# Programming Languages: Theory and Practice

Robert Harper
Carnegie Mellon University

Spring, 2000

# Contents

# Chapter 1

# Inductive Definitions

## 1.1 Introduction

It is common in the study of programming languages to define a set by a collection of *rules* that define the members of that set. Each rule has zero or more *premises*, or *requirements*, and one *conclusion*.

**Example 1 (Numerals, first attempt)**
*We may define the set of unary numerals (i.e., numerals in base 1) for the natural numbers as follows:*

1. **Zero** *is a numeral;*

2. *if* $n$ *is a numeral, then* $\mathbf{Succ}(n)$ *is also a numeral.*

*Equivalently, we might say that the set* $\mathrm{N}$ *of numerals is defined by the following rules:*

1. $\mathbf{Zero} \in \mathrm{N}$*;*

2. *if* $n \in \mathrm{N}$*, then* $\mathbf{Succ}(n) \in \mathrm{N}$.

*Observe that in each formulation the first rule has no premises, whereas the second has one.*

**Example 2 (Binary Trees, first attempt)**
*We may define the set of binary trees by the following rules:*

1. *The empty tree,* **Empty***, is a binary tree.*

2. *If* $t_l$ *and* $t_r$ *are binary trees, then so is* $\mathbf{Node}(t_l, t_r)$.

*Equivalently, we might say that the set* $\mathbf{T}$ *of binary trees is defined by the following rules:*

1. $\mathbf{Empty} \in \mathbf{T}$*;*
2. *if* $t_l \in \mathbf{T}$ *and* $t_r \in \mathbf{T}$*, then* $\mathbf{Node}(t_1, t_2) \in \mathbf{T}$.

*The first rule has no premises; the second has two.*

When thinking about these rules you might well ask yourself what exactly is a node, or what are zero and successor? These are examples of *value constructors*, the "atoms" out of which we build "molecular" data structures. Technically we should say precisely what we mean by these constants and constructors, but for now we'll just assume that there is some way to define them, say in terms of sets.

In ML value constructors are a primitive concept. They are introduced by *datatype* declarations such as these, which closely parallel the definitions given above.

```
datatype num = Zero | Succ of num
datatype tree = Empty | Node of tree * tree
```

These declarations introduce the types `num` and `tree`, respectively, in terms of the given value constructors. Each constructor implicitly defines a "rule" for membership in that type. The constructors `Zero` and `Empty` have no arguments; this corresponds to a rule with no premises. The constructor `Succ` has one argument, corresponding to the single premise in the inductive definition; the constructor `Node` has two arguments, corresponding to the two premises in the inductive definition.

**Exercise 3**
*Give a collection of rules defining the set of strings over an alphabet* $\Sigma$*. Give a corresponding ML* `datatype` *declaration for the same set.*

When we say that a set is defined by a set of rules, what precisely do we mean? Which set do we consider to be defined by those rules? To see why this is an important question, consider the set

$$\mathbf{N}^* = \{\, \mathbf{Zero}, \mathbf{Succ}(\mathbf{Zero}), \mathbf{Succ}(\mathbf{Succ}(\mathbf{Zero})), \ldots \} \cup \{\, \infty \,\},$$

where $\infty = \mathbf{Succ}(\mathbf{Succ}(\ldots))$ is an "infinite" stack of successors, which satisfies the equation $\mathbf{Succ}(\infty) = \infty$. Observe that $\mathbf{Zero} \in \mathbf{N}^*$ and that if $n \in \mathbf{N}^*$, then $\mathbf{Succ}(n) \in \mathbf{N}^*$ — that is, $\mathbf{N}^*$ meets the requirements that "define" $\mathbf{N}$. Yet clearly $\mathbf{N}^* \supsetneq \mathbf{N}$, and hence these requirements do not, by themselves, pick out a unique set.

Incidentally, if you're skeptical about whethere there exists an element $\infty$ described above, here's a way to define it in ML, using a "lazy" `datatype` declaration:[1]

```
datatype lazy num_star = Zero | Succ of num_star
```

Here's $\infty$, spelled out in ML code:

```
val rec lazy infty = Succ infty
```

Here's the predecessor function:

```
fun pred (Succ n) = n
```

You can check by experimetation that `infty` really is an infinite stack of successors as follows:

```
val Succ infty' = infty
val Succ infty'' = infty'
val Succ infty''' = infty''
```

and so on. Each `val` binding strips off one level of `Succ`, leaving another "copy" of infinity behind — $\infty - 1 = \infty$!

For a collection of rules rules to define a set, we must say something more than the set must obey these rules. We need an *extremal clause* that states that "nothing else is in the set except those elements that are required to be there by the rules." This may sound like a bit of legalese, but mathematically it is essential to include the extremal clause in order to rule out unintended interpretations of the rules such as we say with the set $\mathbf{N}^*$. Thus, the definition of $\mathbf{N}$ should really be stated as follows:

**Example 4 (Numerals, revised)**
*The set $\mathbf{N}$ is defined by the following rules:*

1. **Zero** *is a numeral;*

2. *if $n$ is a numeral, then $\mathbf{Succ}(n)$ is also a numeral;*

3. *nothing else is a numeral.*

---

[1]This is valid only for SML/NJ version 110.9 or later. To enable the lazy evaluation primitives, enter `Compiler.Control.lazysml := true` at top level.

*Equivalently, we say that* $\mathbf{N}$ *is the* least, *or* smallest, *set defined by the first two rules, by which we mean precisely that nothing else is in the set except as is forced by the rules.*

To see that $\mathbf{N}^*$ is ruled out by the extremal clause, observe that $\infty$ has no business being in the specified set because it is not forced to be in there by the rules. More precisely, since $\mathbf{N} \subsetneq \mathbf{N}^*$, it follows that $\mathbf{N}^*$ is not the least set closed under the rules for **Zero** and **Succ**(), and hence $\mathbf{N}^*$ is ruled out as a possible interpretation of the preceding definition.

Similarly, we may revise the definition of binary trees by adding an extremal clause as follows.

**Example 5 (Binary Trees, revised)**
*We may define the set of binary trees by the following rules:*

1. *The empty tree,* **Empty**, *is a binary tree.*

2. *If* $t_l$ *and* $t_r$ *are binary trees, then so is* $\mathbf{Node}(t_l, t_r)$.

*Equivalently, we might say that the set* $\mathbf{T}$ *of binary trees is defined by the following rules:*

1. $\mathbf{Empty} \in \mathbf{T}$;

2. *if* $t_l \in \mathbf{T}$ *and* $t_r \in \mathbf{T}$, *then* $\mathbf{Node}(t_1, t_2) \in \mathbf{T}$;

3. *nothing else is a binary tree.*

*The first rule has no premises; the second has two.*

In the presence of the extremal clause, a collection of rules of the kind given above determines a unique set. In practice we do not explicitly state the extremal clause, but rather we state that the set in question is *inductively defined* by a given collection of rules. For example, we may say that the set $\mathbf{N}$ is inductively defined by the two rules membership rules given at the beginning of these notes. In doing so we are implicitly stating that nothing else is to be a member of that set unless it is "forced" to be there by the rules.

Since the smallest set inductively defined by a set of rules contains only those elements explicitly required by the rules, we may envision elements of this set as being built up in "stages". At the first stage we introduce elements that are forced to be in the set by some rule with no premises. (For example, **Empty** would appear at the first stage of the construction of the set $\mathbf{T}$.) Suppose that we have completed stage $s$ of the construction. At stage $s+1$ we add in those elements that are introduced by a rule whose premises are satisfied at stage $s$. (For example, if $t_l$ and $t_r$ are constructed at stage $s$, then $\mathbf{Node}(t_l, t_r)$ is constructed at stage $s+1$.) We may then think of the set inductively defined by these rules as the set of elements that appear at some stage $s$ in the construction.

## 1.2 Rule Induction and Recursion

Why do we say that the least set defined by a collection of rules is "inductively" defined by it? As the terminology suggests, the answer is that there is a close connection with reasoning by mathematical induction. Here's why. Suppose that we wish to prove that every binary tree $t$ has a *height* $h$ satisfying the following two requirements:

1. the height of **Empty** is 0;
2. if $t_l$ and $t_r$ have heights $h_l$ and $h_r$, respectively, then the height of $\mathbf{Node}(t_l, t_r)$ is $1 + \max(h_l, h_r)$.

We call these two conditions the *specification* of the height of a binary tree.

The question is this: how do we know that every binary tree in fact has a height? This might seem like an odd question at first, but consider that the "infinite" binary tree

$$\mathbf{Node}(\mathbf{Node}(\ldots, \ldots), \mathbf{Node}(\ldots, \ldots))$$

has no height in the sense specified! Luckily, the extremal clause rules out such "infinite trees", which makes it possible to assign a height to each binary tree.

How do we prove that every binary tree has a height? That is, how do we prove that for every binary tree $t \in \mathbf{T}$, there exists a natural number $h$ satisfying the specification of the height of a binary tree. Since $\mathbf{T}$ is inductively defined, we proceed by analyzing the rules that define it. By the first rule defining binary trees, $t$ might be **Empty**. But $t$ clearly has a height, namely 0, as given in the specification. By the second rule defining binary trees, $t$ might have the form $\mathbf{Node}(t_l, t_r)$, where $t_l$ and $t_r$ are also binary trees. By induction we may assume that they each have a height, say $h_l$ and $h_r$, respectively. But then the height $h$ of $t$ is uniquely determined by the equation $h = 1 + \max(h_l, h_r)$, as required by the specification. Since $\mathbf{T}$ contains no elements other than these, we may conclude that every binary tree $t \in \mathbf{T}$ has a height $h$.

The principle of proof we have just used is called *proof by rule induction*. The proof proceeds by analyzing the possible forms of $t$ (as given by the rules defining the set of binary trees), and proving that in each case $t$ has a well-defined height. In doing so we may *assume* (for the sake of induction) that the "sub-trees" occurring within $t$ have well-defined heights, which we then use to assign a height to $t$.

All this may seem rather obvious to you. What is important is that you recognize that it is necessary to prove that every tree has a height. Here's another perspective on the same issue, this time drawn from your programming experience. Here's the `height` function written in ML:

```
fun height Empty = 0
  | height (Node (tl, tr)) =
    1 + max (height tl, height tr)
```

Why does this define a function from trees to integers? Given a tree $t$, is there a unique integer $n$ such that `height`$t$ evaluates to $n$? This may be proved by induction on the rules defining trees. If $t$ = `Empty`, then `height(`$t$`)` evaluates to 0. If, on the other hand, $t$ = `Node(`$t_l$`,`$t_r$`)`, then we may assume for the induction that `height(`$t_l$`)` evaluates to $h_l$ and that `height(`$t_r$`)` evaluates to $h_r$. It follows that `height(`$t$`)` evaluates to $1 + \max(h_l, h_r)$, which is uniquely determined from $h_l$ and $h_r$. This completes the proof.

It is important to notice the close interplay between recursive function definitions and inductive proofs illustrated here. Proving by induction and programming by recursion are two sides of the same coin. The concept of an *invariant specification* for a recursive function (describing the input/output behavior of that function) corresponds exactly to the idea of an *inductive hypothesis* for an inductive proof. Cases in a recursive function that do not make recursive calls correspond to the base cases of inductive proofs. Cases that do make recursive calls correspond to inductive steps.

**Exercise 6**
*Using the lazy datatype primitives of SML, define a type* `tree_star` *of (possibly infinite) binary trees, and exhibit an infinite binary tree,* `inf_tree`*, of type* `tree_star` *(there are many possible solutions). Define an analogue of the* `height` *function for the type* `tree_star`*, and observe that* `height inf_tree` *does not terminate — the infinite tree has no height.*

Recall that elements of an inductively-defined set may be considered to be constructed in stages. The elements of the recursive type are those that arise at some finite stage of construction. Using this, we may replace a proof by induction on the rules generating an object $o$ by a proof by ordinary mathematical induction on the stage $s$ at which $o$ is constructed. For example, we may prove that every tree has a height by induction on the stage at which the tree is introduced. At stage 0 we have only the empty tree, whose height is 0. At stage $s+1$, we have in addition those nodes whose children are trees at stage $s$. By the inductive hypothesis the children of a node have a height, so the node must also have a height. The proof is complete because we have considered all possible stages $s$. As you can see considering stages is really just a roundabout way of reasoning directly by rule induction. However, it is occasionally helpful to keep the stage interpretation in mind.

## 1.3 Mathematical Details [Optional]

For the mathematically-inclined student we will give a set-theoretic account of inductive definitions of subsets of some universe $\mathcal{U}$. The general theory of inductive definitions does not rely on any particular properties of $\mathcal{U}$, but for the sake of examples we shall need to assume that $\mathcal{U}$ is rich enough to include objects of interest and over which suitable value constructors may be defined. The existence of a suitable universe for our purposes is not entirely obvious. Suffice it to say that we may take $\mathcal{U}$ to be the set $\mathbb{N}$ of natural numbers. Using techniques introduced by Gödel, we may encode finitary objects such as trees as natural numbers. This method of assigning unique numbers to each data structure is called "Gödel numbering" in honor of Gödel's discovery.

An inductive definition consists of a set of rules. A *rule* is a pair $(X, x)$, where $X$ is a finite[2] subset of $\mathcal{U}$ and $x \in \mathcal{U}$. The idea is that $X$ is the set of requirements, or premises, of the rule, and $x$ is its conclusion. Note that $X$ might be the empty set, in which case the rule has no premises. A rule of the form $(\{\, x_1, \ldots, x_n \,\}, x)$ is usually written $x_1, \ldots, x_n \to x$ or as the figure

$$\frac{x_1 \quad \ldots \quad x_n}{x}.$$

(When $n = 0$ we drop the arrow or horizontal line entirely.) A *rule set*, $\mathcal{R}$, is a (finite or infinite) set of rules.

[2]This restriction is not strictly necessary, but is sufficient for our purposes. One can also consider transfinite inductive definitions, with no significant changes to the development.

**Example 7**
*Here is a rule set corresponding to the inductive definition of numerals:*

$$\begin{aligned} \mathcal{R}_{\mathbf{N}} &= \{\,(\emptyset, \mathbf{Zero})\,\} \cup \{\,(\{\,n\,\}, \mathbf{Succ}(n)) \mid n \in \mathcal{U}\,\} \\ &= \{\,\mathbf{Zero}\,\} \cup \{\,n \to \mathbf{Succ}(n) \mid n \in \mathcal{U}\,\} \\ &= \{\,\overline{\mathbf{Zero}}\,\} \cup \{\,\frac{n}{\mathbf{Succ}(n)}\,\} \end{aligned}$$

Notice that this set consists of an infinite set of rules!

**Exercise 8**
*Give a rule set corresponding to the inductive definitions of binary trees and of strings.*

A subset $A \subseteq \mathcal{U}$ is *closed* under rules $\mathcal{R}$ (or, is $\mathcal{R}$-*closed*) iff for each rule $X \to x$ in $\mathcal{R}$, if $X \subseteq A$, then $x \in A$. That is, $A$ contains all of the elements introduced by an application of a rule in $\mathcal{R}$; it might also contain more than just these.

**Exercise 9**
*Check that both* $\mathbf{N}$ *and* $\mathbf{N}^*$ *are closed under* $\mathcal{R}_{\mathbf{N}}$.

We associate with each rule set $\mathcal{R}$ a monotone operator $\Phi_{\mathcal{R}} : \mathit{Pow}(\mathcal{U}) \to \mathit{Pow}(\mathcal{U})$ defined as follows:

$$\Phi_{\mathcal{R}}(A) = \{\, x \mid X \to x \in \mathcal{R},\ X \subseteq A \,\}$$

The operator $\Phi_{\mathcal{R}}$ corresponds to "one turn of the crank" — the set of elements that arise by applying some rule in $\mathcal{R}$ to the elements of the set $A$. Observe that a set $A \subseteq \mathcal{U}$ is closed under $\mathcal{R}$ iff $\Phi_{\mathcal{R}}(A) \subseteq A$. That is, $A$ contains all of the elements introduced by a single application of a rule in $\mathcal{R}$. For example, $\Phi_{\mathcal{R}_{\mathbf{N}}}(A) = \{\,\mathbf{Zero}\,\} \cup \{\, a \mid a \in A \,\}$.

It is easy to check that $\Phi_{\mathcal{R}}$ is *monotone*: if $A \subseteq B$, then $\Phi_{\mathcal{R}}(A) \subseteq \Phi_{\mathcal{R}}(B)$. Intuitively, if $A \subseteq B$, then the elements emitted by the rules in $\mathcal{R}$ starting $B$ must include at least those emitted by starting from $A$.

The set *inductively defined* by a rule set $\mathcal{R}$, written $\mathcal{I}(\mathcal{R})$, is defined as follows:

$$\mathcal{I}(\mathcal{R}) = \bigcap \{\, A \subseteq \mathcal{U} \mid \Phi_{\mathcal{R}}(A) \subseteq A \,\}$$

That is, $\mathcal{I}(\mathcal{R})$ is the intersection of all $\mathcal{R}$-closed sets $A$.

**Theorem 10**
*Let* $\mathcal{R}$ *be a rule set and let* $I = \mathcal{I}(\mathcal{R})$.

1. *If $A$ is $\mathcal{R}$-closed, then $I \subseteq A$.*

2. *$I$ is $\mathcal{R}$-closed.*

*That is, $I$ is the least set closed under the rules $\mathcal{R}$.*

**Proof:** Let $\Phi = \Phi_{\mathcal{R}}$. If $\Phi(A) \subseteq A$, then $I \subseteq A$ since $I$ is the intersection of all such sets $A$. This establishes (1). To show that $\Phi(I) \subseteq I$, it is sufficient to show that $\Phi(I) \subseteq A$ whenever $\Phi(A) \subseteq A$, since $I$ is the intersection of all such $A$'s. If $\Phi(A) \subseteq A$, then by (1), $I \subseteq A$. Hence by monotonicity, $\Phi(I) \subseteq \Phi(A)$. But then $\Phi(I) \subseteq A$, as required. This establishes (2). ■

**Exercise 11**
*Prove that $I \subseteq \Phi(I)$, and hence that $I = \Phi(I)$.* Hint: *Show that $\Phi(I)$ is $\mathcal{R}$-closed, using part (2) of the Theorem.*

OK, so we've now figured out precisely which set is inductively defined by a collection of rules — it's the intersection of all sets closed under these rules. What matters about this set is what's stated in the Theorem, namely that it's the smallest set closed under the given rules. Why is this important? Because this is how we justify the proofs by rule induction!

Let $I$ be the set inductively defined by some collection of rules $\mathcal{R}$. Suppose we wish to prove that some property holds of every element $x \in I$. Thinking of the property as a subset $P \subseteq I$ (namely, the set of those elements of $I$ that have that property), then what we're trying to do is to prove that $I \subseteq P$ — that is, that everything in $I$ has the property $P$. The theorem tells us that it is sufficient to show that $P$ is closed under $R$, since $I$ is the smallest such set. That is, we must show that $\Phi(P) \subseteq P$.

Let's see what this means by considering an example. Let $\mathbf{T}$ be the set of binary trees inductively defined by the rules in the previous section. Let $P$ be the following set

$$P = \{\, t \in \mathbf{T} \mid \textit{there exists } h \textit{ such that } t \textit{ has height } h \,\}$$

where *$t$ has height $h$* iff either $t = \mathbf{Empty}$ and $h = 0$ or $t = \mathbf{Node}(t_l, t_r)$, $t_l$ has height $h_l$, $t_r$ has height $h_r$, and $h = 1 + \max(h_l, h_r)$. To show that every binary tree has a height, it suffices to show that $P$ is closed under the rules defining the set of binary trees. That is, we must show

1. there exists $h$ such that $\mathbf{Empty}$ has height $h$ (namely, 0);

2. if there exists $h_l$ such that $t_l$ has height $h_l$ and there exists $h_r$ such that $t_r$ has height $h_r$, then there exists $h$ such that $t$ has height $h$ (namely, $h = 1 + \max(h_l, h_r)$).

This is precisely the argument we gave in the previous section!

**Exercise 12**
*Show that every numeral stands for a unique natural number such that* **Zero** *stands for* 0*, and if* $n$ *stands for* $k$*, then* **Succ**($n$) *stands for* $k+1$*. Spell out the property* $P$ *in question, and spell out the conditions for* $P$ *to be closed under the rules defining the numerals. Then prove that* $P$ *is closed under these rules.*

To finish up let's make precise the description of $\mathcal{I}(\mathcal{R})$ as being built up in stages. Fix a rule set $\mathcal{R}$, and let $\Phi = \Phi_{\mathcal{R}}$ and $I = \mathcal{I}(\mathcal{R})$. Define the sets $I_s$ for $s \geq 0$ as follows:

$$\begin{array}{rcl} I_0 & = & \emptyset \\ I_{s+1} & = & \Phi(I_s) \end{array}$$

At each stage we "turn the crank" once to add in all elements required by rules applied to the elements at the previous stage. Observe that $I_s \subseteq I_{s+1}$, for every $s \geq 0$. Now define

$$I_\infty = \bigcup_{s \geq 0} I_s$$

Thus the elements of $I_\infty$ are those that arise at some finite stage.

**Exercise 13**

1. *Prove by induction on* $s \geq 0$ *that* $I_s \subseteq I$*. Conclude that* $I_\infty \subseteq I$*.*

2. *Prove that* $I \subseteq I_\infty$ *by showing that* $I_\infty$ *is* $\Phi$*-closed. Hint: If* $x \in \Phi(I_\infty)$*, then there exists some rule* $X \to x$ *such that* $X \subseteq I_\infty$*. But since* $X$ *is finite (by definition of a rule), there must exist a finite stage* $s \geq 0$ *such that* $X \subseteq I_s$*. Hence* $x \in \Phi(I_s) = I_{s+1}$*. Then apply the previous result.*

# Chapter 2

# Concrete and Abstract Syntax

## 2.1 Introduction

The *concrete syntax* of a language consists of the rules for representing expressions as strings, linear sequences of characters (or symbols) that may be written on a page or entered using a keyboard. The concrete syntax usually is designed to enhance readability and to eliminate ambiguity. While there are good methods (grounded in the theory of formal languages) for eliminating ambiguity, readability is, of course, a matter of taste on which reasonable people may disagree.

Techniques for enhancing readability include the use of suggestive key words and phrases, and punctuation and layout conventions. Techniques for eliminating ambiguity include precedence conventions for binary operators and various forms of parentheses for grouping sub-expressions. You are no doubt familiar with all of these ideas; we will have little more to say about concrete syntax design in this course.

The *abstract syntax* of a language consists of rules defining the fundamental constructs of the language thought of as elements of an inductively-defined set of *abstract syntax trees* or *terms*. The abstract syntax of a language avoids the complications inherent in the representation of expressions as strings. The translation from concrete to abstract syntax is called *parsing*. The job of the parser is to determine the underlying syntactic structure of an expression represented as a string. There can be many different concrete syntaxes for a given abstract syntax (and hence many parsers, one for each concrete syntax), but in practice we normally fix one concrete and one abstract syntax for a language.

## 2.2 Grammars

The standard method for specifying the syntax of a language is a *(context-free) grammar*. A grammar consists of a set of *syntactic categories* whose elements are determined by a set of *productions*. A production may be thought of as a recursive specification of a set of the constructs of a syntactic category described in terms of the elements of some other, or even the very same, syntactic category.

An example will explain everything. Let us give a grammar defining the syntax of a tiny fragment of an ML-like language in which we have both expressions and declarations. Here's the grammar:

| *Category* | *Member* | | *Constructs* |
|---|---|---|---|
| *Variables* | $x$ | $::=$ | $v_1 \mid v_2 \mid \cdots$ |
| *Numbers* | $n$ | $::=$ | $\texttt{0} \mid \texttt{S(}n\texttt{)}$ |
| *Expressions* | $e$ | $::=$ | $x \mid n \mid e_1\texttt{+}e_2 \mid e_1\texttt{*}e_2 \mid \texttt{let}\, d \,\texttt{in}\, e$ |
| *Declarations* | $d$ | $::=$ | $x\texttt{=}e \mid d_1 \,\texttt{and}\, d_2$ |

The symbol "$::=$" may be read as "is" and the symbol "$|$" may be read as "or". Thus a variable $x$ is one of an infinite enumeration $v_1$, $v_2$, $\ldots$; a number $n$ is a base-1 numeral for a natural number, an expression; $e$ is either a variable $x$, a number $n$, the sum or product of two expressions, or a `let` expression; and a declaration $d$ is either a variable binding or a parallel composition of declarations. Each alternative is a production generating an element of the category on the left of the "$::=$" symbol.

The purpose of a grammar is to determine the elements of its syntactic categories. But what are those elements? We can read the grammar in two ways. Under one reading the elements of the syntactic categories are strings (sequences of characters) determined by the productions of the grammar. For example, the string

```
S(0) + S(S(0)) * S(S(S(0)))
```

is an element of the syntactic category of expressions, according to the grammar given above. In the absence of parenthesization or precedence conventions (which are not specified in the grammar) this string is ambiguous; it can be read in one of two ways, either as a sum of a number and a product, or as the product of a sum and a number. The job of the concrete syntax designer is to "beef up" the grammar with enough additional constructs to eliminate these ambiguities (for example, by introducing a precedence relation among binary operators) and to enhance the readability (for example, by admitting ordinary base-10 notation for numbers).

Another reading of the grammar is as a specification of the abstract syntax of the language. Each syntactic category consists of a set of structured *terms*, also known as *abstract syntax trees*, or *ast's*. Under this reading the keywords in the grammar stand for constructors that build ast's out of other ast's. For example, the following term is an element of the category of expressions.

```
+(S(0),*(S(S(0)),S(S(S(0))))),
```

as is the following, different, term

```
*(+(S(0),S(S(0))),S(S(S(0)))).
```

In both cases we do not think of the syntactic elements as strings of characters, but rather as trees whose nodes are labelled with constructors (such as `+`, `*`, `S`, and `0`) and whose children are written as arguments of the constructors. The advantage of this presentation is that there is no ambiguity about how to "read" an element of a category — the term structure specifies the "deep structure" of the expression without recourse to any additional conventions or markings.

It is straightforward to represent the abstract syntax of a language in ML. Each syntactic category is represented by a recursive `datatype` whose constructors are generated by the productions of the grammar. Here is the representation of the above grammar (read as an abstract syntax specification) in ML:

```
datatype
    Var = V of int
and Num = Zero | Succ of Num
and Exp = Var of Var | Num of Num | Plus of Exp * Exp
        | Times of Exp * Exp | Let of Dec * Exp
and Dec = Bind of Var * Exp | And of Dec * Dec
```

The correspondence with the grammar is vivid. We need only choose a value constructor to correspond to the right-hand side of each production, and transliterate the productions into ML notation.

## 2.3 Grammars as Inductive Definitions

The elements of the syntactic categories specified by a grammar may be defined by interpreting the grammar as an inductive definition. The overall idea is to regard each production as a rule, and to take the meaning of the grammar to be the set inductively defined by these rules. What's interesting is that we can recover both interpretations of the grammar simply by changing the meaning of the constructors used in the inductive definition.

There is one complication in rendering a grammar as an inductive definition. Whereas the grammar notation defines *several* sets (one for each category), our discussion of inductive definitions covers only the definition of a *single* set by a set of rules. We can easily generalize the theory of inductive definitions to the simultaneous inductive definition of several sets. We need only "mark" the elements with an indication of the set to which they belong so that we can sort out which rules define elements of which set.

Here is the grammar of the preceding section presented as an inductive definition:

$$\frac{}{v_0\ \mathsf{var}} \qquad \frac{}{v_1\ \mathsf{var}} \qquad \cdots$$

$$\frac{}{Z\ \mathsf{num}} \qquad \frac{n\ \mathsf{num}}{S(n)\ \mathsf{num}}$$

$$\frac{x\ \mathsf{var}}{var(x)\ \mathsf{expr}} \qquad \frac{n\ \mathsf{num}}{num(n)\ \mathsf{expr}}$$

$$\frac{e_1\ \mathsf{expr} \qquad e_2\ \mathsf{expr}}{e_1\texttt{+}e_2\ \mathsf{expr}} \qquad \frac{e_1\ \mathsf{expr} \qquad e_2\ \mathsf{expr}}{e_1\texttt{*}e_2\ \mathsf{expr}}$$

$$\frac{d\ \mathsf{dec} \qquad e\ \mathsf{expr}}{\texttt{let}\, d\, \texttt{in}\, e\ \mathsf{expr}}$$

$$\frac{x\ \mathsf{var} \qquad e\ \mathsf{expr}}{x\texttt{=}e\ \mathsf{dec}} \qquad \frac{d_1\ \mathsf{dec} \qquad d_2\ \mathsf{dec}}{d_1\ \mathsf{and}\ d_2\ \mathsf{dec}}$$

Notice that we mark each object with its syntactic category. For example, the category of expressions consists of those objects $e$ for which $e\ \mathsf{expr}$ is an element of the set inductively defined by the foregoing rules.

To complete the interpretation we must define the objects and constructors used in the rules. If we wish to regard the elements of the categories as strings, we may define these as follows:

$$
\begin{array}{rcl}
v_0 & = & \texttt{"v0"} \\
v_1 & = & \texttt{"v1"} \\
\vdots & & \\
 & & \\
Z & = & \texttt{"0"} \\
S(s) & = & \texttt{"S("} \;\texttt{\textasciicircum}\; s \;\texttt{\textasciicircum}\; \texttt{")"} \\
 & & \\
var(s) & = & s \\
num(s) & = & s \\
plus(s_1, s_2) & = & s_1 \;\texttt{\textasciicircum}\; \texttt{"+"} \;\texttt{\textasciicircum}\; s_2 \\
times(s_1, s_2) & = & s_1 \;\texttt{\textasciicircum}\; \texttt{"*"} \;\texttt{\textasciicircum}\; s_2 \\
let(s_d, s_e) & = & \texttt{"let "} \;\texttt{\textasciicircum}\; s_d \;\texttt{\textasciicircum}\; \texttt{" in "} \;\texttt{\textasciicircum}\; s_e \\
 & & \\
s_x \texttt{=} s_e & = & s_x \;\texttt{\textasciicircum}\; \texttt{"="} \;\texttt{\textasciicircum}\; s_e \\
s_1 \texttt{=} s_2 & = & s_1 \;\texttt{\textasciicircum}\; \texttt{" and "} \;\texttt{\textasciicircum}\; s_2
\end{array}
$$

Here we use an ML-like notation for strings.

