable in such a language differs from the mathematical idea. While functional programming considers a variable just as a name for one particular value, a variable in imperative programming denotes a container whose contents, the value, may change over time.

Starting from a traditional imperative core language, we'll consider the most important kinds of statements, extended the framework to dynamic data structures, and finally consider parameter passing strategies. Background material may be found in the book *Essentials of Programming Languages* by Friedman, Wand, and Haynes.

11.1 Modeling State

The key concept in modeling state is the store. The store is always modeled as a mapping from some kind of locations to values. The choice of locations depends on the particular language modeled. The most straightforward choice is to model the store as a mapping from variable names to values. For example, in a big-step operational semantics, we might have

$$\begin{array}{rcl} \sigma \ni \mathsf{Store} & = & \mathsf{Var} \hookrightarrow \mathsf{Val} \\ y \ni \mathsf{Val} & = & \mathbf{Z} + \dots \end{array}$$

The syntax of the archetypical imperative language `IMP` contains two categories, expressions and statements. We'll stick to the most basic kind of expression, a variable, all extensions that have been discussed earlier can be applied. Statements comprise the variable declaration, the assignment statement, the empty statement, the sequence statement, if-then-else, and while statements.

$$\begin{array}{rcll} e & ::= & v \mid \dots & \\ s & ::= & \texttt{var}\ v & \text{variable declaration} \\ & \mid & v{:=}e & \text{assignment} \\ & \mid & \texttt{skip} & \text{empty statement} \\ & \mid & s;s & \text{sequence} \\ & \mid & \texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ s & \text{conditional} \\ & \mid & \texttt{while}\ e\ \texttt{do}\ s & \text{repetition} \end{array}$$

The meaning of an expression is defined by the familiar judgment

$$\sigma \vdash e \hookrightarrow y$$

that has been discussed previously.

The meaning of a statement is a transformation of the store. Hence,

$$\sigma \vdash s \leadsto \sigma'$$

says that the action of statement s transforms store σ to store σ'. It is defined by the following set of inference rules.

$$\sigma \vdash \texttt{var}\ v \rightsquigarrow \sigma[v \mapsto 0]$$

$$\frac{\sigma \vdash e \hookrightarrow y \qquad v \in \mathit{dom}(\sigma)}{\sigma \vdash v{:=}e \rightsquigarrow \sigma[v \mapsto y]}$$

$$\sigma \vdash \texttt{skip} \rightsquigarrow \sigma$$

$$\frac{\sigma_0 \vdash s_1 \rightsquigarrow \sigma_1 \qquad \sigma_1 \vdash s_2 \rightsquigarrow \sigma_2}{\sigma_0 \vdash s_1{;}s_2 \rightsquigarrow \sigma_2}$$

$$\frac{\sigma \vdash e \hookrightarrow 0 \qquad \sigma \vdash s_2 \rightsquigarrow \sigma'}{\sigma \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \rightsquigarrow \sigma'}$$

$$\frac{\sigma \vdash e \hookrightarrow y \qquad y \neq 0 \qquad \sigma \vdash s_1 \rightsquigarrow \sigma'}{\sigma \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \rightsquigarrow \sigma'}$$

$$\frac{\sigma \vdash e \hookrightarrow 0}{\sigma \vdash \texttt{while}\ e\ \texttt{do}\ s \rightsquigarrow \sigma}$$

$$\frac{\sigma \vdash e \hookrightarrow y \qquad y \neq 0 \qquad \sigma \vdash s \rightsquigarrow \sigma' \qquad \sigma' \vdash \texttt{while}\ e\ \texttt{do}\ s \rightsquigarrow \sigma''}{\sigma \vdash \texttt{while}\ e\ \texttt{do}\ s \rightsquigarrow \sigma''}$$

For constructing a denotational semantics, we have to make the state transformation explicit in the type of the denotations for statements. Hence, we have

$$\begin{array}{rlcl}
y \ni & \mathsf{Val} & = & \mathbf{W}_\perp \oplus \mathbf{Z}_\perp \oplus \ldots \\
\sigma \ni & \mathsf{Store} & = & \mathsf{Var} \to \mathsf{Val} \\
\mu \ni & \mathsf{MStore} & = & \mathsf{Wrong} + \mathsf{Store} \\
\gamma \ni & \mathsf{Command} & = & \mathsf{MStore} \to \mathsf{MStore}
\end{array}$$

where Wrong is the one-point CCPO with only element wrong. For each syntactic category, there is a corresponding semantic function:

$$\begin{array}{lcl}
\mathcal{E} & : & \mathsf{Exp} \to \mathsf{Store} \to \mathsf{Val} \\
\mathcal{S} & : & \mathsf{Stmt} \to \mathsf{Command}
\end{array}$$

The semantics of expressions is straightforward and therefore omitted. We only assume that it returns wrong in case a variable is not defined in the store and that wrong is propagated recursively.

$$
\begin{array}{lcl}
\mathcal{S}[\![\texttt{var}\ v]\!] & = & \lambda\mu.\mathit{case}\ \mu\ \mathit{of}\ \ \begin{array}[t]{l} \mathit{Inr}\ \sigma \to \mathit{Inr}\ \sigma[v \mapsto 0] \\ \mathit{Inl}\ \mathsf{wrong} \to \mathit{Inl}\ \mathsf{wrong} \end{array} \\
\mathcal{S}[\![v\texttt{:=}e]\!] & = & \lambda\mu.\mathit{let}\ y = \mathcal{E}[\![e]\!]\sigma\ \mathit{in} \\
 & & \quad \mathit{case}\ \mu\ \mathit{of} \\
 & & \qquad \mathit{Inr}\ \sigma \to \begin{array}[t]{l} \mathit{if}\ y = \bot \vee \sigma = \bot \\ \mathit{then}\ \bot \\ \mathit{else}\ \mathit{if}\ v \notin \mathit{dom}(\sigma) \\ \mathit{then}\ \mathit{Inl}\ \mathsf{wrong} \\ \mathit{else}\ \mathit{Inr}\ \sigma[v \mapsto y] \end{array} \\
 & & \qquad \mathit{Inl}\ \mathsf{wrong} \to \mathit{Inl}\ \mathsf{wrong} \\
\mathcal{S}[\![\texttt{skip}]\!] & = & \lambda\mu.\mu \\
\mathcal{S}[\![s_1\texttt{;}s_2]\!] & = & \lambda\mu.\mathcal{S}[\![s_2]\!](\mathcal{S}[\![s_1]\!]\mu) \\
\mathcal{S}[\![\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!] & = & \lambda\mu.\mathit{case}\ \mathcal{E}[\![e]\!]\mu\ \mathit{of}\ \ \begin{array}[t]{l} \mathit{In}_0(\mathsf{wrong}) \to \mathit{Inl}\ \mathsf{wrong} \\ \mathit{In}_1(0) \to \mathcal{S}[\![s_2]\!]\mu \\ \_\_ \to \mathcal{S}[\![s_1]\!]\mu \end{array} \\
\mathcal{S}[\![\texttt{while}\ e\ \texttt{do}\ s]\!] & = & \mathit{fix}\ \lambda w.\lambda\mu.\mathit{case}\ \mathcal{E}[\![e]\!]\sigma\ \mathit{of}\ \ \begin{array}[t]{l} \mathit{In}_0(\mathsf{wrong}) \to \mathit{Inl}\ \mathsf{wrong} \\ \mathit{In}_1(0) \to \mu \\ \_\_ \to w(\mathcal{S}[\![s]\!]\mu) \end{array}
\end{array}
$$

The interesting parts in the semantics of statements are the assignment/variable declaration statements and the while statement. The equation for the assignment statement has to take care of two invariants. First, (assuming strict evaluation) the store should never contain undefined values. Second, assignment must be strict with respect to the store: if the store is undefined, it should never again become defined. The equation for the while statement employs a fixpoint to model the repeated execution of the statements in the loop. Hence, proving statements about a while loop requires fixpoint induction!

11.2 Dynamic Data Structures

Modeling dynamic data structures requires the introduction of a notion of locations separate from the variables. This modification, which requires a two-stage access mechanism for variables, frees locations from the second-class status that they assume in many languages. The store model is idealized inasmuch each location can assume any value. While unrealistic, it simplifies the modeling and avoids clutter in the definitions.

$$
\begin{array}{rlcl}
l \ni & \mathsf{Loc} & & \text{unspecified set of locations disjoint from } \mathsf{Var} \text{ etc.} \\
\sigma \ni & \mathsf{Store} & = & \mathsf{Loc} \hookrightarrow \mathsf{Val} \\
y \ni & \mathsf{Val} & = & \mathbf{Z} + \mathsf{Loc} + \dots \\
\rho \ni & \mathsf{Env} & = & \mathsf{Var} \hookrightarrow \mathsf{Loc}
\end{array}
$$

With the refined model, the expression semantics needs both, an environment and a store. The environment is a mapping from variables to locations where the association between variable and location remains fixed over the lifetime of the variable.

The meaning of an expression must now be defined by a judgment

$$\sigma, \rho \vdash e \hookrightarrow y$$

which is defined similar as before. The only difference is the treatment of variables which is given by the following rule

$$\frac{\rho(v) = l \qquad \sigma(l) = y}{\sigma, \rho \vdash v \hookrightarrow y}$$

The meaning of a statement has a judgment of type

$$\sigma, \rho \vdash s \rightsquigarrow \sigma, \rho$$

which is defined analogously to the previous subsection, except for the cases of variable declaration and assignment.

$$\frac{l \notin \mathit{dom}(\sigma)}{\sigma, \rho \vdash \texttt{var}\ v \rightsquigarrow \sigma[l \mapsto 0], \rho[v \mapsto l]}$$

$$\frac{\sigma, \rho \vdash e \hookrightarrow y \qquad l = \rho(v) \qquad l \in \mathit{dom}(\sigma)}{\sigma, \rho \vdash v := e \rightsquigarrow \sigma[l \mapsto y], \rho}$$

The manipulation of dynamic data structures requires further operators. To keep things simple, we add operations for references (pointers to a value): creating a reference, dereferencing it, and updating it with a new value.

$$\begin{array}{lll} e & ::= & \cdots \mid \texttt{get}\ e \\ s & ::= & \cdots \mid v{:=}\texttt{new}\ e \mid \texttt{set}\ e\ e \end{array}$$

All the machinery is already in place, we just need to add new evaluation rules to the semantics.

$$\frac{\sigma, \rho \vdash e \hookrightarrow l \qquad y = \sigma(l)}{\sigma, \rho \vdash \texttt{get}\ e \hookrightarrow y}$$

$$\frac{\sigma, \rho \vdash e \hookrightarrow y \qquad l = \rho(v) \qquad l' \notin \mathit{dom}(\sigma)}{\sigma, \rho \vdash v{:=}\texttt{new}\ e \rightsquigarrow \sigma[l \mapsto l', l' \mapsto y], \rho}$$

$$\frac{\sigma, \rho \vdash e_1 \hookrightarrow l \qquad \sigma, \rho \vdash e_2 \hookrightarrow y}{\sigma, \rho \vdash \texttt{set}\ e_1\ e_2 \rightsquigarrow \sigma[l \mapsto y], \rho}$$

The corresponding extension to the denotational semantics is straightforward and omitted.

11.3 Parameter Passing

In the presence of a store, there are new strategies of parameter passing refining call-by-value and call-by-name. When introducing procedures into the IMP language, the following questions arise.

- Are formal parameters bound to values or to locations?
- In the latter case, are they assigned to newly allocated locations or to existing ones?
- If a parameter is a location, is the referenced data structure copied or aliased?

Varying the answers to these questions yields a number of different parameter passing strategies and behaviors.

Binding formal parameters to values would make them variables in the mathematical sense. In fact, it would enforce the programming convention that parameters should not be changed in a procedure body. However, usually parameters variables are bound to locations, so that they can be assigned to as any other variable.

The question if parameters locations are freshly assigned determines whether an assignment to the parameter is visible at the call site of the procedure. Reusing the location of the call site corresponds to the *call-by-reference* strategy. With that strategy, the caller has to ensure that each actual parameter is represented by a location. If that parameter happens to be an expression, then a fresh location must be allocated to hold the value of the expressions. The language FORTRAN 90 employs call-by-reference throughout.

Allocating a new location and copying the referenced value to the new location corresponds to call-by-value-result (the variant call-by-result just allocates the new location without copying the value). Before returning from the procedure, the value in the new location is copied back to the location at the call site so that the effect of the procedure call becomes visible.

Call-by-reference and call-by-value-result are subtly different as the following example demonstrates. Since values are passed as locations, a procedure like swap2 below can be written.

```
swap2 (x, y) {
  x := x+y;
  y := x-y;
  x := x-y;
}
```

Applying swap2 to a pair of integer variables exchanges their values. However, executing swap2 (z, z) with call-by-reference sets the variable z to zero because the parameters x and y are bound to the same location (they are *aliases*). Call-by-value-result does not suffer this problem, it leaves z unchanged.

The answer to the last question determines how seriously a language implements call-by-value. Most languages adopt the stance that locations are just passed as parameters without further copying taking place. In effect, this means that dynamic data structures are always passed by reference.

The following operational semantics specifies a language which extends `IMP` with procedures with call-by-reference parameter passing. It restricts actual parameters to variables. For simplicity, procedures are toplevel, only.

$$\begin{array}{lcl} p & ::= & d \ldots d; s \\ d & ::= & f(v \ldots v)s \\ s & ::= & f(v \ldots v) \mid \ldots \\ e & ::= & v \mid \ldots \end{array}$$

The semantics of the language is defined by four judgements. A program computes just a final store and environment.

$$\vdash p \rightsquigarrow \sigma, \rho$$

The meaning of a statement is a transformation on stores and environments. The statement may depend on the set of procedure definitions in the program.

$$d^*; \sigma, \rho \vdash s \rightsquigarrow \sigma', \rho'$$

The meaning of an expression is split into the meaning of l-values (in this languages, only variables) and the meaning of expressions. The l-value of a variable is a location. It only depends on the environment.

$$\rho \vdash_l v \hookrightarrow l$$

The meaning of an expression depends on the environment and on the store. It does not change either of these.

$$\sigma, \rho \vdash e \hookrightarrow y$$

We define the inference rules bottom-up, starting with the l-value rule. It just reads from the environment.

$$\frac{l = \rho(v)}{\rho \vdash_l v \hookrightarrow l}$$

The only interesting rule for expressions is the variable rule. It relies on the l-value rule to access the location of a variable.

$$\frac{\rho \vdash_l v \hookrightarrow l \qquad y = \sigma(l)}{\sigma, \rho \vdash v \hookrightarrow y}$$

The new rule for statements is the one for procedure calls. One important detail is that the final environment after executing the body of the called procedure is discarded.

$$\frac{f(v_1 \ldots v_n)s \in d^* \qquad (\forall 1 \leq i \leq n)\ \rho \vdash_l v_i' \hookrightarrow l_i \qquad d^*; \sigma, \rho[v_i \mapsto l_i] \vdash s \rightsquigarrow \sigma', \rho'}{d^*; \sigma, \rho \vdash f(v_1', \ldots, v_n') \rightsquigarrow \sigma', \rho}$$

Finally, the rule for programs initializes the list of definitions to the one from the program, the store and the environment start empty.

$$\frac{d^*; \emptyset, \emptyset \vdash s \rightsquigarrow \sigma, \rho}{\vdash d^*; s \rightsquigarrow \sigma, \rho}$$

11.4 Types for IMP

Adding types to IMP in the style of the simply-typed lambda calculus is straightforward, only the treatment of references requires the addition of an extra set of type assumptions about locations. We will set up the type system in such a way that it guarantees two things. First, that variables are used in a type consistent way. Second, that only defined variables are ever accessed. The following type-related items are involved:

- types of expressions and variables
  $\mathsf{Type} \ni \tau ::= \alpha \mid \texttt{Int} \mid \texttt{Ref}\ \tau$
- types of procedures
  $\mathsf{PType} \ni \pi ::= \tau \times \cdots \times \tau \rightarrow 0$
- $\Gamma : \mathsf{Var} \rightarrow \mathsf{Type}$ type assumptions for variables
- $\Phi : \mathsf{PVar} \rightarrow \mathsf{PType}$ mapping from procedure names to procedure types

For each judgment of the operational semantics there is a corresponding typing judgment.