If we wish to regard the grammar as defining abstract syntax trees, we may define these as follows:

$$
\begin{array}{rcl}
v_0 & = & \texttt{V(0)} \\
v_1 & = & \texttt{V(1)} \\
\vdots & & \\
 & & \\
Z & = & \texttt{Zero} \\
S(t) & = & \texttt{Succ(}t\texttt{)} \\
 & & \\
var(v) & = & \texttt{Var(}v\texttt{)} \\
num(n) & = & \texttt{Num(}n\texttt{)} \\
plus(t_1, t_2) & = & \texttt{Plus(}t_1\texttt{,}t_2\texttt{)} \\
times(t_1, t_2) & = & \texttt{Times(}t_1\texttt{,}t_2\texttt{)} \\
let(t_d, t_e) & = & \texttt{Let(}t_d\texttt{,}t_e\texttt{)} \\
 & & \\
t_x \texttt{=} t_e & = & \texttt{Bind(}t_x\texttt{,}t_e\texttt{)} \\
t_1 \,\texttt{and}\, t_2 & = & \texttt{And(}t_1\texttt{,}t_2\texttt{)}
\end{array}
$$

Here we use the ML `datatype` constructors given earlier to build up the abstract syntax. What is important is that the ranges of distinct constructors are disjoint and that the constructors are injective (1:1) so that a term has a unique outermost constructor from which we can recover its constituent terms.

## 2.4 Structural Induction

Rendering a grammar as an inductive definition allows us to apply the general principle of rule induction to abstract syntax definitions. This is called *proof by structural induction*. The principal of structural induction for expressions states that to prove that a property $P$ holds of every expression $e$, it is enough to prove

1. $P$ holds of every variable;
2. $P$ holds of every number;
3. If $P$ holds of $e_1$ and $e_2$, then $P$ holds of $e_1$`+`$e_2$, and similarly for `Times`.
4. If $P$ holds of $e$, then $P$ holds of `let` $d$ `in` $e$.

In practice it is usually necessary to extend $P$ to the other syntactic categories, as follows. To prove that $P$ holds of a declaration $d$ it is enough to show that:

1. If $P$ holds of $x$ and $e$, then $P$ holds of $x$`=`$e$.
2. If $P$ holds of $d_1$ and $d_2$, then $P$ holds of $d_1$ `and` $d_2$.

Similarly, $P$ holds of every variable $v$ iff $P$ holds of $v_i$ for every $i \geq 0$, and $P$ holds of every number $n$ iff

1. $P$ holds of `Zero`;
2. If $P$ holds of $t$, then $P$ holds of `Succ(`$t$`)`.

We then change the clause for `Let` to

4' If $P$ holds of $d$ and $e$, then $P$ holds of `let` $d$ `in` $e$.

**Exercise 14**
*Define the* depth *of an expression and prove that every expression $e$ has a unique depth by induction on the structure of $e$.* Hint: *First give conditions relating the depth of each form of expression to the depth of its constituent expressions. You will need to define a notion of depth for declarations.*

## 2.5 Binding and Scope

Abstract syntax allows us to separate out what is common to many languages from what is specific to each of them. While well-chosen lexical conventions are crucial for human readers, they are not important for programming or proving properties of languages. The abstract syntax of a phrase reveals its "deep structure" and sheds the niceties designed for human readers. The job of a parser is to translate from concrete to abstract syntax, resolving ambiguities and reducing an expression to its barest representation.

Abstract syntax trees are a natural way to capture the deep structure of a phrase since it delineates the hierarchical relationships among the components of that phrase. For example, the abstract syntax of an English sentence might indicate by its structure that the sentence is a conjunction of two independent clauses, each of which starts with a dependent clause, and so forth. Of course it is not a trivial matter to determine the abstract syntax of a sentence of natural language, but people are pretty good at it and computers are getting much better.

One aspect shared by many programming languages is the notion of *binding* and *scope* of *variables*. For example, in the expression `let` $x$`=`$e$ `in` $e'$ the variable $x$ is *bound* by the `let` expression in its *body* $e'$. That is, we may use the variable $x$ within $e'$ to refer to the variable introduced by the `let` expression. The *name* of the bound variable is not important. We could equally well have chosen the variable $y$ as the variable $x$ in the `let` expression, and used $y$ instead of $x$ in its body to refer to that variable. This is called *renaming of bound variables*, or, for historical reasons, *$\alpha$-conversion*.

Variables that are not bound in an expression are *free*. For example, in the expression `let` $x$`=10 in` $x$`+`$y$ the variable $x$ is bound (by the `let`) in the expression $x$`+`$y$, and the variable $y$ is free in that same expression. Every variable must be bound *somewhere* in a program, but when examining a sub-expression in isolation we may encounter variables without explicit binding occurrences.

Determining where a given variable is bound in a program is called *scope resolution*. It is not entirely trivial because we may re-use bound variables. For example, consider the expression

```
let x=10 in (let x=11 in x+x)+x
```

Both the outer and the inner `let` expressions bind the variable `x`. The inner `let` binds *all* occurrences of `x` within its body; the outer `let` also binds all occurrences of `x` within its body, *except* those that occur within another binding of `x`. This, speaking informally, the value of the above expression would be $32 = (11 + 11) + 10$.

The general rule is this. A non-binding occurrence of a variable is bound at the *nearest enclosing* binder for that variable determined by moving upwards towards the root of the abstract syntax tree in which it occurs. If no binder for that variable is encountered, the variable is free within that phrase. Since a binder for a variable can be enclosed within the scope of another binder for the same variable, the inner occurrence is said to *shadow* the outer variable, or to *puncture the scope* of the outer binder.

Shadowing can always be avoided by renaming bound variables. For example, we can rewrite the next `let` expression given above as follows without changing its meaning, yet removing any doubt about which variable occurrences refer to which binder.

```
let x=10 in (let y=11 in y+y)+x
```

By calling the inner variable `y`, we avoid creating a hole in the scope of the outermost `let` expression.

Bound variables may be thought of as "generalized pronouns". In natural languages (such as English) we use "he", "she", and "it" to refer back to an entity that was previously introduced (bound) in the context of the utterance. We may regard these pronouns as three distinct bound variables that we use to refer to entities "bound" to these variables earlier in the context of the sentence.

**Exercise 15**
*Define a function $FV(e)$ yielding the set of* free variables *in expression $e$. The definition will be by induction on the structure of $e$, and will require an auxiliary function $FV(d)$ to compute the free variables of a declaration $d$.*

While it is handy to assume that distinct bound variables have distinct names, it is an undue burden on the programmer to insist on this rule. After all, suppose that you are linking with a million-line library: how would you know what variable names to avoid, and why on earth should you care? However, since the names of bound variables are not semantically significant, we can accrue the benefits of the "unique name" convention by adopting simple convention — we *identify* (equate) all phrases that differ only in the names of bound variables. For example, we regard the two version of the nested-`let` expression to be *identical*.

According to this convention pieces of abstract syntax are *equivalence classes* of ast's under the equivalence relation given by relating any two expressions that differ only in the names of their bound variables. When writing down an equivalence class, we must choose a representative of it. In doing so we make the tacit assumption that we choose representatives so that distinct bound variables have distinct names and, morever, so that the names are chosen to be apart from any given finite set of names. Thus would we choose the second of the two nested-`let` expressions as representative of the equivalence class. Moreover, if we wish, we can choose another representative in which, say, `x` does not occur at all. To satisfy this condition, we might choose this representative instead:

`let` $z$`=10 in (let` $y$`=11 in` $y$`+`$y$`)+`$z$

The convention of identifying expressions that differ only in the names of bound variables (*i.e.*, are $\alpha$-convertible to one another) is called *higher-order abstract syntax (HOAS)*, in contrast to our earlier notion of *first-order abstract syntax (FOAS)*. FOAS exposes the hierarchical structure of an expression; HOAS exposes also its binding structure. Note, however, that for HOAS to be applicable, we must specify which expression constructors bind which variables in which sub-expressions. For example, we must specify that in the expression `let` $x$`=`$e$ `in` $e'$, we specified above that $x$ is bound by the `let` within its body $e'$.

## 2.6 deBruijn Indices

The passage from FOAS to HOAS can be regarded as passing from tree representations to graph representations. An *expression graph* is a directed graph derived from the first-order abstract syntax tree by replacing occurrences of bound variables by an edge (or "pointer") leading to its binding occurrence. This makes clear the binding structure of the expression without having to resort to arbitrary naming conventions for variables. Each $\alpha$-equivalence class determines a unique expression graph that we can regard as the canonical representation of that expression.

Expression graphs are not very "user-friendly" because they lack an obvious linear representation, and would be difficult to write on a page or manipulate with standard text editors. But for the purposes of abstract syntax they are superior, precisely because they abstract away from niceties such as choice of variable names.

How might an expression graph be represented in ML? The fundamental problem is that bound variable occurrences are "back pointers" that refer higher up in an abstract syntax tree. This is not directly representable using ML data type declarations; some other means must be introduced for referring back to earlier binding occurrences of variables.

One, rather heavy-handed, approach would be to use reference cells to represent general graph structure (say, via adjacency lists) and regard expression graphs as ordinary graphs. This is overkill, however, because there really is no need to admit mutation of an expression graph, other than as a means to create backward references. Judicious use of data abstraction would hide the representation details, but at the expense of pattern matching, which would no longer be possible.

Another approach, invented by the Dutch mathematican N. G. deBruijn, is the technique of *deBruijn indices*. Bound variables are not named; rather, each occurrence is replaced by an index determining the binding occurrence of that variable. The index is the number of nested binders within the scope of the binder to which the index refers.

An example will illustrate the idea. The expression

```
let x=10 in let y=11 in x+y
```

is represented in deBruijn notation by

let 10 in let 11 in $\boxed{2}$+$\boxed{1}$

Here $\boxed{i}$ is a reference to the variable bound by $i$th enclosing `let` expression, starting from 1.

A peculiarity of deBruijn indices is that the same variable may be referred to by several different indices, depending on the nesting depth of the occurrence. For example, the expression

```
let x=10 in (let y=11 in y+y)+x
```

is represented in deBruijn notation by the expression

let 10 in (let 11 in $\boxed{1}$+$\boxed{1}$)+$\boxed{1}$.

All bound variable references have index 1, but the third refers to the outermost binder, whereas the first two refer to the innermost. This is a direct consequence of the convention of counting outwards to the bind occurrence.

DeBruijn indices are often used in implementations of languages, because they eliminate the need to worry about $\alpha$-conversion. They are tricky to handle correctly, in part because of the context-sensitivity of indices. If you're not careful, a reference to one variable can silently become a reference to another, creating havoc for the programmer.

## 2.7 Substitution

It frequently occurs that we will wish to substitute an expression $e'$ for all free occurrences of a given variable $x$ in another expression $e$, written $[e'/x]e$. The key to the proper definition of substitution is to avoid *capture* of free variables in $e'$ by binders within $e$. For example, suppose that $e$ is the expression

```
let y=10 in x+y
```

and $e'$ is the expression

```
2*y+1
```

The result of substituting $e'$ for free occurrences of $x$ in $e$ is *not*

```
let y=10 in (2*y+1)+y
```

but rather

```
let z=10 in (2*y+1)+z
```

Proper substitution requires that we rename `y` in $e$ before substituting to avoid capturing free occurrences of `y` in $e'$. The idea is that the occurrence of `y` in $e'$ is a *different* variable from the one introduced by the `let` occurring in $e$. By renaming the bound variable in $e$ before substituting, we can avoid confusion.

The formal definition of substitution must take account of capture avoidance. We can exploit the higher-order abstract syntax convention by choosing representatives appropriately. Specifically, whenever we wish to perform the substitution $[e'/x]e$, we tacitly choose a representative of the equivalence class of $e$ so that no free variable in $e'$ is bound within $e$. This ensures that capture cannot occur. With this in mind, we can define substitution formally by induction on the structure of $e$ as follows:

$$
\begin{array}{rcll}
[e'/x]x & = & e' & \\
[e'/x]y & = & y & (x \neq y) \\
[e'/x]n & = & n & \\
[e'/x](e_1\texttt{+}e_2) & = & [e'/x]e_1\texttt{+}[e'/x]e_2 & \\
[e'/x](e_1\texttt{*}e_2) & = & [e'/x]e_1\texttt{*}[e'/x]e_2 & \\
[e'/x](\texttt{let}\,d\,\texttt{in}\,e) & = & \texttt{let}\,[e'/x]d\,\texttt{in}\,[e'/x]e & \\
[e'/x](y\texttt{=}e) & = & y\texttt{=}[e'/x]e & (y \notin FV(e')) \\
[e'/x](d_1\,\texttt{and}\,d_2) & = & [e'/x]d_1\,\texttt{and}\,[e'/x]d_2 &
\end{array}
$$

Notice that we may tacitly assume that no binder governs a free variable in $e'$ in this definition.

# Chapter 3

# MinML, A MiniMaL Language: Abstract Syntax

## 3.1 Introduction

The language MinML will serve as the object of study for much of the first half of the course. MinML is a call-by-value, effect-free language with two base types, `int` and `bool`, and (partial) function types $\tau_1 \rightarrow \tau_2$. It is thus a call-by-value variant of Plotkin's PCF language, and an extremely stripped-down version of the ML language.

## 3.2 First-Order Abstract Syntax

The abstract syntax of MinML is divided into three categories, *types*, *expressions*, and *programs*.

The set of types is the least set closed under the following rules:

$$\frac{}{\texttt{int}\ \mathsf{type}} \qquad \frac{}{\texttt{bool}\ \mathsf{type}} \qquad \frac{\tau_1\ \mathsf{type} \quad \tau_2\ \mathsf{type}}{\tau_1 \rightarrow \tau_2\ \mathsf{type}}$$

We let $\rho$, $\sigma$, and $\tau$ range over the set of types.

The set of expressions is the least set closed under the following rules:

$$\frac{n\ \mathsf{num}}{n\ \mathsf{expr}}$$

$$\frac{}{\texttt{true}\ \mathsf{expr}} \qquad \frac{}{\texttt{false}\ \mathsf{expr}}$$

$$\frac{x\ \mathsf{var}}{x\ \mathsf{expr}}$$

$$\frac{o\ \mathsf{op} \quad e_1\ \mathsf{expr} \quad \ldots e_n\ \mathsf{expr}}{o(e_1, \ldots, e_n)\ \mathsf{expr}} \qquad \frac{e\ \mathsf{expr} \quad e_1\ \mathsf{expr} \quad e_2\ \mathsf{expr}}{\mathtt{if}\, e\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2\, \mathtt{fi}\ \mathsf{expr}}$$

$$\frac{f\ \mathsf{var} \quad x\ \mathsf{var} \quad e\ \mathsf{expr}}{\mathtt{fun}\, f\ \mathtt{(}x\mathtt{:}\tau_1\mathtt{)}\mathtt{:}\tau_2\ \mathtt{is}\ e\ \mathtt{end}\ \mathsf{expr}} \qquad \frac{e_1\ \mathsf{expr} \quad e_2\ \mathsf{expr}}{\mathtt{apply(}e_1, e_2\mathtt{)}\ \mathsf{expr}}$$

We use the variable $e$ to stand for a general expression.

We do not specify precisely the sets of numbers or variables. We generally write $x$, $y$, *etc.* for variables, and we write numbers in ordinary decimal notation. The set of primitive operators is inductively defined by these rules:

$$\frac{}{\mathsf{+}\ \mathsf{op}} \qquad \frac{}{\mathsf{*}\ \mathsf{op}} \qquad \frac{}{\mathsf{-}\ \mathsf{op}}$$

$$\frac{}{\mathsf{=}\ \mathsf{op}} \qquad \frac{}{\mathsf{<}\ \mathsf{op}}$$

It is important, in general, to distinguish programs from expression. In the present case programs are just (closed) expressions, which is expressed by the following rule:

$$\frac{e\ \mathsf{expr}}{e\ \mathsf{prog}}$$

That is, a program is nothing more (and nothing less) than an expression. We let $p$ range over the set of programs.

If you prefer grammar notation for specifying abstract syntax, here is an equivalent grammar for the abstract syntax of MinML:

$$\begin{array}{llcl}
\mathit{Variables} & x & ::= & \ldots \\
\mathit{Numerals} & n & ::= & \ldots \\
\mathit{Operators} & o & ::= & \mathsf{+} \mid \mathsf{*} \mid \mathsf{-} \mid \mathsf{=} \mid \mathsf{<} \\
\mathit{Types} & \tau & ::= & \mathtt{int} \mid \mathtt{bool} \mid \tau_1 {\rightarrow} \tau_2 \\
\mathit{Expressions} & e & ::= & x \mid n \mid o(e_1, \ldots, e_n) \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{if}\, e\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2\, \mathtt{fi} \\
 & & \mid & \mathtt{fun}\, f\ \mathtt{(}x\mathtt{:}\tau_1\mathtt{)}\mathtt{:}\tau_2\ \mathtt{is}\ e\ \mathtt{end} \mid \mathtt{apply(}e_1, e_2\mathtt{)} \\
\mathit{Programs} & p & ::= & e
\end{array}$$

Here, as before, we leave the exact definition of variables, numerals, and operators unspecified.

We will make use of informal syntactic conventions that it is hoped are clear from context. Officially we should define a concrete syntax for the language, and a mapping (parser) from the concrete to the abstract syntax. But this would take us too far afield for the present purposes. Here are a few notational conventions:

1. The function space type constructor is right-associative, so that the type int→int→int parses as int→(int→int), rather than as the type (int→int)→int.

2. We will often write expressions such as $e_1\texttt{+}e_2$ for $\texttt{+}(e_1, e_2)$, presuming + to be a primitive operation, and employing the obvious infix syntax.

3. We will often write $e_1\texttt{(}e_2\texttt{)}$, instead of $\texttt{apply(}e_1, e_2\texttt{)}$, to denote function application.

## 3.3 Higher-Order Abstract Syntax

The rules of the *binding* and *scope* of variables are of fundamental importance. While these rules may seem to be a trivial, or at any rate niggling, technical detail, they are in fact of paramount importance. Many languages[1] fail to properly define and maintain sensible binding and scope conventions, with the result that code written in those languages is difficult to read and impossible to maintain.

Every variable has a *binder*, which introduces it, and a *scope*, which is its range of significance. Outside of its scope, the variable has no meaning, nor any particular identity. There is only one binder in MinML, the function expression $\texttt{fun}\,f\,\texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,e\,\texttt{end}$. The variables $f$ and $x$ are *bound* by the function with *scope* the expression $e$. Nested binders may introduce "holes" in the scope of a variable — a variable occurrence in an expression is bound by the nearest enclosing binder for that variable, if there is one, in which case the variable is said to be *bound* by that binder. If no enclosing binder governs that variable, it is said to be *free* in the expression. It may be helpful to think of bound variables as "generalized pronouns" — they are used to refer to an antecedent binding occurrence. (Unlike natural languages, however, we are free to introduce new pronouns as we see fit.)

We may define the set of *free variables* of an expression $e$, $\mathrm{FV}(e)$, as follows:

$$
\begin{array}{rcl}
\mathrm{FV}(x) & = & \{\,x\,\} \\
\mathrm{FV}(n) & = & \emptyset \\
\mathrm{FV}(\texttt{true}) & = & \emptyset \\
\mathrm{FV}(\texttt{false}) & = & \emptyset \\
\mathrm{FV}(o(e_1,\ldots,e_n)) & = & \mathrm{FV}(e_1) \cup \cdots \cup \mathrm{FV}(e_n) \\
\mathrm{FV}(\texttt{if}\,e\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi}) & = & \mathrm{FV}(e) \cup \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(\texttt{fun}\,f\,\texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,e\,\texttt{end}) & = & \mathrm{FV}(e)\setminus\{\,f, x\,\} \\
\mathrm{FV}(\texttt{apply}(e_1, e_2)) & = & \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2)
\end{array}
$$

We say that the variable $x$ is *free* in the expression $e$ iff $x \in \mathrm{FV}(e)$. An expression $e$ is *closed* iff $\mathrm{FV}(e) = \emptyset$; that is, a closed expression has no free variables.

[1] Including the markup language I'm using to write these notes, TEX.

## 3.4 Substitution

subst

*Capture-avoiding substitution* of an expression $e$ for free occurrences of a variable $x$ in another expression $e'$, written $[e/x]e'$, is (partially) defined as follows:

$$\begin{array}{rcll}
[e/x]x & = & e & \\
[e/x]n & = & n & \\
[e/x]\texttt{true} & = & \texttt{true} & \\
[e/x]\texttt{false} & = & \texttt{false} & \\
[e/x]o(e_1, \ldots, e_n) & = & o([e/x]e_1, \ldots, [e/x]e_n) & \\
[e/x]\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} & = & \texttt{if}\, [e/x]e\, \texttt{then}\, [e/x]e_1\, \texttt{else}\, [e/x]e_2\, \texttt{fi} & \\
[e/x]\texttt{fun}\, f\, (y{:}\tau_1){:}\tau_2\, \texttt{is}\, e''\, \texttt{end} & = & \texttt{fun}\, f\, (y{:}\tau_1){:}\tau_2\, \texttt{is}\, e''\, \texttt{end} & \textit{if } x = f \textit{ or } x = y \\
[e/x]\texttt{fun}\, f\, (y{:}\tau_1){:}\tau_2\, \texttt{is}\, e''\, \texttt{end} & = & \texttt{fun}\, f\, (y{:}\tau_1){:}\tau_2\, \texttt{is}\, [e/x]e''\, \texttt{end} & \textit{if } \{\, f, y \,\} \cap (\mathrm{FV}(e) \cup \{\, x \,\}) = \emptyset \\
[e/x]\texttt{apply}(e_1, e_2) & = & \texttt{apply}([e/x]e_1, [e/x]e_2) &
\end{array}$$

*Simultaneous* capture-avoiding substitution, written $[e_1, \ldots, e_n/x_1, \ldots, x_n]e$, is defined in an analogous manner.

Capture-avoiding substitution is *undefined* if the condition in the penultimate equation is not met! In this case free occurrences of $f$ or $y$ in $e$ would be *captured* by the binder for $f$ and $y$, thereby erroneously changing the meanings of the "pronouns". This means, for example, that

$$[x/y]\texttt{fun}\, f\, (x{:}\texttt{int}){:}\texttt{int}\, \texttt{is}\, x + y\, \texttt{end}$$

is *undefined*, rather than equal to

$$\texttt{fun}\, f\, (x{:}\texttt{int}){:}\texttt{int}\, \texttt{is}\, x + x\, \texttt{end},$$

wherein capture of $x$ has occurred.

The possibility of capture during substitution can always be avoided by *renaming of bound variables*. The meaning of an expression should not be affected by the choice of variable names; whether we call a variable $x$, *he*, *she*, *it*, or *baloney_sandwich* should be immaterial, as long as we avoid the clashes (giving two *different* variables the *same* name). Thus

$$\texttt{fun}\, f\, (x{:}\texttt{int}){:}\texttt{int}\, \texttt{is}\, \texttt{apply}(f, y + x + 1)\, \texttt{end}$$

is precisely the same expression as

$$\texttt{fun}\, g\, (z{:}\texttt{int}){:}\texttt{int}\, \texttt{is}\, \texttt{apply}(g, y + z + 1)\, \texttt{end}$$

but is different from the expression

$$\texttt{fun}\, g\, (y{:}\texttt{int}){:}\texttt{int}\, \texttt{is}\, \texttt{apply}(g, y + y + 1)\, \texttt{end}.$$

To express this indifference to the names of bound variables, we introduce an equivalence relation between expressions, called*structural equivalence*, and written $e_1 \equiv e_2$, that is defined to be the smallest binary relation closed under the following rules. The first rule is called the *renaming* axiom.

$$\frac{}{\texttt{fun}\, f\; \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\; \texttt{is}\; e\; \texttt{end} \equiv \texttt{fun}\, f'\; \texttt{(}x'\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\; \texttt{is}\; [f', x'/f, x]e\; \texttt{end}}$$

(where $f'$ and $x'$ are chosen in any way such that substitution is defined). The remaining rules specify that structural equivalence is reflexive, symmetric, and transitive, and that we may perform renaming anywhere in an expression. These rules are sometimes called *congruence* rules.

$$\frac{}{e \equiv e} \qquad \frac{e_2 \equiv e_1}{e_1 \equiv e_2} \qquad \frac{e_1 \equiv e_2 \quad e_2 \equiv e_3}{e_1 \equiv e_3}$$

$$\frac{e_1 \equiv e_1' \quad \ldots \quad e_n \equiv e_n'}{o(e_1, \ldots, e_n) \equiv o(e_1', \ldots, e_n')}$$

$$\frac{e \equiv e' \quad e_1 \equiv e_1' \quad e_2 \equiv e_2'}{\texttt{if}\; e\; \texttt{then}\; e_1\; \texttt{else}\; e_2\; \texttt{fi} \equiv \texttt{if}\; e'\; \texttt{then}\; e_1'\; \texttt{else}\; e_2'\; \texttt{fi}}$$

$$\frac{e \equiv e'}{\texttt{fun}\, f\; \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\; \texttt{is}\; e\; \texttt{end} \equiv \texttt{fun}\, f\; \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\; \texttt{is}\; e'\; \texttt{end}}$$

$$\frac{e_1 \equiv e_1' \quad e_2 \equiv e_2'}{\texttt{apply(}e_1, e_2\texttt{)} \equiv \texttt{apply(}e_1', e_2'\texttt{)}}$$

Any binary relation closed under the congruence rules above is said to be a *congruence* relation. Structural equivalence is the *least congruence* closed under the renaming axiom.

In order to express the irrelevance of the names of bound variables, we implicitly identity expressions that are structurally equivalent. That is, in what follows, we never work directly with expressions, but rather with *equivalence classes* of expressions, modulo structural equivalence. When choosing a representative of a structural equivalence class, we do so in accordance with the *variable convention*, according to which the names of the bound variables are chosen to be distinct from one another and distinct from the name of any free variable in the context of use.

One consequence of this convention is that substitution is *never* undefined when applied to expressions taken modulo structural equivalence. The reason is that if the variable conditions on substitution that prevent capture were not met, we could always choose another representative so that they are met. This means, for example, that

$$[x/y]\texttt{fun}\, f\; \texttt{(}x\texttt{:int)}\texttt{:int}\; \texttt{is}\; x + y\; \texttt{end}$$

is equivalent to

$$\texttt{fun}\,f\,\texttt{(}z\texttt{:int):int is}\;z+x\;\texttt{end}$$

(for any variable $z \neq x$), but *not* to

$$\texttt{fun}\,f\,\texttt{(}x\texttt{:int):int is}\;x+x\;\texttt{end}$$

(wherein captured has occurred).

## 3.5 Summary

This completes the presentation of the abstract syntax of MinML. The key points are these

1. Inductive definitions of the syntactic categories.
2. Definition of binding and scope.
3. Definition of capture-avoiding substitution.
4. Structural equivalence to identity expressions differing only in the choice of names of bound variable.

# Chapter 4

# MinML Static Semantics

## 4.1 Introduction

In this note we consider the *static semantics*, or *typing rules*, for MinML. The type system of MinML is given by an inductive definition consisting of a set of rules for deriving *typing judgements*.

## 4.2 Type System

Not all expressions in MinML are well-formed. For example, the expression `if` 3 `then` 1 `else` 0 `fi` is not well-formed because 3 is an integer, whereas the conditional test expects a boolean. In other words, this expression is *ill-typed* because the expected constraint is not met. Expressions which do satisfy these constraints are said to be *well-typed*.

Typing is clearly context-sensitive. The expression $x + 3$ may or may not be well-typed, according to the type we assume for the variable $x$. That is, it depends on the surrounding context whether this sub-expression is well-typed or not.

The definition of well-typed expressions is given by a three-place *typing relation*, or *typing judgement*, written $\Gamma \vdash e : \tau$, where $\Gamma$ is a partial function with finite domain mapping variables to types, and $\mathrm{FV}(e) \subseteq \mathrm{dom}(\Gamma)$. This relation may be read as "the expression $e$ has type $\tau$, under the assumption that its free variables have the types given by $\Gamma$." The function $\Gamma$ may be thought of as a "symbol table" recording the types of the free variables of the expression $e$; the type $\tau$ is the type of $e$ under the assumption that its free variables have the types assigned by $\Gamma$. When $e$ is closed (has no free variables), we write simply $e : \tau$ instead of the more unwieldy $\emptyset \vdash e : \tau$.

We write $\Gamma(x)$ for the unique type $\tau$ (if any) assigned to $x$ by $\Gamma$. The function $\Gamma[x{:}\tau]$, where $x \notin \mathrm{dom}(\Gamma)$, is defined by the following equation

$$\Gamma[x{:}\tau](y) = \begin{cases} \tau & \textit{if } y = x \\ \Gamma(y) & \textit{otherwise} \end{cases}$$

The typing relation is inductively defined by the following rules:

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ [\textbf{VarTyp}] \qquad \frac{}{\Gamma \vdash n : \texttt{int}}\ [\textbf{NumTyp}]$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ [\textbf{TrueTyp}] \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\ [\textbf{FalseTyp}]$$

In the rule for variables, it is understood that if $\Gamma(x)$ is undefined, then no type for $x$ is derivable from assumptions $\Gamma$.

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{+}(e_1, e_2) : \texttt{int}}\ [\textbf{PlusTyp}]$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{*}(e_1, e_2) : \texttt{int}}\ [\textbf{TimesTyp}]$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{-}(e_1, e_2) : \texttt{int}}\ [\textbf{MinusTyp}]$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{=}(e_1, e_2) : \texttt{bool}}\ [\textbf{EqualsTyp}]$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{<}(e_1, e_2) : \texttt{bool}}\ [\textbf{LessTyp}]$$

The typing rules for the arithmetic and boolean primitive operators are as expected.

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} : \tau}\ [\textbf{IfTyp}]$$

Notice that the "then" and the "else" clauses must have the same type!

$$\frac{\Gamma[f{:}\tau_1 \to \tau_2][x{:}\tau_1] \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\, f\ \texttt{(}x : \tau_1\texttt{)} : \tau_2\ \texttt{is}\, e\, \texttt{end} : \tau_1 \to \tau_2}\ [\textbf{FunTyp}]$$

Here we require that the variables $f$ and $x$ be chosen (by suitable renaming of the function expression) so that $\{ f, x \} \cap \mathrm{dom}(\Gamma) = \emptyset$.

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{apply(}e_1, e_2\texttt{)} : \tau}\ [\textbf{AppTyp}]$$

This completes the typing rules for MinML.

## 4.3 Properties of Typing

It is useful at this stage to catalogue some properties of the typing relation. We will make use of the principle of *induction on typing derivations*, or *induction on the typing rules*.

A key observation about the typing rules is that there is exactly one rule for each form of expression — that is, there is one rule for the each of the boolean constants, one rule for functions, *etc.*. The typing relation is therefore said to be *syntax-directed*; the form of the expression determines the typing rule to be applied. While this may seem inevitable at this stage, we will later encounter type systems for which this is not the case.

**Exercise 16**
*Prove by induction on the structure of $e$ that for every $e$ and every $\Gamma$ there exists at most one $\tau$ such that $\Gamma \vdash e : \tau$.* Hint: *use rule induction for the rules defining the abstract syntax of expressions. The proof in each case is straightforward; what is important about this exercise is that you know how to present the proof clearly and rigorously using the machinery we've introduced.*

A simple — but important — consequence of syntax-directedness are the following *inversion principles* for typing. The typing rules define *sufficient* conditions for typing. For example, to show that $\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} : \tau$, it suffices to show that $\Gamma \vdash e : \texttt{bool}$, $\Gamma \vdash e_1 : \tau$, and $\Gamma \vdash e_2 : \tau$, because of rule [**IfTyp**]. Since there is exactly one typing rule for each expression, the typing rules also express *necessary* conditions for typing. For example, if $\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} : \tau$, then $\Gamma \vdash e : \texttt{bool}$, $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$. That is, we can "invert" each rule to obtain a necessary typing condition. This is the content of the following theorem.

**Theorem 17 (Inversion)**

1. *If $\Gamma \vdash x : \tau$, then $\Gamma(x) = \tau$.*
2. *If $\Gamma \vdash n : \tau$, then $\tau = \texttt{int}$.*
3. *If $\Gamma \vdash \texttt{true} : \tau$, then $\tau = \texttt{bool}$, and similarly for $\texttt{false}$.*
4. *If $\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} : \tau$, then $\Gamma \vdash e : \texttt{bool}$, $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$.*
5. *If $\Gamma \vdash \texttt{fun}\, f\, (x{:}\tau_1){:}\tau_2\, \texttt{is}\, e\, \texttt{end} : \tau$, then $\tau = \tau_1 \to \tau_2$ and then $\Gamma[f{:}\tau_1 \to \tau_2][x{:}\tau_1] \vdash e : \tau_2$.*

6. *If* $\Gamma \vdash$ `apply(`$e_1, e_2$`)` $: \tau$, *then there exists* $\tau_2$ *such that* $\Gamma \vdash e_1 : \tau_2 \to \tau$ *and* $\Gamma \vdash e_2 : \tau_2$.

**Proof:** Each case is proved by induction on typing. In each case exactly one rule applies, from which the result is obvious. ∎

**Exercise 18**
*Can you think of a type system for a variant of MinML in which inversion fails? What form would such a type system have to take?* Hint: *think about overloading arithmetic operations.*

**Lemma 19**

1. *Typing is not affected by "junk" in the symbol table. If* $\Gamma \vdash e : \tau$ *and* $\Gamma' \supseteq \Gamma$, *then* $\Gamma' \vdash e : \tau$.

2. *Substitution for a variable with type* $\tau$ *by an expression of the same type doesn't affect typing. If* $\Gamma[x{:}\tau] \vdash e' : \tau'$, *and* $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash [e/x]e' : \tau'$.

**Proof:**

1. By induction on the typing rules. For example, consider the typing rule for applications. Inductively we may assume that if $\Gamma' \supseteq \Gamma$, then $\Gamma' \vdash e_1 : \tau_2 \to \tau$ and if $\Gamma' \supseteq \Gamma$, then $\Gamma' \vdash e_2 : \tau_2$. Consequently, if $\Gamma' \supseteq \Gamma$, then $\Gamma' \vdash$ `apply(`$e_1, e_2$`)` $: \tau$, as required. The other cases follow a similar pattern.

2. By induction on the derivation of the typing $\Gamma[x{:}\tau] \vdash e' : \tau'$. We will consider several rules to illustrate the idea.

   **[VarTyp]** We have that $e'$ is a variable, say $y$, and $\tau' = \Gamma[x{:}\tau](y)$. If $y \neq x$, then $[e/x]y = y$ and $\Gamma[x{:}\tau](y) = \Gamma(y)$, hence $\Gamma \vdash y : \Gamma(y)$, as required. If $x = y$, then $\tau' = \Gamma[x{:}\tau](x) = \tau$, and $[e/x]x = e$. By assumption $\Gamma \vdash e : \tau$, as required.

   **[FunTyp]** We have that $e' =$ `fun` $f$ `(`$y$`:`$\tau_1$`):`$\tau_2$ `is` $e_2$ `end` and $\tau' = \tau_1 \to \tau_2$. We may assume that $f$ and $y$ are chosen so that

$$\{ f, y \} \cap (\mathrm{FV}(e) \cup \{ x \} \cup \mathrm{dom}(\Gamma)) = \emptyset.$$

By definition of substitution,

$$[e/x]e' = \texttt{fun}\, f\ \texttt{(}y\,{:}\,\tau_1\texttt{)}\,{:}\,\tau_2\ \texttt{is}\ [e/x]e_2\ \texttt{end}.$$

Applying the inductive hypothesis to the premise of the rule [**FunTyp**],

$$\Gamma[x{:}\tau][f{:}\tau_1{\rightarrow}\tau_2][y{:}\tau_1] \vdash e_2 : \tau_2,$$

it follows that

$$\Gamma[f{:}\tau_1{\rightarrow}\tau_2][y{:}\tau_1] \vdash [e/x]e_2 : \tau_2.$$

Hence

$$\Gamma \vdash \texttt{fun}\, f\ \texttt{(}y\,{:}\,\tau_1\texttt{)}\,{:}\,\tau_2\ \texttt{is}\ [e/x]e_2\ \texttt{end} : \tau_1{\rightarrow}\tau_2,$$

as required.

$\blacksquare$

## 4.4 Summary

This completes the definition of the *static semantics*, or *typing rules*, for MinML. The main points are these:

1. Typing is context-sensitive. The type (if any) of an expression with free variables in it is sensitive to the types we assume for those variables.

2. The typing relation is given by an inductive definition. This licenses proofs by induction on the typing rules.

3. The typing relation is syntax-directed: there is one rule for each form of expression.

# Chapter 5

# MinML Dynamic Semantics

## 5.1 Introduction

In this section we consider the *dynamic semantics*, or *rules of evaluation*, for MinML. We will use a technique called *structured operational semantics (SOS)* to specify the evaluation rules for MinML.

The dynamic semantics of MinML is an example of what is known as a *language-based* model of computation, as opposed to a more familiar *machine-based* model. Before giving details, let us first discuss the distinction between these two approaches.

In a machine-based model the dynamic semantics of a language is specified by describing how the language is compiled into code for an (abstract or concrete) machine. That is, we define the language by prescribing how it is to be implemented. This is common practice for languages such as C, which are designed for writing low-level systems code where access to the "bare metal" is essential. However, such code is notoriously non-portable (because real machines differ in critical details such as byte-order, memory model, or data representation). Worse, machine models of computation are inherently "indirect" in the sense that the meaning of a program is defined only in terms of the meaning of another program (the code to which it compiles), rather than in terms of the language itself. This means that programs can exhibit behavior (such as "bus errors" or "segmentation faults") that can only be understood in terms of the implementation, rather than in terms of the language itself. While this can be important for certain systems applications, the vast majority of applications do not (and, arguably, should not) rely on such details.

In contrast, a language-based model is one for which the semantics of a program is given entirely in terms of the constructs of the language itself. This means that no mysterious implementation-specific problems can arise, greatly enhancing portability. Moreover, language-based models support reasoning about program behavior entirely at the level of the program itself, avoiding the need to understand how the language is compiled on a given platform. The overall idea of a language-based model is to treat computation as a form of calculation. In high school you learned to calculate with polynomials. Given a polynomial, you can readily "plug in" (substitute) a value for one or more variables, then carry out the arithmetic steps to determine the final result. You can think of a polynomial as a kind of program, and you can think of the calculation steps you used as a kind of "dynamic semantics" for "executing" polynomials. To put it another way, you can think of a program as a kind of "generalized polynomial" in which we compute by substituting values for variables and executing primitive instructions generalizing the arithmetic operations.

Before going into details, it is important to dispel a possible misunderstanding. The choice of a language-based *vs.* a machine-based model of computation is not independent of the language itself. That is, we cannot expect to use either approach with any given language. Languages that admit language-based models are sometimes called *abstract* languages because they abstract away from the details of the implementation. MinML is an abstract language in this sense, and we shall give it a language-based semantics. Languages that do not admit language-based models are sometimes called *concrete* languages since they are closely tied to actual machines. Nearly all assembly languages, and early dialects of the C language[1], are concrete languages, and are usually given a machine-based semantics. Recent languages, such as Java, occupy a middle ground. The semantics of Java is given by a mapping to an abstract machine, the Java Virtual Machine (JVM). Thus Java is, in a sense, a concrete language, but relative to an abstract machine! The JVM is given a semantics entirely in its own terms, without reference to a lower-level platform.

## 5.2 The Dynamic Semantics of MinML

The dynamic semantics of MinML is given by (what else?) an inductive definition of the *one-step evaluation* relation, $e \mapsto e'$, between *closed* expressions. Recall that we are modelling computation in MinML as a form

[1]Subsequent standardization efforts have attempted to give a more abstract semantics to C.

of "in place" calculation; the relation $e \mapsto e'$ means that $e'$ is the result of performing a single step of computation starting with $e$. To calculate the value of an expression $e$, we repeatedly perform single calculation steps until we reach a *value*, $v$: a number, a boolean constant, or a function.

The rules defining the dynamic semantics of MinML may be classified into two categories: rules defining the fundamental computation steps (or, *instructions*) of the language, and rules for determining where the next instruction is to be executed. The purpose of the search rules is to ensure that the dynamic semantics is *deterministic*, which means that for any expression there is at most one "next instruction" to be executed.[2]

First the instructions governing the primitive operations. We assume that each primitive operation $o$ defines a total function — given values $v_1$, $\ldots$, $v_n$ of appropriate type for the arguments, there is a unique value $v$ that is the result of performing operation $o$ on $v_1, \ldots, v_n$. For example, for addition we have the following primitive instruction:

$$\frac{}{\texttt{+}(m, n) \mapsto m + n} \; [\textbf{PlusNums}]$$

The other primitive operations are defined similarly.

The primitive instructions for conditional expressions are as follows:

$$\frac{}{\texttt{if}\,\texttt{true}\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi} \mapsto e_1} \; [\textbf{IfTrue}]$$

$$\frac{}{\texttt{if}\,\texttt{false}\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi} \mapsto e_2} \; [\textbf{IfFalse}]$$

The primitive instruction for application is as follows:

$$\frac{(v = \texttt{fun}\,f\,\texttt{(}x\,\texttt{:}\,\tau_1\texttt{)}\,\texttt{:}\,\tau_2\,\texttt{is}\,e\,\texttt{end})}{\texttt{apply(}v, v_1\texttt{)} \mapsto [v, v_1/f, x]e} \; [\textbf{CallFun}]$$

To apply the function $v = \texttt{fun}\,f\,\texttt{(}x\,\texttt{:}\,\tau_1\texttt{)}\,\texttt{:}\,\tau_2\,\texttt{is}\,e\,\texttt{end}$ to an argument $v_1$ (which must be a value!), we substitute the function itself, $v$, for $f$, and the argument value, $v_1$, for $x$ in the body, $e$, of the function. By substituting $v$ for $f$ we are "unrolling" the recursive function as we go along.

This completes the primitive instructions of MinML. The "search" rules, which determine which instruction to execute next, follow.

For the primitive operations, we specify a left-to-right evaluation order. For example, we have the following two rules for addition:

$$\frac{e_1 \mapsto e_1'}{\texttt{+}(e_1, e_2) \mapsto \texttt{+}(e_1', e_2)} \; [\textbf{PlusArg}_1] \qquad \frac{e_2 \mapsto e_2'}{\texttt{+}(v_1, e_2) \mapsto \texttt{+}(v_1, e_2')} \; [\textbf{PlusArg}_2]$$

[2] Some languages are, by contrast, *non-determinstic*, notably those involving concurrent interaction. We'll come back to those later.

The other primitive operations are handled similarly.

For the conditional, we evaluate the test expression.

$$\frac{e \mapsto e'}{\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \mapsto \texttt{if}\, e'\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi}} \text{ [IfTest]}$$

For applications, we first evaluate the function position; once that is complete, we evaluate the argument position.

$$\frac{e_1 \mapsto e_1'}{\texttt{apply}(e_1, e_2) \mapsto \texttt{apply}(e_1', e_2)} \text{ [AppFun]} \qquad \frac{e_2 \mapsto e_2'}{\texttt{apply}(v_1, e_2) \mapsto \texttt{apply}(v_1, e_2')} \text{ [AppArg]}$$

This completes the definition of the MinML one-step evaluation relation.

The *multi-step evaluation relation*, $e \mapsto^* e'$, is inductively defined by the following rules:

$$\frac{}{e \mapsto^* e} \qquad \frac{e \mapsto e' \quad e' \mapsto^* e''}{e \mapsto^* e''}$$

In words: $e \mapsto^* e'$ iff performing zero or more steps of evaluation starting from the expression $e$ yields the expression $e'$. The relation $\mapsto^*$ is sometimes called the *Kleene closure*, or *reflexive-transitive closure*, of the relation $\mapsto$.

## 5.3 Properties of the Dynamic Semantics

Let us demonstrate that the dynamic semantics of MinML is well-defined in the sense that it assigns at most one value to each expression. (We should be suspicious if this weren't true of the semantics, for it would mean that programs have no definite meaning.)

First, observe that if $v$ is a value, then there is no $e$ (value or otherwise) such that $v \mapsto e$. Second, observe that the evaluation rules are arranged so that at most one rule applies to any given form of expression, *even though* there are, for example, $n + 1$ rules governing each $n$-argument primitive operation. These two observations are summarized in the following lemma.

**Lemma 20**
*For every closed expression $e$, there exists at most one $e'$ such that $e \mapsto e'$. In other words, the relation $\mapsto$ is a partial function.*

**Proof:** By induction on the structure of $e$. We leave the proof as an exercise to the reader. Be sure to consider *all* rules that apply to a given expression $e$! ■

It follows that evaluation to a value is deterministic:

**Lemma 21**
*For every closed expression $e$, there exists at most one value $v$ such that $e \mapsto^* v$.*

**Proof:** Follows immediately from the preceding lemma, together with the observation that there is no transition from a value. ■

## 5.4 Summary

This completes the dynamic semantics of MinML. The main points are:

1. It is an example of *language-based* model of computation. The semantics is specified entirely in terms of the language itself, without reference to an implementation on an abstract or concrete machine.

2. It is specified using the technique of *structured operational semantics*, a collection of rules that determine the "next instruction" to be executed in a program, and for determining the result of that execution step.

# Chapter 6

# MinML Type Safety

## 6.1 Introduction

Programming languages such as ML and Java are said to be "safe" (or, "type safe", or "strongly typed"). Informally, this means that certain kinds of mismatches cannot arise during execution. For example, it will never arise that an integer is to be applied to an argument, nor that two functions could be added to each other. The goal of this section is to make this informal notion precise. What is remarkable is that we will be able to clarify the idea of type safety without making reference to an implementation. Consequently, the notion of type safety is extremely robust — it is shared by *all* correct implementations of the language.

Type safety is a *relation* between the static and dynamic semantics. It tells us something about the execution of well-typed programs; it says nothing about the execution of ill-typed programs. In implementation terms, we expect ill-typed programs to be rejected by the compiler, so that nothing need be said about their execution behavior (just as syntactically incorrect programs are rejected, and nothing is said about what such a program might mean).

In the framework we are developing, type safety amounts to the following two conditions:

1. **Preservation**. If $e$ is a well-typed program, and $e \mapsto e'$, then $e'$ is also a well-typed program.

2. **Progress**. If $e$ is a well-typed program, then either $e$ is a value, or there exists $e'$ such that $e \mapsto e'$.

Preservation tells us that the dynamic semantics doesn't "run wild". If we start with a well-typed program, then each step of evaluation will necessarily lead to a well-typed program. We can never find ourselves lost in the tall weeds. Progress tells us that evaluation never "gets stuck", unless the computation is complete (*i.e.*, the expression is a value). An example of "getting stuck" is provided by the expression $\mathtt{apply}(3,4)$ — it is easy to check that no transition rule applies. Fortunately, this expression is also ill-typed! Progress tells us that this will always be the case.

Neither preservation nor progress can be expected to hold without some assumptions about the primitive operations. For preservation, we must assume that if the result of applying operation $o$ to arguments $v_1, \ldots, v_n$ is $v$, and $o(v_1, \ldots, v_n) : \tau$, then $v : \tau$. For progress, we must assume that if $o(v_1, \ldots, v_n)$ is well-typed, then there exists a value $v$ such that $v$ is the result of applying $o$ to the arguments $v_1, \ldots, v_n$. For the primitive operations we're considering, these assumptions make sense, but they do preclude introducing "partial" operations, such as division, that are undefined for some arguments. We'll come back to this later in the course.

## 6.2 Type Safety of MinML

**Theorem 22 (Preservation)**
*If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

**Proof:** Note that we are proving not only that $e'$ is well-typed, but that it has the same type as $e$. The proof is by induction on the rules defining one-step evaluation. We will consider each rule in turn.

**[PlusNums]** Here $e = \mathtt{+}(m, n)$, $\tau = \mathtt{int}$, and $e' = m + n$. Clearly $e' : \mathtt{int}$, as required. The other primitive operations are handled similarly.

**[IfTrue]** Here $e = \mathtt{if}\,\mathtt{true}\,\mathtt{then}\,e_1\,\mathtt{else}\,e_2\,\mathtt{fi}$ and $e' = e_1$. Since $e : \tau$, by inversion $e_1 : \tau$, as required.

**[IfFalse]** Here $e = \mathtt{if}\,\mathtt{false}\,\mathtt{then}\,e_1\,\mathtt{else}\,e_2\,\mathtt{fi}$ and $e' = e_2$. Since $e : \tau$, by inversion $e_2 : \tau$, as required.

**[CallFun]** Here $e = \mathtt{apply}(v_1, v_2)$, where $v_1 = \mathtt{fun}\,f\,(x{:}\tau_2){:}\tau\,\mathtt{is}\,e_2\,\mathtt{end}$, and $e' = [v_1, v_2/f, x]e_2$. By inversion applied to $e$, we have $v_1 : \tau_2 \rightarrow \tau$ and $v_2 : \tau_2$. By inversion applied to $v_1$, we have $[f{:}\tau_2 \rightarrow \tau][x{:}\tau_2] \vdash e_2 : \tau$. Therefore, by substitution we have $[v_1, v_2/f, x]e_2 : \tau$, as required.

**[PlusArg$_1$]** Here $e = \texttt{+}(e_1, e_2)$, $e' = \texttt{+}(e_1', e_2)$, and $e_1 \mapsto e_1'$. By inversion $e_1 : \texttt{int}$, so that by induction $e_1' : \texttt{int}$, and hence $e' : \texttt{int}$, as required.

**[PlusArg$_2$]** Here $e = \texttt{+}(v_1, e_2)$, $e' = \texttt{+}(v_1, e_2')$, and $e_2 \mapsto e_2'$. By inversion $e_2 : \texttt{int}$, so that by induction $e_2' : \texttt{int}$, and hence $e' : \texttt{int}$, as required.

The other primitive operations are handled similarly.

**[IfTest]** Here $e = \texttt{if}\, e_1 \,\texttt{then}\, e_2 \,\texttt{else}\, e_3 \,\texttt{fi}$ and $e' = \texttt{if}\, e_1' \,\texttt{then}\, e_2 \,\texttt{else}\, e_3 \,\texttt{fi}$. By inversion we have that $e_1 : \texttt{bool}$, $e_2 : \tau$ and $e_3 : \tau$. By inductive hypothesis $e_1' : \texttt{bool}$, and hence $e' : \tau$.

**[AppFun]** Here $e = \texttt{apply}(e_1, e_2)$ and $e' = \texttt{apply}(e_1', e_2)$. By inversion $e_1 : \tau_2 \rightarrow \tau$ and $e_2 : \tau_2$, for some type $\tau_2$. By induction $e_1' : \tau_2 \rightarrow \tau$, and hence $e' : \tau$.

**[AppArg]** Here $e = \texttt{apply}(v_1, e_2)$ and $e' = \texttt{apply}(v_1, e_2')$. By inversion, $v_1 : \tau_2 \rightarrow \tau$ and $e_2 : \tau_2$, for some type $\tau_2$. By induction $e_2' : \tau_2$, and hence $e' : \tau$.

■

The type of a closed value "predicts" its form.

**Lemma 23 (Canonical Forms)**
*Suppose that $v : \tau$ is a closed, well-formed value.*

1. *If $\tau = \texttt{bool}$, then either $v = \texttt{true}$ or $v = \texttt{false}$.*
2. *If $\tau = \texttt{int}$, then $v = n$ for some $n$.*
3. *If $\tau = \tau_1 \rightarrow \tau_2$, then $v = \texttt{fun}\, f\, \texttt{(}x \,\texttt{:}\, \tau_1\texttt{)} \,\texttt{:}\, \tau_2 \,\texttt{is}\, e \,\texttt{end}$ for some $f$, $x$, and $e$.*

**Proof:** By induction on the typing rules, using the fact that $v$ is a value.

■

**Exercise 24**
*Give a proof of the canonical forms lemma.*

**Theorem 25 (Progress)**
*If $e : \tau$, then either $e$ is a value, or there exists $e'$ such that $e \mapsto e'$.*

**Proof:** The proof is by induction on the typing rules.

**[VarTyp]** Cannot occur, since $e$ is closed.

**[NumTyp],[TrueTyp],[FalseTyp],[FunTyp]** In each case $e$ is a value, which completes the proof.

**[PlusTyp]** Here $e = \texttt{+}(e_1, e_2)$ and $\tau = \texttt{int}$, with $e_1 : \texttt{int}$ and $e_2 : \texttt{int}$. By induction we have either $e_1$ is a value, or there exists $e_1'$ such that $e_1 \mapsto e_1'$ for some expression $e_1'$. In the latter case it follows that $e \mapsto e'$, where $e' = \texttt{+}(e_1', e_2)$. In the former case, we note that by the canonical forms lemma $e_1 = n_1$ for some $n_1$, and we consider $e_2$. By induction either $e_2$ is a value, or $e_2 \mapsto e_2'$ for some expression $e_2'$. If $e_2$ is a value, then by the canonical forms lemma $e_2 = n_2$ for some $n_2$, and we note that $e \mapsto e'$, where $e' = n_1 + n_2$. Otherwise, $e \mapsto e'$, where $e' = \texttt{+}(v_1, e_2')$, as desired.

**[IfTyp]** Here $e = \texttt{if}\, e_1 \,\texttt{then}\, e_2 \,\texttt{else}\, e_3 \,\texttt{fi}$, with $e_1 : \texttt{bool}$, $e_2 : \tau$, and $e_3 : \tau$. By the first inductive hypothesis, either $e_1$ is a value, or there exists $e_1'$ such that $e_1 \mapsto e_1'$. If $e_1$ is a value, then we have by the Canonical Forms Lemma, either $e_1 = \texttt{true}$ or $e_1 = \texttt{false}$. In the former case $e \mapsto e_2$, and in the latter $e \mapsto e_3$, as required. If $e_1$ is not a value, then $e \mapsto e'$, where $e' = \texttt{if}\, e_1' \,\texttt{then}\, e_2 \,\texttt{else}\, e_3 \,\texttt{fi}$, by rule [**IfTest**].

**[AppTyp]** Here $e = \texttt{apply}(e_1, e_2)$, with $e_1 : \tau_2 \rightarrow \tau$ and $e_2 : \tau_2$. By the first inductive hypothesis, either $e_1$ is a value, or there exists $e_1'$ such that $e_1 \mapsto e_1'$. If $e_1$ is not a value, then $e \mapsto \texttt{apply}(e_1', e_2)$ by rule [**AppFun**], as required. By the second inductive hypothesis, either $e_2$ is a value, or there exists $e_2'$ such that $e_2 \mapsto e_2'$. If $e_2$ is not a value, then $e \mapsto e'$, where $e' = \texttt{apply}(e_1, e_2')$, as required. Finally, if both $e_1$ and $e_2$ are values, then by the Canonical Forms Lemma, $e_1 = \texttt{fun}\, f\, \texttt{(}x\,\texttt{:}\,\tau_2\texttt{)}\,\texttt{:}\,\tau \,\texttt{is}\, e'' \,\texttt{end}$, and $e \mapsto e'$, where $e' = [e_1, e_2/f, x]e''$, by rule [**CallFun**].

∎

**Theorem 26 (Safety)**
*If $e$ is well-typed, then evaluation of $e$ can only terminate with a value of the same type. In particular, evaluation cannot "abort" due to an ill-defined state.*

## 6.3 Summary

The main points are these:

1. Type safety expresses the coherence of the static and dynamic semantics. The type of an expression accurately "predicts" the form of its value.

2. Type safety is formalized by a combination of the *preservation* and *progress* theorems. Preservation states that evaluation doesn't "go off into the weeds". Progress states that "you can't get stuck in an undefined state."

# Chapter 7

# Evaluation Semantics and Dynamic Typing

## 7.1 Overview

The dynamic semantics of MinML is given by a transition relation $e \mapsto e'$ defined using Plotkin's method of Structured Operational Semantics (SOS). A significant advantage of SOS is that it admits a crisp statement and proof of type safety, in the form of the Progress and Preservation theorems. A disadvantage of SOS is that it is rather detailed in that it defines the evaluation of expressions step-by-step.

Here we consider another formalism for giving dynamic semantics, called *evaluation semantics (ES)*, in which we define directly the evaluation relation, $e \Downarrow v$, by a set of inference rules. These rules define the unique value $v$ (if such exists) for every expression $e$. In doing so they specify the evaluation order only implicitly, but in the process achieve an appealing simplicity that is lacking in the SOS formulation. On the other hand, evaluation semantics does not admit as a clean formulation of type safety as does SOS, so it is a question of intent as to which is the preferable formalism. Generally speaking, SOS is preferable for proving properties of languages and ES is preferable for describing languages to programmers.

## 7.2 Evaluation Semantics

The evaluation relation, $e \Downarrow v$, for MinML is inductively defined by the following inference rules.

$$\frac{}{n \Downarrow n}$$

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{+(e_1, e_2) \Downarrow n_1 + n_2}$$

(and similarly for the other primitive operations).

$$\frac{}{\texttt{true} \Downarrow \texttt{true}} \qquad \frac{}{\texttt{false} \Downarrow \texttt{false}}$$

$$\frac{e \Downarrow \texttt{true} \quad e_1 \Downarrow v}{\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \Downarrow v}$$

$$\frac{e \Downarrow \texttt{false} \quad e_2 \Downarrow v}{\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \Downarrow v}$$

$$\frac{}{\texttt{fun}\, f\, \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\, \texttt{is}\, e\, \texttt{end} \Downarrow \texttt{fun}\, f\, \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\, \texttt{is}\, e\, \texttt{end}}$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad [v_1, v_2/f, x]e \Downarrow v}{\texttt{apply(}e_1, e_2\texttt{)} \Downarrow v}$$

(where $v_1 = \texttt{fun}\, f\, \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\, \texttt{is}\, e\, \texttt{end}$.)

This concludes the definition of the evaluation semantics of MinML. As you can see, the specification is quite small and is very intuitively appealing.

## 7.3 Correspondences

The precise relationship between SOS and ES is given by the following theorem.