- For expressions, the usual judgment $\Gamma \vdash e : \tau$ applies.
- For l-values, a similar judgment $\Gamma \vdash_l l : \tau$ is constructed.
- For statements, the judgment $\Gamma \vdash s \Rightarrow \Gamma$ only needs to check the consistency of the assumptions with their use in the expressions and in the assignment statement. The outgoing Γ is not strictly required. It enforces the declare-before use policy for variables.

$$\Gamma \vdash \texttt{var}\ v \Rightarrow \Gamma, v : \tau$$

$$\frac{\Gamma(v) = \tau \qquad \Gamma \vdash e : \tau}{\Gamma \vdash v{:=}e \Rightarrow \Gamma}$$

$$\Gamma \vdash \texttt{skip} \Rightarrow \Gamma$$

$$\frac{\Gamma_0 \vdash s_1 \Rightarrow \Gamma_1 \qquad \Gamma_1 \vdash s_2 \Rightarrow \Gamma_2}{\Gamma_0 \vdash s_1;s_2 \Rightarrow \Gamma_2}$$

$$\frac{\Gamma \vdash e : \texttt{Int} \qquad \Gamma \vdash s_1 \Rightarrow \Gamma_1 \qquad \Gamma \vdash s_2 \Rightarrow \Gamma_2}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Rightarrow \Gamma}$$

$$\frac{\Gamma \vdash e : \texttt{Int} \qquad \Gamma \vdash s \Rightarrow \Gamma'}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ s \Rightarrow \Gamma}$$

Since a variable declaration does not contain a type in the syntax, the typing rule nondeterministically “guesses” a type. The program typechecks if this guess turns out to be consistent with the rest of the program.

At an assignment, the type of the variable should be equal to the type of the expression on the right-hand side. The rules for the `skip` statement and the sequence statement are straightforward.

The rules for the conditional and the while statement both allow for local variable definitions inside the branches or the body of the loop which are forgotten in the context. The operational semantics is more permissive, but in the case of the while, it is impossible to guarantee that the body is executed at least once (so that the variable is defined). For the conditional, the condition could be weakened to just ask for identical typing assumptions after both branches or even to filter out the intersection of the final typing assumptions.

For this subset of the language and any extension with further primitive types, we can prove the following result.

Lemma 36 *Suppose that* $\emptyset \vdash s \Rightarrow \Gamma$. *If* $\mu = \mathcal{S}[\![s]\!](\mathit{Inr}\ \lambda x\,.\,\mathit{In}_0(\mathsf{wrong}))$ *then* $\mu = \mathit{Inr}\ \sigma \in \mathit{Inr}\ (\mathsf{Store})$ *and* $(\forall v \in \mathit{dom}(\Gamma))\ \sigma(v) \in \mathcal{T}[\![\Gamma(v)]\!]$.

The next block of rules deals with the typing of references.

$$\frac{\Gamma \vdash e : \mathtt{Ref}\ \tau}{\Gamma \vdash \mathtt{get}\ e : \tau}$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma(v) = \mathtt{Ref}\ \tau}{\Gamma \vdash v{:=}\mathtt{new}\ e \Rightarrow \Gamma}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{Ref}\ \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{set}\ e_1\ e_2 \Rightarrow \Gamma}$$

In the presence of references, the type system does not guarantee that values of type `Ref` τ are properly initialized (although it could be changed to do so). For illustration, just consider the statement sequence

```
var x;
var p;
if e then p := new 0 else skip;
x := get p;
```

Depending on the value of `e`, variable `p` either holds a defined location (that contains the value `0`) or it holds a null pointer. The latter case yields an execution error at `get p` (or at least undefined behavior). Hence, a type soundness result would have to be appropriately weakened.

This behavior can be improved by integrating statements with expressions and returning variables to their mathematical status, that is, by regarding statements as expressions that return a fixed value `()` (*unit* or *void*). This unification of the operational semantics and the typing rules is left as an exercise.

The last block of rules considers procedures. Their types are collected in a special typing assumption, Φ. Again, due to the lack of procedure declarations, procedure types are nondeterministically guessed and only verified for consistency. Since procedure calls are statements, the rules for expressions remain as before. The judgment for statements is extended with Φ which is collected from the list of procedure definitions in an auxiliary judgment and does not change thereafter. Hence, Φ is just passed down through all statement rules, which now have the form $\Phi; \Gamma \vdash s \Rightarrow \Gamma$.

$$\frac{\Phi(f) = \tau_1 \times \cdots \times \tau_n \to 0 \qquad (\forall 1 \leq i \leq n)\ \Gamma \vdash v'_i : \tau_i}{\Phi; \Gamma \vdash f(v'_1, \ldots, v'_n) \Rightarrow \Gamma}$$

While the last rule checks a use of a procedure call in a statement, another rule is required to check a procedure declaration.

$$\frac{\Phi(f) = \tau_1 \times \cdots \times \tau_n \to 0 \qquad \Phi; \{v_i : \tau_i\} \vdash s \Rightarrow \Gamma'}{\Phi \vdash f(v_1, \ldots, v_n)s}$$

Finally, the rule that links all parts of a program together ensures that declarations and uses are mutually consistent.

$$\frac{(\forall 1 \leq i \leq n)\ \Phi \vdash d_i \qquad \Phi; \Gamma \vdash s \Rightarrow \Gamma'}{\Phi; \Gamma \vdash d_1 \ldots d_n; s \Rightarrow \Gamma'}$$

In most cases, this rule will be applied to the empty typing environment for variables. However, if certain variables were used to provide input to the program, then this practice would have to be reflected in the environment Γ accordingly.

12 Objects and Classes

The material in this section is derived from Chapter 5 of EOPL (*Essentials of Programming Languages* by Friedman, Wand, and Haynes, second edition).

The main contribution of object-oriented programming is the packaging of local state together with operations that operate on this local state into an object. Ideally, the local state is encapsulated so that information about it is only available through the operations.

From a pragmatic point of view, an object is nothing but a record where some fields are designated to hold the state of the object (the *attributes* or *instance variables*) and the remaining fields hold the operations (the *methods*). Each operation is a function that (implicitly) takes the object itself as a parameter. Object-oriented languages provide syntactic support, *method invocation*, to apply the method functions in the right way.

Languages with this functionality are called *object-based*. The classical example is the language Self, the main contemporary contender is JavaScript. Such languages provide a cloning facility to create new objects from existing ones. A typical programming idiom is the creation of a *trait* or *prototype* object, which is equipped with the intended set of operations. The clone operation is then used to create objects from the prototypes as they are needed.

Class-based object-oriented language provide *classes* as a means of specifying the fields and methods of many similar objects. Usually, each class comes with a facility to generate objects (*instances* of the class) according to the class's specification.

Many class-based languages have a concept of *inheritance*, whereby a class may import field and method specifications from another class. The idea is that inheritance can model a hierarchical refinement of object behaviors. If class B inherits from class A, then B is a *subclass* of A and A is a *superclass* of B. Some languages (*e.g.*, C++) also support heterarchies of classes, that is, a class may inherit from more than one superclass (*multiple inheritance*).

A further important feature of object-oriented languages is *polymorphism* (the quality or state of being able to assume different forms, according to Merriam-Webster). In the context of object-orientation, it is taken to mean *subtype* or *inclusion polymorphism*, that is, an object an object of a class B can play the role of an object of any superclass of B. However, other equally useful forms of polymorphism exist.

12.1 Syntax

We will define a class-based object-oriented language with single inheritance (*i.e.*, each class has at most one superclass). In the syntax, every class definition will refer to a superclass. The superclass may be the predefined class `object`.

Let $c \in \mathsf{CVar}$ range over class names, $m \in \mathsf{MVar}$ over method names.

$$\begin{array}{lcl}
p & ::= & d^*; e \\
d & ::= & \texttt{class}\ c\ \texttt{extends}\ c\ f\ m^* \\
f & ::= & \texttt{fields}\ v^* \\
m & ::= & \texttt{method}\ m(v^*)\ e \\
e & ::= & \texttt{let}\ v = e\ \texttt{in}\ e \mid v := e \mid \texttt{new}\ c(e^*) \mid e.m(e^*) \mid \dots
\end{array}$$

Again, for simplicity, statements are integrated into expressions. The sequence statement $e_1;e_2$ can be expressed by $\texttt{let}\ v = e_1\ \texttt{in}\ e_2$ where $v \notin FV(e_2)$. The absence of a special field access notation is deliberate, only methods are accessible from outside the object itself.

A program is executed by evaluating its body expression. The $\texttt{new}\ c(\dots)$ expression creates a new object of class c and runs its initialization method with the supplied parameters. The expression $e.m(e^*)$ is a method call. It expects e to evaluate to an object (reference) and then invokes the method m of the object with the supplied parameters. Among the expressions are constants, the usual arithmetic operations, and conditionals.

12.2 Examples

Taken from EOPL.

12.2.1 Terminology example

- `i` and `j` are also called *member* or *instance variables*
- Methods are are also called *member functions.* A methods declaration consists of a *method name*, *method parameters*, and a *method body.*
- Method names are sometimes called *messages.*
- Methods may be mutually recursive.

```
class c1 extends object
  fields i, j

  method new (x) {
    i := x;
    j := 0-x
  }
  method countup (d) {
    i := i+d;
    j := j-d;
  }
  method getstate () {
    list (i, j)
  }

  let t1 = 0
      t2 = 0
      o1 = new c1 (3)
  in  t1 := o1.getstate ();
      o1.countup (2);
      t2 := o1.getstate ();
      list (t1, t2)
```

12.2.2 Dynamic dispatch

```
class interior_node extends object
  fields left, right

  method new (l, r) {
    left := l;
    right := r
  }
  method sum () {
    left.sum() + right.sum()
  }

class leaf_node extends object
  fields value

  method new (v) {
    value := v
  }
  method sum () {
    value
  }

let o1 = new interior_node (
           new interior_node (
             new leaf_node (3),
             new leaf_node (4)),
           new leaf_node (5))
in o1.sum ()
```

12.2.3 Inheritance

```
class point extends object
  fields x, y

  method new (initx, inity) {
    x := initx;
    y := inity
  }
  method move (dx, dy) {
    x := x+dx;
    y := y+dy
  }
  method get_location () {
    list (x, y)
  }

class colorpoint extends point
  fields color

  method set_color (c) {
    color := c
  }
  method get_color () {
    color
  }

let p  = new point (3, 4)
    cp = new colorpoint (10, 20)
in  p.move (3, 4);
    cp.set_color (87);
    cp.move (10, 20);
    list (p.get_location (),
          cp.get_location (),
          cp.get_color())
```

12.2.4 Shadowing of instance variables

A field in a class shadows a field of the same name in any superclass.

12.2.5 Method override

A method declared in a class overrides a previous method definition for the same method name in any superclass. This may be more refined by taking into account the arity of the method and the argument types if they are available.

12.2.6 Static vs. dynamic method dispatch

Interpretation of `o2.m2 ()`?

```
class c1 extends object
  method new () { 1 }
  method m1  () { 1 }
  method m2  () { self.m1 () }

class c2 extends c1
  method m1 ()  { 2 }

let o1 = new c1 ()
    o2 = new c2 ()
in  list (o1.m1 (),
          o1.m2 (),
          o2.m1 (),
          o2.m2 ())
```

12.2.7 The `super` object

The object `super` refers to the part of the object that corresponds to the super-class of the class in which the current method is defined.

```
class c1 extends object
  method new () { 1 }
  method m1  () { self.m2 () }
  method m2  () { 13 }

class c2 extends c1
  method m1 () { 22 }
  method m2 () { 23 }
  method m3 () { super.m1 () }

class c3 extends c2
  method m1 () { 32 }
  method m2 () { 33 }

let o3 = new c3 ()
in  o3.m3()
```

12.3 Recursive Record Semantics

Denotationally, object-orientation with inheritance can be modeled using records and fixpoints. This classical encoding is described in detail in: W. Cook and J. Palsberg. *A denotational semantics of inheritance and its correctness.* Inf. & Comp., 114(2):329–350, 1995.

The underlying idea is to model an object as a record of methods. For example, objects of the `Point` class

```
class Point extends object
  fields x, y
  method distFrom0 () {
    sqrt (square (self.x) + square (self.y))
  }
  method closer (p) {
    self.distFrom0 () < p.distFrom0 ()
  }
```

are represented by a record of type

$$\{x : \texttt{Int}, y : \texttt{Int}, \mathit{distFrom0} : () \to \texttt{Int}, \mathit{closer} : \texttt{point} \to \texttt{Bool}\}$$

with *generator* function

$$\begin{array}{lll} \multicolumn{3}{l}{\texttt{MakePoint}(\texttt{initx}, \texttt{inity}) =} \\ \quad \lambda\texttt{self}.\{ & \texttt{x} = & \texttt{initx}, \\ & \texttt{y} = & \texttt{inity}, \\ & \texttt{distFrom0} = & \lambda().\texttt{sqrt}(\texttt{square}(\texttt{self.x}) + \texttt{square}(\texttt{self.y})), \\ & \texttt{closer} = & \lambda\texttt{p}.\texttt{self.distFrom0}() < \texttt{p.distFrom0}() \\ \quad \} & & \end{array}$$

Creation of a new `Point` object boils down to taking the fixpoint of the result of the `MakePoint` function:

$$\texttt{new Point}(x, y) = \mathit{fix}\ (\texttt{MakePoint}(x, y))$$

Using inheritance, the `Point` class may be extended to a `Circle` class.

```
class Circle extends Point
  fields radius

  method distFrom0 () {
    max (super.distFrom0 () - self.radius, 0)
  }
```

Modeling inheritance requires the definition of a *wrapper* function for `Circle`. A wrapper parameterizes the construction of the underlying record over `self` and `super`, the record modeling the superclass.

$$\begin{array}{lll} \multicolumn{3}{l}{\texttt{WrapCircle}(r) = \lambda\texttt{self}.\lambda\texttt{super}.} \\ \quad \{ & \texttt{radius} = & r, \\ & \texttt{distFrom0} = & \lambda().\texttt{max}(\texttt{super.distFrom0}() - \texttt{self.radius}, 0) \\ \quad \} & & \end{array}$$

Wrappers are combined with generators using *wrapper application* $\boxed{\triangleright}$ to yield a new generator.

$$w \boxed{\triangleright} g = \lambda\texttt{self}.(\lambda r.w(\texttt{self})(r) \oplus r)(g(\texttt{self}))$$

Hence, the generator function for `Circle` objects is

$$\texttt{MakeCircle}(\texttt{initx}, \texttt{inity}, \texttt{r}) = \texttt{WrapCircle}(r) \boxed{\triangleright} \texttt{MakePoint}(\texttt{initx}, \texttt{inity})$$

with objects constructed by

$$\texttt{new Circle}(x, y, r) = \mathit{fix}\ (\texttt{MakeCircle}(x, y, r))$$

Cook and Palsberg show that this denotational model is equivalent to an informally described operational model.