**Theorem 27**

1. *If $e \Downarrow v$, then $e \mapsto^* v$.*
2. *If $e \mapsto e'$ and $e' \Downarrow v$, then $e \Downarrow v$. Consequently, if $e \mapsto^* v$, then $e \Downarrow v$.*

**Proof:**

1. By induction on the rules defining the evaluation relation. The result is clearly true for values, since trivially $v \mapsto^* v$. Suppose that $e = \texttt{apply(}e_1, e_2\texttt{)}$ and assume that $e \Downarrow v$. Then $e_1 \Downarrow v_1$, where $v_1 = \texttt{fun}\, f\, \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\, \texttt{is}\, e\, \texttt{end}$, $e_2 \Downarrow v_2$, and $[v_1, v_2/f, x]e \Downarrow v$. By induction we have that $e_1 \mapsto^* v_1$, $e_2 \mapsto^* v_2$ and $[v_1, v_2/f, x]e \mapsto^* v$. It follows that $\texttt{apply(}e_1, e_2\texttt{)} \mapsto^* \texttt{apply(}v_1, e_2\texttt{)} \mapsto^* \texttt{apply(}v_1, v_2\texttt{)} \mapsto [v_1, v_2/f, x]e \mapsto^* v$, as required. The other cases are handled similarly.

2. By induction on the rules defining single-step transition. Suppose that $e = \texttt{apply(}v_1, v_2\texttt{)}$, where $v_1 = \texttt{fun}\,f\ \texttt{(}x\,\texttt{:}\,\tau_1\texttt{)}\,\texttt{:}\,\tau_2\ \texttt{is}\ e\ \texttt{end}$, and $e' = [v_1, v_2/f, x]e$. Suppose further that $e' \Downarrow v$; we are to show that $e \Downarrow v$. Since $v_1 \Downarrow v_1$ and $v_2 \Downarrow v_2$, the result follows immediately from the assumption that $e' \Downarrow v$. Now suppose that $e = \texttt{apply(}e_1, e_2\texttt{)}$ and $e' = \texttt{apply(}e_1', e_2\texttt{)}$, where $e_1 \mapsto e_1'$. Assume that $e' \Downarrow v$; we are to show that $e \Downarrow v$. It follows that $e_1' \Downarrow v_1$, $e_2 \Downarrow v_2$, and $[v_1, v_2/f, x]e \Downarrow v$. By induction $e_1 \Downarrow v_1$, and hence $e \Downarrow v$. The remaining cases are handled similarly. It follows by induction on the rules defining multi-step evaluation that if $e \mapsto^* v$, then $e \Downarrow v$. The base case, $v \mapsto^* v$, follows from the fact that $v \Downarrow v$. Now suppose that $e \mapsto e' \mapsto^* v$. By induction $e' \Downarrow v$, and hence $e \Downarrow v$ by what we have just proved.

■

## 7.4 Type Safety For Evaluation

One disadvantage of evaluation semantics is that it is not sufficiently detailed as to admit a precise definition of type safety. Recall that in the SOS framework we defined type safety as the conjunction of preservation and progress. Preservation states that the type of an expression doesn't change during evaluation; progress states that well-typed programs cannot "get stuck". Evaluation semantics admits an analogue of type preservation (if $e : \tau$ and $e \Downarrow v$, then $v : \tau$), but there appears to be no way to capture the notion of progress (what would it mean to "get stuck"?).

**Exercise 28**
*Prove type preservation for evaluation semantics by induction on the rules defining evaluation.*

A standard technique for expressing progress in the framework of evaluation semantics is to reduce progress to preservation for an extended evaluation relation. Specifically, we extend the evaluation relation with rules governing "stuck" expressions stating that the value of such an expression is a special token wrong that is, by definition, ill-typed. Since preservation ensures that the value of a well-typed expression is well-typed, the value cannot be wrong. This leads to the slogan "well-typed programs cannot go wrong."

More precisely, we introduce a special syntactic category of *answers* defined by the following grammar:

$$\textit{Answers} \quad a \quad ::= \quad \mathsf{ok}(v) \mid \mathsf{wrong}$$

Here $v$ is a syntactic value, as defined earlier.

Typing for answers is inductively defined by the following single rule:

$$\frac{\vdash v : \tau}{\vdash \mathsf{ok}(v) : \tau}$$

Observe that we consider only closed answers to be well-typed, and that $\mathsf{wrong}$ is ill-typed because there is no typing rule governing it.

The evaluation relation is defined as a binary relation $e \Downarrow a$ between expressions $e$ and answers $a$ defined inductively as follows:

$$\frac{}{n \Downarrow \mathsf{ok}(n)}$$

$$\frac{e_1 \Downarrow \mathsf{ok}(n_1) \quad e_2 \Downarrow \mathsf{ok}(n_2)}{+(e_1, e_2) \Downarrow \mathsf{ok}(n_1 + n_2)}$$

$$\frac{e_1 \Downarrow \mathsf{wrong}}{+(e_1, e_2) \Downarrow \mathsf{wrong}}$$

$$\frac{e_1 \Downarrow \mathsf{ok}(v_1) \quad e_2 \Downarrow \mathsf{wrong}}{+(e_1, e_2) \Downarrow \mathsf{wrong}}$$

$$\frac{e_1 \Downarrow \mathsf{ok}(v_1) \quad v_1 \neq n_1}{+(e_1, e_2) \Downarrow \mathsf{wrong}}$$

$$\frac{e_1 \Downarrow \mathsf{ok}(n_1) \quad e_2 \Downarrow \mathsf{ok}(v_2) \quad v_2 \neq n_2}{+(e_1, e_2) \Downarrow \mathsf{wrong}}$$

(The other primitive operations are handled similarly.)

$$\frac{}{\texttt{true} \Downarrow \mathsf{ok}(\texttt{true})} \qquad \frac{}{\texttt{false} \Downarrow \mathsf{ok}(\texttt{false})}$$

$$\frac{e \Downarrow \mathsf{ok}(\texttt{true}) \quad e_1 \Downarrow a}{\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \Downarrow a}$$

$$\frac{e \Downarrow \mathsf{ok}(\texttt{false}) \quad e_2 \Downarrow a}{\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \Downarrow a}$$

$$\frac{e \Downarrow \mathsf{wrong}}{\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \Downarrow \mathsf{wrong}}$$

$$\frac{e \Downarrow \mathsf{ok}(v) \quad v \notin \{\,\mathtt{true}, \mathtt{false}\,\}}{\mathtt{if}\; e\; \mathtt{then}\; e_1\; \mathtt{else}\; e_2\; \mathtt{fi} \Downarrow \mathsf{wrong}}$$

$$\frac{}{\mathtt{fun}\; f\; (x\!:\!\tau_1)\!:\!\tau_2\; \mathtt{is}\; e\; \mathtt{end} \Downarrow \mathsf{ok}(\mathtt{fun}\; f\; (x\!:\!\tau_1)\!:\!\tau_2\; \mathtt{is}\; e\; \mathtt{end})}$$

$$\frac{e_1 \Downarrow \mathsf{ok}(v_1) \quad e_2 \Downarrow \mathsf{ok}(v_2) \quad [v_1, v_2/f, x]e \Downarrow a}{\mathtt{apply}(e_1, e_2) \Downarrow a}$$

(where $v_1 = \mathtt{fun}\; f\; (x\!:\!\tau_1)\!:\!\tau_2\; \mathtt{is}\; e\; \mathtt{end}$.)

$$\frac{e_1 \Downarrow \mathsf{wrong}}{\mathtt{apply}(e_1, e_2) \Downarrow \mathsf{wrong}}$$

$$\frac{e_1 \Downarrow \mathsf{ok}(v_1) \quad e_2 \Downarrow \mathsf{wrong}}{\mathtt{apply}(e_1, e_2) \Downarrow \mathsf{wrong}}$$

$$\frac{e_1 \Downarrow \mathsf{ok}(v_1) \quad v_1 \neq \mathtt{fun}\; f\; (x\!:\!\tau_1)\!:\!\tau_2\; \mathtt{is}\; e\; \mathtt{end}}{\mathtt{apply}(e_1, e_2) \Downarrow \mathsf{wrong}}$$

The changes we have made are these:

1. Introduce rules that check for malformed arguments such as a non-numeral value as argument to a numeric primitive operation, a non-boolean value as a test in a conditonal, or a non-function value being applied to an argument.

2. Introduce rules that propagate wrong from sub-expressions. Once evaluation goes wrong it stays wrong!

Why does this work? We have the following correspondence:

**Theorem 29**

*1. $e \Downarrow \mathsf{ok}(v)$ iff $e \mapsto^* v$.*

*2. $e \Downarrow \mathsf{wrong}$ iff $e \mapsto^* e' \not\mapsto$, where $e'$ is not a value.*

Thus evaluation to wrong corresponds exactly to getting stuck, and evaluation to $\mathsf{ok}(v)$ corresponds exactly to successful evaluation to a value.

**Exercise 30**

*Re-prove preservation for the extended evaluation semantics by induction on evaluation. For each rule yielding wrong as answer, you must argue that the rule cannot apply, given that the expression is well-typed. For example, if $+(e_1, e_2)$ is well-typed, then by inversion it must have type `int`, as must both $e_1$ and $e_2$. By the canonical forms lemma and induction, the evaluation of $e_1$ and $e_2$ must yield well-typed answers of the form $\mathsf{ok}(n_1)$ and $\mathsf{ok}(n_2)$, respectively. Therefore none of the three rules yielding wrong can apply.*

It is important that you understand how the proof of progress is embedded (contrapositively) in the proof of preservation for the extended semantics.

## 7.5 Dynamic Typing

The introduction of an instrumented semantics that checks for type errors during execution suggests another method for ensuring safety. The key observation is that the instrumented semantics makes it possible to execute a program regardless of whether it is well-typed — "stuck" expressions are simply those that evaluate to wrong, rather than those for which evaluation is undefined. This suggests that we can dispense with the static type system entirely, relying instead on the instrumentation to catch type faults during execution. Such languages are said to be *dynamically typed.* The best-known (and first) dynamically-typed language is Lisp.

To understand what is at stake, it is helpful to consider what it would mean to implement the instrumented semantics for MinML given above. This question doesn't arise as long as we restrict our attention to (statically) well-typed programs — the extended preservation theorem ensures that none of the additional rules can ever be executed. But if we are to dispense with static typing we must, of course, implement them.

The key issue is how to implement a rule such as this:

$$\frac{e_1 \Downarrow \mathsf{ok}(v_1) \quad v_1 \neq \mathtt{fun}\, f\ (x : \tau_1) : \tau_2\ \mathtt{is}\ e\ \mathtt{end}}{\mathtt{apply}(e_1, e_2) \Downarrow \mathsf{wrong}}$$

in which we check whether the value of the function position of an application is indeed a function. Inside of a machine, all values are simply bit patterns. Functions might be represented as pointers to regions of memory containing machine language instructions. Faced with a word, how can we determine whether or not it is a valid pointer to a valid function? In general we cannot, so we must instead plan to support these checks in our representations of data objects.

Fundamentally, we must be prepared to support the following operations:

1. Testing whether a given value an integer, a boolean, or a function.

2. Extracting from a valid integer its underlying integer value, or from a valid boolean its underlying truth value, or from a valid function its underlying code.

3. Marking a value as an integer, boolean, or function.

One way to implement these operations is via a *tagging* mechanism whereby all values are marked with a tag indicating whether it is an integer, boolean, or function.[1] It is important to distinguish tags from types! The tag on a value only indicates its *shape*, or *outermost form*; it does not indicate anything deeper about the value. For example, all functions would be tagged as functions, without regard to their domain or range types. This is enough to ensure safety; anything more than this is overkill (and would require much more work to implement.)

The main ideas of tag checking can be illustrated by a few rules. Let us define the set of *tagged values* by the following grammar:

$$t \quad ::= \quad \texttt{Int}\,(n) \mid \texttt{Bool}\,(\texttt{true}) \mid \texttt{Bool}\,(\texttt{false}) \mid \texttt{Func}\,(\texttt{fun}\,f\,(x\,{:}\,\tau_1)\,{:}\,\tau_2\ \texttt{is}\ e\ \texttt{end})$$

The set of answers is re-defined in terms of tagged values:

$$a \quad ::= \quad \mathsf{ok}(t) \mid \mathsf{wrong}$$

The evaluation rules are modified to make explicit the manipulation of tags. For example, the rules for numbers are as follows:

$$\frac{}{n \Downarrow \mathsf{ok}(\texttt{Int}\,(n))}$$

$$\frac{e_1 \Downarrow \mathsf{ok}(\texttt{Int}\,(n_1)) \quad e_2 \Downarrow \mathsf{ok}(\texttt{Int}\,(n_2))}{+(e_1, e_2) \Downarrow \mathsf{ok}(\texttt{Int}\,(n_1 + n_2))}$$

$$\frac{e_1 \Downarrow \mathsf{ok}(\texttt{Bool}\,(v_1))}{+(e_1, e_2) \Downarrow \mathsf{wrong}}$$

$$\frac{e_1 \Downarrow \mathsf{ok}(\texttt{Func}\,(v_1))}{+(e_1, e_2) \Downarrow \mathsf{wrong}}$$

$$\frac{e_1 \Downarrow \mathsf{ok}(\texttt{Int}\,(n_1)) \quad e_2 \Downarrow \mathsf{ok}(\texttt{Bool}\,(v_2))}{+(e_1, e_2) \Downarrow \mathsf{wrong}}$$

$$\frac{e_1 \Downarrow \mathsf{ok}(\texttt{Int}\,(n_1)) \quad e_2 \Downarrow \mathsf{ok}(\texttt{Func}\,(v_2))}{+(e_1, e_2) \Downarrow \mathsf{wrong}}$$

The other rules are handled similarly. Observe that evaluation of an integer constant demands work to attach a tag to the value, and that the evaluation

[1] In a richer language more tags would be required, for example to indicate whether a value is a pair or a floating point number, etc.

of a primitive operation requires that these tags be checked for validity and that the result of a valid addition be tagged as an integer.

If we were assuming a static type discipline, all this machinery would be unfortunate because we are forced to pay the cost of tag manipulation, even though the static type checks prove that it is unnecessary. This semantics accrues advantage only in the case that we are interested in running programs that are not statically typed (or even statically typable). Examples include

1. Conditionals with branches of differing type. For example, in a dynamically typed language the expression `if true then 1 else true fi + 7` is safe to execute and will not incur a run-time type error. Yet it is statically ill-typed.

2. Aggregates, such as lists, with values of different type. For example, in a dynamically typed variant of ML we could form the list `[1,"one",true,fn x=>x]`, even though it is ill-typed.

These examples are said to be useful, precisely because they avoid rejecting programs that do not incur a type fault at run-time, even though it looks as though they might when analyzed purely statically (without regard to run-time behavior). In this sense type faults are isolated *as late as possible*, rather than *as early as possible* (and hence conservatively) as in a statically-typed framework.

To make this work well a dynamically typed language must support run-time type checking (really, run-time tag checking). For example, if $l$ is a heterogeneous list and we wish to compute with its third element, we must perform a run-time test to determine whether it is an integer, or boolean, or function. Having made this determination we can use it safely, without fear of incurring a run-time type fault.

Much energy has been expended on debating the relative merits of static and dynamic typing, with the interlocutors in the debate typically advocating one position or the other as if they are in opposition to one another. But this debate misses a fundamental point: *static and dynamic typing are not opposite viewpoints on language design, rather dynamic typing is a special case of static typing (in a sufficiently rich type system)!*

The point can be made quite succinctly by observing the encoding of dynamic typing directly in ML using recursive data types. Consider the following program:

```
(* The type of tagged values. *)
datatype tagged =
  Int of int | Bool of bool | Function of tagged -> tagged
exception TypeError
fun checked_add (m:tagged, n:tagged):tagged =
    case (m,n) of
      (Int a, Int b) => Int (a+b)
    | (_, _) => raise TypeError
fun checked_apply (f:tagged, a:tagged):tagged =
    case f of
      Function g => g a
    | _ => raise TypeError
```

Here we introduce a type of tagged data values, and define primitive operations on it corresponding to the "checked" version of those operations in the instrumented dynamic semantics. This representation of tagging is fully faithful to the checked operational semantics. Moreover, it is that the virtue that we pay for tag checking exactly to the extent that we use it! Addition still corresponds to a single machine instruction, provided that we're adding integers; if, instead, we're adding tagged values, we pay the cost of pattern matching against the tag and tagging the result. If we wish to use dynamic checking, we can, but we are not forced to do so. In contrast a dynamically typed language admits *only* the checked operations; we cannot recover the more efficient behavior of a statically typed language without sacrificing safety.

# Chapter 8

# Abstract Machines

## 8.1 Overview

Long considered to be a topic of primarily academic interest, *abstract*, or *virtual*, *machines* are now attracting attention in the software industry. The main idea is to define an instruction set for a "pseudo-computer", the abstract machine, that may be used as the object code for compiling a high-level language (such as ML or Java) and that may be implemented with reasonable efficiency on a wide variety of stock platforms. This means that the high-level language must be implemented only once, for the abstract machine, but that the abstract machine must itself be implemented once per platform. One advantage is that it is, in principle, much easier to port the abstract machine than it is to re-implement the language for each platform. More importantly, this architecture supports the exchange of object code across the network — if everyone implements the abstract machine, then code can migrate from one computer to another without modification. Web sites all over the world exploit this capability to tremendous advantage, using the Java Virtual Machine.

Before we get started, let us ask ourselves the question: what is an abstract machine? In other words, what is a computer? The fundamental idea of computation is the notion of step-by-step execution of *instructions* that transform the *state* of the computer in some determinate fashion.[1] Each instruction should be executable in a finite amount of time using a finite amount of information, and it should be clear how to effect the required

[1]The question of determinacy is increasingly problematic for real computers, largely because of the aggressive use of parallelism in their implementation. We will gloss over this issue here.

state transformation using only physically realizable methods.[2] Execution of a program consists of initializing the machine to a known *start state*, executing instructions one-by-one until no more instructions remains; the result of the computation is the *final state*. Thus an abstract machine is essentially a *transition system* between states of that machine.

According to this definition the dynamic semantics of MinML is an abstract machine, the M machine. The states of the M machine are closed MinML expressions $e$, and the transitions are given by the one-step evaluation relation $e \mapsto_{\mathsf{M}} e'$ defined earlier. This machine is quite high-level in the sense that the instructions are fairly complex compared to what are found in typical concrete machines. For example, the M machine performs substitution of a value for a variable in one step, a decidedly large-scale (but nevertheless finite and effective) instruction. This machine is also odd in another sense: rather than have an analogue of a program counter that determines the next instruction to be executed, we instead have "search rules" that traverse the expression to determine what to do next. As you have no doubt observed, this can be quite an involved process, one that is not typical of real computers. We will begin to address these concerns by first looking at the management of the flow of control in an abstract machine, and then considering the management of bindings of values to variables.

## 8.2 Control Flow

Rather than repeatedly traverse an expression looking for the next instruction to execute, we can maintain an explicit record of what to do next in the computation using an abstract *control stack* that maintains a record of the work remaining to be done (in reverse order) to finish evaluating an expression. We will call this machine the C *machine*, to remind us that it is defined to capture the idea of control flow in a computation.

The states of the C machine have the form $(k, e)$, where $k$ is a control stack and $e$ is a closed expression. Control stacks are inductively defined by the following rules:

$$\frac{}{\bullet\ \mathsf{stack}} \qquad \frac{f\ \mathsf{frame} \qquad k\ \mathsf{stack}}{f \triangleright k\ \mathsf{stack}}$$

[2] For example, an instruction that, given the representation of a program, sets register zero to zero iff there is an input on which that program halts when executed could not be regarded as the instruction of any computing device that we could ever physically realize, because of the unsolvability of the halting problem.

The set of *stack frames* is inductively defined by these rules:

$$\frac{}{+(\Box, e_2)\ \mathsf{frame}} \qquad \frac{}{+(v_1, \Box)\ \mathsf{frame}}$$

(There are analogous frames associated with the other primitive operations.)

$$\frac{}{\texttt{if}\,\Box\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi}\ \mathsf{frame}}$$

$$\frac{}{\texttt{apply(}\Box, e_2\texttt{)}\ \mathsf{frame}} \qquad \frac{}{\texttt{apply(}v_1, \Box\texttt{)}\ \mathsf{frame}}$$

Thus a control stack is a sequence of frames $f_1 \triangleright \cdots f_n \triangleright \bullet$ (implicitly right-associated), where $\bullet$ is the empty stack and each $f_i$ $(1 \leq i \leq n)$ is a stack frame. Each stack frame represents one step in the process of searching for the next position to evaluate in an expression.

The transition relation for the $\mathsf{C}$ machine is inductively defined by a set of transition rules. We begin with the rules for addition; the other primitive operations are handled similarly.

$$(k, +(e_1, e_2)) \mapsto_{\mathsf{C}} (+(\Box, e_2) \triangleright k, e_1)$$

$$(+(\Box, e_2) \triangleright k, v_1) \mapsto_{\mathsf{C}} (+(v_1, \Box) \triangleright k, e_2)$$

$$(+(n_1, \Box) \triangleright k, n_2) \mapsto_{\mathsf{C}} (k, n_1 + n_2)$$

The first two rules capture the left-to-right evaluation order for the arguments of addition. The top stack frame records the current position within the argument list; when the last argument has been evaluated, the operation is applied and the stack is popped.

Next, we consider the rules for the conditional.

$$(k, \texttt{if}\,e\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi}) \mapsto_{\mathsf{C}} (\texttt{if}\,\Box\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi} \triangleright k, e)$$

$$(\texttt{if}\,\Box\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi} \triangleright k, \texttt{true}) \mapsto_{\mathsf{C}} (k, e_1)$$

$$(\texttt{if}\,\Box\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi} \triangleright k, \texttt{false}) \mapsto_{\mathsf{C}} (k, e_2)$$

These rules follow the same pattern. First, the test expression is evaluated, recording the pending conditional branch on the stack. Once the value of the test has been determined, we branch to the appropriate arm of the conditional.

Finally, we consider the rules for application of functions.

$$(k, \texttt{apply(}e_1, e_2\texttt{)}) \mapsto_{\mathsf{C}} (\texttt{apply(}\Box, e_2\texttt{)} \triangleright k, e_1)$$

$$(\texttt{apply(}\Box, e_2\texttt{)} \triangleright k, v_1) \mapsto_{\mathsf{C}} (\texttt{apply(}v_1, \Box\texttt{)} \triangleright k, e_2)$$

$$(\texttt{apply}(v_1, \Box) \triangleright k, v_2) \mapsto_{\mathsf{C}} (k, [v_1, v_2/f, x]e)$$

The last rule applies in the case that $v_1 = \texttt{fun}\, f\, \texttt{(}x\,\texttt{:}\,\tau_1\texttt{)}\,\texttt{:}\,\tau_2\, \texttt{is}\, e\, \texttt{end}$. These rules ensure that the function is evaluated before the argument, applying the function when both have been evaluated.

The final states of the C machine have the form $(\bullet, v)$ consisting of the empty stack (no further work to do) and a value $v$.

The rules defining the C machine have *no premises* — they are all simple transitions, without any hypotheses. We've made explicit the management of the "subgoals" required for evaluating expressions using the M by introducing a stack of pending sub-goals that specifies the order in which they are to be considered. In this sense the C is less abstract than the M machine. It is interesting to examine your implementation of the M machine, and compare it to an implementation of the C machine. The M machine implementation makes heavy use of the ML runtime stack to implement the recursive calls to the MinML interpreter corresponding to premises of the evaluation rules. The runtime stack is required because the interpreter is *not* a tail recursive function. In contrast an implementation of the C *is* tail recursive, precisely because there are no premises on any of the transitions rules defining it.

What is the relationship between the M machine and the C machine? Do they define the same semantics for the MinML language? Indeed they do, but a rigorous proof of this fact is surprisingly tricky to get right. The hardest part is to figure out how to state the correspondence precisely; having done that, the verification is not difficult.

The first step is to define a correspondence between C states and M states. Intuitively the control stack in the C corresponds to the "surrounding context" of an expression, which is saved for consideration once the expression has been evaluated. Thus a C state may be thought of as representing the M state obtained by "unravelling" the control stack and plugging in the current expression to reconstruct the entire program as a single expression. The function that does this, written $k \,@\, e$, is defined by induction on the structure of $k$ as follows:

$$\begin{array}{rcl}
\bullet \,@\, e & = & e \\
+(\Box, e_2) \triangleright k \,@\, e_1 & = & k \,@\, +(e_1, e_2) \\
+(v_1, \Box) \triangleright k \,@\, e_2 & = & k \,@\, +(v_1, e_2) \\
\texttt{if}\, \Box\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \triangleright k \,@\, e & = & k \,@\, \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \\
\texttt{apply}(\Box, e_2) \triangleright k \,@\, e & = & k \,@\, \texttt{apply}(e, e_2) \\
\texttt{apply}(v_1, \Box) \triangleright k \,@\, e & = & k \,@\, \texttt{apply}(v_1, e)
\end{array}$$

The precise correspondence between the two machines is given by the following theorem.

**Theorem 31**

1. *If* $(k, e) \mapsto_{\mathsf{C}} (k', e')$, *then either* $k \mathbin{@} e = k' \mathbin{@} e'$, *or* $k \mathbin{@} e \mapsto_{\mathsf{M}} k' \mathbin{@} e'$.

2. *If* $e \mapsto_{\mathsf{M}} e'$ *and* $(k, e') \mapsto^*_{\mathsf{C}} (\bullet, v)$, *then* $(k, e) \mapsto^*_{\mathsf{C}} (\bullet, v)$.

The first part of the Theorem states that the C transitions are either "bookkeeping" steps that move a piece of the program onto the control stack without materially changing the overall program, or "instruction" steps that correspond to transitions in the M machine. The second part is a bit tricky to understand, at first glance. It says that if the M machine moves from a state $e$ to a state $e'$, and the C machine runs to completion starting from $e'$ and an arbitrary stack $k$, then it also runs to completion starting from $e$ and $k$.[3]

**Proof:**

1. By a straightforward induction on the definition of the C machine. The "search" rules do not change the program as a whole; the "instruction" rules correspond to transitions in the M machine. Observe that the next M machine instruction to be executed in $k \mathbin{@} e$ must lie within $e$, and that the result of executing that instruction must be $k \mathbin{@} e'$ for some $e'$.

2. By induction on the MinML dynamic semantics. We will do the cases for application here; the remainder follow a similar pattern.

   (a) $e = \texttt{apply}(v_1, v_2) \mapsto_{\mathsf{M}} [v_1, v_2/f, x]e_2 = e'$, where the value $v_1 = \texttt{fun}\, f\, (x{:}\tau_2){:}\tau\, \texttt{is}\, e_2\, \texttt{end}$. Suppose that $(k, e') \mapsto^*_{\mathsf{C}} (\bullet, v)$. By the definition of the C transition relation,

   $$\begin{array}{rll} (k, e) & \mapsto_{\mathsf{C}} & (\texttt{apply}(\Box, v_2) \triangleright k, v_1) \\ & \mapsto_{\mathsf{C}} & (\texttt{apply}(v_1, \Box) \triangleright k, v_2) \\ & \mapsto_{\mathsf{C}} & (k, e') \end{array}$$

   From this, the result follows immediately.

   (b) $e = \texttt{apply}(e_1, e_2) \mapsto_{\mathsf{M}} \texttt{apply}(e_1', e_2) = e'$, where $e_1 \mapsto_{\mathsf{M}} e_1'$. Suppose that $(k, e') \mapsto^*_{\mathsf{C}} (\bullet, v)$. Since $e' = \texttt{apply}(e_1', e_2)$, and since

[3] Half the battle in establishing a correspondence between the two machines was to find the proper statement of the correspondence! So you should not be dismayed if it takes some time to understand what is being said here, and why.

the $\mathsf{C}$ machine is deterministic, this transition sequence must have the form

$$(k, e') = (k, \texttt{apply}(e_1', e_2)) \mapsto_{\mathsf{C}} (\texttt{apply}(\Box, e_2) \triangleright k, e_1') \mapsto_{\mathsf{C}}^* (\bullet, v)$$

By the inductive hypothesis, using the enlarged stack, it follows that

$$(\texttt{apply}(\Box, e_2) \triangleright k, e_1) \mapsto_{\mathsf{C}}^* (\bullet, v).$$

Now since

$$(k, e) = (k, \texttt{apply}(e_1, e_2)) \mapsto_{\mathsf{C}} (\texttt{apply}(\Box, e_2) \triangleright k, e_1)$$

the result follows immediately.

(c) $e = \texttt{apply}(v_1, e_2) \mapsto_{\mathsf{M}} \texttt{apply}(v_1, e_2') = e'$, where $e_2 \mapsto_{\mathsf{M}} e_2'$. Suppose that $(k, e') \mapsto_{\mathsf{C}}^* (\bullet, v)$. Since $e' = \texttt{apply}(v_1, e_2')$, and since the $\mathsf{C}$ machine is deterministic, this transition sequence must have the form

$$(k, e') = (k, \texttt{apply}(v_1, e_2')) \mapsto_{\mathsf{C}} (\texttt{apply}(v_1, \Box) \triangleright k, e_2') \mapsto_{\mathsf{C}}^* (\bullet, v)$$

By the inductive hypothesis, using the enlarged stack, it follows that

$$(\texttt{apply}(v_1, \Box) \triangleright k, e_2) \mapsto_{\mathsf{C}}^* (\bullet, v).$$

Now since

$$(k, e) = (k, \texttt{apply}(v_1, e_2)) \mapsto_{\mathsf{C}} (\texttt{apply}(v_1, \Box) \triangleright k, e1)$$

the result follows immediately.

■

**Exercise 32**
*Finish the proof of the theorem by giving a complete proof of part (1), and filling in the missing cases in part (2).*

**Corollary 33**

1. *If* $(k, e) \mapsto_{\mathsf{C}}^* (\bullet, v)$, *then* $k \mathbin{@} e \mapsto_{\mathsf{M}}^* v$. *Hence if* $(\bullet, e) \mapsto_{\mathsf{C}}^* (\bullet, v)$, *then* $e \mapsto_{\mathsf{M}}^* v$.
2. *If* $e \mapsto_{\mathsf{M}}^* e'$ *and* $(k, e') \mapsto_{\mathsf{C}}^* (\bullet, v)$, *then* $(k, e) \mapsto_{\mathsf{C}}^* (\bullet, v)$. *Hence if* $e \mapsto_{\mathsf{M}}^* v$, *then* $(\bullet, e) \mapsto_{\mathsf{C}}^* (\bullet, v)$.

**Proof:**

1. By induction on the transition sequence, making use of part (1) of the theorem. Then take $k = \bullet$.

2. By induction on transition sequence, making use of part (2) of the theorem. Then take $e' = v$ and $k = \bullet$.

■

## 8.3 Environments

The C is still quite "high level" in that function application is performed by substitution of the function itself and its argument into the body of the function, a rather complex operation. This is unrealistic for two reasons. First, substitution is a complicated process, not one that we would ordinarily think of as occurring as a single step of execution of a computer. Second, and perhaps more importantly, the use of substitution means that the program itself, and not just the data it acts upon, changes during evaluation. This is a radical departure from more familiar models of computation, which maintain a rigorous separation between program and data. In this section we will present another abstraction machine, the E machine, which avoids substitution by introducing an *environment* that records the bindings of variables.

The basic idea is simple: rather than *replace* variables by their bindings when performing a function application, we instead *record* the bindings of variables in a data structure, and, correspondingly, *look up* the bindings of variables when they are used. In a sense we are performing substitution "lazily", rather than "eagerly", to avoid unnecessary duplication and to avoid modifying the program during execution. The main complication introduced by environments is that we must exercise considerable caution to ensure that we do not confuse the scopes of variables.[4] It is remarkably easy, if we are not careful, to confuse the bindings of variables that happen to have the same name. We avoid difficulties by introducing *closures*, data structures that package an expression together with an environment.

To see the point, let's first sketch out the structure of the E machine. A state of the E machine has the form $(K, E, e)$, where $K$ is a *machine stack*,

[4] In fact, the notion of "dynamic scope" arose as a result of an error in the original Lisp interpreter (circa 1960) that confused the scopes of variables.

$E$ is an *environment*, a finite function mapping variables to *machine values*, and $e$ is an open expression such that $\mathrm{FV}(e) \subseteq \mathrm{dom}(E)$. Machine values are values "inside the machine", distinct from the syntactic notion of value used in the M and C machines. The reason for the distinction arises from the replacement of substitution by binding.

Since the M and C machines perform function application by substitution, there is never any need to consider expressions with free variables in them; the invariant that the expression part of the state is closed is maintained throughout evaluation. The whole point of the E, however, is to avoid substitution by maintaining an environment that records the bindings of free variables. When a function is called, the parameter is bound to the argument, the function name is bound to the function itself, and the body is evaluated; when that is complete the bindings of the function name and parameter can be released, and evaluation continues.

This suggests that the environment is a global, stack-like data structure onto which arguments are pushed and popped during evaluation — values are pushed on function call and popped on function return. In fact, the environment might be called the *data stack* for precisely this reason. However, a moment's thought reveals that this characterization is a tad too simplistic, because it overlooks a crucial issue in the implementation of functional languages, namely the ability to return functions as results of function applications. Suppose that $f$ is a function of type $\texttt{int}\rightarrow\texttt{int}\rightarrow\texttt{int}$. When applied to an integer $n$, the result $\texttt{apply}(f, n)$ yields a function of type $\texttt{int}\rightarrow\texttt{int}$. For example, $f$ might be the following function:

$$\texttt{fun}\,\_\,(x\,\texttt{:int})\,\texttt{:int}\rightarrow\texttt{int is fun}\,\_\,(y\,\texttt{:int})\,\texttt{:int is}\ x\ \texttt{end end},$$

Observe that the function returned by $f$ contains a free occurrence of the parameter $x$ of $f$. If we follow the simple stack-like discipline of function call and return, we will, upon calling $f$, bind $x$ to 1, yielding the value

$$\texttt{fun}\,\_\,(y\,\texttt{:int})\,\texttt{:int is}\ x\ \texttt{end},$$

then pop the binding of $x$ from the environment. *But wait a minute!* The returned value is a function that contains a free occurrence of $x$, and we've just deleted the binding for $x$ from the environment! Subsequent uses of this function will either capture some other binding for $x$ that happens to be in the environment at the time it is used, violating the static scoping principle,[5], or incur an unbound variable error if no binding for $x$ happens to be available.

[5]This is the error in the original implementation of Lisp referred to earlier.

This problem is avoided by the use of *closures*. The value returned by the application $\texttt{apply(}f, 1\texttt{)}$ is the closure

$$\texttt{fun}\,\_\,\texttt{(}y\texttt{:int):int is}\ x\ \texttt{end}[E[x \mapsto 1]]^6$$

where $E$ is the environment in effect at the point of the call. When $f$ returns the binding for $x$ is indeed popped from the *global* environment, but a *local* copy of it is retained in the closure returned by $f$. This way no confusion or capture is possible, and the static scoping discipline is maintained, even in the absence of substitution.

The need for closures motivates the distinction between syntactic values, $v$, and machine values, $V$. The latter are inductively defined by the following rules:

$$\frac{}{n\ \mathsf{mvalue}} \qquad \frac{}{\texttt{true}\ \mathsf{mvalue}} \qquad \frac{}{\texttt{false}\ \mathsf{mvalue}}$$

$$\frac{}{\texttt{fun}\ f\ \texttt{(}x\texttt{:}\tau_1\texttt{):}\tau_2\ \texttt{is}\ e\ \texttt{end}[E]\ \mathsf{mvalue}}$$

An *environment*, $E$, is a finite function mapping variables to machine values.

The set of *machine stacks* is inductively defined by the following rules:

$$\frac{}{\bullet\ \mathsf{mstack}} \qquad \frac{F\ \mathsf{mframe} \qquad K\ \mathsf{mstack}}{F \triangleright K\ \mathsf{mstack}}\ .$$

Here $F$ is a *machine frame*. The set of machine frames is inductively defined by these rules:

$$\frac{}{+(\Box, e_2)[E]\ \mathsf{mframe}} \qquad \frac{}{+(V_1, \Box)\ \mathsf{mframe}}$$

$$\frac{}{\texttt{if}\ \Box\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \texttt{fi}[E]\ \mathsf{mframe}}$$

$$\frac{}{\texttt{apply(}\Box, e_2\texttt{)}[E]\ \mathsf{mframe}} \qquad \frac{}{\texttt{apply(}V_1, \Box\texttt{)}\ \mathsf{mframe}}$$

The notation for $\mathsf{E}$ machine frames is deceptively similar to the notation for $\mathsf{C}$ frames. Note, however, that $\mathsf{E}$ frames involve machine values, and that in many cases the frame is closed with respect to an environment recording the bindings of the free variables in the expressions stored in the frame. The second form of addition and application frames need no environment; do you see why?

The $\mathsf{E}$ machine has two kinds of states: $(K, E, e)$, described earlier, and "auxiliary" states of the form $(K, V)$, where $K$ is a machine stack and $V$ is

[6]In this case the rest of the environment, $E$, is superfluous. In general we can cut down the closing environment to just those variables that actually occur in the body of the function. We will ignore this optimization for the time being.

a machine value. The auxiliary state represents the passage of a machine value to the top frame of the machine stack. (In the $\mathsf{C}$ machine this is accomplished by simply filling the hole in the stack frame, but here a bit more work is required.)

The $\mathsf{E}$ is inductively defined by a set of rules for transitions of one of the following four forms:

$$
\begin{array}{rcll}
(K, E, e) & \mapsto_{\mathsf{E}} & (K', E', e') & \textit{process expression} \\
(K, E, v) & \mapsto_{\mathsf{E}} & (K', V') & \textit{pass value from expression to stack} \\
(K, V) & \mapsto_{\mathsf{E}} & (K', V') & \textit{pass value to stack} \\
(K, V) & \mapsto_{\mathsf{E}} & (K', E', e') & \textit{process pending expression}
\end{array}
$$

We will use the same transition relation for all four cases, relying on the form of the states to disambiguate which is intended.

To evaluate a variable $x$, we look up its binding and pass the associated value to the top frame of the control stack.

$$(K, E, x) \mapsto_{\mathsf{E}} (K, E(x))$$

Similarly, to evaluate numeric or boolean constants, we simply pass them to the control stack.

$$(K, E, n) \mapsto_{\mathsf{E}} (K, n)$$

$$(K, E, \texttt{true}) \mapsto_{\mathsf{E}} (K, \texttt{true}) \qquad (K, E, \texttt{false}) \mapsto_{\mathsf{E}} (K, \texttt{false})$$

To evaluate a function expression, we close it with respect to the current environment to ensure that its free variables are not inadvertently captured, and pass the resulting closure to the control stack.

$$(K, E, \texttt{fun}\, f\; \texttt{(}x\,\texttt{:}\,\tau_1\texttt{)}\,\texttt{:}\,\tau_2\; \texttt{is}\; e\; \texttt{end}) \mapsto_{\mathsf{E}} (K, \texttt{fun}\, f\; \texttt{(}x\,\texttt{:}\,\tau_1\texttt{)}\,\texttt{:}\,\tau_2\; \texttt{is}\; e\; \texttt{end}[E])$$

To evaluate a primitive operation, we start by evaluating its first argument, pushing a frame on the control stack that records the need to evaluate its remaining arguments.

$$(K, E, +(e_1, e_2)) \mapsto_{\mathsf{E}} (+(\Box, e_2)[E] \triangleright K, E, e_1)$$

Notice that the frame is closed in the current environment to avoid capture of free variables in the remaining arguments.

To evaluate a conditional, we evaluate the test expression, pushing a frame on the control stack to record the two pending branches, once again closed with respect to the current environment.

$$(K, E, \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi}) \mapsto_{\mathsf{E}} (\texttt{if}\, \Box\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi}[E] \triangleright K, E, e)$$

To evaluate an application, we begin by evaluating the function position, pushing a frame to record the pending evaluation of the argument, closed with respect to the current environment.

$$(K, E, \texttt{apply}(e_1, e_2)) \mapsto_{\mathsf{E}} (\texttt{apply}(\square, e_2)[E] \triangleright K, E, e_1)$$

To complete the definition of the $\mathsf{E}$ machine, we must define the transitions governing the auxiliary states.

Pending argument evaluations for primitive operations are handled as follows. If more arguments remain to be evaluated, we switch states to process the next argument.

$$(+(\square, e_2)[E] \triangleright K, V_1) \mapsto_{\mathsf{E}} (o(V_1, \square) \triangleright K, E, e_2)$$

Notice that the environment of the frame is used to evaluate the next argument. If no more arguments remain to be evaluated, we pass the result of executing the primitive operation to the rest of the stack.

$$(+(n_1, \square) \triangleright K, n_2) \mapsto_{\mathsf{E}} (K, n_1 + n_2)$$

Pending conditional branches are handled in the obvious manner.

$$(\texttt{if}\,\square\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi}[E] \triangleright K, \texttt{true}) \mapsto_{\mathsf{E}} (K, E, e_1)$$

$$(\texttt{if}\,\square\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi}[E] \triangleright K, \texttt{false}) \mapsto_{\mathsf{E}} (K, E, e_2)$$

Notice that the environment of the frame is restored before evaluating the appropriate branch of the conditional.

Pending function applications are handled as follows.

$$(\texttt{apply}(\square, e_2)[E] \triangleright K, V) \mapsto_{\mathsf{E}} (\texttt{apply}(V, \square) \triangleright K, E, e_2)$$

Observe that the environment of the frame is restored before evaluating the argument of the application, and that the function value (which is, presumbly, a closure) is stored intact in the new top frame of the stack.

Once the argument has been evaluated, we call the function.

$$(\texttt{apply}(V, \square) \triangleright K, V_2) \mapsto_{\mathsf{E}} (K, E[f \mapsto V][x \mapsto V_2], e)$$

where

$$V = \texttt{fun}\,f\,\texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,e\,\texttt{end}[E].$$

To call the function we *bind* $f$ to $V$ and $x$ to $V_2$ in the environment of the closure, continuing with the evaluation of the body of the function. Observe that since we use the environment of the closure, extended with bindings for the function and its parameter, we ensure that the appropriate bindings for the free variables of the function are employed.

The final states of the $\mathsf{E}$ machine have the form $(\bullet, V)$, with final result $V$. Notice that the result is a machine value. If the type of the entire program is `int` or `bool`, then $V$ will be a numeral or a boolean constant, respectively. Otherwise the value will be a closure.

A correspondence between the $\mathsf{E}$ and the $\mathsf{C}$ along the lines of the correspondence between the $\mathsf{C}$ and the $\mathsf{M}$ machine may be established. However, since the technical details are rather involved, we will not pursue a rigorous treatment of the relationship here. Suffice it to say that if $e$ is a closed MinML program of base type (`int` or `bool`), then $(\bullet, e) \mapsto_{\mathsf{C}} (\bullet, v)$ iff $(\bullet, \emptyset, e) \mapsto_{\mathsf{E}} (\bullet, v)$. (The restriction to base type is necessary if we are to claim that both machines return the *same* value.)

# Chapter 9

# Computational Effects

## 9.1 Introduction

MinML is a purely functional language, in two senses. First, the only control mechanism is procedure call and return. Procedures may call themselves recursively, which is the sole means of looping in the language. Second, evaluation is purely a matter of calculation. A program is an expression whose value we wish to calculate. Expressions "stand for" their values; the primitive operations are defined in terms of values.

This simple model of evaluation must be generalized to account for *computational effects* (or simply *effects*) such as mutation of storage or non-local transfer of control. The purpose of this note is to study the extension of MinML with effects. First, we will add *mutable reference cells* to MinML in the style of Standard ML. With this extension programs not only have values, as before, but also engender *store effects*, modifications to mutable cells. Second, we will add *exceptions* to MinML that allow non-local transfers of control. Programs may now also exhibit *control effects*, or changes to the execution context effected by branching to an exception handler.

## 9.2 References

The MinML type language is extended with *reference types* $\tau\,\texttt{ref}$ whose elements are to be thought of as mutable storage cells. We correspondingly extend the expression language with these primitive operations:

$$e \ ::= \ l \mid \texttt{ref}(e) \mid \texttt{!}e \mid e_1\texttt{:=}e_2$$

As in Standard ML, $\texttt{ref(}e\texttt{)}$ allocates a "new" reference cell, $\texttt{!}e$ retrieves the contents of the cell $e$, and $e_1\texttt{:=}e_2$ sets the contents of the cell $e_1$ to the value $e_2$. The variable $l$ ranges over a set of *locations*, an infinite set of identifiers disjoint from variables. These are needed for the dynamic semantics, but are not expected to be notated directly by the programmer. The set of *values* is extended to include locations.

Typing judgments have the form $\Lambda;\Gamma \vdash e : \tau$, where $\Lambda$ is a *location typing*, a finite function mapping locations to types; the other components of the judgement are as for MinML. The location typing $\Lambda$ records the types of allocated locations during execution; this is critical for a precise statement and proof of type soundness.

The typing rules are those of MinML (extended to carry a location typing), plus the following rules governing the new constructs of the language:

$$\frac{(\Lambda(l) = \tau)}{\Lambda;\Gamma \vdash l : \tau\,\texttt{ref}} \qquad \frac{\Lambda;\Gamma \vdash e : \tau}{\Lambda;\Gamma \vdash \texttt{ref(}e\texttt{)} : \tau\,\texttt{ref}}$$

$$\frac{\Lambda;\Gamma \vdash e : \tau\,\texttt{ref}}{\Lambda;\Gamma \vdash \texttt{!}e : \tau} \qquad \frac{\Lambda;\Gamma \vdash e_1 : \tau_2\,\texttt{ref} \qquad \Lambda;\Gamma \vdash e_2 : \tau_2}{\Lambda;\Gamma \vdash e_1\texttt{:=}e_2 : \tau_2}$$

Notice that the location typing is not extended during type checking! Locations arise only during execution, and are not part of complete programs, which must not have any free locations in them. The role of the location typing will become apparent in the proof of type safety for MinML extended with references.

A *memory* is a finite function mapping locations to closed values (but possibly involving locations). The dynamic semantics of MinML with references is given by an abstract machine. The states of this machine have the form $(M, e)$, where $M$ is a memory and $e$ is an expression possibly involving free locations in the domain of $M$. The locations in $\mathrm{dom}(M)$ are bound simultaneously in $(M, e)$; the names of locations may be changed at will without changing the identity of the state.

The transitions for this machine are similar to those of the M machine, but with these additional steps:

$$\frac{e \mapsto e'}{(M, \texttt{ref(}e\texttt{)}) \mapsto (M, \texttt{ref(}e'\texttt{)})} \qquad \frac{(l \notin \mathrm{dom}(M))}{(M, \texttt{ref(}v\texttt{)}) \mapsto (M[l{=}v], l)}$$

$$\frac{e \mapsto e'}{(M, \texttt{!}e) \mapsto (M, \texttt{!}e')} \qquad \frac{(l \in \mathrm{dom}(M))}{(M, \texttt{!}l) \mapsto (M, M(l))}$$

$$\frac{e_1 \mapsto e_1'}{(M, e_1\texttt{:=}e_2) \mapsto (M, e_1'\texttt{:=}e_2)} \qquad \frac{e_2 \mapsto e_2'}{(M, v_1\texttt{:=}e_2) \mapsto (M, v_1\texttt{:=}e_2')}$$

$$\frac{(l \in \mathrm{dom}(M))}{(M, l\,\mathtt{:=}v) \mapsto (M[l{=}v], v)}$$

A state $(M, e)$ is *final* iff $e$ is a value (possibly a location).

To prove type safety for this extension we will make use of some auxiliary relations. Most importantly, the typing relation between memories and location typings, written $\vdash M : \Lambda$, is inductively defined by the following rule:

$$\frac{\mathrm{dom}(M) = \mathrm{dom}(\Lambda) \quad \forall l \in \mathrm{dom}(\Lambda)\ \Lambda; \bullet \vdash M(l) : \Lambda(l)}{\vdash M : \Lambda}$$

It is very important to study this rule carefully! First, we require that $\Lambda$ and $M$ govern the same set of locations. Second, for each location $l$ in their common domain, we require that the value at location $l$, namely $M(l)$, have the type assigned to $l$, namely $\Lambda(l)$, relative to the *entire* location typing $\Lambda$. This means, in particular, that memories may be "circular" in the sense that the value at location $l$ may contain an occurrence of $l$, for example if that value is a function.

The typing rule for memories is reminiscent of the typing rule for recursive functions — we are allowed to assume the typing that we are trying to prove while trying to prove it. This similarity is no accident, as the following example shows. Here we use ML notation, but the example can be readily translated into MinML extended with references:

```
(* loop forever when called *)
fun diverge (x:int):int = diverge x
(* allocate a reference cell *)
val fc : (int->int) ref = ref (diverge)
(* define a function that ``recurs'' through fc *)
fun f 0 = 1 | f n = n * ((!fc)(n-1))
(* tie the knot *)
val _ = fc := f
(* now call f *)
val n = f 5
```

This technique is called *backpatching*. It is used in some compilers to implement recursive functions (and other forms of looping construct).

**Exercise 34**

1. *Sketch the contents of the memory after each step in the above example. Observe that after the assignment to* `fc` *the memory is "circular" in the sense that some location contains a reference to itself.*

2. *Prove that every cycle in well-formed memory must "pass through" a function. Suppose that $M(l_1) = l_2$, $M(l_2) = l_3$, $\ldots$, $M(l_n) = l_1$ for some sequence $l_1, \ldots, l_n$ of locations. Show that there is no location typing $\Lambda$ such that $\vdash M : \Lambda$.*

The well-formedness of a machine state is inductively defined by the following rule:

$$\frac{\vdash M : \Lambda \qquad \Lambda; \bullet \vdash e : \tau}{(M, e)\ \mathsf{ok}}$$

That is, $(M, e)$ is well-formed iff there is a location typing for $M$ relative to which $e$ is well-typed.

**Theorem 35 (Preservation)**
*If $(M, e)$ $\mathsf{ok}$ and $(M, e) \mapsto (M', e')$, then $(M', e')$ $\mathsf{ok}$.*

**Proof:** The trick is to prove a stronger result by induction on evaluation: if $(M, e) \mapsto (M', e')$, $\vdash M : \Lambda$, and $\Lambda; \bullet \vdash e : \tau$, then there exists $\Lambda' \supseteq \Lambda$ such that $\vdash M' : \Lambda'$ and $\Lambda'; \bullet \vdash e' : \tau$. ∎

**Exercise 36**
*Prove Theorem 35. The strengthened form tells us that the location typing, and the memory, increase monotonically during evaluation — the type of a location never changes once it is established at the point of allocation. This is crucial for the induction.*

**Theorem 37 (Progress)**
*If $(M, e)$ $\mathsf{ok}$ then either $(M, e)$ is a final state or there exists $(M', e')$ such that $(M, e) \mapsto (M', e')$.*

**Proof:** The proof is by induction on typing: if $\vdash M : \Lambda$ and $\Lambda; \bullet \vdash e : \tau$, then either $e$ is a value or there exists $M' \supseteq M$ and $e'$ such that $(M, e) \mapsto (M', e')$. ∎

**Exercise 38**
*Prove Theorem 37 by induction on typing of machine states.*

## 9.3 Exceptions

The purpose of an exception mechanism is to provide a means of effecting a non-local transfer of control to an exception handler. Exceptions are typically used to signal the occurrence of an unexpected or unusual event requiring special treatment. A significant advantage of an exception mechanism over explicit error checking is that the flow of control is not disrupted by the unusual case. In particular explicit error checks are easily forgotten or neglected, whereas exceptions cannot be simply ignored. For example, the typical case for opening a new file for writing might be to return a file descriptor, which only rarely fails due to file system overflow. By arranging the code for opening a new file to signal an exception in this case we can prevent accidental neglect of this condition without introducing unduly complex code.

We'll consider the extension of MinML with an exception mechanism similar to that of Standard ML, with the significant simplification that no value is associated with the exception — we simply signal the exception and thereby invoke the nearest dynamically enclosing handler. We'll come back to consider value-passing exceptions later.

The following grammar describes the extensions to MinML to support valueless exceptions:

$$e \quad ::= \quad \ldots \mid \texttt{fail} \mid \texttt{try}\, e_1 \,\texttt{ow}\, e_2$$

The expression `fail` raises an exception. The expression $\texttt{try}\, e_1 \,\texttt{ow}\, e_2$ evaluates $e_1$. If it terminates normally, we return its value; otherwise, if it fails, we continue by evaluating $e_2$.

The static semantics of exceptions is quite straightforward:

$$\frac{}{\Gamma \vdash \texttt{fail} : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{try}\, e_1 \,\texttt{ow}\, e_2 : \tau}$$

Observe that a failure can have any type, precisely because it never returns. Both clauses of a handler must have the same type, to allow for either possible outcome of evaluation.

The dynamic semantics of exceptions is given in terms of the C machine with an explicit control stack. The set of frames is extended with the following additional clause:

$$\frac{e_2 \ \mathsf{expr}}{\texttt{try}\, \Box \,\texttt{ow}\, e_2 \ \mathsf{frame}}$$

The evaluation rules are extended as follows:

$$\frac{}{(K, \texttt{try}\, e_1 \,\texttt{ow}\, e_2) \mapsto (\texttt{try}\, \Box \,\texttt{ow}\, e_2 \triangleright K, e_1)}$$

$$\frac{}{(\texttt{try}\, \Box \,\texttt{ow}\, e_2 \triangleright K, v) \mapsto (K, v)}$$

$$\frac{}{(\mathtt{try}\,\Box\,\mathtt{ow}\,e_2 \triangleright K, \mathtt{fail}) \mapsto (K, e_2)}$$

$$\frac{(F \neq \mathtt{try}\,\Box\,\mathtt{ow}\,e_2)}{(F \triangleright K, \mathtt{fail}) \mapsto (K, \mathtt{fail})}$$

To evaluate $\mathtt{try}\,e_1\,\mathtt{ow}\,e_2$ we begin by evaluating $e_1$. If it achieves a value, we "pop" the pending handler and yield that value. If, however, it fails, we continue by evaluating the "otherwise" clause of the nearest enclosing handler. Notice that we explicitly "pop" non-handler frames while processing a failure; this is sometimes called *unwinding* the control stack. Finally, we regard the state $(\bullet, \mathtt{fail})$ as a final state of computation, corresponding to an uncaught exception.

**Exercise 39**
*Hand-simulate the evaluation of a few simple expressions with exceptions and handlers to get a feeling for how it works.*

To prove safety we define well-formedness of machine states by the following rule:

$$\frac{\vdash K : \tau\,\mathtt{stack} \qquad \vdash e : \tau}{(K, e)\ \mathsf{ok}}$$

That is, a state $(K, e)$ is well-formed iff $e$ is an expression of type $\tau$ and $K$ is a $\tau$-accepting control stack. The latter is defined by the following rules:

$$\frac{}{\vdash \bullet : \tau\,\mathtt{stack}} \qquad \frac{\vdash F : (\tau,\tau')\ \mathtt{frame} \quad \vdash K : \tau'\,\mathtt{stack}}{\vdash F \triangleright K : \tau\,\mathtt{stack}}$$

A stack is well-formed iff its frames compose properly. The type of a frame is defined by the following rules:

$$\frac{\vdash e_2 : \mathtt{int}}{\vdash +(\Box, e_2) : (\mathtt{int},\mathtt{int})\ \mathtt{frame}} \qquad \frac{v_1\ \mathsf{value} \quad \vdash v_1 : \mathtt{int}}{\vdash +(v_1, \Box) : (\mathtt{int},\mathtt{int})\ \mathtt{frame}}$$

$$\frac{\vdash e_1 : \tau \quad \vdash e_2 : \tau}{\vdash \mathtt{if}\,\Box\,\mathtt{then}\,e_1\,\mathtt{else}\,e_2\,\mathtt{fi} : (\mathtt{bool},\tau)\ \mathtt{frame}}$$

$$\frac{\vdash e_2 : \tau_2}{\vdash \mathtt{apply}(\Box, e_2) : (\tau_2{\rightarrow}\tau,\tau)\ \mathtt{frame}} \qquad \frac{v_1\ \mathsf{value} \quad \vdash v_1 : \tau_2{\rightarrow}\tau}{\vdash \mathtt{apply}(v_1, \Box) : (\tau_2,\tau)\ \mathtt{frame}}$$

$$\frac{\vdash e_2 : \tau}{\vdash \mathtt{try}\,\Box\,\mathtt{ow}\,e_2 : (\tau,\tau')\ \mathtt{frame}}$$

Intuitively, a frame of type $(\tau_1,\tau_2)$ $\mathtt{frame}$ takes an "argument" of type $\tau_1$ and yields a "result" of type $\tau_2$. The argument is represented by the "$\Box$" in the frame; the result is the type of the frame once its hole has been filled with an expression of the given type.

With this in place we can state and prove safety.

**Theorem 40 (Preservation)**
*If* $(K, e)$ ok *and* $(K, e) \mapsto (K', e')$, *then* $(K, e)$ ok.

**Proof:** By induction on evaluation. ∎

**Exercise 41**
*Prove Theorem 40.*

**Theorem 42 (Progress)**
*If* $(K, e)$ ok *then either*

1. $K = \bullet$ *and* $e$ value, *or*
2. $K = \bullet$ *and* $e =$ `fail`, *or*
3. *there exists* $K'$ *and* $e'$ *such that* $(K, e) \mapsto (K', e')$.

**Proof:** By induction on typing. ∎

**Exercise 43**
*Prove Theorem 42.*

**Exercise 44**
*Combine the treatment of references and exceptions to form a language with both of these features. You will face a choice of how to define the interaction between mutation and exceptions:*

1. *As in ML, mutations are irrevocable, even in the face of exceptions that "backtrack" to a surrounding handler.*
2. *Invocation of a handler rolls back the memory to the state at the point of installation of the handler.*

*Give a dynamic semantics for each alternative, and argue for and against each choice.*

The dynamic semantics of exceptions is somewhat unsatisfactory because of the explicit unwinding of the control stack to find the nearest enclosing handler. While this does effect a non-local transfer of control, it does so by rather crude means, rather than by a direct "jump" to the handler. In practice exceptions are implemented as jumps, using the following ideas. A dedicated register is set aside to contain the "current" exception handler.

When an exception is raised, the current handler is retrieved from the exception register, and control is passed to it. Before doing so, however, we must reset the exception register to contain the nearest handler enclosing the new handler. This ensures that if the handler raises an exception the correct handler is invoked. How do we recover this handler? We maintain a stack of pending handlers that is pushed whenever a handler is installed, and popped whenever a handler is invoked. The exception register is the top element of this stack. Note that we must restore the control stack to the point at which the handler was installed before invoking the handler!

This can be modelled by a machine with states of the form $(H, K, e)$, where

- $H$ is a handler stack;
- $K$ is a control stack;
- $e$ is a closed expression

A handler stack consists of a stack of pairs consisting of a handler together its associated control stack:

$$\frac{}{\bullet\ \mathsf{hstack}} \qquad \frac{K\ \mathsf{stack} \quad e\ \mathsf{expr} \quad H\ \mathsf{hstack}}{(K, e) \triangleright H\ \mathsf{hstack}}$$

A handler stack element consists of a "freeze dried" control stack paired with a pending handler.