12.4 Operational Semantics

The operational semantics generally binds variables to locations. Each location in the store contains either an integer or an object. Each object is represented by a triple, where the first component assigns field names to locations, the second assigns method names to method closures (a list of formal variables and an expression), and the third refers to the superclass part of the object. Hence, each object may be considered as a list of records, where each list entry corresponds to one step in the inheritance chain. The root class `object` is represented by the value 0.

$$\begin{array}{lcl}
\mathsf{Obj} & = & \mathsf{Fields} \times \mathsf{Methods} \times \mathsf{Loc} \\
\mathsf{Fields} & = & \mathsf{Var} \hookrightarrow \mathsf{Loc} \\
\mathsf{Methods} & = & \mathsf{MVar} \hookrightarrow (\mathsf{Var}^* \times \mathsf{Exp}) \\
\mathsf{Storable} & = & \mathbf{Z} + \mathsf{Loc} + \mathsf{Obj} \\
\mathsf{Store} & = & \mathsf{Loc} \hookrightarrow \mathsf{Storable} \\
\mathsf{Env} & = & \mathsf{Var} \hookrightarrow \mathsf{Loc} \\
\mathsf{Val} & = & \mathbf{Z} + \mathsf{Loc}
\end{array}$$

The judgment $\rho, \sigma \vdash e \rightsquigarrow y, \sigma'$ for evaluating an expression maps an environment, a store, and an expression to a location and a final store. The location contains the value of the expression. Evaluation of expressions relies on two auxiliary judgments for creating new objects and for method calls.

$$\frac{\rho, \sigma \vdash e_1 \rightsquigarrow y_1, \sigma' \qquad \rho[v \mapsto l], \sigma'[l \mapsto y_1] \vdash e_2 \rightsquigarrow y_2, \sigma'' \qquad l \notin dom(\sigma')}{\rho, \sigma \vdash \texttt{let}\ v = e_1\ \texttt{in}\ e_2 \rightsquigarrow y_2, \sigma''}$$

$$\frac{\rho, \sigma \vdash e \rightsquigarrow y, \sigma' \qquad \rho(v) = l \qquad l \in dom(\sigma')}{\rho, \sigma \vdash v{:=}e \rightsquigarrow y, \sigma'[l \mapsto y]}$$

$$\frac{\sigma, c \vdash_n l, \sigma' \qquad \rho, \sigma', l \vdash_m \texttt{new}(e_1, \ldots, e_n) \rightsquigarrow y, \sigma''}{\rho, \sigma \vdash \texttt{new}\ c(e_1, \ldots, e_n) \rightsquigarrow l, \sigma''}$$

$$\frac{\rho, \sigma \vdash e \rightsquigarrow l, \sigma' \qquad \rho, \sigma', l \vdash_m m(e_1, \ldots, e_n) \rightsquigarrow y, \sigma''}{\rho, \sigma \vdash e.m(e_1, \ldots, e_n) \rightsquigarrow y, \sigma''}$$

Object construction takes an initial state and a class and returns a pointer to the constructed object and the final state. It binds the variable `self` in the bottommost record to refer to the entire object.

$$\frac{l \notin \mathit{dom}(\sigma) \qquad \sigma[l \mapsto 0], c, l, l \vdash'_n \sigma''}{\sigma, c \vdash_n l, \sigma''}$$

$$\sigma, \texttt{object}, l_s, l \vdash'_n \sigma[l \mapsto ([\texttt{self} \mapsto l_s], \emptyset, 0)]$$

$$\frac{\begin{array}{c}\texttt{class}\ c\ \texttt{extends}\ c'\ \texttt{fields}\ v_1 \ldots v_n\ m^* \\ l' \notin \mathit{dom}(\sigma) \qquad \sigma[l' \mapsto 0], c', l_s, l' \vdash'_n \sigma' \\ \rho = [v_1 \mapsto l_1, \ldots, v_n \mapsto l_n] \qquad (i \neq j \Rightarrow l_i \neq l_j) \qquad l_i \notin \mathit{dom}(\sigma') \\ \sigma'' = \sigma'[l_1 \mapsto 0, \ldots, l_n \mapsto 0]\end{array}}{\sigma, c, l_s, l \vdash'_n \sigma''[l \mapsto (\rho, m^*, l')]}$$

Method invocation takes an object reference and a method call besides environment, store, and the `self` pointer.

$$\frac{(\forall 1 \leq i \leq n)\ \rho, \sigma_{i-1} \vdash e_i \rightsquigarrow y_i, \sigma_i \qquad \sigma_n, l \vdash'_m m(y_1, \ldots, y_n) \rightsquigarrow y, \sigma'}{\rho, \sigma_0, l \vdash_m m(e_1, \ldots, e_n) \rightsquigarrow y, \sigma'}$$

$$\frac{\begin{array}{c}\sigma(l) = (\rho, \mu, l') \qquad \mu(m) = (v_1 \ldots v_n, e) \\ \sigma' = \sigma[l_i \mapsto y_i] \qquad l_i \notin \mathit{dom}(\sigma) \qquad (i \neq j \Rightarrow l_i \neq l_j) \\ \sigma \vdash_e l' \rightsquigarrow \rho' \qquad (\rho \oplus \rho')[v_i \mapsto l_i, \texttt{super} \mapsto l'], \sigma' \vdash e \rightsquigarrow y, \sigma''\end{array}}{\sigma, l \vdash'_m m(y_1, \ldots, y_n) \rightsquigarrow y, \sigma''}$$

$$\frac{\begin{array}{c}\sigma(l) = (\rho, \mu, l') \qquad m \notin \mathit{dom}(\mu) \\ \sigma, l' \vdash'_m m(y_1, \ldots, y_n) \rightsquigarrow y, \sigma''\end{array}}{\sigma, l \vdash'_m m(y_1, \ldots, y_n) \rightsquigarrow y, \sigma''}$$

$$\sigma, 0 \vdash'_m m(y_1, \ldots, y_n) \rightsquigarrow \text{message not understood}$$

The method environment is constructed by composing the environments bottom-up from the object chain. Environment composition $\rho_1 \oplus \rho_2$ is defined by

$$(\rho_1 \oplus \rho_2)(v) = \begin{array}{ll} \rho_2(v) & v \in \mathit{dom}(\rho_2) \\ \rho_1(v) & \text{otherwise} \end{array}$$

The composition operation is iterated to compute the environment in which a method call executes.

$$\frac{\sigma(l) = (\rho', \mu', l') \qquad \sigma \vdash_e l' \rightsquigarrow \rho}{\sigma \vdash_e l \rightsquigarrow \rho \oplus \rho'}$$

$$\sigma \vdash_e 0 \rightsquigarrow \{\ \}$$

12.5 Types

Apart from runtime errors caused by illegal uses of primitive operations, the operational semantics flags only one kind of error, "message not understood". As usual, the job of a type system is to avoid those errors.

As we have seen in the recursive record semantics, objects may be modeled with records. However, simple types for records lack the flexibility to model the polymorphism due to inheritance properly. The missing ingredient is *subtyping*, which is expressed as a binary relation <: on types (in fact, a partial ordering relation).

The basic intuition of subtyping is that whenever $A <: B$ (read: A is a subtype of B or B is a supertype of A) then a value of type A can be used wherever a value of type B is expected. In a type system, this intuition is formalized by the *subsumption rule* (which relies on the subtyping relation being formalized by a judgement $\Gamma \vdash \tau <: \tau'$):

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \tau <: \tau'}{\Gamma \vdash e : \tau'}$$

A naive interpretation of subtyping is inclusion of the types' carrier sets. For example, each subrange type $[m..n]$ is a subtype of the type of natural numbers $\mathbf{N}$. Subtyping may also involve *coercion* (an injective function that maps values of the subtype to values of the supertype), for example, when considering the types of integers $\mathbf{Z}$ as a subtype of the set of fractions $\mathbf{Q}$.

For object-oriented programming, it is more relevant to consider subtyping for record types. Some languages (e.g., OCaml) define a dual notion for variant types, but this is not a common feature. The key notion for records is *width subtyping*: a record with more fields can be used wherever a record with fewer fields is expected. If a language supports *breadth subtyping*, too, then the types of the shared fields may be refined to subtypes.

Here is the inference rule for width subtyping

$$\frac{i \neq j \Rightarrow l_i \neq l_j}{\Gamma \vdash [l_1 : \tau_1, \dots, l_n : \tau_n] <: [l_1 : \tau_1, \dots, l_n : \tau_n, l_{n+1} : \tau_{n+1}, \dots, l_{n+m} : \tau_{n+m}]}$$

Breadth subtyping is specified by the following rule.

$$\frac{i \neq j \Rightarrow l_i \neq l_j \qquad \Gamma \vdash \tau_1 <: \tau_1' \qquad \dots \qquad \Gamma \vdash \tau_n <: \tau_n'}{\Gamma \vdash [l_1 : \tau_1, \dots, l_n : \tau_n] <: [l_1 : \tau_1', \dots, l_n : \tau_n']}$$

Usually, both, breadth and width subtyping, apply in combination.

In the presence of further type constructors (pairs, sums, functions, etc), we must extend the subtyping relation to types formed with these constructors. The extension is straightforward with pairs and sums, which are considered componentwise, but there is a twist for functions.

Consider a function like the `distFrom0` method above:

$$\begin{aligned}\texttt{distFrom0} &: \{\texttt{x} : \texttt{Real}, \texttt{y} : \texttt{Real}\} \to \texttt{PosReal}\\ \texttt{distFrom0} &= \lambda\texttt{p}.\texttt{sqrt}(\texttt{square}(\texttt{p.x}) + \texttt{square}(\texttt{p.y}))\end{aligned}$$

Clearly, `distFrom0` is applicable to any record that provides fields `x` and `y` of type `Real` assuming `square :  Real -> Real`. Here are some suitable arguments:

$$\begin{array}{lcl} \mathtt{p}_1 & : & \{\mathtt{x} : \mathtt{Real}, \mathtt{y} : \mathtt{Real}\} \\ \mathtt{p}_2 & : & \{\mathtt{x} : \mathtt{Real}, \mathtt{y} : \mathtt{Real}, \mathtt{z} : \mathtt{Real}\} \\ \mathtt{p}_3 & : & \{\mathtt{x} : \mathtt{Int}, \mathtt{y} : \mathtt{Int}\} \end{array}$$

The last value is only suitable if $\mathtt{Int} <: \mathtt{Real}$.

Finding a subtype for the function type involves the question which functions can be substituted for `distFrom0` without violating the typing of an application of the function. It would be unsound to substitute a function that expects an additional field, say, `z`, because this would make application to $\mathtt{p}_1$ and $\mathtt{p}_3$ above fail. It would also be unsound to substitute a function returning a supertype of `PosReal` because this might violate the constraints of a context that expects only positive numbers.

In summary, the point is that a function `f` may be substituted for `distFrom0` if it makes *less* demands on the argument and guarantees (the same or) *more* constraints on the result. In short, to form a subtype of a function type, the new argument type must be a supertype of the original argument type whereas the new result type must be a subtype of the original result type. This reversal of the subtyping relation in the argument position of the function type constructor is called *contravariance*. In the result position, the orientation of the subtyping relation remains same, this is called *covariance*.

Formally, subtyping for function types is governed by the following inference rule.

$$\frac{\Gamma \vdash \tau_1' <: \tau_1 \qquad \Gamma \vdash \tau_2 <: \tau_2'}{\Gamma \vdash \tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}$$

Type inference is possible for some systems with records, but it is a highly non-trivial task. All uses of unification in the algorithm for inferring simple types must be replaced by an enforcement of the subtyping relation because the subsumption rule may be used at any point in a type derivation. Depending on the particular features of the system, the subtyping relation may be undecidable. Some systems abstract the record types to class types (a class type is a set of class names), which are generated from the class definitions, to facilitate type inference.

13 On Understanding Types, Data Abstraction, and Polymorphism

Excerpted from: Luca Cardelli, Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. ACM Computing Surveys, 17(4):471–522, 1985.

Kinds of Polymorphism

- Monomorphic languages:
  - All functions and procedures have unique type.
  - All values and variables of one and only type.
  - Comparable to Pascal or C type systems.
- Polymorphic languages:
  - Values and variables may have more than one type.
  - Polymorphic functions admit operands of more than one type.
- Universal polymorphism:
  - Function works uniformly on range of types.
  - Parametric and inclusion polymorphism.
- Ad-hoc polymorphism:
  - Function works on several unrelated types.
  - Overloading and coercion.

Universal Polymorphism

- Parametric polymorphism:
  - Actual type is a function of type parameters.
  - Each application of polymorphic function substitutes the type parameters.
  - Generic functions:
    - ”Same” work is done for arguments of many types.
    - Length function over lists.
- Inclusion polymorphism:
  - Value belongs to several types related by inclusion relation.
  - Object-oriented type systems.

Ad-hoc Polymorphism

- Overloading

  - Same name denotes different functions.
  - Context decides which function is denoted by particular occurence of a name.
  - Preprocessing may eliminate overloading by giving different names to different functions.

- Coercion
  - Type conversions convert an argument to a type expected by a function.
  - May be provided statically at compile time.
  - May be determined dynamically by run-time tests.
- Only apparent polymorphism
  - Operators/functions only have one type.
  - Only syntax "pretends" polymorphism.

Overloading and Coercion

- Distinction may be blurred:

```
3   + 4
3.0 + 4
3   + 4.0
3.0 + 4.0
```

- Different explanations possible:
  - `+` has four overloaded meanings.
  - `+` has two overloaded meanings (integer and real addition) and integers may be coerced to reals.
  - `+` is real addition and integers are always coerced to reals.
- Overloading and/or coercion or both!

Preview of Fun

- lambda-calculus based language
  - Basis is first-order typed lambda-calculus.
  - Enriched by second-order features for modeling polymorphism and object-oriented languages.
- First-order types
  - Bool, Int, Real, String.

- Various forms of type quantifiers

$$\begin{array}{lll} \mathit{Type} & ::= & \cdots \mid \mathit{QuantifiedType} \\ \mathit{QuantifiedType} & ::= & \forall A.\mathit{Type} \mid \exists A.\mathit{Type} \mid \forall A \subseteq \mathit{Type}.\mathit{Type} \mid \exists A \subseteq \mathit{Type}.\mathit{Type} \end{array}$$

- Modeling of advanced type systems:
  - Universal quantification: parameterized types.
  - Existential quantifiers: abstract data types.
  - Bounded quantification: type inheritance.

The Typed lambda-Calculus

- Extension of Untyped lambda-Calculus
  - Every variable must be explicitly typed when introduced as typed variable
  - Result types can be deduced from function body.
- Examples
  - `value succ = fun(x:Int) x+1`
  - `value twice = fun(f:  Int -> Int) fun(y:Int) f(f(y))`
- Type declarations:
  - `type IntPair = Int x Int`
  - `type IntFun = Int -> Int`
- Type annotations/assertions:
  - `(3, 4):  IntPair`
  - `value intPair:  IntPair = (3, 4)`
- Local variables
  - `let a = 3 in a+1`
  - `let a:  Int = 3 in a+1`

13.1 Universal Quantification

- Typed lambda-calculus describes monomorphic functions.
  - Not sufficient to describe functions that behave the same way for argumentes of different types.
- Introduce types as parameters:

```
value id = all[a] fun(x:a) x
id[Int](3)

id : forall a. a -> a
id[Int] : Int -> Int
```

- May omit type information:

```
value id = fun(x) x
id(3)
```

  - Type inference (type reconstruction) reintroduces `all[a]`, `a`, and `[Int]`

- Polymorphic types:

```
type GenericId = forall a. a -> a
id: GenericId
-- examples
value inst = fun(f: forall a. a -> a) (f[Int], f[Bool])
value intid: Int -> Int = fst(inst(id))
value boolid: Bool -> Bool = snd(inst(id))
```

Polymorphic Functions

- First version of polymorphic `twice`:

```
 value twice1 = all[t] fun(f: forall a. a -> a)
                           fun(x: t) f[t](f[t](x))

 twice1[Int](id)(3)     -- legal.
 twice1[Int](succ)      -- illegal!
```

- Second version of polymorphic twice:

```
value twice2 = all[t] fun(f: t -> t) fun(x: t) f(f(x))

twice2[Int](succ)       -- legal.
twice2[Int](id[Int])(3) -- legal.
```

- Both versions different in nature of f:
  - In `twice1`, `f` is a polymorphic function of type `forall a:  a -> a`.
  - In `twice2`, `f` is a monomorphic function of type `t -> t` (for some instantiation of `t`)

Parametric Types

- Type definitions with similar structure:

```
type BoolPair = Bool x Bool
type IntPair = Int x Int
```

- Use parametric definition:

```
type Pair[T] = T x T
type PairOfBool = Pair[Bool]
type PairOfInt = Pair[Int]
```

- Type operators are not types:

```
type A[T] = T -> T
type B = forall T. T -> T
```

  – Different notions!

Recursive Definitions

- Recursively defined type operators:

```
rec type List[Item] =
   [nil: Unit
   ,cons: {head: Item, tail: List[Item]} ]
```

- Constructing values of recursive types:

```
value nil: forall Item.List[Item] = all[Item]. [nil = ()]
value intNil: List[Int] = nil[Int]
value cons:
   forall Item. (Item x List[Intem]) -> List[Item] =
      all[Item].
         fun(h Item, t: List[Item])
            [cons = {head = h, tail = t}]
```

13.2 Existential Quantification

- Existential type quantification:
  - `p:  exists a.  t(a)`
  - For some type cfta, `p` has type `t(a)`
- Examples:
  - `(3, 4):  exists a.  a x a`
  - `(3, 4):  exists a.  a`

  - A value can satisfy different existential types!

- Sample existential types:
  - `type Top = exists a.  a` (type of any value)
  - `exists a.  exists b.  a x b` (type of any pair)
- Particularly useful: "existential packaging"
  - `x:  exists a.  a x (a -> Int)`
  - `(snd(x))(fst(x))`
  - `(3, succ)` has this type
  - `([1,2,3], length)` has this type

Information Hiding

- Abstract types:
  - Unknown representation type.
  - Packaged with operations that may be applied to representation.
- Another example:
  - `x:  exists a.  {const:  a, op:  a -> Int}`
  - `x.op(x.const)`
- Restrict use of abstract types:
  - Simplify type checking.

  -
    ```
    value p: exists a. a x (a -> Int)
            = pack[a = Int in a x (a -> Int)](3, succ)
    ```
  - Value `p` is a *package*
  - Type `a x (a -> Int)` is the *interface*.
  - Binding `a=Int` is the type *representation*.
- General form:
  - pack [a = typerep in interface](contents)

Use of Packages

- Package must be opened before use:

  -
    ```
    value p = pack[a = Int in a x (a -> Int)]
                  (3, succ)
    open p as x in (snd(x))(fst(x))
    ```

  – `value p = pack[a = Int in {arg: a, op: a -> Int}]`

```
                    (3, succ)
    open p as x in x.op(x.arg)
```

- Reference to hidden type: `open p as x[b] in ...fun(y:b) (snd(x))(y)` ...

Packages and Abstract Data Types
Modeling of Ada type system:

- Records with function components model Ada packages.
- Existential quantification models Ada type abstraction.

```
type Point = Real x Real
type Point1 =
   {makepoint: (Real x Real) -> Point,
      x_coord: Point x Real,
      y_coord: Point x Real}

value point1: Point1 =
   {makepoint = fun(x:Real, y:Real)(x, y),
      x_coord = fun(p:Point) fst(p),
      y_coord = fun(p:Point) snd(p)}
```

Ada Packages

```
package point1 is
   function makepoint(x: Real, y: Real) return Point;
   function x_coord(P: Point) return Real;
   function y_coord(P: Point) return Real;
end point1;

package body point1 is
   function makepoint(x: Real, y: Real) return Point;
      -- implementation of makepoint
   function x_coord(P: Point) return Real;
      -- implementation of x_coord
   function y_coord(P: Point) return Real;
      -- implementation of y_coord
end point1;
```

Hidden Data Structures

- Ada:

  ```
  package body localpoint is
     point: Point;
  ```

```
     procedure makePoint(x, y: Real); ...
     function x_coord return Real; ...
     function y_coord return Real; ...
   end localpoint
```

- Fun:

```
   value localpoint =
     let p: Point = ref((0,0)) in
      {makepoint = fun(x: Real, y: Real) p := (x, y),
         x_coord = fun() fst(p)
         y_coord = fun() snd(p)}
```

- First-order information hiding: Use let construct to restrict scoping at value level (hide record components).

Hidden Data Types

Second-order information hiding: Use existential quantification to restrict scoping at type level (hide type representation).

```
package point2
   type Point is private;
   function makepoint(x: Real, y: Real) return Point;
   ...
   private
   -- hidden local definition of type Point
end point2;

type Point2 =
   exists Point.
      {makepoint: (Real x Real) -> Point,
         ...}

type Point2WRT[Point] =
      {makepoint: (Real x Real) -> Point,
         ...}

value point2: Point2 = pack[Point = (Real x Real) in
   Point2WRT[Point]] point1
```

Combining Universal and Existential Quantification

- Universal quantification: generic types.
- Existential quantification: abstract data types.
- Combination: parametric data abstractions.

The following signature of list and array operations is used in the examples.

```
nil: forall a. List[a]
cons: forall a. (a x List[a]) -> List[a]
hd: forall a. List[a] -> a
tl: forall a. List[a] -> List[a]
null: forall a. List[a] -> Bool

array: forall a. Int -> Array[a]
index: forall a. (Array[a] x Int) -> a
update: forall a. (Array[a] x Int x a) -> Unit
```

Concrete Stacks

```
type IntListStack =
  {emptyStack: List[Int],
   push: (Int x List[Int]) -> List[Int]
   pop: List[Int] -> List[Int],
   top:List[Int] -> Int}

value intListStack: IntListStack =
  {emptyStack = nil[Int],
   push = fun(a: Int, s: List[Int]) cons[Int](a,s),
   pop =  fun(s: List[Int]) tl[Int](s)
   top =  fun(s: List[Int]) hd[Int](s)}

type IntArrayStack =
  {emptyStack: (Array[Int] x Int),
   push: (Int x (Array[Int] x Int)) -> (Array[Int] x Int),
   pop:  (Array[Int] x Int) -> (Array[Int] x Int),
   top:  (Array[Int] x Int) -> Int}

value intArrayStack: IntArrayStack =
  {emptyStack = (Array[Int](100), -1) ...}
```

Generic Element Types

```
type GenericListStack =
   forall Item.
     {emptyStack: List[Item],
      push: (Item x List[Item]) -> List[Item]
      pop:  List[Item] -> List[Item],
      top:  List[Item] -> Item}

value genericListStack: GenericListStack =
   all[Item]
     {emptyStack = nil[Item],
      push = fun(a: Item, s: List[Item]) cons[Item](a,s),
      pop  = fun(s: List[Item]) tl[Item](s)
      top  = fun(s: List[Item]) hd[Item](s)}

type GenericArrayStack =
   ...

value genericArrayStack: GenericArrayStack =
   ...
```

Hiding the Representation

```
type GenericStack =
   forall Item. exists Stack. GenericStackWRT[Item][Stack]

type GenericStackWRT[Item][Stack] =
  {emptyStack: Stack,
   push: (Item x Stack) -> Stack
   pop: Stack -> Stack,
   top: Stack -> Item}

value listStackPackage: GenericStack =
   all[Item]
      pack[Stack = List[Item]
         in GenericStackWRT[Item][Stack]]
      genericListStack[Item]
value useStack =
   fun(stackPackage: GenericStack)
      open stackPackage[Int] as p[stackRep]
      in p.top(p.push(3, p.emptystack))

useStack(listStackPackage)
```

Quantification and Modules

- Modules
  - Abstract data type packaged with operators.
  - Can import other (known) modules.
  - Can be parameterized with (unknown) modules.
- Parametric modules
  - Functions over existential types.

Example

```
type PointWRT[PointRep] =
  {mkPoint: (Real x Real) -> PointRep,
   x-coord: PointRep -> Real,
   y-coord: PointRep -> Real}

type Point = exists PointRep. PointWRT[PointRep]


value cartesianPointOps =
  {mkpoint = fun(x: Real, y: Real) (x,y),
   x-coord = fun(p: Real x Real) fst(p),
   y-coord = fun(p: Real x Real) snd(p)}

value cartesianPointPackage: Point =
   pack[PointRep = Real x Real
      in PointWRT[PointRep]]
   (cartesianPointOps)

value polarPointPackage: Point =
   pack[PointRep = Real x Real in PointWRT[PointRep]]
  {mkpoint = fun(x: Real, y: Real) ...,
   x-coord = fun(p: Real x Real)   ...,
   y-coord = fun(p: Real x Real)   ...}
```

Parametric Modules

```
type ExtendedPointWRT[PointRep] =
   PointWRT[PointRep] &
   {add: (PointRep x PointRep) -> PointRep}

type ExtendedPoint =
   exists PointRep. ExtendedPointWRT[PointRep]

value extendPointPackage =
   fun(pointPackage: Point)
   open pointPackage as p[PointRep] in
      pack[PointRep' = PointRep in ExtendedPointWRT[PointRep']]
      p & {add = fun(a: PointRep, b: PointRep)
                   p.mkpoint(p.x-coord(a)+p.x-coord(b),
                             p.y-coord(a)+p.x-coord(b))}

value extendedCartesianPointPackage =
   extendPointPackage(cartesianPointPackage)
```

A Circle Package

```
type CircleWRT2[CircleRep, PointRep] =
  {pointPackage: PointWRT[PointRep],
   mkcircle: (PointRep x Real) -> CircleRep,
   center: CircleRep -> PointRep, ...}

type CircleWRT1[PointRep] =
   exists CircleRep. CircleWRT2[CircleRep, PointRep]

type Circle =
   exists PointRep. CircleWRT1[PointRep]

value circleModule: CircleModule =
all[PointRep]
   fun(p: PointWRT[PointRep])
      pack[CircleRep = PointRep x Real
         in CircleWRT2[CircleRep,PointRep]]
     {pointPackage = p,
      mkcircle = fun(m: PointRep, r: Real)(m, r) ...}

value cartesianCirclePackage =
   open CartesianPointPackage as p[Rep] in
      pack[PointRep = Rep in CircleWRT1[PointRep]]
         circleModule[Rep](p)

open cartesianCirclePackage as c0[PointRep] in
open c0 as c[CircleRep] in
  ...c.mkcircle(c.pointPackage.mkpoint(3, 4), 5) ...
```

A Rectangle Package

```
type RectWRT2[RectRep, PointRep] =
  {pointPackage: PointWRT[PointRep],
   mkrect: (PointRep x PointRep) -> RectRep, ...}

type RectWRT1[PointRep] =
   exists RectRep. RectWRT2[RectRep, PointRep]

type Rect =
   exists PointRep. RectWRT1[PointRep]

type RectModule = forall PointRep.
   PointWRT[PointRep] -> RectWRT1[PointRep]

value rectModule: RectModule =
all[PointRep]
   fun(p: PointWRT[PointRep])
      pack[PointRep' = PointRep
```

```
        in RectWRT1[PointRep']]
    {pointPackage = p,
     mkrect = fun(tl: PointRep, br: PointRep) ...}
```

A Figures Package

```
type FiguresWRT3[RectRep, CircleRep, PointRep] -
  {circlePackage: CircleWRT[CircleRep, PointRep],
   rectPackage: RectWRT[RectRep, PointRep],
   boundingRect: CircleRep -> RectRep}

type FIguresWRT1[PointRep] =
   exists RectRep. exists CircleRep.
      FigureWRT3[RectRep, CircleRep, PointRep]

type Figures =
   exists PointRep. FIgureWRT1[PointRep]

type FiguresModule = forall PointRep.
   PointWRT[PointRep] -> FiguresWRT1[PointRep]

value figuresModule: FIguresModule =
all[PointRep]
   fun(p: PointWRT[PointRep])
      pack[PointRep' = PointRep
         in FiguresWRT1[PointRep]]
      open circleModule[PointRep](p) as c[CircleRep] in
         open rectModule[PointRep](p) as r[RectRep] in
           {circlePackage = c, ...}
```

13.3 Bounded Quantification

- Type inclusion:
  - Type `A` is included in type `B` when all values of A are also values of B.
  - Inclusion relation on subranges, records, variants, function, universally and existentially quantified types.
- Integer subrange type `n..m`
  - `n..m` <: `n'..m'` iff $\mathtt{n'} \leq \mathtt{n} \wedge \mathtt{m} \leq \mathtt{m'}$

  -
    ```
    value f = fun(x: 2..5) x+1
    f: 2..5 -> 3..6
    f(3)
    value g = fun(y: 3..4) f(y)
    ```

- Function type
  - `s -> t <: s' -> t'` iff `s' <: s` and `t <: t'`
  - Function of type `3..7 -> 7..9` can be also considered as function of type `4..6 -> 6..10`

Bounded Quantification and Subtyping

- Mix subtyping and polymorphism.

  ```
  value f0 = fun(x: {one: Int}) x.one
  f0({one = 3, two = true})

  value f = all[a] fun(x: {one: a}) x.one
  f[Int]({one = 3, two = true})
  ```

- Constraint `all[a <: T] e`

  ```
  value g0 = all[a <: {one: Int}] fun(x: a) x.one
  g0[{one:Int, two:Bool}]({one=3, two=true})
  ```

- Two forms of inclusion constraints:
  - In `f0`, implicit by function parameters.
  - In `g0`, explicit by bounded quantification.
  - Type expressions:

    ```
    g0: forall a <: {one: Int}. a -> Int
    ```

  - Type abstraction:

    ```
    value g = all[b] all[a <: {one: b}] fun(x:a)x:one
    g[Int][({one:Int,two:Bool})]({one=3,...})
    ```

Object Oriented Programming

```
type Point = {x: Int, y: Int}

value moveX0 =
   fun(p: Point, dx: Int) p.x := p.x + dx; p
value moveX =
   all[P <: Point] fun(p:P, dx: Int) p.x := p.x + dx; p

type Tile = {x: Int, y: Int, hor: Int, ver: Int}
moveX[Tile]({x = 0, y = 0, hor - 1, ver = 1}, 1).hor
```

- Result of `moveX` is same as argument type.
- `moveX` can be applied to objects of (yet) unknown type.

Bounded Existential Quantification and Partial Abstraction

- Bounding existential quantifiers:
  - `exists a <:  t.  t'`
  - `exists a.  t` is short for `exists a <:  Top.  t`
- Partially abstract types:
  - `a` is abstract.
  - We know `a` is subtype of `t`.
  - `a` is not more abstract than `t`.
- Modified packing construct:

  ```
  pack [a <= t = t' in t"] e
  ```

Points and Tiles

```
type Tile = exists P. exists T <= P. TileWRT2[P, T]

type TileWRT2[P, T] =
  {mktile: (Int x Int x Int x Int) -> T,
   origin: T -> P,
   hor: T -> Int,
   ver: T -> Int}

type TileWRT[P] = exists T <= P. TileWRT2[P, T]
type Tile = exists P. TileWRT[P]

type PointRep = {x: Int, y: Int}
type TileRep = {x: Int, y: Int, hor: Int, ver: Int}

pack [P = PointRep in TileWRT[P]]

pack [T <= PointRep = TileRep in TileWRT2[P, T]]
  {mktile = fun(x:Int, y: Int, hor: Int, ver: Int)
     {x=x, y-y, hor=hor, ver=ver},
      origin = fun(t: TileRep) t,
      hor = fun(t: TileRep) t.hor,
      ver = fun(t: TileRep) t.ver}

fun(tilePack: Tile)
   open tilePack as t[pointRep][tileRep]
      let f = fun(p: pointRep) ...
      in f(t.tile(0, 0, 1, 1))
```

13.4 Summary

- Three main principles
  - Universal type quantification (polymorphism).
  - Existential type quantification (abstraction).
  - Bounded type quantification (subtyping).
- Resulting programs may be statically type-checked.
  - Bottom-construction of types.
  - More sophisticated type inference possible (ML).
- More general type systems.
  - Type-checking typically not decidable any more.
  - Dependent types (Martin-Löf).
  - Calculus of constructions (Coquand and Huet)..

14 Typed Object-Oriented Programming

See Martin Abadi and Luca Cardelli, *A Theory of Objects*, Springer Verlag, 1996, for background material.

14.1 Object Types

Subtyping is a relation between (object) types, having to do with keeping track of the messages that objects can accept. Inheritance is a relation between classes, having to do with the superclass-subclass relation and with method reuse. The usual connection between inheritance and subtyping is the following: a subclass generates objects whose type is a subtype of the objects generated by a superclass.

Initially, object-oriented languages confused classes with object types, and therefore ended up blurring the distinction between implementations (classes) and interfaces (object types), and between inheritance (code sharing) and subtyping (interface sharing). Recent object-oriented languages are designed to make these distinctions and to take advantage of them.

The main idea is to separate the definition of the object types that are to be used as interfaces for using them from the classes that implement those types. The object type only fixes the protocol used with the objects, not their implementation.

For example, there might be two interfaces that are subtypes of each other, but their implementations can be completely unrelated. Similarly, inheritance may be employed to reuse code from different classes, but the type of the newly formed class need not be a subtype of the type of any superclass.

The former concept is present in Java in the form of interfaces. The latter concept is—to a limited degree—obtainable with nested classes.

14.2 Separating Subtyping from Subclassing

Since the subtyping relation no longer derives from the inheritance relation, other ways of defining it must be considered.

1. *Structural subtyping* derives the subtyping relation from the structure of the types. A disadvantage of structural subtyping is accidental matching.
2. *Nominal subtyping* is based on an (arbitrary) ordering on named types. The ordering must be compatible with the underlying structure and it is tedious to specify precisely.

With structural subtyping, object types (typed records) support multiple subtyping. Since a subclass can only extend the interface of an existing class, it follows that the object type of an object of a subclass is always a subtype of the superclass's object type. That is: subclassing implies subtyping.

14.3 Bounded Type Parameters

Consider two object type definitions

```
ObjectType Person {
  ...
  method eat (food: Food)
}

ObjectType Vegetarian {
  ...
  method eat (food: Vegetables)
}
```

where `Vegetables <: Food`. The intention is that a vegetarian is a special person. However, the `Vegetarian` type is **not** a subtype of the `Person` type because the food occurs contravariantly in the type of `eat`: `Food -> Unit <: Vegetables -> Unit`!

One solution is to parameterize the two object types over the kind of food and bound the food accordingly.