The key transitions of the machine are given by the following rules. On failure we pop the control stack and pass to the exception stack:

$$\frac{}{((K', e') \triangleright H, K, \mathtt{fail}) \mapsto (H, K', e')}$$

We pop the handler stack, "thaw" the saved control stack, and invoke the saved handler expression. If there is no pending handler, we stop the machine:

$$\frac{}{(\bullet, K, \mathtt{fail}) \mapsto (\bullet, \bullet, \mathtt{fail})}$$

To install a handler we preserve the handler code and the current control stack:

$$\frac{}{(H, K, \mathtt{try}\, e_1\, \mathtt{ow}\, e_2) \mapsto ((K, e_2) \triangleright H, \mathtt{try}\, \Box\, \mathtt{ow}\, e_2 \triangleright K, e_1)}$$

We "freeze dry" the control stack, associate it with the unevaluated handler, and push it on the handler stack. We also push a frame on the control stack to remind us to remove the pending handler from the handler stack in the case of normal completion of evaluation of $e_1$:

$$\frac{}{((K, e_2) \triangleright H, \mathtt{try}\, \Box\, \mathtt{ow}\, e_2 \triangleright K, v_1) \mapsto (H, K, v_1)}$$

**Exercise 45**
*State and prove the safety of this formulation of exceptions.*

The idea of "freeze-drying" an entire control stack and "thawing" it later may seem like an unusually heavy-weight operation. However, a key invariant governing a machine state $(H, K, e)$ is the following *prefix property*: if $H = (K', e') \triangleright H'$, then $K'$ is a prefix of $K$. This means that we can store a control stack by simply keeping a "finger" on some initial segment of it, and can restore a saved control stack by popping up to that finger.

**Exercise 46**
*Prove that the prefix property is preserved by every step of evaluation.*

Finally, let us consider value-passing exceptions such as are found in Standard ML. The main idea is to replace the failure expression, `fail`, by a more general *raise* expression, `raise(`$e$`)`, which associates a value (that of $e$) with the failure. Handlers are generalized so that the "otherwise" clause is a function accepting the value associated with the failure, and yielding a value of the same type as the "try" clause. Here is a sketch of the static semantics for this variation:

$$\frac{\Gamma \vdash e : \tau_{\mathsf{exn}}}{\Gamma \vdash \mathtt{raise}(e) : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau_{\mathsf{exn}} \to \tau}{\Gamma \vdash \mathtt{try}\, e_1 \,\mathtt{ow}\, e_2 : \tau}$$

These rules are parameterized by the type of values associated with exceptions, $\tau_{\mathsf{exn}}$.

The question is: what should be the type $\tau_{\mathsf{exn}}$? The first thing to observe is that *all* exceptions should be of the same type, otherwise we cannot guarantee type safety. The reason is that a handler might be invoked by *any* raise expression occurring during the execution of its "try" clause. If one exception raised an integer, and another a boolean, the handler could not safely dispatch on the exception value. Given this, we must choose a type $\tau_{\mathsf{exn}}$ that supports a flexible programming style.

For example, we might choose, say, `string`, for $\tau_{\mathsf{exn}}$, with the idea that the value associated with an exception is a description of the cause of the exception. For example, we might write

```
fun div (m, 0) = raise "Division by zero attempted."
  | div (m, n) = ... raise "Arithmetic overflow occurred." ...
```

However, consider the plight of the poor handler, which may wish to distinguish between division-by-zero and arithmetic overflow. How might it do that? If exception values were strings, it would have to parse the string, relying on the message to be in a standard format, and dispatch based on the parse. This is manifestly unworkable. For similar reasons we wouldn't choose $\tau_{\mathsf{exn}}$ to be, say, `int`, since that would require coding up exceptions as numbers, much like "error numbers" in Unix. Again, completely unworkable in practice, and completely unmodular (different modules are bound to conflict over their numbering scheme).

A more reasonable choice would be to define $\tau_{\mathsf{exn}}$ to be a given datatype `exc`. For example, we might have the declaration

```
datatype exc = Div | Overflow | Match | Bind
```

as part of the implicit prelude of every program. Then we'd write

```
fun div (m, 0) = raise Div
  | div (m, n) = ... raise Overflow ...
```

Now the handler can easily dispatch on `Div` or `Overflow` using pattern matching, which is much better. However, this choice restricts all programs to a *fixed* set of exceptions, the value constructors associated with the pre-declared `exc` datatype.

To allow extensibility Standard ML includes a special *extensible datatype* called `exn`. Values of type `exn` are similar to values of a datatype, namely they are constructed from other values using a constructor. Moreover, we may pattern match against values of type `exn` in the usual way. But, in addition, we may introduce *new* constructors of type `exn` "on the fly", rather than declare a fixed set at the beginning of the program. Such new constructors are introduced using an exception declaration such as the following:

```
exception Div
exception Overflow
```

Now `Div` and `Overflow` are constructors of type `exn`, and may be used in a `raise` expression or matched against by an exception handler. Exception declarations can occur anywhere in the program, and are guaranteed (by $\alpha$-conversion) to be distinct from all other exceptions that may occur elsewhere in the program, even if they happen to have the same name. If two modules declare an exception named `Error`, then these are *different* exceptions; no confusion is possible.

The interesting thing about the `exn` type is that *it has nothing whatsoever to do with the exception mechanism* (beyond the fact that it is the type of values associated with exceptions). In particular, the `exception` declaration introduces a value constructor that has no inherent connection with the exception mechanism. We may use the `exn` type for other purposes; indeed, Java has an analogue of the type `exn`, called `Object`. This is the basis for downcasting and so-called typecase in Java.

# Chapter 10

# First-Class Continuations

## 10.1 Introduction

The treatment of exceptions in terms of a handler and control stack relies on the *reification* of control stacks as values that can be pushed on the handler stack. At first glance this appears to be a rather heavyweight operation that would involve copying the entire control stack when establishing a handler, and restoring it when raising an exception. However, we observed that the machine satisfies a crucial invariant, namely that the saved control stack is always an initial segment of the current control stack. This allows us to reify a control stack as a "finger" in the control stack, and to install it by popping the stack back to the finger. This is a formal justification of an implementation based on the `setjmp` and and `longjmp` constructs of the C language. Unlike `setjmp` and `longjmp`, the exception mechanism is completely safe — it is impossible to return past the "finger" yet later attempt to "pop" the control stack to that point. In C the fingers are kept as addresses (pointers) in memory, and there is no discipline for ensuring that the set point makes any sense when invoked later in a computation.

The idea of reification of control stacks can be taken a step further, by allowing them to be passed as values within a program and to be restored at a later point, *even if* control has long since returned past the point of reification. Reified control stacks of this kind are called *first-class continuations*, where the qualification "first class" stresses that they are ordinary values with an indefinite lifetime that can be passed and returned at will in a computation. In contrast to set points in C first-class continuations never "expire", and it is always sensible to reinstate a continuation without compromising safety. Thus first-class continuations support unlimited "time travel" — we can go back to a previous point in the computation and then return to some point in its future, at will.

How is this achieved? The key to implementing first-class continuations is to arrange that control stacks are *persistent* data structures, just like any other data structure in ML that does not involve mutable references. By a persistent data structure we mean one for which operations on it yield a "new" version of the data structure without disturbing the old version. For example, lists in ML are persistent in the sense that if we cons an element to the front of a list we do not thereby destroy the original list, but rather yield a new list with an additional element at the front, retaining the possibility of using the old list for other purposes. In this sense persistent data structures allow time travel — we can easily switch between several versions of a data structure without regard to the temporal order in which they were created. This is in sharp contrast to more familiar *ephemeral* data structures for which operations such as insertion of an element irrevocably mutate the data structure, preventing any form of time travel.

Returning to the case in point, the standard implementation of a control stack is as an ephemeral data structure, a pointer to a region of mutable storage that is overwritten whenever we push a frame. This makes it impossible to maintain an "old" and a "new" copy of the control stack at the same time, making time travel impossible. If, however, we represent the control stack as a persistent data structure, then we can easily reify a control stack by simply binding it to a variable, and continue working. If we wish we can easily return to that control stack by referring to the variable that is bound to it. This is achieved in practice by representing the control stack as a list of frames in the heap so that the persistence of lists can be extended to control stacks. While we will not be specific about implementation strategies in this note, it should be born in mind when considering the semantics outlined below.

Why are first-class continuations useful? Fundamentally, they are representations of the control state of a computation at a given point in time. Using first-class continuations we can "checkpoint" the control state of a program, save it in a data structure, and return to it later. In fact this is precisely what is necessary to implement *threads* (concurrently executing programs) — the thread scheduler must be able to checkpoint a program and save it for later execution, perhaps after a pending event occurs or another thread yields the processor. Later in the course we will show how to build a threads package for concurrent programming using continuations.

## 10.2 Informal Overview of Continuations

We will extend MinML with the type $\tau$ `cont` of continuations accepting values of type $\tau$. A continuation will, in fact, be a control stack of type $\tau$ `stack`, but rather than expose this representation to the programmer, we will regard $\tau$ `cont` as an abstract type supporting two operations, `letcc` $x$ `in` $e$ and `throw` $e_1$ `to` $e_2$.[1]

Informally, evaluation of `letcc` $x$ `in` $e$ binds the *current continuation*[2] to $x$ and evaluates $e$. The current continuation as, as we've discussed, a reification of the current control stack, which represents the current point in the evaluation of the program. The type of $x$ is $\tau$ `cont`, where $\tau$ is the type of $e$. The intuition is that the current continuation is the point to which $e$ returns when it completes evaluation. Consequently, the control stack expects a value of type $\tau$, which then determines how execution proceeds. Thus $x$ is bound to a stack expecting a value of type $\tau$, that is, a value of type $\tau$ `cont`. Note that this is the *only* way to obtain a value of type $\tau$ `cont`; there are no expressions that evaluate to continuations. (This is similar to our treatment of references — values of type $\tau$ `ref` are locations, but locations can only be obtained by evaluating a `ref` expression.)

We may "jump" to a saved control point by *throwing* a value to a continuation, written `throw` $e_1$ `to` $e_2$. The expression $e_2$ must evaluate to a $\tau_1$ `cont`, and $e_1$ must evaluate to a value of type $\tau_1$. The current control stack is abandoned in favor of the reified control stack resulting from the evaluation of $e_2$; the value of $e_1$ is then passed to that stack.

Here is a simple example, written in Standard ML notation. The idea is to multiply the elements of a list, short-circuiting the computation in case `0` is encountered. Here's the code:

```
fun mult_list (l:int list):int =
    callcc
      (fn ret =>
        let fun mult (nil) = 1
              | mult (0::_) = throw ret 0
              | mult (n::l) = n * mult (l)
         in mult (l) end)
```

Ignoring the `callcc` for the moment, the body of `mult_list` is a `let` expression that defines a recursive procedure `mult`, and applies it to the argument

[1] Close relatives of these primitives are available in SML/NJ in the following forms: for `letcc` $x$ `in` $e$, write `callcc (fn` $x$ `=>` $e$`)`, and for `throw` $e_1$ `to` $e_2$, write `throw (`$e_2$`,`$e_1$`)`.

[2] Hence the name "`letcc`".

of mult_list. The job of mult is to return the product of the elements of the list times the value of the accumulator; by calling mult with l and 1, we obtain the product of the elements of l. Ignoring the second line of mult, it should be clear why and how this code works.

Now let's consider the second line of mult, and the outer application of callcc. Intuitively, the purpose of the second line of mult is to short circuit the multiplication, returning 0 immediately in the case that a 0 occurs in the list. This is achieved by throwing the value 0 (the final answer) to the continuation bound to the variable ret. This variable is bound by the call to callcc surrounding the body of mult_list. What continuation is it? It's the continuation that runs upon completion of the body of mult_list. This continuation would be executed in the case that no 0 is encountered and evaluation proceeds normally. In the unusual case of encountering a 0 in the list, we branch directly to the return point, passing the value 0, effecting an early return from the procedure with result value 0.

Here's a slicker formulation of the same function:

```
val mult_list : int list -> int =
    let fun mult ret nil = 1
          | mult ret (0::_) = throw ret 0
          | mult ret (n::l) = n * mult ret l
     in callcc mult end
```

Here the inner loop is parameterized by the return continuation for early exit. The multiplication loop is obtained by calling mult with the current continuation at the exit point of mult_list so that throws to ret effect an early return from mult_list, as desired.

**Exercise 47**
*Study this example carefully to be sure you understand why it works!*

Let's look at another example: given a continuation $k$ of type $\tau$ cont and a function $f$ of type $\tau' \rightarrow \tau$, return a continuation $k'$ of type $\tau'$ cont with the following behavior: throwing a value $v'$ of type $\tau'$ to $k'$ throws the value $f(v')$ to $k$. This is called *composition of a function with a continuation.* We wish to fill in the following template:

```
fun compose (f:τ'->τ,k:τ cont):τ' cont = ...
```

The function compose will have type

```
((τ' -> τ) * τ cont) -> τ' cont
```

**Exercise 48**
*This is a very difficult programming problem! But please take a few moments to try to solve it before reading on. The solution is very instructive, but is, for most people, rather hard to think up.*

The first problem is to obtain the continuation we wish to return. The second problem is how to return it. The continuation we seek is the one in effect at the point of the ellipsis in the expression `throw` $f$`(...) to` $k$. This is the continuation that, when given a value $v'$, applies $f$ to it, and throws the result to $k$. We can seize this continuation using `letcc`, writing

```
throw f(letcc x:τ' cont in ...) to k
```

At the point of the ellipsis the variable $x$ is bound to the continuation we wish to return. How can we return it? By using the same trick as we used for short-circuiting evaluation above! We don't want to actually throw a value to this continuation (yet), instead we wish to abort it and return it as the result. Here's the final code, in SML/NJ notation:

```
fun compose (f, k) =
    callcc
      (fn return =>   (* return point *)
          throw k (f (callcc (fn k' => throw return k'))))
```

The type of `return` is $\tau'$ `cont cont`, a continuation expecting a continuation expecting a value of type $\tau'$!

We can do without first-class continuations by "rolling our own". The idea is that we can perform (by hand or automatically) a systematic program transformation in which a "copy" of the control stack is maintained as a function, called a continuation. Every function takes as an argument the control stack to which it is to pass its result by applying given stack (represented as a function) to the result value. Functions never return in the usual sense; they pass their result to the given continuation. Programs written in this form are said to be in *continuation-passing style*, or *CPS* for short.

Here's the code to multiply the elements of a list (without short-circuiting) in continuation-passing style:

```
fun cps_mult nil k = k 1
  | cps_mult (n::l) k = cps_mult l (fn r => k (n * r))
fun mult l = cps_mult l (fn r => r)
```

It's easy to implement the short-circuit form by passing an additional continuation, the one to invoke for short-circuiting the result:

```
fun cps_mult_list l k =
    let fun cps_mult nil k0 k = k 1
      | fun cps_mult (0::_) k0 k = k0 0
      | fun cps_mult (n::l) k0 k = cps_mult k0 l (fn p => k (n*p))
    in cps_mult l k k end
```

Notice that `k0` never changes; it is always the return continuation for `cps_mult_list`!

Observe that the type of the first version of `cps_mult` becomes

$$\texttt{int list}\rightarrow(\texttt{int}\rightarrow\alpha)\rightarrow\alpha,$$

and that the type of the second version becomes

$$\texttt{int list}\rightarrow(\texttt{int}\rightarrow\alpha)\rightarrow(\texttt{int}\rightarrow\alpha)\rightarrow\alpha,$$

These transformations are representative of the general case.

## 10.3 Semantics of Continuations

The informal description of evaluation is quite complex, as you no doubt have observed. Here's an example where a formal semantics is *much* clearer, and can serve as a useful guide for understanding how all of this works. The semantics is suprisingly simple and intuitive.

First, the abstract syntax. We extend the language of MinML types with continuation types of the form $\tau\,\texttt{cont}$. We extend the language of MinML expressions with these additional forms:

$$e \quad ::= \quad \ldots \mid \texttt{letcc}\,x\,\texttt{in}\,e \mid \texttt{throw}\,e_1\,\texttt{to}\,e_2 \mid K$$

In the expression $\texttt{letcc}\,x\,\texttt{in}\,e$ the variable $x$ is bound in $e$. As usual we rename bound variables implicitly as convenient. We include control stacks $K$ as expressions for the sake for the sake of the dynamic semantics, much as we included locations as expressions when considering reference types. We define continuations thought of as expressions to be values:

$$\frac{K\ \mathsf{stack}}{K\ \mathsf{value}}$$

Stacks are as defined for the $\mathsf{C}$ machine, extended with these additional frames:

$$\frac{e_2\ \mathsf{expr}}{\texttt{throw}\,\Box\,\texttt{to}\,e_2\ \mathsf{frame}} \qquad \frac{v_1\ \mathsf{value}}{\texttt{throw}\,v_1\,\texttt{to}\,\Box\ \mathsf{frame}}$$

Second, the static semantics. The typing rules governing the continuation primitives are these:

$$\frac{\Gamma[x{:}\tau\,\texttt{cont}] \vdash e : \tau}{\Gamma \vdash \texttt{letcc}\, x\, \texttt{in}\, e : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_1\,\texttt{cont}}{\Gamma \vdash \texttt{throw}\, e_1\, \texttt{to}\, e_2 : \tau'}$$

The result type of a `throw` expression is arbitrary because it does not return to the point of the call. The typing rule for continuation values is as follows:

$$\frac{\vdash K : \tau\,\texttt{stack}}{\Gamma \vdash K : \tau\,\texttt{cont}}$$

That is, a continuation value $K$ has type $\tau\,\texttt{cont}$ exactly if it is a stack accepting values of type $\tau$. This relation is as defined in our treatment of exceptions, extended to include the additional frames mentioned above.

Finally, the dynamic semantics. We use the $\mathsf{C}$ machine as a basis. We extend the language of expressions to include control stacks $K$ as values. Like locations, these arise only during execution; there is no explicit notation for continuations in the language. The key transitions are as follows:

$$\frac{}{(K, \texttt{letcc}\, x\, \texttt{in}\, e) \mapsto (K, [K/x]e)} \qquad \frac{}{(\texttt{throw}\, v\, \texttt{to}\, \Box \triangleright K, K') \mapsto (K', v)}$$

In addition we specify the order of evaluation of arguments to `throw`:

$$\frac{}{(K, \texttt{throw}\, e_1\, \texttt{to}\, e_2) \mapsto (\texttt{throw}\, \Box\, \texttt{to}\, e_2 \triangleright K, e_1)}$$

$$\frac{}{(\texttt{throw}\, \Box\, \texttt{to}\, e_2 \triangleright K, v_1) \mapsto (\texttt{throw}\, v_1\, \texttt{to}\, \Box \triangleright K, e_2)}$$

Notice that evaluation of `letcc` *duplicates* the control stack, and that evaluation of `throw` *eliminates* the current control stack.

**Exercise 49**

*Simulate the evaluation of* `compose (f, k)` *on the empty stack. Observe that the control stack substituted for $x$ is*

$$\texttt{apply}(f, \Box) \triangleright \texttt{throw}\, \Box\, \texttt{to}\, k \triangleright \bullet$$

*This stack is returned from* `compose`*. Next, simulate the behavior of throwing a value $v'$ to this continuation. Observe that the above stack is reinstated and that $v'$ is passed to it.*

The safety of this extension of $\mathsf{MinML}$ may be established using familiar techniques. First we must define well-formedness for machine states. As before, we define $(K, e)$ $\mathsf{ok}$ iff $\vdash K : \tau\,\texttt{stack}$ and $\vdash e : \tau$.

The preservation theorem is stated as follows:

**Theorem 50 (Preservation)**
*If* $(K, e)$ ok *and* $(K, e) \mapsto (K', e')$, *then* $(K', e')$ ok.

**Proof:** The proof is by induction on evaluation. The verification is left as an exercise. ∎

To establish progress we need the following extension to the canonical forms lemma:

**Lemma 51 (Canonical Forms)**
*If* $\vdash v : \tau$ `cont`, *then* $v = K$ *for some control stack* $K$ *such that* $\vdash K : \tau$ `stack`.

Finally, progress is stated as follows:

**Theorem 52 (Progress)**
*If* $(K, e)$ ok *then either* $K = \bullet$ *and* $e$ value, *or there exists* $K'$ *and* $e'$ *such that* $(K, e) \mapsto (K', e')$.

**Proof:** By induction on typing. The verification is left as an exercise. ∎

## 10.4 Coroutines

Some problems are naturally implemented using *coroutines*, two (or more) routines that interleave their execution by an explicit hand-off of control from one to the other. In contrast to conventional sub-routines neither routine is "in charge", with one calling the other to execute to completion. Instead, the control relationship is symmetric, with each yielding control to the other during excecution.

A classic example of coroutining is provided by the producer-consumer model of interaction. The idea is that there is a common, hidden resource that is supplied by the producer and utilized by the consumer. Production of the resource is interleaved with its consumption by an explicit hand-off from producer to consumer. Here is an outline of a simple producer-consumer relationship:

```
val buf : int ref = ref 0
fun produce (n:int, cons:state) =
    (buf := n; produce (n+1, resume cons))
fun consume (prod:state) =
    (print (!buf); consume (resume prod))
```

There the producer and consumer share an integer buffer. The producer fills it with successive integers; the consumer retrieves these values and prints them. The producer yields control to the consumer after filling the buffer; the consumer yields control to the producer after printing its contents. Since the handoff is explicit, the producer and consumer run in strict synchrony, alternating between production and consumption.

The key to completing this sketch is to detail the handoff protocol. The overall idea is to represent the state of a coroutine by a continuation, the point at which it should continue executing when it is resumed by another coroutine. The function `resume` captures the current continuation and throws it to the argument continuation, transferring control to the other coroutine and, simultaneously, informing it how to resume the caller. This means that the state of a coroutine is a continuation accepting the state of (another) coroutine, which leads to a recursive type. This leads to the following partial solution:

```
datatype state = S of state cont

fun resume (S k : state) : state =
    callcc (fn k' : state cont => throw k (S k'))

val buf : int ref = ref 0

fun produce (n:int, cons:state) =
    (buf := n; produce (n+1, resume cons))

fun consume (prod:state) =
    (print (Int.toString(!buf)); consume (resume prod))
```

All that remains is to initialize the coroutines. It is natural to start by executing the producer, but arranging to pass it a coroutine state corresponding to the consumer. This can be achieved as follows:

```
fun run () =
    consume (callcc (fn k : state cont => produce (0, S k)))
```

■

Because of the call-by-value semantics of function application, we first seize the continuation corresponding to passing an argument to `consume`, then invoke `produce` with initial value `0` and this continuation. When `produce` yields control, it throws its state to the continuation that invokes `consume` with that state, at which point the coroutines have been initialized — further hand-off's work as described earlier.

This is, admittedly, a rather simple-minded example. However, it illustrates an important idea, namely the symmetric hand-off of control between routines. The difficulty with this style of programming is that the hand-off protocol is "hard wired" into the code. The producer yields control to the consumer, and *vice versa*, in strict alternating order. But what if there are multiple producers? Or multiple consumers? How would we handle priorities among them? What about asynchronous events such as arrival of a network packet or completion of a disk I/O request?

An elegant solution to these problems is to generalize the notion of a coroutine to the notion of a *user-level thread*. As with coroutines, threads enjoy a symmetric relationship among one another, but, unlike coroutines, they do not explicitly hand off control amongst themselves. Instead threads run as coroutines of a *scheduler* that mediates interaction among the threads, deciding which to run next based on considerations such as priority relationships or availability of data. Threads yield control to the scheduler, which determines which other thread should run next, rather than explicitly handing control to another thread.

Here is a simple interface for a user-level threads package:

```
signature THREADS = sig
  exception NoMoreThreads
  val fork : (unit -> unit) -> unit
  val yield : unit -> unit
  val exit : unit -> 'a
end
```

The function `fork` is called to create a new thread executing the body of the given function. The function `yield` is called to cede control to another thread, selected by the thread scheduler. The function `exit` is called to terminate a thread.

User-level threads are naturally implemented as continuations. A thread is a value of type `unit cont`. The scheduler maintains a queue of threads that are ready to execute. To dispatch the scheduler dequeues a thread from the ready queue and invokes it by throwing `()` to it. Forking is implemented by creating a new thread. Yielding is achieved by enqueueing the current thread and dispatching; exiting is a simple dispatch, abandoning the current thread entirely. This implementation is suggestive of a slogan suggested by Olin Shivers: "A thread is a trajectory through continuation space". During its lifetime a thread of control is represented by a succession of continuations that are enqueued onto and dequeued from the ready queue.

Here is a simple implementation of threads:

```
structure Threads :> THREADS = struct
    open SMLofNJ.Cont
    exception NoRunnableThreads
    type thread = unit cont
    val readyQueue : thread Queue.queue = Queue.mkQueue()
    fun dispatch () =
        let
            val t = Queue.dequeue readyQueue
                    handle Queue.Dequeue => raise NoRunnableThreads
        in
            throw t ()
        end
    fun exit () = dispatch()
    fun enqueue t = Queue.enqueue (readyQueue, t)
    fun fork f =
        callcc (fn parent => (enqueue parent; f (); exit()))
    fun yield () =
        callcc (fn parent => (enqueue parent; dispatch()))
end
```

Using the above thread interface we may implement the simple producer-consumer example as follows:

```
structure Client = struct
    open Threads
    val buffer : int ref = ref (~1)
    fun producer (n) =
        (buffer := n ; yield () ; producer (n+1))
    fun consumer () =
        (print (Int.toString (!buffer)); yield (); consumer())
    fun run () =
        (fork (consumer); producer 0)
end
```

This example is excessively naïve, however, in that it relies on the strict FIFO ordering of threads by the scheduler, allowing careful control over the order of execution. If, for example, the producer were to run several times in a row before the consumer could run, several numbers would be omitted from the output.

Here is a better solution that avoids this problem (but does so by "busy waiting"):

```
structure Client = struct
    open Threads
    val buffer : int option ref = ref NONE
    fun producer (n) =
        (case !buffer
           of NONE => (buffer := SOME n ; yield() ; producer (n+1))
            | SOME _ => (yield (); producer (n)))
    fun consumer () =
        (case !buffer
           of NONE => (yield (); consumer())
            | SOME n =>
              (print (Int.toString n); buffer := NONE; yield(); consumer()))
    fun run () =
        (fork (consumer); producer 0)
end
```

There is much more to be said about threads! We will return to this later in the course. For now, the main idea is to give a flavor of how first-class continuations can be used to implement a user-level threads package with very little difficulty. A more complete implementation is, of course, somewhat more complex, but not much more. We can easily provide all that is necessary for sophisticated thread programming in a few hundred lines of ML code.

# Chapter 11

# Type Reconstruction

## 11.1 Introduction

One criticism that has been made of statically typed languages is that the type information required on programs is excessively verbose. While this may be a valid criticism of some statically-typed languages, it overlooks the possibility of automatically inferring type information from context to lessen the notational burdens of static typing. This approach to type checking was pioneered by Robin Milner in the design of ML, and has since been exploited in a number of settings, including a type system for Cobol (!) programs to detect Year 2000 problems. The purpose of this note is to explain how type inference is accomplished.

To get a feeling for what's involved, let's consider the `map` function, written in a modest extension of MinML with multi-argument functions and lists. To make things as simple as possible, the example is written using the list primitives `nil` (empty list), `hd` (head of list), `tl` (tail of list), `cons` (construct a list), and `null` (test for nil), rather than pattern-matching.

Here's `map`, written in this extension of MinML:

```
fun map (f, l) is
  if null(l) then
     nil
  else
     cons (f (hd l), map (f, (tl l)))
  fi
end
```

We may think of type reconstruction as being performed in several stages. First, we introduce *type variables* (written $\alpha$, $\beta$, $\gamma$, etc) in place of the omitted type information on functions. These are intended as placeholders for type information that we will appear in the reconstructed typed term. We obtain the following *term scheme*, a pattern, or scheme, for a typed term:

```
fun map (f:α, l:β):γ is
  if null(l) then
     nil
  else
     cons (f (hd(l)), map (f, tl(l)))
  fi
end
```

The type variables $\alpha$ and $\beta$ are placeholders for the argument types of `map`, and the type variable $\gamma$ is a placeholder for the its result type.

In the next phase we examine the code of `map` to determine whether the type variables can be instantiated so as to obtain a well-typed reconstruction of the original untyped function. This is achieved by a form of type checking in which we generate *constraints* between *type schemes*. A *type scheme* is, as the name suggestions, a type expression that involves type variables acting as placeholders for types. For example, $\alpha \rightarrow \alpha\,\texttt{list}$ is a type scheme, with scheme variable $\alpha$. It stands for all types of the form $\tau \rightarrow \tau\,\texttt{list}$, where $\tau$ is a type. A *constraint* is an equation between type schemes. During inspection of the code we generate constraints corresponding, for example, to the requirement that in an application, the domain type of the function must be equal to the type of the argument.

To see what's going on, let's go through the code of `map` to determine the constraints that govern its reconstruction.

1. The expression `null(l)` induces the requirement that the type $\beta$ of `l` be a list type, which is expressed by the equation $\beta = \beta'\,\texttt{list}$, where $\beta'$ is a "fresh" type variable (one that we haven't used before). The type of `null(l)` is `bool`.

2. The expression `nil` occurring in the "then" clause of the conditional has type $\delta\,\texttt{list}$, where $\delta$ is a fresh type variable representing the as-yet-undetermined type of the empty list.

3. The expressions `hd(l)` and `tl(l)` induce the requirements that $\beta = \beta''\,\texttt{list}$ and $\beta = \beta'''\,\texttt{list}$, where $\beta''$ and $\beta'''$ are fresh type variables. The type of `hd(l)` is therefore $\beta''$ and the type of `tl(l)` is $\beta'''\,\texttt{list}$.

4. The expression `map (f, tl(l))` induces the requirements that $\alpha = \alpha$ and $\beta = \beta'''\,\texttt{list}$, which arise from equating corresponding argument and parameter types. The type of the expression is therefore $\gamma$, the result type of `map`.

5. The expression `f (hd(l))` induces the requirement that $\alpha = \beta'' \rightarrow \alpha'$, where $\alpha'$ is a fresh type variable, corresponding to the requirement that the type of $f$ be a function type whose domain is the type of the argument of the application. The type of `f (hd(l))` is therefore $\alpha'$.