```
ObjectOperator PersonEating[F <: Food] {
  ...
  method eat (food: F)
}

ObjectOperator VegetarianEating[F <: Vegetables] {
  ...
  method eat (food: F)
}
```

- `VegetarianEating[Vegetables]` is a well-formed type
- `VegetarianEating[Food]` is not well-formed

There is no direct relation between `PersonEating` and `VegetarianEating`, however, it holds that

```
forall F <: Vegetables. VegetarianEating[F] <: PersonEating[F]
```

Hence, we obtain that

```
Vegetarian = VegetarianEating[Vegetables]  <: PersonEating[Vegetables]
```

Another possibility is a model with bounded existential quantification or bounded abstract types. The idea here is to provide the food edible by the object in a field with a bounded parameterized type.

```
ObjectType Person {
  type F <: Food
  field lunch: F
  method eat (food: F)
}

ObjectType Vegetarian {
  type F <: Vegetables
  field lunch: F
  method eat (food: F)
}
```

An object can only be constructed by providing it with suitable food. After construction, the specifics about the food are forgotten. It is only ensured that eating the food originally provided is OK.

With this construction, it holds that `Vegetarian <: Person` because `eat` can only be applied to the value of the `lunch` field. However, this subsumption would be void in the presence of imperative field updates because it would be possible to first subsume a `Vegetarian` to a `Person`, and then update the `lunch` to `Meatballs`, say.

14.4 Subclassing without Subtyping

- *inheritance is not subtyping*
- Motivation: admit contravariant occurences of `self` in method argument; *binary methods*

An example:

```
ObjectType Max {
  field n: Int
  method max(other: Max): Max
}

ObjectType MinMax {
  field n: Int
  method max(other: MinMax): MinMax
  method min(other: MinMax): MinMax
}
```

These object types are defined recursively!

Now, we want to define two classes corresponding to types `Max` and `MinMax`.

```
class maxClass {
  field n: Int = 0
  method max(other: Self): Self {
    if (self.n > other.n) {
      return self
    } else {
      return other
    }
  }
}

class minMaxClass extends maxClass {
  method min(other: Self): Self {
    if (self.n < other.n) {
      return self
    } else {
      return other
    }
  }
}
```

- `min` and `max` are binary methods, with `other` being a contravariant argument of type `self`.
- `minMaxClass` inherits `n` and `min` from `maxClass`.
- objects of `maxClass` have type `Max`
- objects of `minMaxClass` have type `MinMax`

`MinMax` is not a subtype of `Max`:

```
class minMaxClass' extends minMaxClass {
  override max(other: Self): Self {
    if (other.min(self) == other) {
      return self
    } else {
      return other
    }
  }
}
```

Objects of this class have type `m1 : MinMax`. Assuming that `MinMax <: Max`, subsumption yields `m1 : Max`. The type of `max` allows it to apply `m1.max` to another object `m2 : Max` as in `m1.max(m2)`. However, as `m2` does not necessarily have a `min` method, the call `other.min` may fail. This contradicts the assumption.

14.5 Object Protocols

When subclassing does not lead to a subtype, there might still be a useful relation between the types induced by a class and a subclass. However, this relation does not have the subsumption property.

We would like to capture the fact that any object that supports the operators from `MinMax` also supports the operators from `Max`. We'll call that suite of operators the *protocol* of the objects. The protocols may be formalized using type operators.

```
ObjectOperator MaxProtocol[X] {
  field n: Int
  method max(other: X): X
}

ObjectOperator MinMaxProtocol[X] {
  field n: Int
  method max(other: X): X
  method min(other: X): X
}
```

In general, any recursive object type can be converted into its associated protocol by abstracting over the recursive occurrences. The original type is then the fixpoint of the protocol operator.

For the example in question,

```
Max    = fix X. MaxProtocol[X]
MinMax = fix X. MinMaxProtocol[X]
```

There are two possible formal relationships between `MinMax` and `Max`. First

```
MinMax <: MaxProtocol[MinMax]
```

Second, we can define a subtyping relation between type operators:

```
P <* P'     iff    for all T. P[T] <: P'[T]
```

With this definition

```
MinMaxProtocol <* MaxProtocol
```

Either one may be the basis for a subprotocol relation:

1. *F*-bounded parameterization

   ```
   S subprotocol T     if    S <: T-Protocol[S]
   ```

   Then we might define object types as follows

   ```
   ObjectOperator P1[X <: MaxProtocol[X]] {...}
   ```

   `P1[MinMax]` is a legal instantiation

2. higher-order bounded parameterization

   ```
   S subprotocol T     if    S-Protocol <* T-Protocol
   ```

with object types defined as follows

```
ObjectOperator P2[P <* MaxProtocol] {...}
```

`P2[MinMaxProtocol]` is a legal instantiation

One way of avoiding the complications of working with type operators, types can be equipped with a matching relation `<#`. The matching relation can be defined through either of the `subprotocol` definitions above. In any case, matching does not enjoy a subsumption property, as we demonstrated above. However, it can be used for bounded parameterization:

```
ObjectOperator P3[X <# Max] { ... }
```

With this declaration, the instantiation `P3[MinMax]` is legal.

15 An Interpretation of Objects and Object Types

(see paper by Abadi, Cardelli, and Viswanathan in POPL'96)

15.1 Object Calculus

Syntax

$$\begin{array}{rcll} e & ::= & x & \text{variables} \\ & \mid & [l_i = \varsigma(x_i)e_i]_{i=1}^n & \text{objects} \\ & \mid & e.l_j & \text{method invocation} \\ & \mid & e.l_j \Leftarrow \varsigma(x)e & \text{method update} \end{array}$$

Reduction rules; let $a = [l_i = \varsigma(x_i)e_i]_{i=1}^n$

$$\begin{array}{lcl} a.l_j & \longrightarrow & e_j[x_j \mapsto a] \\ a.l_j \Leftarrow \varsigma(x)e & \longrightarrow & [l_1 = \varsigma(x_1)e_1, \ldots l_j = \varsigma(x)e, \ldots] \end{array}$$

In an operational semantics with weak reduction, objects serve as values.

Syntax of types

$$\begin{array}{rcll} A, B & ::= & \mathit{Top} & \text{supertype of all types} \\ & \mid & [l_i : B_i]_{i=1}^n & \text{object type} \end{array}$$

Typing rules

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A}$$

$$\frac{(\forall 1 \leq j \leq n)\ \Gamma, x_j : [l_i : B_i]_{i=1}^n \vdash e_j : B_j}{\Gamma \vdash [l_i = \varsigma(x_i)e_i]_{i=1}^n : [l_i : B_i]_{i=1}^n}$$

$$\frac{\Gamma \vdash e : [l_i : B_i]_{i=1}^n}{\Gamma \vdash e.l_j : B_j}$$

$$\frac{\Gamma \vdash e : [l_i : B_i]_{i=1}^n \qquad \Gamma, x : [l_1 : B_1, \dots l_j : B_j, \dots] \vdash e' : B_j}{\Gamma \vdash e.l_j \Leftarrow \varsigma(x)e' : [l_i : B_i]_{i=1}^n}$$

Subtyping (breadth only)

15.2 Main Result

There is a typing preserving embedding of the (explicitly typed) object calculus into (essentially) Fun, the polymorphic lambda calculus.

Encoding of Object-types into Fun-types

$$\begin{array}{lcll} \mathit{Top}^* & = & \mathit{Top} & \\ {[l_i : B_i]_{i=1}^n}^* & = & \mu(Y)\exists(X <: Y)\ \{ & l_i^{sel} : X \to B_i^*, \\ & & & l_i^{upd} : (X \to B_i^*) \to X, \\ & & & \mathit{self} : X \\ & & \} & \end{array}$$

- B_i^* occurs both, covariantly and contravariantly

Idea of the term encoding

$$\begin{array}{lcl} [\![e.l_j]\!] & = & [\![e]\!].l_j^{sel}([\![e]\!].\mathit{self}) \\ [\![e.l_j \Leftarrow \varsigma(x)e']\!] & = & [\![e]\!].l_j^{upd}(\lambda(x)[\![e']\!]) \end{array}$$

The creation of objects is more complicated. Due to the presence of method update it is not sufficient to just define a recursive record because it would lead to a premature binding of *self*.

$$\begin{array}{lll} [\![[l_i = \varsigma(x_i)e_i]_{i=1}^n]\!] = & \mathtt{letrec}\ \mathit{create}(f_1)\dots(f_n) = & \{\ l_i^{sel} = f_i, \\ & & \quad l_i^{upd} = \lambda(g)\mathit{create}(f_1)\dots(g)\dots(f_n), \\ & & \quad \mathit{self} = \mathit{create}(f_1)\dots(f_n)\} \\ & \mathtt{in}\ \mathit{create}(\lambda(x_1)[\![e_1]\!])\dots(\lambda(x_n)[\![e_n]\!]) & \end{array}$$

It remains to wrap this correctly into the typed language. First, we define some abbreviations. Let $A = [l_i : B_i]_{i=1}^n$, then

$$\begin{array}{lcl} C_A[X] & = & \{ \; l_i^{sel} : X \to B_i^*, \\ & & \quad l_i^{upd} : (X \to B_i^*) \to X, \\ & & \quad \mathit{self} : X \\ & & \} \\ \\ L_{l,B} & = & \mu(Y)\exists(X <: Y)\{l^{sel} : X \to B^*, \mathit{self} : X\} \end{array}$$

The typed translation

$$\begin{array}{lcl} [\![x]\!] & = & x \\ \\ [\![[l_i = \varsigma(x_i : A)e_i]_{i=1}^n]\!] & = & \texttt{letrec}\ \mathit{create}(f_1 : A^* \to B_1^*) \ldots (f_n : A^* \to B_1^*) : A^* = \\ & & \quad \texttt{pack}\ [X <: A^* = A^*\ \texttt{in}\ C_A[X]] \\ & & \qquad \{ \; l_i^{sel} = f_i, \\ & & \qquad\quad l_i^{upd} = \lambda(g : A^* \to B_i^*)\mathit{create}(f_1) \ldots (g) \ldots (f_n), \\ & & \qquad\quad \mathit{self} = \mathit{create}(f_1) \ldots (f_n)\} \\ & & \texttt{in}\ \mathit{create}(\lambda(x_1 : A^*)[\![e_1]\!]) \ldots (\lambda(x_n : A^*)[\![e_n]\!]) \\ \\ [\![e.l]\!] & = & \texttt{open}\ [\![e]\!]\ \texttt{as}\ X <: L_{l,B}, x : \{l^{sel} : X \to B^*, \mathit{self} : X\}\ \texttt{in}\ x.l^{sel}(x.\mathit{self}) \\ \\ [\![e.l_j \Leftarrow \varsigma(x)e']\!] & = & \texttt{open}\ [\![e]\!]\ \texttt{as}\ X <: A^*, y : C_A[X]\ \texttt{in}\ (y.l^{upd})(\lambda(x : X)[\![e']\!]) \end{array}$$

Summary of technical results (informal)

1. If $\Gamma \vdash e : A$ in the object calculus, then $\Gamma^* \vdash [\![e]\!] : A^*$ in the polymorphic lambda calculus.

2. If $\Gamma \vdash e : A$ and $e \longrightarrow e'$ in the object calculus, then $[\![e]\!] \longrightarrow^* [\![e']\!]$ in the polymorphic lambda calculus.

3. If $\emptyset \vdash e : A$, then e terminates iff $[\![e]\!]$ terminates.

4. If $\emptyset \vdash e : A$ and $\emptyset \vdash e' : A$, then $[\![e]\!] = [\![e']\!] : A^*$ implies $e = e' : A$.

Further results:

1. imperative variant of calculus and translation; requires an additional field $\mathit{clone} : \{\} \to X$

2. variance annotations determine if a method is invoke-only, update-only, or both

3. self types: just use the X from $\exists X <: Y$ in record type.

16 Logic Programming

Further background material may be found in "Computing with Logic" by David Maier and David S. Warren, Benjamin Cummings, 1988.

The idea of logic programming is to turn knowledge and facts expressed in terms of formulae in a logic into a program. The most successful logic programming language, Prolog, relies on predicate logic as its underlying theory. A logic program might also be considered as a means of specifying a relation. Hence, a typical program execution lists tuples that satisfy the program relation.

16.1 First-order Predicate Logic

The language of first-order predicate logic is defined by the following grammar.

$$
\begin{array}{lcll}
t & ::= & & \text{terms} \\
 & & x & \text{variables} \\
 & \mid & c & \text{constants} \\
 & \mid & f(t, \dots, t) & f \in \text{function symbols} \\
A & ::= & & \text{formulae} \\
 & & p(t, \dots, t) & p \in \text{relation symbols} \\
 & \mid & \texttt{true} & \\
 & \mid & \texttt{false} & \\
 & \mid & \neg A \mid A \wedge A \mid A \vee A \mid A \rightarrow A \mid A \leftrightarrow A & \\
 & \mid & (\exists x) A \mid (\forall x) A &
\end{array}
$$

Formulae of the form $p(t_1, \dots, t_n)$ are *atoms*. A *literal* is an atom (positive literal) or a negated atom (negative literal). A *clause* is a formula of the form

$$\forall \overline{x} L_1 \vee \cdots \vee L_m$$

where each L_i is a literal and $\overline{x}$ are all variables occurring in $L_1, \dots, L_m$. Each clause can be written in *clausal form*

$$A_1, \dots, A_k \leftarrow B_1, \dots, B_n$$

where $A_1, \dots, A_k$ are the positive literals among the L_i and $B_1, \dots, B_n$ are the negative literals stripped of the negation. The $\overline{A}$ are the *conclusions* and the $\overline{B}$ are the *premises* of the clause.

Example:

$$(\forall x)\ (\forall y)\ p(x) \vee \neg A \vee \neg q(y) \vee B$$

written in clausal form

$$p(x), B \leftarrow A, q(y)$$

A clause with only one conclusion is a *definite clause* or *Horn clause.* If there is no conclusion, the clause is a *goal.*

Definition 53 A *logic program* is a finite nonempty set of definite clauses.

A Horn clause $A \leftarrow B_1, \dots, B_n$ can be read in two different ways. The *procedural interpretation* is: "to solve A you must solve $B_1, \dots, B_n$." The *declarative interpretation* is: "$B_1, \dots, B_n$ imply A."

16.2 Procedural Interpretation—SLD-resolution

We start by considering a logic program where all clauses are variable-free (*ground*). Let

$$N = \leftarrow G_1, \dots, G_m$$

be a ground goal and $C = G_i \leftarrow B_1, \dots, B_n$ be a program clause. Then

$$N' = \leftarrow G_1, \dots, G_{i-1}, B_1, \dots, B_n, G_i, \dots, G_m$$

is the *resolvent* of N and C with the property that $N' \wedge C \Rightarrow N$. A *derivation* is a sequence of *resolvents*. If a derivation ends with the empty clause, it is called a *refutation* of the original goal.

In general, variables may be present in goals as well as in a program. Hence, the process of finding a resolvent may require that a goal literal and the conclusion of a program clause are unified. Let again,

$$N = \leftarrow G_1, \dots, G_m$$

be a goal, $C = A \leftarrow B_1, \dots, B_n$ be a program clause, and θ be the most general unifier (mgu) of A and G_i. Then

$$N' = \leftarrow \theta(G_1, \dots, G_{i-1}, B_1, \dots, B_n, G_i, \dots, G_m)$$

is a *resolvent of N and C with mgu θ.*

An SLD-derivation of goal N with respect to a program is a maximal sequence $N_0, N_1, \dots$ of goals with $N = N_0$, a sequence $C_0, C_1, \dots$ of program clauses, and a sequnce $\theta_0, \theta_1, \dots$ of substitutions such that

- N_{i+1} is a resolvent of N_i and C_i with mgu θ_i,
- C_i has no variable in common with $N_0, C_0, C_1, \dots$.

An SLD-derivation is an *SLD-refutation* if there is an i such that N_i is empty. Otherwise, the derivation is either *failed* (if it is finite) or infinite.

A goal may have arbitrary many SLD-derivations, depending on the choice of the subgoal and the program rule for each resolvent. Hence, the set of SLD-derivations is often depicted as a tree where each node represents a goal and each edge represents a choice (annotated with C_i and θ_i).

A typical example for computing with relations is in studying family relations.

```
male (kronos).
female (rhea).
female (gaia).
male (zeus). parent (kronos, zeus). parent (rhea, zeus).
female (hera). parent (kronos, hera). parent (rhea, hera).
male (poseidon). parent (kronos, poseidon). parent (rhea, poseidon).
female (demeter). parent (kronos, demeter). parent (rhea, demeter).
female (methis).
female (athene). parent (zeus, athene). parent (methis, athene).
female (persephone). parent (zeus, persephone). parent (demeter, persephone).
male (zagreus). parent (zeus, zagreus). parent (persephone, zagreus).
female (leto).
female (artemis). parent (zeus, artemis). parent (leto, artemis).
male (apollon). parent (zeus, apollon). parent (leto, apollon).
male (ares). parent (zeus, ares). parent (hera, ares).

person (X) :- male (X).
person (X) :- female (X).
father (X, Y) :- parent (X, Y), male (X).
mother (X, Y) :- parent (X, Y), female (X).
sibling (X, Y) :- father (A, X), mother (B, X), father (A, Y), mother (B, Y).
half_sibling (X, Y) :- parent (A, X), parent (A, Y).
grandfather (X, Y) :- father (X, A), parent (A, Y).
```

```
:- grandfather (zeus, zagreus).
% 1.  X1 -> zeus, Y1 -> zagreus
:- father (zeus, A1), parent (A1, zagreus).
% 2a. A1 -> artemis
:- parent (zeus, artemis), parent (artemis, zagreus) .
% 3a. parent (zeus, artemis)
:- parent (artemis, zagreus) .
% not refutable
% 2b. A1 -> persephone
:- parent (zeus, persephone), parent (persephone, zagreus).
% 3b. parent (zeus, persephone)
:- parent (persephone, zagreus)
% 4b. parent (persephone, zagreus)
:-
% 1, 2b, 3b, 4b is an SLD-refutation!
% One solution: X1 -> zeus, Y1 -> zagreus, A1 -> persephone
```

Another example.

```
path (X, Z) :- arc (X, Y), path (Y, Z).
path (X, X).
arc (b, c).
```

Build an SLD-tree for `path (X,c)`.

A Prolog interpreter traverses an SLD-tree depth-first starting from the goal input by the programmer. It always chooses the first subgoal and then tries to resolve it with the rule heads in program ordering. If the interpreter has no more rules available to resolve a subgoal, then it backtracks to the last successful resolvent, undoes it, and attempts the remaining program clauses.

This method is sound but incomplete since depth-first search may run into an infinite branch of the SLD-tree and thus never reach an existing refutation.

Practical implementations contain further ways of controlling backtracking. The best-known of these is the cut-operator `!`. As an example, consider the rule

```
r (X) :- f (X), !, g (X).
r (X) :- h (X).
```

Once the predicate `f (X)` succeeds, the cut instructs the Prolog interpreter to commit to the current resolution of `r (...)` by the rule containing the cut. In other words, even if `g (X)` fails, the second rule with body `h (X)` is never considered. Instead, the resolvent for `r (...)` is considered failed and the failure is propagated upwards. The same happens on failure in a later goal, after successfully processing `g (X)`.

Thus the behavior of the above code fragment is comparable to the function

```
r (x) = if if (x) then g (x) else h (x)
```

The cut operator often improves efficiency and termination behavior by shortcutting parts of the search space and by avoiding repeated enumeration of results. For example, here is a predicate that tests membership in a set represented by a list (potentially with repetitions).

```
myelem (X, [X|T]) :- ! .
myelem (X, [Y|T]) :- myelem (X, T).
```

Without the cut in the first clause, the predicate `myelem` would suffer some problems.

1. `myelem' (a, [a,a,a])` would succeed three times.
2. `myelem' (a, X)` would not terminate.

In addition, arithmetic operators are usually built in as well as several non-logical predicates, for example, to assert and retract rules from the program.

16.3 Semantics

An *interpretation* I for a first-order language consists of

- a nonempty set D, the domain of I,
- a constant assignment: $c \mapsto c_I \in D$,
- a function assignment: $f^{(n)} \mapsto f_I \in D^n \to D$,
- a relation assignment: $r^{(n)} \mapsto r_I \subseteq \mathcal{P}(D^n)$.

A variable assignment σ is a function from variables to D. Every variable assignment σ can be extended uniquely to a function $\hat{\sigma}$ from terms to D:

- $\hat{\sigma}(x) = \sigma(x)$
- $\hat{\sigma}(c) = c_I$

- $\hat{\sigma}(f(t_1, \ldots, t_n)) = f_I(\hat{\sigma}(t_1), \ldots, \hat{\sigma}(t_n))$

A formula A is true under variable assignment σ iff $I \models_\sigma A$ holds. The latter and its converse, $I \not\models_\sigma A$, are defined inductively by

- $I \models_\sigma r(t_1, \ldots, t_n)$ iff $(\hat{\sigma}(t_1), \ldots, \hat{\sigma}(t_n)) \in p_I$
- $I \models_\sigma \texttt{true}$
- $I \models_\sigma \neg A$ iff $I \not\models_\sigma A$
- $I \models_\sigma A \vee B$ iff $I \models_\sigma A$ or $I \models_\sigma B$
- $I \models_\sigma (\forall x)A$ iff $I \models_{\sigma[x \mapsto d]} A$, for all $d \in D$
- $I \not\models_\sigma r(t_1, \ldots, t_n)$ iff $(\hat{\sigma}(t_1), \ldots, \hat{\sigma}(t_n)) \notin p_I$
- $I \not\models_\sigma \texttt{false}$
- $I \not\models_\sigma \neg A$ iff $I \models_\sigma A$
- $I \not\models_\sigma A \vee B$ iff $I \not\models_\sigma A$ and $I \not\models_\sigma B$
- $I \not\models_\sigma (\forall x)A$ iff $I \not\models_{\sigma[x \mapsto d]} A$, for some $d \in D$

A formula A is true in interpretation I if $I \models_\sigma A$, for all variable assignments σ.

If T is a set of formulae, then I is a model for T if every formula in T is true in I. If T has a model, then T is satisfiable (consistent). Otherwise T is unsatisfiable (inconsistent). If every interpretation is a model for T, then T is *valid*.

A set of formulae T *semantically implies* T' ($T \models T'$) if every model of T is also a model of T'.

Thus armed, it is easy to see that SLD-resolution is sound. For a goal $N = \leftarrow B_1, \ldots, B_n$, let $\tilde{N} = B_1 \wedge \cdots \wedge B_n$.

Lemma 37 *If M is a resolvent of N and clause C with mgu θ, then $C \models \tilde{M} \rightarrow \theta(\tilde{N})$.*

Theorem 12 *Let P be a logic program and N a goal with an SLD-refutation with substitutions $\theta_0, \ldots, \theta_n$. Then $\theta_n \cdots \theta_1 \theta_0(\tilde{N})$ is a semantic consequence of P.*

16.4 Herbrand Models

Each logic program has a natural model, the Herbrand model. The *Herbrand universe* U_P is the set of all ground terms of P. The *Herbrand base* B_P is the set of all ground atoms.

Definition 54 The *Herbrand interpretation* for P has

- U_P as its domain
- the constant assignment $c \mapsto c \in U_P$
- the function assignment $f^{(n)} \mapsto \lambda(t_1, \ldots, t_n) f(t_1, \ldots, t_n)$
- each n-ary relation symbol is assigned an n-ary relation over U_P

The last item (and hence the entire Herbrand interpretation) is uniquely determined by a subset $I \subseteq B_P$ as follows:

$$r^{(n)} \mapsto \{(t_1, \ldots, t_n) \mid r(t_1, \ldots, r_n) \in I\}$$

The *immediate consequence operator* T_P maps a Herbrand interpretation to another Herbrand interpretation as follows:

Definition 55 Let A be a ground atom and I be a Herbrand interpretation. $A \in T_P(I)$ iff $B \leftarrow B_1, \ldots, B_n$ is a program clause, θ is a substitution such that $A = \theta(B)$, and $I \models \theta(B_1 \wedge \cdots \wedge B_n)$.

Lemma 38 *Let P be a program and I a Herbrand interpretation. I is a model of P iff $T_P(I) \subseteq I$ (I is a pre-fixpoint of T_P).*

The set of subsets of the Herbrand base for a program is a complete lattice with operations set union and set intersection. The operator T_P is monotonic with respect to set inclusion. Hence, the following definition:

$$T^{(0)}(I) = I \qquad T^{(i+1)}(I) = T(T^{(i)}(I)) \qquad T^{(\omega)}(I) = \bigcup_{i \in \mathbf{N}} T^{(i)}(I)$$

Theorem 13 *Let P be a program. Then P has a Herbrand model M_P such that*

- *M_P is the least Herbrand model of P*
- *M_P is the least pre-fixpoint of T_P*
- *M_P is the least fixpoint of T_P*
- *$M_P = T_P^{(\omega)}(\emptyset)$*

NB, this theorem suggests another complete way of computing a goal by constructing the least Herbrand model bottom-up.

17 Continuations

Continuations are a functional-programming based concept for expressing exceptions, backtracking, coroutines, and multi-threading. The main idea is to adopt a programming style which places some restrictions on user-defined functions:

- all functions are tail-recursive
- each function has one or more continuation parameters, each of which is a function
- instead of returning a value, each function invokes a continuation and passes the value as a parameter.

A continuation function is an abstraction of the rest of the computation. Since it is manifest in the program, it can be manipulated, stored, and modified like any other value. This gives rise to a number of interesting manipulations of control, *i.e.*, higher-order control structures.

17.1 Motivation: Exceptions

Consider a function that multiplies the elements of a tree of numbers.

```
product xt =
  if null xt
  then 1
  else product (left xt) * value xt * product (right xt)
```

If `xt` contains `0` then `product xt` is also `0`. However, the function still performs two multiplications multiplications for each inner node of the tree to compute the result. Here is an attempt to improve on the performance of `product`.

```
product1 xt =
  if null xt
  then 1
  else if value xt == 0
  then 0
  else product (left xt) * value xt * product (right xt)
```

While `product1` performs much fewer multiplications on trees containing `0`, we would like to perform no multiplications at all in such a case.

```
product2 xt =
  prod xt (\x -> x)
  where
  prod xt c =
    if null xt
    then c 1
    else if value xt == 0
    then c 0
    else prod (left xt) (\l -> prod (right xt) (\r -> c (l * r * value xt)))
```

The function `prod` is written in *continuation-passing style* and the argument `c` is the continuation. Each function call (to `prod` and `c`) is tail-recursive and results are returned by passing them to the respective continuation. However, the function still behaves exactly like `product1`!

Next, we exploit presence of continuations. By just returning `0` instead passing it to the continuation, we make the function call `prod xt (\x -> x)` (and hence `product3 xt`) return immediately with result `0`:

```
product3 xt =
  prod xt (\x -> x)
  where
  prod xt c =
    if null xt
    then c 1
    else if value xt == 0
    then 0
    else prod (left xt) (\l -> prod (right xt) (\r -> c (l * r * value xt)))
```

This break of continuation-passing style corresponds to the use of a so-called *control operator*

This programming style is useful but clumsy, so the Scheme programming language provides a primitive `call-with-current-continuation` which we abbreviate with `call/cc`. It enables us to rewrite `product2` without using continuation-passing style.

```
product3 xt =
  call/cc (\ return ->
    let prod xt =
      if null xt
      then 1
      else if value xt == 0
      then return 0
      else prod (left xt) * value xt * prod (right xt)
    in prod xt
```

We'll look later at how `call/cc` may be implemented.

17.2 Motivation: Backtracking

Consider the subset sum problem which is a specialized version of the knapsack problem: You are given a positive integer C (the weight that you can carry) and a list of positive integers (the weights of items you want to carry). Is there a subset of the items the weights of which add up to C? (This problem is known to be NP-complete.) An implementation in pseudo code might look like this:

```
subsetsum target items =
  let work path target items =
    if target == 0
    then RESULT path
    else if null items
    then FAIL
    else if (head items) <= target
    then TRY (work (head items : path) (target - head items) (tail items))
         ANDTHEN work path target (tail items)
    else work path target (tail items)
  in work [] target items
```

This function uses a number of primitives to express backtracking. `RESULT` announces a result. `FAIL` declares the current invokation to fail. `TRY ...ANDTHEN ...` searches for a result in the first argument and then in the second. (Similar operators might be used in a Prolog interpreter.)

The function `subsetsum` can be implemented easily using continuations. The idea is to pass two continuations to each function call, one for the successful case and another for the failure case.

```
subsetsum1 target items =
  let work path target items succ fail =
    if target == 0
    then succ path
    else if null items
    then fail ()
    else let hi = head items in
    if hi <= target
    then work (hi : path) (target - hi) (tail items) succ (\ () ->
         work path target (tail items) succ fail)
    else work path target (tail items) succ fail
  in work [] target items (\ x -> True) (\ () -> False)
```

The use of the success continuation in this example seems rather trivial. In fact, replacing `succ path` with just `True` would not change the behavior of the program. However, the success continuation may be used to compose a list of all results as follows.

```
subsetsum1 target items =
  let work path target items succ fail =
    if target == 0
    then succ path fail
    else if null items
    then fail ()
    else let hi = head items in
    if hi <= target
    then work (hi : path) (target - hi) (tail items) succ (\ () ->
         work path target (tail items) succ fail)
     else work path target (tail items) succ fail
   let results = ref []
   in work [] target items (\ path fail -> results := path : !results;
                                    fail ())
                        (\ () -> return !results)
```

17.3 Motivation: Coroutines

Coroutines are program components like subroutines. They differ from the latter in that they do not impose a caller-callee hierarchy on the components. Instead coroutines come as a set of rountines that exist on equal footing. Exactly one of them is active at each time while the others are sleeping. They do not call one another but the active coroutine can yield control (inclusive parameter passing) to another coroutine, which resumes execution from where that coroutine last yielded control. Passing of control suspends the yielding coroutine and activates the other. The active coroutine may also choose to terminate, in which case the entire set terminates. Coroutines are well suited for implementing programming patterns such as cooperative tasks, iterators, infinite lists, and pipes.

Here is an example where coroutines may be used. Consider to functions, one that reads an input stream and performs a simple run-length decompressions, and one that gobbles characters into tokens (part of a scanner).