6. The expression `cons(f (hd(l)), map(f, tl(l)))` induces the requirement that $\gamma$, the type of the recursive call to `map`, be $\gamma'\,\texttt{list}$ and that the type of `f(hd(l))` be $\gamma'$. That is, we add $\gamma = \gamma'\,\texttt{list}$ and $\alpha' = \gamma'$ to the set of constraints. The type of the `cons` expression is therefore $\gamma'\,\texttt{list}$.

7. The conditional expression induces the requirement that the types of the `then` and `else` clauses are equal, that is that $\delta\,\texttt{list} = \gamma'\,\texttt{list}$.

8. Finally, the result type of the function must be the type of the body: $\gamma = \delta\,\texttt{list}$.

This completes the generation of constraints.

Here is a summary of the constraints we have generated from this program:

$$
\begin{array}{rcl}
\beta & = & \beta'\,\texttt{list} \\
\beta & = & \beta''\,\texttt{list} \\
\beta & = & \beta'''\,\texttt{list} \\
\alpha & = & \alpha \\
\beta & = & \beta'''\,\texttt{list} \\
\alpha & = & \beta'' \rightarrow \alpha' \\
\gamma & = & \gamma'\,\texttt{list} \\
\alpha' & = & \gamma' \\
\delta\,\texttt{list} & = & \gamma'\,\texttt{list} \\
\gamma & = & \delta\,\texttt{list}
\end{array}
$$

The *solutions* to these constraints determine all of the possible reconstructions of the untyped term. If the constraints have no solution, the original expression is ill-formed and must be rejected. The determination of whether a given set of constraints has a solution and, if so, determining the set of possible solutions, is performed by a process similar to Gaussian elimination. Solving these constraints leads to the following equations

$$\begin{array}{rcl}\alpha & = & \beta'\to\gamma' \\ \beta & = & \beta'\,\texttt{list} \\ \gamma & = & \gamma'\,\texttt{list}\end{array}$$

Since the solution involves the unspecified type variables $\beta'$ and $\gamma'$, there are many reconstructions of `map` corresponding to how we choose to instantiate $\beta'$ and $\gamma'$. That is, all reconstructions of `map` are instances of the following term scheme (renaming type variables for readability):

```
fun map (f:α→β,x:α list):β list is
  if null(l) then
     nil
  else
     cons (f (hd(l)), map (f, tl(l)))
  fi
end
```

The remainder of this note consists of a systematic development of these ideas.

## 11.2 Type Reconstruction

The syntax of MinML is *explicitly typed* (or, more briefly, *typed*) in the sense that every expression has a unique type, if it has one at all. This is accomplished by requiring the programmer to provide explicitly the argument and result types of functions. In extensions to MinML (such as the ones you will explore in homework) additional type information is often required to ensure that the uniqueness of types property is maintained.

In practice it can be quite burdensome to be forced to supply the argument and result types of functions. It would be convenient to be able to omit this information whenever it can be deduced from context. For example, we may wish to write the factorial function in the form

```
fun f(x) is if x=0 then 1 else x * f (x-1) fi end
```

rather than in the fully explicit form

```
fun f(x:int):int is if x=0 then 1 else x * f (x-1) fi end
```

The problem is even worse in cases such as the following, where the return type is a function:

```
fun f(x) is fun g(y) is x+y end end
```

In explicit form we would have to write

```
fun f(x:int):int->int is fun g(y:int):int is x+y end end
```

As programs grow larger it becomes more and more burdensome to supply the required type information.

This suggests that we introduce an *implicitly typed* (briefly, *untyped*) syntax for MinML. The untyped MinML expressions, $u$, are inductively-defined by the following rules:

$$\frac{}{x} \qquad \frac{}{n} \qquad \frac{}{\texttt{true}} \qquad \frac{}{\texttt{false}}$$

$$\frac{u_1 \quad \ldots u_n}{o(u_1, \ldots, u_n)} \qquad \frac{u \quad u_1 \quad u_2}{\texttt{if}\, u \,\texttt{then}\, u_1 \,\texttt{else}\, u_2 \,\texttt{fi}}$$

$$\frac{u}{\texttt{fun}\, f\texttt{(}x\texttt{)}\, \texttt{is}\, u \,\texttt{end}} \qquad \frac{u_1 \quad u_2}{\texttt{apply(}u_1, u_2\texttt{)}}$$

The only difference is that no type information is required on functions; otherwise the syntax is the same as for MinML.

There is a simple *erasure* operation that maps a typed MinML expression $e$ to a corresponding untyped MinML expression $e^-$ by removing all type information. It is defined by induction on the structure of $e$ as follows:

$$\begin{array}{rcl}
x^- & = & x \\
n^- & = & n \\
\texttt{true}^- & = & \texttt{true} \\
\texttt{false}^- & = & \texttt{false} \\
o(e_1, \ldots, e_n)^- & = & o(e_1^-, \ldots, e_n^-) \\
\texttt{if}\, e \,\texttt{then}\, e_1 \,\texttt{else}\, e_2 \,\texttt{fi}^- & = & \texttt{if}\, e^- \,\texttt{then}\, e_1^- \,\texttt{else}\, e_2^- \,\texttt{fi} \\
\texttt{fun}\, f\, \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2 \,\texttt{is}\, e \,\texttt{end}^- & = & \texttt{fun}\, f\texttt{(}x\texttt{)}\, \texttt{is}\, e^- \,\texttt{end}
\end{array}$$

It is a simple matter to check that these equations define a unique function from typed MinML to untyped MinML.

We say that the typed expression $e$ is a *reconstruction* of the untyped expression $u$ iff the erasure of $e$ is $u$, *i.e.*, $e^- = u$. Notice that a given untyped expression may have many (in fact, infinitely many) distinct reconstructions! For example, the untyped expression `fun f(x) is x end` can be reconstructed to any typed expression of the form `fun f(x:`$\tau$`):`$\tau$` is x end`, where $\tau$ is a type.

How are the typed and untyped syntaxes of MinML related? We will define a four-place *elaboration* relation $\Gamma \vdash u \rightsquigarrow e : \tau$, with the intended meaning that $e$ is a reconstruction of $u$ with type $\tau$, relative to the context $\Gamma$. Since a given untyped expression $u$ may have many distinct reconstructions (or none at all, if the untyped expression is ill-formed) the elaboration relation will not be a *function* of $\Gamma$ and $u$. The elaboration relation is inductively-defined by the following rules.

Variables reconstruct to themselves, with the type given by the context.

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

Numbers and boolean constants reconstruct to themselves with the expected types.

$$\frac{}{\Gamma \vdash n \leadsto n : \mathtt{int}}$$

$$\frac{}{\Gamma \vdash \mathtt{true} \leadsto \mathtt{true} : \mathtt{bool}}$$

$$\frac{}{\Gamma \vdash \mathtt{false} \leadsto \mathtt{false} : \mathtt{bool}}$$

Primitive operations are reconstructed by reconstructing the constituent expressions at the required types.

$$\frac{\Gamma \vdash u_1 \leadsto e_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash u_n \leadsto e_n : \tau_n}{\Gamma \vdash o(u_1, \ldots, u_n) \leadsto o(e_1, \ldots, e_n) : \tau}$$

where $o$ takes arguments of type $\tau_1,\ldots,\tau_n$ and yields values of type $\tau$.

Conditionals reconstruct to conditionals, with the constituent expressions reconstructed at the appropriate type. Notice that we require that $u$ reconstruct to $e$ with type `bool`.

$$\frac{\Gamma \vdash u \leadsto e : \mathtt{bool} \qquad \Gamma \vdash u_1 \leadsto e_1 : \tau \qquad \Gamma \vdash u_2 \leadsto e_2 : \tau}{\Gamma \vdash \mathtt{if}\, u\, \mathtt{then}\, u_1\, \mathtt{else}\, u_2\, \mathtt{fi} \leadsto \mathtt{if}\, e\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2\, \mathtt{fi} : \tau}$$

Functions are reconstructed by "guessing" domain and range types, which are then assigned to the function name and parameter while reconstructing the body.

$$\frac{\Gamma[f{:}\tau_1{\to}\tau_2][x{:}\tau_1] \vdash u \leadsto e : \tau_2}{\Gamma \vdash \mathtt{fun}\, f\,\mathtt{(}x\mathtt{)}\, \mathtt{is}\, u\, \mathtt{end} \leadsto \mathtt{fun}\, f\, \mathtt{(}x : \tau_1\mathtt{)} : \tau_2\, \mathtt{is}\, e\, \mathtt{end} : \tau_1{\to}\tau_2}$$

Formally, this rule states that we may reconstruct an untyped function expression to any typed function expression with chosen domain and range types, provided that the body may be reconstructed with the specified range type, assuming the appropriate types for the function itself and its parameter.

Applications are reconstructed by "guessing" the mediating type $\tau_2$ for the argument.

$$\frac{\Gamma \vdash u_1 \leadsto e_1 : \tau_2 {\rightarrow} \tau \qquad \Gamma \vdash u_2 \leadsto e_2 : \tau_2}{\Gamma \vdash \texttt{apply}(u_1, u_2) \leadsto \texttt{apply}(e_1, e_2) : \tau}$$

In other words, we may reconstruct an application expression in potentially many different ways, corresponding to the choice of the type $\tau_2$ for the argument.

Since the elaboration rules involve arbitrary choices of type information, the elaboration relation is sometimes said to be *nondeterministic*, or *indeterminate*. This is just another way of saying that the relation is not a function of $\Gamma$ and $u$ — in general there are many possible reconstructions for given $\Gamma$ and $u$. Although it can be helpful to think in terms of "guessing" implicit type information, it is important to realize that the elaboration relation is fully and precisely specified by these rules. There is no imprecision in the definition.

**Exercise 53**
*Using the elaboration rules, show that the following untyped term can be reconstructed as a term of type* $\texttt{int}{\rightarrow}\texttt{int}$. *That is, find* $e$ *such that* $\vdash u \leadsto e : \texttt{int}{\rightarrow}\texttt{int}$, *where* $u$ *is the following untyped term:*

```
(fn f in (fn x in f (f x) end) end)(fn x in x end)
```

*(Here* $\texttt{fn}\,x\;\texttt{in}\,e\;\texttt{end}$ *stands for* $\texttt{fun}\,f\,\texttt{(}x\texttt{)}\;\texttt{is}\,e\;\texttt{end}$, *where* $f \notin \mathrm{FV}(e)$.*) Observe the points at which the "guesses" are necessary to complete the reconstruction. How did you make the appropriate choices?*

The crucial properties of the elaboration relation are summarized in the following theorem.

**Theorem 54**

1. *The elaboration relation is* sound*: if* $\Gamma \vdash u \leadsto e : \tau$, *then* $\Gamma \vdash e : \tau$ *and* $e^- = u$.

2. *The elaboration relation is* complete*: if* $\Gamma \vdash e : \tau$ *and* $e^- = u$, *then* $\Gamma \vdash u \leadsto e : \tau$.

Soundness ensures that given $\Gamma$ and $u$ we can derive *only* well-typed reconstructions of $u$ relative to $\Gamma$. Completeness ensures that we can derive *all* well-typed reconstructions of $u$, relative to $\Gamma$.

**Proof:**

1. By induction on the elaboration relation. We proceed by considering each rule in turn, applying the inductive hypothesis to the premises of the rule. We will illustrate one case here, leaving the rest as an exercise for the reader.

   Suppose that

   $$\Gamma \vdash \texttt{fun}\, f\,\texttt{(}x\texttt{)}\;\texttt{is}\; u\;\texttt{end} \leadsto \texttt{fun}\, f\;\texttt{(}x:\tau_1\texttt{)}:\tau_2\;\texttt{is}\; e\;\texttt{end} : \tau_1{\rightarrow}\tau_2,$$

   where

   $$\Gamma[f{:}\tau_1{\rightarrow}\tau_2][x{:}\tau_1] \vdash u \leadsto e : \tau_2.$$

   By induction we know that

   $$\Gamma[f{:}\tau_1{\rightarrow}\tau_2][x{:}\tau_1] \vdash e : \tau_2.$$

   Therefore

   $$\Gamma \vdash \texttt{fun}\, f\;\texttt{(}x:\tau_1\texttt{)}:\tau_2\;\texttt{is}\; e\;\texttt{end} : \tau_1{\rightarrow}\tau_2,$$

   as required.

2. By induction on the MinML typing relation, making use of the definition of the erasure function. We will illustrate one case here, leaving the rest as an exercise for the reader.

   Suppose that $e = \texttt{fun}\, f\;\texttt{(}x:\tau_1\texttt{)}:\tau_2\;\texttt{is}\; e_2\;\texttt{end}$ and $\tau = \tau_1{\rightarrow}\tau_2$, where

   $$\Gamma \vdash \texttt{fun}\, f\;\texttt{(}x:\tau_1\texttt{)}:\tau_2\;\texttt{is}\; e_2\;\texttt{end} : \tau_1{\rightarrow}\tau_2,$$

   by

   $$\Gamma[f{:}\tau_1{\rightarrow}\tau_2][x{:}\tau_1] \vdash e_2 : \tau_2,$$

   and that $\texttt{fun}\, f\;\texttt{(}x:\tau_1\texttt{)}:\tau_2\;\texttt{is}\; e_2\;\texttt{end}^- = u$. We are to show that $\Gamma \vdash u \leadsto e : \tau$. By the definition of the erasure function, $u = \texttt{fun}\, f\,\texttt{(}x\texttt{)}\;\texttt{is}\; u_2\;\texttt{end}$, where $u_2 = e_2^-$. Since $\tau = \tau_1{\rightarrow}\tau_2$, it suffices to show that

   $$\Gamma[f{:}\tau][x{:}\tau_1] \vdash u_2 \leadsto e_2 : \tau_2,$$

   for then it follows immediately that

   $$\Gamma \vdash u \leadsto e : \tau,$$

   as required. But this follows immediately by the inductive hypothesis, since $\Gamma[f{:}\tau][x{:}\tau_1] \vdash e_2 : \tau_2$ and $u_2 = e_2^-$.

■

**Exercise 55**
*Complete the proof of soundness and completeness of the elaboration relation.*

## 11.3 Constraint Generation

The elaboration relation is a simple and elegant definition of type reconstruction for MinML, but it is not at all obvious how to implement it! The difficulty lies in the use of non-deterministic rules to "guess" omitted type information to achieve a valid reconstruction. How can these choices be made by an algorithm? The key is to defer making these choices as late as possible to take account of the constraints imposed by the context in which the expression occurs. For example, in the expression `hd(l)`, the choice of a type for `l` is constrained by the requirement that it be a list type. If this expression occurs as `hd(l)+2`, then the choice of type for `l` is further constrained to `int list` by the fact that `hd(l)` is added to the integer constant 2. Our approach to type reconstruction, then, is to first collect all of the constraints on the choices of type information by making a pass over the program, then solving these constraints to determine the missing type information. The constraints may have no solution, because the program is ill-formed, exactly one solution, because there is only one consistent set of choices for missing type information, or infinitely many solutions, because there are many ways to make consistent choices.

How do the constraints arise? Careful inspection of the elaboration rules reveals that there are implicit requirements that two types be equal in order for an expression to be well-formed. For example, the reconstruction rule for application expressions requires that the domain type of the function be equal to the argument type. Similarly, the reconstruction rule for conditionals requires that the types of the "then" and "else" clauses be equal. These conditions limit the possible choices of type information that can be used to reconstruct an untyped expression. Each of the choices is represented by an unknown whose value is constrained by a set of equations that are gleaned from an analysis of the program. By solving the for the unknowns we resolve, as late as possible, the set of choices that could be made during elaboration.

A *type scheme* is a type that may contain one or more *type variables*. We let $\alpha$, $\beta$, and $\gamma$ stand for type variables, and write $\tau^*$ and $\sigma^*$ for type schemes (to clearly distinguish them from types, which do not have type variables). Formally, the set of type schemes is inductively defined by the following rules:

$$\frac{}{\alpha} \qquad \frac{}{\texttt{int}} \qquad \frac{}{\texttt{bool}} \qquad \frac{\tau_1^* \quad \tau_2^*}{\tau_1^* \rightarrow \tau_2^*}$$

The set $\mathrm{TV}(\tau^*)$ is the set of type variables occurring in the type scheme

$\tau^*$. A *term scheme* is a MinML term built up using type schemes (rather than just types). We write $e^*$ for term schemes, to emphasize that they may involve type variables. Similarly, we write $\Gamma^*$ for contexts that map variables to type schemes.

A *constraint*, $\phi$, is a formula expressing equations between type schemes. The set of constraints is inductively defined by the following rules:

$$\frac{}{\top} \qquad \frac{}{\tau_1^*{=}\tau_2^*} \qquad \frac{\phi_1 \quad \phi_2}{\phi_1 \wedge \phi_2}$$

The *constraint generation* relation is written $\Gamma^* \vdash u \rightsquigarrow e^* : \tau^* \; [\phi]$. The intended meaning is that in context $\Gamma^*$ the untyped term $u$ reconstructs to term scheme $e^*$ with type scheme $\tau^*$, subject to the constraints $\phi$. (The exact meaning will be given by Theorem 56 below.) This relation will be *total* function of $\Gamma^*$ and $u$, provided that $\mathrm{FV}(u) \subseteq \mathrm{dom}(\Gamma)$ — that is, for every $\Gamma^*$ and $u$ there will be a *unique* $e^*$ and $\tau^*$ such that $\Gamma^* \vdash u \rightsquigarrow e^* : \tau^* \; [\phi]$, even if $u$ is ill-formed.

The constraint generation relation is inductively-defined by the following rules.

Variables and constants are straightforward; no constraints are generated.

$$\frac{}{\Gamma^* \vdash x \rightsquigarrow x : \Gamma^*(x) \; [\top]}$$

$$\frac{}{\Gamma^* \vdash n \rightsquigarrow n : \texttt{int} \; [\top]}$$

$$\frac{}{\Gamma^* \vdash \texttt{true} \rightsquigarrow n : \texttt{bool} \; [\top]}$$

$$\frac{}{\Gamma^* \vdash \texttt{false} \rightsquigarrow n : \texttt{bool} \; [\top]}$$

Primitive operations generate constraints corresponding to the types of the arguments and results of the operation. We assume that $o$ has argument types $\tau_1, \ldots, \tau_n$ and result type $\tau$.

$$\frac{\Gamma^* \vdash u_1 \rightsquigarrow e_1^* : \tau_1^* \; [\phi_1] \quad \ldots \quad \Gamma^* \vdash u_n \rightsquigarrow e_n^* : \tau_n^* \; [\phi_n]}{\Gamma^* \vdash o(u_1, \ldots, u_n) \rightsquigarrow o(e_1^*, \ldots, e_n^*) : \tau \; [\phi_1 \wedge \cdots \wedge \phi_n \wedge \tau_1^* = \tau_1 \wedge \cdots \wedge \tau_n^* = \tau_n]}$$

Conditionals generate constraints for the sub-expressions, together with the requirement that the type of the test expression be `bool` and that the “then” and “else” clauses have the same type. The type variable $\alpha$ is uniquely associated with this occurrence of the conditional; it may not otherwise appear in a constraint.

$$\frac{\Gamma^* \vdash u \rightsquigarrow e^* : \tau^* \; [\phi] \qquad \Gamma^* \vdash u_1 \rightsquigarrow e_1^* : \tau_1^* \; [\phi_1] \qquad \Gamma^* \vdash u_2 \rightsquigarrow e_2^* : \tau_2^* \; [\phi_2]}{\begin{array}{l}\Gamma^* \vdash \texttt{if}\, u\, \texttt{then}\, u_1\, \texttt{else}\, u_2\, \texttt{fi} \rightsquigarrow \texttt{if}\, e^*\, \texttt{then}\, e_1^*\, \texttt{else}\, e_2^*\, \texttt{fi} : \alpha \\ \qquad\qquad\qquad\qquad [\phi \wedge \phi_1 \wedge \phi_2 \wedge \tau^*{=}\texttt{bool} \wedge \alpha{=}\tau_1^* \wedge \alpha{=}\tau_2^*]\end{array}}$$

Functions are handled by introducing fresh type variables to stand for the type of the parameter and result of the function; these variables are uniquely associated with this occurrence of the function. Constraints are then generated by examining the body of the function.

$$\frac{\Gamma^*[f{:}\alpha_1{\rightarrow}\alpha_2][x{:}\alpha_1] \vdash u \leadsto e^* : \tau^* \ [\phi]}{\Gamma^* \vdash \texttt{fun}\, f\texttt{(}x\texttt{)}\,\texttt{is}\, e\, \texttt{end} \leadsto \texttt{fun}\, f\ \texttt{(}x\,\texttt{:}\,\alpha_1\texttt{)}\,\texttt{:}\,\alpha_2\, \texttt{is}\, u\, \texttt{end} : \alpha_1{\rightarrow}\alpha_2\ [\phi \wedge \tau^*{=}\alpha_2]}$$

Applications are handled by introducing a fresh type variable to stand for the result type of the application.

$$\frac{\Gamma^* \vdash u_1 \leadsto e_1^* : \tau_1^* \ [\phi_1] \qquad \Gamma^* \vdash u_2 \leadsto e_2^* : \tau_2^* \ [\phi_2]}{\Gamma^* \vdash \texttt{apply(}u_1, u_2\texttt{)} \leadsto \texttt{apply(}e_1^*, e_2^*\texttt{)} : \alpha\ [\phi_1 \wedge \phi_2 \wedge \tau_1^*{=}\tau_2^*{\rightarrow}\alpha]}$$

To state precisely the properties of the constraint generation algorithm, it is necessary to introduce the notion of a *solution* to a constraint, a choice of bindings for the type variables in the constraint making all of the equations true. For example, replacing $\alpha$ and $\beta$ with the type `int` makes the equation $\alpha{\rightarrow}\beta{=}\texttt{int}{\rightarrow}\texttt{int}$ true; it is a solution of this constraint.

A *substitution* is a function mapping type variables to type schemes that is identity almost everywhere. This means that a function $S$ mapping type variables to type schemes is a substitution iff $S(\alpha) = \alpha$ for all but finitely many type variables $\alpha$. The *domain* $\mathrm{dom}(S)$ of the substitution $S$ is defined by $\mathrm{dom}(S) = \{\, \alpha \mid S(\alpha) \neq \alpha \,\}$. It is the set of type variables on which $S$ is not the identity. A substitution $S$ is *idempotent* iff $S \circ S = S$ — that is, $S(\alpha) = \hat{S}(S(\alpha))$ for all type variables $\alpha$. This means that the type variables occurring in the range of $S$ are not also in its domain. The substitution $S$ such that $S(\alpha) = \beta{\rightarrow}\texttt{int}$ and $S(\beta) = \texttt{bool}$ is not idempotent, since $\hat{S}(S(\alpha)) = \texttt{bool}{\rightarrow}\texttt{int} \neq \beta{\rightarrow}\texttt{int} = S(\alpha)$.

The application of a substitution, $S$, to a type scheme, $\tau^*$, written $\hat{S}(\tau^*)$ is defined by induction on the structure of $\tau^*$ as follows:

$$\begin{array}{rcl} \hat{S}(\alpha) & = & S(\alpha) \\ \hat{S}(\texttt{int}) & = & \texttt{int} \\ \hat{S}(\texttt{bool}) & = & \texttt{bool} \\ \hat{S}(\tau_1^*{\rightarrow}\tau_2^*) & = & \hat{S}(\tau_1^*){\rightarrow}\hat{S}(\tau_2^*) \end{array}$$

In other words, we replace all of the type variables in $\tau^*$ by their bindings in $S$. The type scheme $\hat{S}\tau^*$ is called an *instance* of the type scheme $\tau^*$. Abusing notation, we write $\hat{S}(\Gamma^*)$ for the result of applying $S$ to every type scheme in $\Gamma^*$. By a similar abuse, we also write $\hat{S}(e)$ for the application of a substitution $S$ to a term scheme $e$, replacing all type variables by their bindings in $S$. We leave the precise definition to the reader.

Substitutions may be composed by defining $(S_1 \circ S_2)(\alpha) = \hat{S}_1(S_2(\alpha))$. That is, we first retrieve the binding of $\alpha$ from $S_2$, then apply $S_1$ to the resulting type scheme. The *identity* substitution, $I$, is defined by $I(\alpha) = \alpha$ for every type variable $\alpha$. The *singleton* substitution $[\alpha{=}\tau^*]$ sends $\alpha$ to $\tau^*$ and is the identity elsewhere. We will often use singletons in combination with composition. The substitution $[\alpha{=}\tau^*] \circ S$ maps $\alpha$ to $[\tau^*/\alpha]S(\alpha)$. That is, it substitutes $\tau^*$ for $\alpha$ in every binding in $S$.

A substitution $S$ is a *solution* for a constraint $\phi$, written $S \models \phi$, every equation in $\phi$ is true when its type variables are instantiated by $S$. More precisely,

$$\begin{array}{rll} S \models \top & \textit{iff} & \textit{always} \\ S \models \tau_1^*{=}\tau_2^* & \textit{iff} & \hat{S}(\tau_1^*) = \hat{S}(\tau_2^*) \\ S \models \phi_1 \wedge \phi_2 & \textit{iff} & S \models \phi_1 \textit{ and } S \models \phi_2 \end{array}$$

We may now state the key properties of the constraint generation relation.

**Theorem 56**

1. *Constraint generation is a total function: For every $\Gamma^*$ and $u$ there exists a unique $e^*$, $\tau^*$, and $\phi$ such that $\Gamma^* \vdash u \rightsquigarrow e^* : \tau^*\ [\phi]$.*

2. *Constraint generation is sound: If $\Gamma^* \vdash u \rightsquigarrow e^* : \tau^*\ [\phi]$ and $S \models \phi$, then $\hat{S}\Gamma^* \vdash u \rightsquigarrow \hat{S}e^* : \hat{S}\tau^*$.*

3. *Constraint generation is complete: Suppose that $\Gamma^* \vdash u \rightsquigarrow e^* : \tau^*\ [\phi]$. Let $S$ be a substitution such that $\hat{S}\Gamma^*$ contains no type variables. If $\hat{S}\Gamma^* \vdash u \rightsquigarrow e : \tau$, then there exists substitutions $S_1$ and $S_2$ such that $S_2 = S_1 \circ S$, $\mathrm{TV}(\phi) \subseteq \mathrm{dom}(S_2)$, $S_2 \models \phi$, and $e = \hat{S}_2 e^*$ and $\tau = \hat{S}_2\tau^*$.*

It is worthwhile to contemplate the meaning of this theorem before considering its proof. The first statement asserts that the process of constraint generation always terminates with a term scheme, type scheme, and constraint. This is easily proved by induction on the structure of $u$. The second statement asserts that *any* solution of the constraints determines a valid reconstruction of $u$, relative to the instance of the context. This is proved by rule induction on the constraint generation relation. The third statement is perhaps the hardest to understand. As a first cut, let us assume that $\Gamma^*$ is empty and $S$ is the identity. The completeness property says that *every* reconstruction of $u$ is an instance of its schematic reconstruction by a substitution solving the constraints associated with $u$. Completeness is proved by induction on the elaboration relation. To complete the proof it is necessary to consider the generalization to non-empty contexts $\Gamma^*$, in the form stated.

## 11.4 Solving Constraints

solve

Constraint solving (also known as *unification*) is performed by a process similar to Guassian elimination. We accumulate a solution, $S$, by incrementally simplifying the constraint, gradually accumulating bindings for the variables. This process is naturally expressed by a transition system with states of the form $(S, \phi)$, where $S$ is an idempotent substitution and $\phi$ is a constraint such that $\mathrm{dom}(S) \cap \mathrm{TV}(\phi) = \emptyset$. The restriction to idempotent substitutions means that we can regard $S$ as a set of ordered pairs $(\alpha, \tau)$ where $\mathrm{TV}(\tau) \cap \mathrm{dom}(S) = \emptyset$. The requirement that the type variables of $\phi$ be disjoint the domain of $S$ amounts to the requirement that solved variables not occur in the remaining set of constraints.

A *state* $(S, \phi)$ consists of a partial solution, $S$, of a constraint, together with a remaining constraint that is yet to be solved to form a complete solution to the original constraint. The *initial state* is $(I, \phi)$, with the entire constraint remaining to be solved, and no partial solution having yet been attained for any of the type variables in $\phi$. A *final state* has of the form $(S, \top)$, consisting of a complete solution, with no further constraints to be solved.

To simplify the presentation of the transition system, we impose the following structural equivalences on constraints:

$$
\begin{array}{rcl}
\top \wedge \phi & \equiv & \phi \\
\phi \wedge \top & \equiv & \phi \\
\phi_1 \wedge \phi_2 & \equiv & \phi_2 \wedge \phi_1 \\
\phi_1 \wedge (\phi_2 \wedge \phi_3) & \equiv & (\phi_1 \wedge \phi_2) \wedge \phi_3 \\
\tau_1^* = \tau_2^* & \equiv & \tau_2^* = \tau_1^*
\end{array}
$$

Informally, this means that we may regard a constraint as a multiset (set with repetitions) of unordered pairs of type schemes. This allows us to avoid a bunch of bookkeeping that would distract from the presentation.

The rules of the transition system are as follows.

First, equations between identical base types and identical type variables may be dropped.

$$\begin{array}{rcl}(S, \texttt{int}=\texttt{int} \wedge \phi) & \mapsto & (S, \phi) \\ (S, \texttt{bool}=\texttt{bool} \wedge \phi) & \mapsto & (S, \phi) \\ (S, \alpha=\alpha \wedge \phi) & \mapsto & (S, \phi)\end{array}$$

Second, an equation between function types reduces to two equations between their respective domains and ranges.

$$(S, \tau_1^* \rightarrow \tau_2^* = \sigma_1^* \rightarrow \sigma_2^* \wedge \phi) \mapsto (S, \tau_1^* = \sigma_1^* \wedge \tau_2^* = \sigma_2^* \wedge \phi)$$

Third, an equation of the form $\alpha=\tau^*$ partially determines the solution for $\alpha$, provided that $\alpha$ does not occur in $\tau^*$. (This is the so-called *occurs check*.)

$$(S, \alpha=\tau^* \wedge \phi) \mapsto ([\alpha=\tau^*] \circ S, [\tau^*/\alpha]\phi)$$

*provided that* $\alpha \notin \mathrm{TV}(\tau^*)$. The reason for this restriction is that an equation such as $\alpha=\alpha\rightarrow\alpha$ is unsolvable, since any solution for $\alpha$ would have to be an infinite type expression, which is impossible.