```
decompress () {
    while ((c = getchar()) != EOF) {
        if (c == 0xFF) {
            len = getchar();
            c = getchar();
            while (len--)
                emit(c);
        } else
            emit(c);
    }
    emit(EOF);
}

scanner () {
    while ((c = getchar()) != EOF) {
        if (isalpha(c)) {
            do {
                add_to_token(c);
                c = getchar();
            } while (isalpha(c));
            got_token(WORD);
        }
        add_to_token(c);
        got_token(PUNCT);
    }
}
```

Both pieces of code are very simple and easy to understand. But now suppose we wish to construct a scanner that works on compressed documents. The usual approach is to rewrite one of the functions so that it can call (can be called) from the other: either rewrite the scanner so that it takes one character at a time (at the expense of keeping local state across invocations of the scanner) or rewrite the decompressor so that it returns one character at the time (keeping local state across invocations of the decompressor). However, using coroutines we might write the following code.

```
scanner (COROUTINE producer) {
    while ((c = yield(producer)) != EOF) {
        if (isalpha(c)) {
            do {
                add_to_token(c);
                c = yield(producer);
            } while (isalpha(c));
            got_token(WORD);
        }
        add_to_token(c);
        got_token(PUNCT);
    }
}
```

The decompression function is changed to perform the job of the driver:

```
decompress (COROUTINE consumer) {
    while ((c = getchar ()) != EOF) {
        if (c == 0xFF) {
            len = getchar();
            c = getchar();
            while (len--)
                yield(consumer, c);
        } else
            yield(consumer, c);
    }
    yield(consumer, EOF);
}
```

Those two pieces of functionality need just a bit of external glue to be put together.

```
run (COROUTINE producer, COROUTINE consumer) {
  do {
    c = yield (producer);
    yield (consumer, c);
  } while (c != EOF);
}

...
COROUTINE producer = make_coroutine (decompress);
COROUTINE consumer = make_coroutine (scanner);
COROUTINE driver = make_coroutine (run);

driver (producer, consumer);
...
```

Coroutines may be readily implemented using `call/cc`. The basic idea is to represent each coroutine state by the coroutine's current continuation.

```
running = ref (ref Nothing)
make_coroutine f =
  let mycont = ref Nothing
  in  \ x ->
      running := Just mycont
      case cont of
        Nothing -> f x
        Just ff -> ff x
yield g y =
  call/cc (\ resume ->
    !running := Just resume;
    g y)
```

Exercise: Can you reduce the implementation of `yield` to a function call, *e.g.*, `yield g y = g y`?

17.4 Motivation: Threads

Threads are program components that may be executed concurrently and that run on a shared state. Threads come in several variants

- native threads vs. simulated threads. The former rely on the operating system and may be executed on different processors. The latter simulate concurrency inside of a sequential process.
- preemptive vs. cooperative. In each thread implementation, a scheduler determines which thread becomes active next. With preemption, the scheduler runs at regular time intervals. It suspends the currently active thread and selects another thread from a pool of suspended threads to run in the next time slice. With cooperative threading, a thread remains active until it explicitly relinquishes control or until it gets blocked due to an I/O operation.

Having `call/cc` as a primitive enables a simple user-level implementation of cooperative threads. In contrast to a coroutine, a thread does not explicitly yield control to another thread but leaves that decision to the scheduler. In addition, threads communicate exclusively via shared state. They cannot receive parameters or return values while they are running.

A typical thread interface offers the following functionality.

```
spawn :: (Unit -> Unit) -> Thread
yield :: Unit -> Unit
terminate :: Unit -> Unit
```

```
currentThread = NULL
runQueue = emptyQueue

spawn f =
  enqueue (runQueue, makeThread f)

makeThread f =
  { cont =
      \ () ->
        f (); terminate ()
    ...
  }

terminate () =
  scheduleThread (dequeue (runQueue))

scheduleThread (thread) =
  currentThread = thread;
  currentThread.cont ()

yield () =
  call/cc (\ myself ->
    currentThread.cont = myself;
    enqueue (runQueue, currentThread);
    scheduleThread (dequeue (runQueue));
    )
```

To implement a pair of cooperating functions like scanner and decompress with threads requires communication via shared state. Before looking at means to achieve that, we first look at an implementation of call/cc.

17.5 Implementing first-class continuations

The easiest way of implementing first-class continuations is via an interpreter. Let's consider the following language.

$$e \quad ::= \quad x \mid \lambda x.e \mid e\ e \mid \mathtt{0} \mid e{+}e \mid \mathit{if}\ e\ e\ e \mid \mathtt{call/cc}$$

The interpretation is inspired by denotational semantics.

$$\begin{array}{rlcl}
y \in & \mathsf{Val} & = & \mathbf{Z} + (\mathsf{Val} \to \mathsf{Comp}) \\
\kappa \in & \mathsf{Cont} & = & \mathsf{Val} \to \mathsf{Answer} \\
 & \mathsf{Comp} & = & \mathsf{Cont} \to \mathsf{Answer} \\
\rho \in & \mathsf{Env} & = & \mathsf{Var} \to \mathsf{Val} \\
 & \mathcal{E} & : & \mathsf{Exp} \to \mathsf{Env} \to \mathsf{Comp}
\end{array}$$

$$\begin{array}{lcl}
\mathcal{E}[\![x]\!]\rho\kappa & = & \kappa(\rho(x)) \\
\mathcal{E}[\![\lambda x.e]\!]\rho\kappa & = & \kappa(\lambda y.\mathcal{E}[\![e]\!]\rho[x \mapsto y]) \\
\mathcal{E}[\![e_1\ e_2]\!]\rho\kappa & = & \mathcal{E}[\![e_1]\!]\rho(\lambda y_1.\mathcal{E}[\![e_2]\!]\rho(\lambda y_2.y_1\ y_2\ \kappa)) \\
\mathcal{E}[\![\mathsf{0}]\!]\rho\kappa & = & \kappa(\mathsf{0}) \\
\mathcal{E}[\![e_1{+}e_2]\!]\rho\kappa & = & \mathcal{E}[\![e_1]\!]\rho(\lambda y_1.\mathcal{E}[\![e_2]\!]\rho(\lambda y_2.y_1{+}y_2)) \\
\mathcal{E}[\![\mathit{if}\ e_1\ e_2\ e_2]\!]\rho\kappa & = & \mathcal{E}[\![e_1]\!]\rho(\lambda y.\mathtt{if}\ y\ (\mathcal{E}[\![e_2]\!]\rho\kappa)\ (\mathcal{E}[\![e_3]\!]\rho\kappa)) \\
\mathcal{E}[\![\mathtt{call/cc}]\!]\rho\kappa & = & \kappa(\lambda f.\lambda\kappa.f(\lambda y.\lambda\kappa'.\kappa y))
\end{array}$$

Notice that the interpreter is written in continuation-passing style. Moreover, it is written in such a way that the underlying program is evaluated using call-by-value, regardless of the evaluation-strategy used for evaluating the interpreter.

Alternatively, we may transform a program with `call/cc` into one without by applying a CPS transformation. The first published formal study of such a transformation is due to Plotkin. However, his specification makes it hard to prove its properties. Hence, we'll also consider a version due to Danvy and Filinski which simplifies the proofs a lot.

18 The CPS Transformation

Continuations have also become important in compiler construction so that CPS warrants a closer look and more systematic study. Indeed, understanding CPS is a prerequisite to many newer research papers in compiler construction. At the heart of CPS is the *CPS transformation* which transforms a term in the lambda calculus into one using explicit continuations.

The presentation here follows the work by Danvy and Filinski [1].

For expository purposes, we will revert to the pure lambda calculus. All results will carry over smoothly to its applied variants. However, the notation of application (formerly $e_1\ e_2$) will change to $@e_1\ e_2$.

18.1 Classical CPS transformation

The central idea of the classical CPS transformation is by Plotkin and Fischer [2, 3]. Recall that the interpreter used abstractions to represent continuations. The CPS transformation does the same.

Definition 56 (Fischer/Plotkin Call-by-Value CPS Transformation) It is a function $[\![\_]\!] : E \to E$:

$$\begin{aligned}
[\![x]\!] &:= \lambda k.@\ k\ x \\
[\![\lambda x.e]\!] &:= \lambda k.@\ k\ (\lambda x.[\![e]\!]) \\
[\![@\ e_1\ e_2]\!] &:= \lambda k.@\ [\![e_1]\!]\ (\lambda v_1.@\ [\![e_2]\!]\ (\lambda v_2.@\ (@\ v_1\ v_2)\ k)) \quad (v_1,\ v_2 \text{ fresh})
\end{aligned}$$

□

The Fischer/Plotkin CPS transformation is simple enough, and the following statement states its correctness with respect to call-by-value evaluation:

Theorem 14 (Simulation) *Let e be a lambda term. Let furthermore* eval_v *be call-by-value evaluation.*

$$\mathrm{eval}_v(e) = \mathrm{eval}_v(@\ [\![e]\!](\lambda x.x))$$

□

Moreover, the CPS transformation has a pleasant side effect:

Theorem 15 (Indifference) *Let e be a lambda term. Let furthermore* eval_v *be call-by-value evaluation and* eval_n *be call-by-name evaluation.*

$$\mathrm{eval}_n(@[\![e]\!](\lambda x.x)) = \mathrm{eval}_v(@[\![e]\!](\lambda x.x))$$

□

The consequence of the indifference property is that the CPS-transformed term is indifferent to the evaluation strategy: Call-by-name and call-by-value will produce the same result on a CPS term. Consequently, the CPS interpreter is now independent of the evaluation strategy of the metalanguage, which brings it a significant step closer to being definitional.

18.2 Avoiding administrative β redexes

Unfortunately, the Fischer/Plotkin CPS transformation is not suitable for direct application in realistic compilers: it produces humungous result terms. For example, the CPS version of $@(\lambda x.x)(@y\, y)$ is this:

$$\begin{aligned}&\lambda k.@(\lambda k.@\, k(\lambda x.\lambda k.@k\, x))\\&\qquad(\lambda m.@(\lambda k.@(\lambda k.@k\, y)(\lambda m.@(\lambda k.@k\, y)(\lambda n.@(@m\, n)\, k)))(\lambda n.@(@m\, n)\, k))\end{aligned}$$

This term contains a large number of β redexes—in addition to the *beta* redex already present in the original term. Reducing those *administrative redexes* leads to the following, much more acceptable term:

$$\lambda k.@(y\, y)\ (\lambda a.@(@(\lambda x.\lambda k.(@k\, x))\ a)\ (\lambda a.@k\, a))$$

Hence, for practical intents and purposes, the Fischer/Plotkin CPS transformation needs to be accompanied by a post-reducer which removes the β reductions introduced by the vanilla transformation. This approach has the disadvantage that it still constructs the intermediate, large CPS term only to replace it immediately by something much smaller. It is much more desirable to compute the final result directly without large intermediate terms.

The method to achieve this "on-the-fly" post-reduction is to classify the abstractions and applications on the right-hand sides of the transformation into those which will be part of an administrative β redex and those which will not. With the straightforward Fischer/Plotkin transformation, this is not possible—some abstractions and applications sometimes do take part in adminstrative redexes, and sometimes do not. However, it is possible to perform η expansion on the right-hand sides in a few, select instances, and then perform the classification.

The new transformation resulting from this has annotations on each λ and each @ indicating its classification: $\overline{\lambda}$ is for *static* abstractions that are part of administrative redexes (and therefore do not show up in the result term), and $\underline{\lambda}$ is for *dynamic* abstractions which definitely are part of the transformed term. Analogously, $\underline{@}$ denotes a dynamic application, and $\overline{@}$ a static one. The reformulation of the transformation is due to Danvy and Filinski [1], hence:

Definition 57 (Danvy/Filinski CPS Transformation) Let e be a lambda term. The *Danvy/Filinski CPS Transformation* is a function $\llbracket\_\rrbracket : E \to (E \to E) \to E$:

$$\begin{aligned}\llbracket x\rrbracket &:= \overline{\lambda}\kappa.\overline{@}\kappa\, x\\ \llbracket\lambda x.e\rrbracket &:= \overline{\lambda}\kappa.\overline{@}\kappa(\underline{\lambda}x.\underline{\lambda}k.(\overline{@}\llbracket e\rrbracket(\overline{\lambda}v.\underline{@}k\, v))\\ \llbracket @e_1\, e_2\rrbracket &:= \overline{\lambda}\kappa.\overline{@}\llbracket e_1\rrbracket\ (\overline{\lambda}v_1.\overline{@}\llbracket e_2\rrbracket\ (\overline{\lambda}v_2.\underline{@}(\underline{@}v_1\, v_2)\ (\underline{\lambda}a.\overline{@}\kappa\, a)))\end{aligned}$$

□

Note that, in this definition, κ stands for a continuation *at transformation time*; only k ever appears in the output of the transformation

The corresponding correctness statement is this:

Theorem 16 *For a lambda term e, $\underline{\lambda}\kappa.\overline{@}[\![e]\!](\overline{\lambda}v.\underline{@}\kappa\, v)$ is $\beta\eta$-equivalent to the corresponding result term of the Fischer/Plotkin transformation.* □

The implementation of the Danvy/Filinski transformation is straightforward: Static abstractions and applications become the corresponding constructs in the metalanguage, and their dynamic counterparts become syntax constructors.

The introduction of additional η redexes allows for the classification of the applications and abstractions of a lambda term. Mostly, they participate in administrative redexes and therefore do not show up in the resulting term. However, there is an exception:

$$\overline{@}[\![\lambda f.@f\ x]\!](\overline{\lambda}v.v) = \lambda f.\lambda\kappa.@(@f\ x)\ (\lambda a.@\kappa\, a)$$

The residual term still contains an η redex introduced by the CPS transformation. This has potentially serious consequences, as the application in the original term is a tail call. Some modern programming languages based on the lambda calculus (most notably Scheme) demand that tail calls do not create new continuations. However, the above η redex is just that.

Therefore, it is necessary to augment the transformation to guard against this case. The idea is to duplicate the transformation rules: A new version of the rules is for "trivial" continuations of the form $\overline{\lambda}v.\underline{@}\kappa\, v$; the new rules avoid building the redex in the residual rules. The copied rules are called $[\![\_]\!]'$ in the following definition.

Definition 58 (Tail-Recursive Danvy/Filinski CPS Transformation) Let e be a lambda term. The *tail-recursive Danvy/Filinski CPS Transformation* is a function $[\![\_]\!] : E \to (E \to E) \to E$:

$$\begin{aligned}
[\![x]\!] &:= \overline{\lambda}\kappa.\overline{@}\kappa\ x\\
[\![\lambda x.e]\!] &:= \overline{\lambda}\kappa.\overline{@}\kappa(\underline{\lambda}x.\underline{\lambda}k.(\overline{@}[\![e]\!]'k)\\
[\![@e_1\ e_2]\!] &:= \overline{\lambda}\kappa.\overline{@}[\![e_1]\!]\ (\overline{\lambda}v_1.\overline{@}[\![e_2]\!]\ (\overline{\lambda}v_2.\underline{@}(\underline{@}v_1\ v_2)\ (\underline{\lambda}a.\overline{@}\kappa\ a)))\\[1ex]
[\![x]\!]' &:= \overline{\lambda}k.\underline{@}k\ x\\
[\![\lambda x.e]\!]' &:= \overline{\lambda}k.\underline{@}k(\underline{\lambda}x.\underline{\lambda}k.(\overline{@}[\![e]\!]'k)\\
[\![@e_1\ e_2]\!] &:= \overline{\lambda}k.\overline{@}[\![e_1]\!]\ (\overline{\lambda}v_1.\overline{@}[\![e_2]\!]\ (\overline{\lambda}v_2.\underline{@}(\underline{@}v_1\ v_2)\ k))
\end{aligned}$$

□

Note that Theorem 16 requires a slight reformulation: The result of transforming a term e into CPS in a dynamic context is given by $\underline{\lambda}\kappa.\overline{@}[\![e]\!]'\kappa$.

19 Concurrent Access to Shared Resources

This section relies on material from Concepts, Techniques, and Models of Computer Programming by Peter Van Roy and Saif Haridi (to appear with MIT Press).

To coordinate currently running threads it is not sufficient to pass a parameter at the beginning and reap a result at the end. Instead, the threads must communicate while they are running, but at the same time be as independent as possible.

The simple-minded approach of letting threads access shared state without further precaution is doomed, as the following example shows. The underlying assumption is that assignment and read operations are *atomic*, that is, they cannot be interrupted in transit.

```
var m

thread A {
  x = get m;     // A1
  set m (x+1);  // A2
  y = get m;     // A3
  print y;       // A4
}


  ...
  m = 0;
  spawn(A);
  spawn(A);
  ...
```

Depending on the exact interleaving of the operations, the program may print out any of the following four different results (with B denoting the second thread):

```
1 1     // A1, B1, [A2, A3, A4, B2, B3, B4] any interleaving
1 2     // A1, A2, A3, [A4, B1, B2, B3, B4] any interleaving
2 1     // A1, A2, A3, B1, B2, B3, B4, A4
2 2     // A1, A2, B1, B2, [A3, A4, B3, B4] any interleaving
```

Since the result depends on the interleaving, it is very hard to write correct programs using this model. Nevertheless, there are algorithms (*e.g.*, Dekker's algorithm) that guarantee *mutual exclusion*: each thread has designated *critical regions* in which it modifies shared state. The job of a mutual exclusion algorithm is to ensure that

1. at any time, at most one thread is in a critical region,
2. if more than one thread attempts to enter a critical region at the same time, one of the threads is elected to proceed (and the others are blocked) in a finite amount of time,

3. stopping a thread outside its critical region does not influence the other threads.

However, algorithms like Dekker's are too low-level. They are unflexible to use and hard to reason about. Hence, programming languages have facilities (or libraries) to ease the management of concurrent access to shared memory cells.

Suppose, we wish to implement a stack datastructure for use in concurrent threads. Let's start naively:

```
let stack = ref nil
    push (x) {
      s = !stack;
      stack := cons x s
    }
    pop () {
      case !stack of
        x:s -> stack := s; return x
        nil -> raise StackEmpty
    }
in  { push = push, pop = pop }
```

Unfortunately, there is no guarantee that `stack` does not change between `s = !stack` and the following update operation in `push`, so that stack entries may be lost. Similarly, there is no guarantee that `stack` does not change between the match and the update in `pop` so that a stack entry may be popped multiple times.

There are three general techniques for avoiding such incidents: locks, monitors, and transactions.

- Locks allow grouping of atomic operations to larger atomic operations.
- Monitors extend locks with *wait points* where threads may be parked just outside the lock.
- Transactions refine locks to have to possible exits: a normal one and an exceptional one. The latter can be taken at any time with the result that the transaction has no effect on the state.

19.1 Locks

A lock guards the access of a thread to a critical region. Each lock should be responsible for a shared resource which may be internal to the program or external. For internal resources (like objects implementing some service) guarantees can be given about the behavior of the program. For external resources, locks can be created dynamically and it is the programmers responsibility to match critical regions and the shared resources accessed in them with their locks.

A typical API for locks:

- `newLock :  () -> Lock`
  creates a new lock
- `isLock :  Object -> Bool`
  checks if an object is a lock.
- `withLock :  Lock -> (Unit -> X) -> X` "guarded statement"
  runs the given thunk under control of the lock. That is, `withLock l th` blocks until there is no thread active inside a region protected with lock `l`. Then one of the blocked threads is chosen to execute its thunk.

A program may have many guarded statements with the same lock. Typically, locks are *reentrant*, that is, if the thread that already has the lock attempts to execute another guarded statement for the same lock, then it is immediately granted access. This is useful if an public operation calls another public operation that requires access to the same resource.

19.1.1 Concurrent Stack ADT

Here is an implementation of the stack using locks. It creates the lock object at the same time as the stack and uses guarded statements to lump together the operations on the `stack` cell.

```
let stack = ref nil
    l = newLock ()
    push (x) {
      withLock (l, function () {
                     s = !stack;
                     stack := cons x s
                   })
    }
    pop () {
      withLock (l, function () {
      case !stack of
        x:s -> stack := s; return x
        nil -> raise StackEmpty
      })
    }
in  { push = push, pop = pop }
```

Note that the implementation of exceptions must be integrated with lock management. Since raising the exception escapes from the scope of the guarded statement, the implementation of `withLock` must capture all exceptions, release the lock, and the reraise the exception.

19.1.2 Tuple Spaces

David Gelernter has invented an abstraction called "Tuple Spaces" for concurrent programming. It provides a coordination model between concurrently execution threads and can be added to any programming language.

A tuples space consists of a multi-set of tuples, `TS`, with three operations

- `write :: TupleSpace -> Tuple -> Unit`
  adds a tuple to the tuple space
- `read :: TupleSpace -> Pattern -> Tuple`
  blocks until the tuple space contains a tuple matching `Pattern`. Then it removes the tuples from the space and returns it
- `readNonBlock :: TupleSpace -> Pattern -> Maybe Tuple`
  If the tuple space contains a tuple `t` matching `Pattern`, then `t` is removed from the space and the result is `Just t`. Otherwise, the result is `Nothing`.

Hence, tuple spaces have two important properties:

1. They are content-addressed.
2. They are asynchronous in that readers are decoupled from writers.

Here is an implementation of the stack with a tuple space.

```
let ts = newTupleSpace ()
    push (x) {
      let stack (s) = read ts (stack (X))
      in  write ts (stack (cons x s))
    }
    pop () {
      let stack (s) = read ts (stack (cons X S))
      case s of
        x:s -> write ts (stack s); return x
        nil -> raise StackEmpty                // won't happen
    }
in write (ts, stack (nil));
   return { push = push, pop = pop }
```

Note that the `pop` operation never raises an exception. Instead, due to the pattern, it blocks until a value is available.

A tuple space may be implemented with a lock, a dictionary, and a queue datatype. The idea is that the tuples are stored in the dictionary with queues containing entries that match the same pattern. A single lock governs access to the dictionary and the queues contained within.

19.1.3 Implementing Locks

Implementing a lock requires an atomic `exchange` operation for a memory cell. Such an operation is provided by all common processors.

```
type State = Empty | Occupied
newLock () {
  return { state = ref Empty }
}
withLock (l, thunk) {
  let oldstate = ref Occupied in
  while (1) {
    exchange( oldstate, l.state);
    if (!oldstate == Empty) {
      try {
        r = thunk ();
        return r;
      } finally {
        oldstate := Empty;
        exchange( oldstate, l.state);
      }
    } else {
      yield ();
    }
  }
}
```

This implementation does not guarantee fairness, *i.e.*, that every thread attempting to obtain the lock eventually gets it. It is also expensive in that a thread may check the state over and over again even if the state has not changed. For that reason, the implementation of locks is usually entangled with the implementation of threads. Each lock has a separate queue of threads waiting for the lock. If a thread cannot immediately be granted the lock, then the thread is suspended and put on the lock's queue (**not** on the scheduler's general run queue). Whenever the lock is released, the top entry of the lock's queue is moved to the run queue.

To obtain reentrant locks requires an additional field for storing the current thread in the lock data structure. If the lock is occupied, then `withLock` first checks if it is occupied by the same thread and, if so, the thunk is started immediately.

19.2 Monitors

Locks alone do not provide enough information, in many cases. For example, consider an ADT implementing a concurrent, bounded buffer. A thread that wants to deposit an element in the buffer may discover that the buffer is full (and the be unable to proceed) even though no other thread is currently accessing the buffer. While the latter can be regulated with a lock, the former requires a way to block a thread until a certain condition is fulfilled. This condition may be communicated in the form of a message that a reader of the buffer sends to a potential writer.

The standard abstraction for this situation is a *monitor*. It has been introduced by Per Brinch-Hansen and further developed by Hoare. Monitors are also built into the Java language.

A monitor is a lock extended with program control over how waiting threads enter and exit the lock. A monitor may be structured in one or more queues of waiting threads. We'll consider the simpler case (one queue) first.

A monitor adds two new operations, `wait` and `notify`, to the API of locks. Both are only acceptable from inside a guarded statement (this can be guaranteed by the compiler if monitors are built into the language). In one variant, a monitor consists of an internal lock and a set of waiting threads (the *wait set*).

- `wait :  Unit -> Unit`

  - suspends the current thread and
  - puts it in the monitor's internal wait set
  - releases the lock

- `notify :  Unit -> Unit`

  - selects a thread `t` from the internal wait set
  - places `t` on the queue for the lock

- `notifyAll :  Unit -> Unit`
  performs `notify` for all threads in the internal wait set

As an example, here is the code for a sequential bounded buffer.

```
create (n) {
  let buf = newArray n 0
      first = ref 0
      last = ref 0
  in
  { put = function (x) {
            if ((!last+1) % n == !first) { raise BufferFull }
            buf [!first] := x;
            first := !first + 1;
          },
    get = function () {
            if (!last == !first) { raise BufferEmpty }
            let x = buf [!last]
            in  last := (!last+1) % n;
                return x
          }
  }
}
```

With a monitor, instead of raising an exception, the implementation could wait until the condition is fulfulled. This implementation gives raise to a typical programming pattern where the body of each operation is guarded by the monitor and starts with waiting until a precondition is fulfilled.

```
create (n) {
  let buf = newArray n 0
      first = ref 0
      last = ref 0
      m = newMonitor ()
  in
  { put = function (x) {
          withMonitor m (function () {
            while ((!last+1) % n == !first) {
              wait (m);
            }
            buf [!first] := x;
            first := !first + 1;
            notifyAll (m)
          })},
    get = function () {
          withMonitor m (function () {
            while (!last == !first) {
              wait (m);
            }
            let x = buf [!last]
            in  last := (!last+1) % n;
                notifyAll (m);
                return x
          })}
  }
}
```

A variation of the semantics of a monitor is to have `notify` remove a thread from the wait set and then immediately pass the lock to that thread. Since there is now a fixed association between occurrences of `notify` and preconditions, it is common in this variation to have multiple wait sets each of which comes with a *condition variable* (with operations `wait` and `notify`). In the bounded buffer example, there would be two condition variables, one for the buffer full condition and another for buffer empty.

19.3 Transactions

A transaction is any operation that satisfies the ACID properties:

- A for *atomic*: no intermediate states of a transaction's execution are observable. A transaction either *commits* (finishes successfully and the changed state becomes visible instantaeously) or it *aborts*, in which case the whole transaction has no effect on the state.

- C for *consistent*: observable states respect the system's invariants. This is a responsibility of the programmer.
- I for *isolation* (serializability): several transaction can take place concurrently without inference.
- D for *durability* (persistency): observable state changes survive system crashes.

Outside the database world, not all four properties of transactions are always required. Concepts like durability and atomicity also make sense by themselves. Also subsets are possible. For example, an operation satisfying ACI may be called a *lightweight transaction*.

A typical programming setting might be an operation that throws an exception in the middle of some state transformation. The problem is that the current state may not be consistent because the exception occured in the middle of some reorganization. This might be handled in the following ways.

- The caller is responsible for cleaning up and getting back to a consistent state.
- The operation may be put inside a transaction where raising an exception correponsinds to aborting the transaction.

Another motivation is fault tolerance, where transactions can help to contain the effect of a fault in a limited part of the application.

Transaction may also be nested to gain the same results for smaller units inside a larger transaction.

As already stated, some of the transactional properties can be useful on their own. This and how they can be put back together to implement full transactions is explained in greater detail in the paper: Composing first-class transactions. Nicholas Haines, Darrell Kindred, J. Gregory Morrisett, Scott M. Nettles, Jeannette M. Wing. ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 16 Issue 6, November 1994.

19.3.1 Persistence

A persistent value is one that outlives the computation that created it. In particular, a persistent value will survive a crash. In the *orthogonal persistence* model, all data reachable by pointer dereferencing from a distinguished location, the persistent root, are persistent. When a persistency transaction terminates, all persistent data modified by the transaction are atomically saved to stable storage. If a crash occurs during the transaction, the last committed state is recoverable from stable storage. All data not reachable from the persistent root are lost. Conceptually, a crash aborts all top-level transactions (of any flavor) and terminates all threads, so there is no mechanism for a persist-only transaction to abort in isolation. A variety of approaches can be taken to guarantee that the effects of top-level transactions on stable storage are atomic. For example, the effects of nested transactions may be made permanent only when the enclosing top-level transaction commits. This approach simplifies crash recovery but assumes that the number of modifications done by nested transactions is relatively small. An alternative approach would make nested transactions effects permanent when they commit, but then crash recovery would have to undo such effects.

19.3.2 Undoability

A top-level undo-only transaction has no special effect if it commits. If it aborts then all changes it made to the store are undone. One possible semantics for undo reverts changes to volatile data in addition to changes to persistent data. At the start of an undo-only transaction (conceptually) a checkpoint of the store is made. If the transaction terminates successfully, then nothing unusual happens; if not, then the effects of the transaction are rolled back to the checkpointed state, at which point a possibly different computation can begin. Undo-only transactions may commit or abort regardless of whether they are nested. However, since a nested transaction's commit is relative to the action of its parent, if the parent aborts then the effects of the (committed) nested transaction must be undone along with the parent's other changes. Thus, when a child transaction commits it hands back (antiinherits) to its parent its set of changes to the store.

19.3.3 Locks

Two-phase reader/writer (R/ W) locks are a well-known mechanism for ensuring serializability. Alone, they provide no support for commit or abort. A transaction acquires a R/W lock and holds it until the transaction commits or aborts, thereby avoiding the problem of cascading aborts. Write locks guarantee that any two concurrent transactions modify disjoint sets of data in the store, unless one is a descendant of the other. Under Moss's standard locking rules for nested transactions [Moss 1985], transactions acquire locks subject to the following rules:

- A transaction may acquire a read lock if all writers are ancestors of the transaction.
- A transaction may acquire a write lock if all readers and writers are ancestors of the transaction.
- When a transaction commits, all its locks are antiinherited, *i.e.*, handed off to the parent or released if the transaction is top-level. If the transaction aborts, all its locks are released.

However, a parent transaction may run concurrently with its children, so we must check that the read (or write) condition holds not only when a lock is acquired, but also every time the transaction reads (or writes) the associated data object. This check is reasonable for programs, which use mutable data infrequently.

19.3.4 APIs

```
signature PERS =
  sig val persist : (a -> b) -> a -> b
      val bind : identifier * a -> unit
      val unbind : identifier -> unit
      val retrieve : identifier -> a
      ...
  end
```

The function `(persist f)` behaves in the save way as `f`, extensionally. However, once it has terminated, all changes that `f` made to the persistent store (via `bind`) are saved to stable storage.

```
signature UNDD =
  sig val undoably : (a-> b) -> a-> b
      exception Restore of exn
      ...
  end
```

The function `(undoably f)` behaves like unless the exception `Restore` is raised inside `f`. In that case, *all modifications to the store since f was entered are reverted.* The implementation logs all store operations and replays the log in reverse order when the exception is raised. In case of a nested transaction, the undo log of a child is spliced into the parent's log.

```
signature RW_REF =
  sig type RW_Ref (a)
      type Lock
      exception Read_Not_Held
      exception Write_Not_Held
      val create_rw_ref : a * Lock -> RW_Ref (a)
      val rw_get        : RW_Ref (a) -> a
      val rw_set        : RW_Ref (a) -> a -> Unit
      val lock_of       : RW_Ref (a) -> Lock
      ...
  end
```

Creates reference cells that are under the control of a lock. Locks may be acquired for reading and for writing. Locks are released implicitly at the end of the transaction (or antiinherited to the parent in case of a nested transaction).

References

[1] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2:361–391, 1992.

[2] Michael J. Fischer. Lambda-calculus schemata. *Lisp and Symbolic Computation*, 6(3/4):259–288, 1993.

[3] Gordon Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.