Notice that we replace all occurrences of $\alpha$ in the remaining constraints $\phi$ by $\tau^*$ so that $\alpha$ no longer occurs in the constraints after this step is taken. Furthermore, we compose the singleton substitution with $S$ to ensure that occurrences of $\alpha$ in the range of $S$ are instantiated by $\tau^*$. Both of these steps ensure that the invariants governing the state are maintained.

An *irreducible state* $(S, \phi)$ is one for which there is no state $(S', \phi')$ such that $(S, \phi) \mapsto (S', \phi')$. Clearly every final state is irreducible, but not all irreducible states are final. For example, the state $(S, \texttt{int}=\texttt{bool})$ is irreducible. Irreducible, non-final states are called *stuck* states. The solution process has stopped, but no solution has been found. An examination of the transition rules reveals that stuck states must consist entirely of equations of one of the following forms:

1. $\texttt{int}=\texttt{bool}$
2. $\tau_1^* \rightarrow \tau_2^* = \tau^*$, where $\tau^*$ is neither a function type nor a type variable.
3. $\alpha=\tau^*$, where $\alpha \in \mathrm{TV}(\tau^*)$.

It is obviously decidable whether or not a state is stuck. It is worthwhile to check that if a constraint $\phi$ contains an equation not of these three forms, then progress can be made.

An important property of the transition system is that there can be no infinite transition sequences $(S_1, \phi_1) \mapsto (S_2, \phi_2) \mapsto \cdots$. To prove this, we define a "size measure" for states that is decreased by each transition. Examining the transitions in turn, we can make the following observations:

1. The first set of transitions reduces the size of the constraint (measured as the sum of the sizes of the type schemes occurring in it).

2. The second transition also reduces the size of the constraint, since it eliminates two occurrences of the $\rightarrow$ symbol.

3. The third transition, however, can *increase* the size of the constraint because the substitution may replicate $\tau^*$, thereby blowing up its size. However, the number of variables occurring in the constraint definitely decreases, because all occurrences of $\alpha$ (of which there is at least one, prior to the transition) are eliminated.

This suggests that we assign the pair of numbers $(n, s)$ to the state $(S, \phi)$, where $s$ is the sum of the sizes of the type schemes occurring in $\phi$ (not $S$!) and $n$ is the number of type variables occurring in $\phi$. If we order these pairs lexicographically by defining $(n, s) < (n', s')$ to hold iff $n < n'$ or $n = n'$ and $s < s'$,[1] then it is easy to see that each transition reduces the size measure in the sense of this ordering. In the first case $n$ may decrease, or $n$ may remain the same, but $s$ decreases. In the second, $n$ remains the same, but $s$ decreases. In the third, $s$ may increase dramatically, but nevertheless $n$ decreases. Thus we have the following theorem.

**Theorem 57 (Termination)**
*There is no infinite sequence of states* $(S_1, \phi_1) \mapsto (S_2, \phi_2) \mapsto \cdots$.

The next observation concerns the relationship between successive states. A *complete solution* for a state $(S, \phi)$ consists of a substitution $T$ such that $T = U \circ S$ for some $U$ and $T \models \phi$. In other words, $T$ is an extension of the partial solution $S$ to a solution for $\phi$. It is easy to check that stuck states have no complete solution, and that for a final state the partial solution contained within it is a complete solution.

The following technical lemma will prove useful.

**Lemma 58**
$S \models [\tau^*/\alpha]\phi$ *iff* $S \circ [\alpha{=}\tau^*] \models \phi$.

[1] This is the "dictionary ordering" for two-letter words, where the "letters" are natural numbers.

**Proof:** By induction on the structure of $\phi$. ∎

The key property of the transition system is that it *preserves* and *reflects* complete solutions of states.

**Theorem 59**
*Suppose that $(S, \phi) \mapsto (S', \phi')$. Then $T$ is a complete solution for $(S, \phi)$ iff $T$ is a complete solution for $(S', \phi')$.*

**Proof:** We consider each of the transitions in turn. The case of transitions that drop identical base types or identical type variables are trivial — the presence or absence of such constraints does not change the complete solution. The transition for an equation between function types is similarly straightforward; a complete solution to the left-hand state is clearly a complete solution to the right-hand state, and vice versa.

Finally, let us consider the third transition, for which $\phi = \alpha{=}\tau^* \wedge \psi$, $\phi' = [\tau^*/\alpha]\psi$, and $S' = [\alpha{=}\tau^*] \circ S$.

Suppose that $T = U' \circ S'$ and that $T \models \phi'$; we are to show that there exists $U$ such that $T = U \circ S$ and $T \models \phi$. Taking $U = U' \circ [\alpha{=}\tau^*]$, we have that

$$\begin{array}{rcll} T & = & U' \circ S' & \text{by asssumption on } T \\ & = & U' \circ [\alpha{=}\tau^*] \circ S & \text{by assumption on } S' \\ & = & U \circ S & \text{by definition of } U \end{array}$$

Now we must show that $T \models \alpha{=}\tau^* \wedge \psi$. First, observe that by the invariants on the state, $\hat{S}(\tau^*) = \tau^*$ and $S(\alpha) = \alpha$. Next, note that

$$\begin{array}{rcll} T(\alpha) & = & \hat{U}(S(\alpha)) & \text{since } T = U \circ S \\ & = & \hat{U}(\alpha) & \text{by state invariant} \\ & = & \hat{U'}(\tau^*) & \text{by definition of } U \end{array}$$

and that

$$\begin{array}{rcll} \hat{T}(\tau^*) & = & \hat{U}(\hat{S}(\tau^*)) & \text{since } T = U \circ S \\ & = & \hat{U}(\tau^*) & \text{by state invariants} \\ & = & \hat{U'}([\tau^*/\alpha]\tau^*) & \text{by definition of } U \\ & = & \hat{U'}(\tau^*) & \text{by the occurs check, } \alpha \notin \mathrm{TV}(\tau^*) \end{array}$$

Thus $T \models \alpha{=}\tau^*$, and hence $T = T \circ [\alpha{=}\tau^*]$. But since $T \models [\tau^*/\alpha]\psi$, it follows that $T \models \psi$.

Conversely, suppose that $T = U \circ S$ and $T \models \phi$. We are to show that there exists a sustitution $U'$ such that $T = U' \circ S'$ and $T \models \phi'$. Since

$T \models \phi$, $T(\alpha) = \hat{T}(\tau^*)$ and $T \models \psi$. Consequently, $T = T \circ [\alpha{=}\tau^*]$, and so $T \models [\tau^*/\alpha]\psi$. It remains to find $U'$ such that $T = U' \circ S'$. First, observe that

$$\begin{array}{rcll} U(\alpha) & = & \hat{U}(S(\alpha)) & \text{by the state invariants} \\ & = & T(\alpha) & \text{since } T = U \circ S \\ & = & \hat{T}(\tau^*) & \text{since } T \models \alpha{=}\tau^* \\ & = & \hat{U}(\hat{S}(\tau^*)) & \text{since } T = U \circ S \\ & = & \hat{U}(\tau^*). & \text{by the state invariants} \end{array}$$

Now define $U'(\alpha) = \alpha$ and $U'(\beta) = U(\beta)$ (for $\beta \neq \alpha$). Note that $U' \circ [\alpha{=}\tau^*] = U$. For if $\beta \neq \alpha$, then

$$\begin{array}{rcll} (U' \circ [\alpha{=}\tau^*])(\beta) & = & U'(\beta) & \text{since } \beta \neq \alpha \\ & = & U(\beta) & \text{by definition of } U' \end{array}$$

and

$$\begin{array}{rcll} (U' \circ [\alpha{=}\tau^*])(\alpha) & = & \hat{U'}(\tau^*) & \text{by composition of substitutions} \\ & = & \hat{U}(\tau^*) & \text{since } \alpha \notin \mathrm{TV}(\tau^*) \\ & = & U(\alpha) & \text{since } U(\alpha) = \hat{U}(\tau^*) \end{array}$$

Therefore, $U' \circ S' = U \circ [\alpha{=}\tau^*] \circ S = U \circ S = T$. ■

An important corollary of Theorems 59 and 57 states that the transition system computes *principal solutions* of constraints.

**Corollary 60**
*Suppose that $(I, \phi) \mapsto^* (S, \top)$. The substitution $T$ solves $\phi$ iff $T = U \circ S$ for some substitution $U$.*

The substitution $S$ is called the *principal solution* (or *most general unifier*) for $\phi$.

**Proof:** It follows from Theorem 59 by induction on the length of the transition sequence from the inital to the final state that $T$ is a complete solution to $(I, \phi)$ iff $T$ is a complete solution to $(S, \top)$. Hence $T \models \phi$ iff $T = U \circ S$ for some $U$. In other words, all and only the solutions to $\phi$ arise as specializations of $S$. ■

Combining the previous results we obtain the following theorem.

**Theorem 61 (Unification Algorithm)**
*It is decidable whether or not a constraint $\phi$ has a solution. If $\phi$ is solvable, then we may find the principal solution for it.*

**Proof:** Find an irreducible state $(S, \phi')$ such that $(I, \phi) \mapsto^* (S, \phi')$. If $(S, \phi)$ is final, then $S$ is the principal solution for $\phi$; if it is stuck, then $\phi$ has no solution. ■

Combining the results of this section with those of the previous section, we obtain the following result. Given a closed context $\Gamma$ and an untyped term $u$, we can decide whether or not $u$ is reconstructible to a well-typed term, and, if so, we can recover all possible reconstructions of $u$ as instances of a term and type scheme that we may obtain by a one-pass analysis of $u$. To decide whether $u$ is reconstructible (relative to $\Gamma$),

1. Find $e^*$, $\tau^*$, and $\phi$ such that $\Gamma \vdash u \rightsquigarrow e^* : \tau^*\ [\phi]$.

2. Find the principal solution $S_0$ (if any) of $\phi$.

If $\phi$ has no solution, then $u$ is not reconstructible (relative to $\Gamma$). If $\phi$ has a principal solution $S_0$, then all and only the reconstructions of $u$ are instances of $\hat{S_0}(e^*)$ and $\hat{S_0}(\tau^*)$.

# Chapter 12

# Expression Equivalence

## 12.1 Introduction

One of the beauties of functional programming is the ease with which we may reason about equivalence of expressions. Informally, we say that two expressions $e_1$ and $e_2$ of the same type are *equivalent* iff replacing $e_1$ by $e_2$ in a complete program doesn't change its final result. By a "complete program" we mean a closed expression of type `int` or `bool`; the final result of a complete program is therefore a number or a Boolean constant. What is important here is that the final outcome be *finitely observable* — we can see immediately that the answer is `false` or 17. Since functions are essentially "infinite" objects (in the sense that the graph of a function on the integers is infinite), we would not regard functions as observable outcomes of a complete program.

We can think of a usage of an expression in a complete program as an "experiment" or "observation" performed on that expression. The idea is that the program "uses" the expression to compute an observable quantity that we can regard as a kind of test performed on that expression. For this reason the notion of equivalence just described is sometimes called *observational equivalence* — two expressions are observationally equivalent iff any experiment performed on one yields the same observable outcome as the same experiment performed on the other. This relation is also called *contextual equivalence* to emphasize that equivalence is determined by considering all contexts in which the two expressions might be used to form a complete program. In philosophical logic this relation is known as *Leibniz's Principle of Identity of Indiscernibles* — two things are equal iff we cannot tell them apart.

Observational equivalence is very difficult to handle. To determine whether or not $e_1$ and $e_2$ are observationally equivalent requires us to consider *all possible programs* that use them to compute an integer! This quickly gets out of hand. What we need are alternative criteria for establishing observational equivalences that avoid the need to explicitly consider all possible usages of the expressions in question. Unfortunately, a rigorous development of such alternatives would take us far beyond the scope of the course. We will content ourselves with stating, without proof, a collection of laws of equivalence that are useful for deriving equations between expressions such as these:

1. $x + (y + z)$ is equivalent to $(x + y) + z$.
2. $\texttt{rev(rev(}x\texttt{))}$ is equivalent to $x$ (where $\texttt{rev}$ is the list reversal function).
3. If $v = \texttt{fun}\, f\, \texttt{(}x\,\texttt{:}\,\tau_1\texttt{)}\,\texttt{:}\,\tau_2\ \texttt{is}\ e\ \texttt{end}$ and $v{:}\tau_1$ is a value, then $v\texttt{(}v_1\texttt{)}$ is equivalent to $[v, v_1/f, x]e$.

In the next section we make precise the definition of observational equivalence, and state an alternative characterization of it that is often easier to handle. In the subsequent section we enumerate a collection of valid principles of equivalence for variants of MinML.

## 12.2 Expression Equivalence

We begin with the notion of *Kleene equality* between complete programs. Kleene equivalence captures what we mean by "same outcome" for complete programs. Two complete programs $p_1$ and $p_2$ are *Kleene equivalent*, written $p_1 \simeq p_2$, exactly when either $p_1 \Downarrow v$ iff $p_2 \Downarrow v$. That is, either both $p_1$ and $p_2$ diverge (fail to halt), or both converge to the same number or Boolean constant (perhaps in very different ways, using very different amounts of time and space).

A *context* $C$ is a complete program with a single "hole" into which we may insert an expression. That is, $C$ has the form $\ldots \bullet \ldots$, where the $\bullet$ indicates the "hole" in the program. We write $C\{e\}$ for the result of filling the hole in $C$ with the expression $e$ to obtain $\ldots e \ldots$. The expression $e$ might have free variables that are captured when inserted into the hole. For example, if $C$ is the program context $\texttt{(fun}\, f\, \texttt{(}x\,\texttt{:int):int is}\, \bullet\, \texttt{end)(3)}$ and $e$ is the expression $x + 5$, then $C\{x + 5\}$ is the program

$$\texttt{(fun}\, f\, \texttt{(}x\,\texttt{:int):int is}\ x + 5\ \texttt{end)(3)}.$$

Suppose that $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$. We define the *observational equivalence* relation $\Gamma \vdash e_1 \cong_{obs} e_2 : \tau$ to hold iff for every program context $C$ such that $C\{e_1\}$ and $C\{e_2\}$ are programs, $C\{e_1\} \simeq C\{e_2\}$. That is, every use of $e_1$ has the same observable outcome as the corresponding use of $e_2$, and vice-versa.

As we remarked earlier, it is rather difficult to establish that two expressions are observationally equivalent. A direct application of the definition leaves us no recourse but to consider all possible program contexts $C$, which quickly gets out of hand.

A more usable characterization of observational equivalence, called *applicative equivalence*, is defined as follows. For closed expressions $e_1$ and $e_2$ of type $\tau$, we define $e_1 \cong_{app}^{closed} e_2 : \tau$ by induction on the structure of $\tau$ as follows:

- If $\tau = \texttt{int}$, or $\tau = \texttt{bool}$, then $e_1 \cong_{app}^{closed} e_2 : \tau$ iff $e_1 \simeq e_2$.
- If $\tau = \tau_1 \rightarrow \tau_2$, then $e_1 \cong_{app}^{closed} e_2 : \tau$ iff for every $v : \tau_1$, $e_1\texttt{(}v\texttt{)} \cong_{app}^{closed} e_2\texttt{(}v\texttt{)} : \tau_2$.

This relation is extended to open expressions by substitution of closed values of appropriate type for the free variables. Let $\Gamma$ be the context $x_1{:}\tau_1, \ldots, x_n{:}\tau_n$. We define $\Gamma \vdash e_1 \cong_{app} e_2 : \tau$ to hold iff

$$[v_1, \ldots, v_n/x_1, \ldots, x_n]e_1 \cong_{app}^{closed} [v_1, \ldots, v_n/x_1, \ldots, x_n]e_2$$

for every substitution of closed values $v_1,\ldots,v_n$ of type $\tau_1,\ldots, \tau_n$ for $x_1,\ldots,x_n$.

An important result of Milner's states that applicative and observational equivalence coincide. The proof is non-trivial, and is omitted from this brief exposition.

**Theorem 62 (Milner's Context Lemma)**
*$\Gamma \vdash e_1 \cong_{obs} e_2 : \tau$ iff $\Gamma \vdash e_1 \cong_{app} e_2 : \tau$.*

The point of the context lemma is that we may interpret the relation $\Gamma \vdash e_1 \cong_{obs} e_2 : \tau$ as expressing the universally-quantified formula

$$\forall v_1 : \tau_1 \ \ldots \ \forall v_n : \tau_n \, [v_1, \ldots, v_n/x_1, \ldots, x_n]e_1 \cong_{app} [v_1, \ldots, v_n/x_1, \ldots, x_n]e_2 : \tau$$

where the quantifiers range over *closed values* of the appropriate type. For example,

$$x{:}\texttt{int}, y{:}\texttt{int} \vdash x + y \cong_{obs} y + x : \texttt{int}$$

means that for every $m$ and $n$, $m + n \simeq n + m$.

## 12.3 Some Laws of Equivalence

In this section we summarize some useful principles of equivalence. These are all valid observational equivalences, but we will not prove this. What is important is to get a feeling for what are some valid principles of equivalence, and how to use them in practice. Since observational and applicative equivalence coincide, we will write $\Gamma \vdash e_1 \cong e_2 : \tau$ for equality of expressions of type $\tau$ relative to a context $\Gamma$, where we tacitly assume that $\Gamma \vdash e_i : \tau$ for $i = 1, 2$.

In the presentation of the rules, we use $v$ to stand for an *open value*, either a variable, a constant, or an function expression (perhaps with free variables occurring within it). We admit variables as values because, in a call-by-value language, variables are only ever bound to values, and hence may be taken as standing for a fixed, but unknown, value.

It will be convenient to make use of a designated non-terminating expression of each type $\tau$, written $\Omega_\tau$, which is defined to be the expression

```
(fun f(x:int):τ is f(x) end)(0).
```

It is easy to check that $\Omega_\tau$ diverges (loops forever) when evaluated.

### 12.3.1 General Laws

First, equivalence is indeed an equivalence relation — it is reflexive, symmetric, and transitive.

$$\frac{}{\Gamma \vdash e \cong e : \tau}$$

$$\frac{\Gamma \vdash e_2 \cong e_1 : \tau}{\Gamma \vdash e_1 \cong e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 \cong e_2 : \tau \qquad \Gamma \vdash e_2 \cong e_3 : \tau}{\Gamma \vdash e_1 \cong e_3 : \tau}$$

Second, equivalence is a *congruence* — we may replace a sub-expression of any expression by an equivalent one to obtain an equivalent expression. This is most easily stated by a collection of rules that ensure that we may replace equivalent sub-expressions to obtain equivalent expressions. We will give just a few of these here; the rest follow a similar pattern.

$$\frac{\Gamma \vdash e_1 \cong e_1' : \tau_1 \quad \cdots \quad \Gamma \vdash e_n \cong e_n' : \tau_n}{\Gamma \vdash o(e_1, \ldots, e_n) \cong o(e_1', \ldots, e_n') : \tau}$$

$$\frac{\Gamma \vdash e_1 \cong e_1' : \tau_2 \to \tau \qquad \Gamma \vdash e_2 \cong e_2' : \tau_2}{\Gamma \vdash e_1(e_2) \cong e_1'(e_2') : \tau}$$

$$\frac{\Gamma[f{:}\tau_1{\rightarrow}\tau_2][x{:}\tau_1] \vdash e \cong e' : \tau_2}{\Gamma \vdash \texttt{fun}\, f\ \texttt{(}x : \tau_1\texttt{)} : \tau_2\ \texttt{is}\ e\ \texttt{end} \cong \texttt{fun}\, f\ \texttt{(}x : \tau_1\texttt{)} : \tau_2\ \texttt{is}\ e'\ \texttt{end} : \tau_1{\rightarrow}\tau_2}$$

Finally, equivalence is stable under substitution of *values* for free variables.

$$\frac{\Gamma[x{:}\tau] \vdash e \cong e' : \tau'}{\Gamma \vdash [v/x]e \cong [v/x]e' : \tau'}$$

The restriction to values is essential; this rule is *not* true for general expression substitution! A counterexample is given in the next subsection.

### 12.3.2 Symbolic Evaluation

Evaluation of an expression in accordance with the rules of the operational semantics results in an equivalent expression. This is called "symbolic evaluation" because the transformations may involve expressions with free variables, which are regarded as values for the purposes of these rules.

An application of a primitive operation may be simplified if we know the values of its arguments:

$$\frac{}{\Gamma \vdash o(v_1, \ldots, v_n) \cong v : \tau,}$$

where $v$ is the result of applying $o$ to $v_1, \ldots, v_n$.

Similarly, if we know the result of the boolean test, then a conditional may be simplified:

$$\frac{}{\Gamma \vdash \texttt{if}\,\texttt{true}\,\texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \cong e_1 : \tau}$$

$$\frac{}{\Gamma \vdash \texttt{if}\,\texttt{false}\,\texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \cong e_2 : \tau}$$

An application may be simplified if we know the function and the argument is a value. Note that either the function or argument may be *open* expressions (containing free variables)!

$$\frac{}{\Gamma \vdash v\texttt{(}v_1\texttt{)} \cong [v, v_1/f, x]e : \tau_2}$$

where $v = \texttt{fun}\, f\ \texttt{(}x : \tau_1\texttt{)} : \tau_2\ \texttt{is}\ e\ \texttt{end}$.

**Exercise 63**

*Using these rules, check that* `(fn x in 3 end)(z)` *is equivalent to* 3. *Show that* `(fn x in 3 end)(`$\Omega_{\texttt{int}}$`)` *is* not *equivalent to* 3. *Conclude that substitution of non-values for free variables does not preserve equivalence.*

### 12.3.3 Extensionality

Two functions are equivalent if they are equivalent on all arguments.

$$\frac{\Gamma[x{:}\tau_1] \vdash e(x) \cong e'(x) : \tau_2}{\Gamma \vdash e \cong e' : \tau_1 \to \tau_2}$$

In other words, if two functions are equal for all closed argument values, then they are equal.

### 12.3.4 Strictness Properties

The evaluation rules of MinML impose a *call-by-value* evaluation order on function applications and primitive operations. This can be captured equationally by a set of *strictness* equations that are defined in terms of the divergent expressions $\Omega_\tau$. We may state that an expression $e$ of type $\tau$ diverges as an equation by stating that $e \cong \Omega_\tau$. The following rules give some conditions under which expressions are divergent.

If any argument of a primitive operation is divergent, so is the whole expression:

$$\frac{}{\Gamma \vdash o(e_1, \ldots, e_{i-1}, \Omega_{\tau_i}, e_{i+1}, \ldots, e_n) \cong \Omega_\tau : \tau}$$

If the test expression of a conditiona is divergent, so is the conditional.

$$\frac{}{\Gamma \vdash \texttt{if}\,\Omega_{\texttt{bool}}\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi} \cong \Omega_\tau : \tau}$$

If the function or argument of an application is divergent, so is the entire expression:

$$\frac{}{\Gamma \vdash \Omega_{\tau_2 \to \tau}(e_2) \cong \Omega_\tau : \tau}$$

$$\frac{}{\Gamma \vdash e_1(\Omega_{\tau_2}) \cong \Omega_\tau : \tau}$$

### 12.3.5 Arithmetic Laws

Arithmetic and comparison operations behave as expected. For example, addition is associative and commutative, and equality test on integers is an equivalence relation. In general appropriate laws governing the primitive operations on integers hold, provided that they hold mathematically. The same could not be said for floating point (for which addition is not even associative!). Observe that these laws *fail*, in general, in the presence of effects such as writing to the screen or destructively updating a reference cell! In that case we must restrict attention to values, not general expressions.

$$\frac{}{\Gamma \vdash e_1\texttt{+}e_2 \cong e_2\texttt{+}e_1 : \texttt{int}}$$

$$\frac{}{\Gamma \vdash e_1\texttt{+(}e_2\texttt{+}e_3\texttt{)} \cong \texttt{(}e_1\texttt{+}e_2\texttt{)+}e_3 : \texttt{int}}$$

$$\frac{}{\Gamma \vdash e_1\texttt{=}e_2 \cong e_2\texttt{=}e_1 : \texttt{bool}}$$

### 12.3.6 Products

For the extension of MinML to product types,[1] we have the following symbolic evaluation rule:

$$\frac{}{\Gamma \vdash \texttt{bind}\,\texttt{(}x_1\texttt{:}\tau_1\texttt{,}x_2\texttt{:}\tau_2\texttt{)}\,\texttt{to}\,\texttt{(}v_1\texttt{,}v_2\texttt{)}\,\texttt{in}\,e\,\texttt{end} \cong [v_1, v_2/x_1, x_2]e}$$

We may, in general, replace equals by equals:

$$\frac{\Gamma \vdash e_1 \cong e_1' : \tau_1 \qquad \Gamma \vdash e_2 \cong e_2' : \tau_2}{\Gamma \vdash \texttt{(}e_1\texttt{,}e_2\texttt{)} \cong \texttt{(}e_1'\texttt{,}e_2'\texttt{)} : \tau_1\texttt{*}\tau_2}$$

$$\frac{\Gamma \vdash e_1 \cong e_1' : \tau_1\texttt{*}\tau_2 \quad \Gamma[x_1{:}\tau_1][x_2{:}\tau_2] \vdash e_2 \cong e_2' : \tau'}{\Gamma \vdash \texttt{bind}\,\texttt{(}x_1\texttt{:}\tau_1\texttt{,}x_2\texttt{:}\tau_2\texttt{)}\,\texttt{to}\,e_1\,\texttt{in}\,e_2\,\texttt{end} \cong \texttt{bind}\,\texttt{(}x_1\texttt{:}\tau_1\texttt{,}x_2\texttt{:}\tau_2\texttt{)}\,\texttt{to}\,e_1'\,\texttt{in}\,e_2'\,\texttt{end}}$$

Finally, the expected strictness properties hold:

$$\frac{}{\Gamma \vdash \texttt{(}\Omega_{\tau_1}\texttt{,}e_2\texttt{)} \cong \Omega_{\texttt{(}\tau_1\texttt{,}\tau_2\texttt{)}}}$$

$$\frac{}{\Gamma \vdash \texttt{(}e_1\texttt{,}\Omega_{\tau_2}\texttt{)} \cong \Omega_{\texttt{(}\tau_1\texttt{,}\tau_2\texttt{)}}}$$

$$\frac{}{\Gamma \vdash \texttt{bind}\,\texttt{(}x_1\texttt{:}\tau_1\texttt{,}x_2\texttt{:}\tau_2\texttt{)}\,\texttt{to}\,\Omega_{\tau_1\texttt{*}\tau_2}\,\texttt{in}\,e\,\texttt{end} \cong \Omega_{\tau'}}$$

### 12.3.7 Lists

We may extend MinML with list types $\tau\,\texttt{list}$ by adding the expressions `nil`, $e_1\texttt{::}e_2$, and $\texttt{listcase}\,e\,\texttt{of}\,\texttt{nil}\,\texttt{=>}\,e'\,\texttt{|}\,x\texttt{::}y\,\texttt{=>}\,e''\,\texttt{end}$, with the following typing rules:

$$\frac{}{\Gamma \vdash \texttt{nil} : \tau\,\texttt{list}}$$

$$\frac{}{\Gamma \vdash e_1\texttt{::}e_2 : \tau\,\texttt{list}}$$

[1] We do not consider nested or wildcard patterns, for the sake of simplicity. It is a simple matter to extend these rules to the more general case.

$$\frac{\Gamma \vdash e : \tau\,\texttt{list} \qquad \Gamma \vdash e' : \tau' \qquad \Gamma[x{:}\tau][y{:}\tau\,\texttt{list}] \vdash e'' : \tau'}{\Gamma \vdash \texttt{listcase}\, e\, \texttt{of}\, \texttt{nil} \,\texttt{=>}\, e' \,\texttt{|}\, x\,\texttt{::}\,y \,\texttt{=>}\, e'' \,\texttt{end} : \tau'}$$

The following symbolic evaluation and strictness rules express the evaluation of these constructs:

$$\overline{\Gamma \vdash \texttt{listcase}\,\texttt{nil}\,\texttt{of}\,\texttt{nil} \,\texttt{=>}\, e' \,\texttt{|}\, x\,\texttt{::}\,y \,\texttt{=>}\, e'' \,\texttt{end} \cong e' : \tau'}$$

$$\overline{\Gamma \vdash \texttt{listcase}\, v_1\,\texttt{::}\,v_2\, \texttt{of}\, \texttt{nil} \,\texttt{=>}\, e' \,\texttt{|}\, x_1\,\texttt{::}\,x_2 \,\texttt{=>}\, e'' \,\texttt{end} \cong [v_1, v_2/x_1, x_2]e'' : \tau'}$$

$$\overline{\Gamma \vdash \Omega_\tau\,\texttt{::}\,e \cong \Omega_{\tau\,\texttt{list}} : \tau\,\texttt{list}}$$

$$\overline{\Gamma \vdash e\,\texttt{::}\,\Omega_{\tau\,\texttt{list}} \cong \Omega_{\tau\,\texttt{list}} : \tau\,\texttt{list}}$$

$$\overline{\Gamma \vdash \texttt{listcase}\, \Omega_{\tau\,\texttt{list}}\, \texttt{of}\, \texttt{nil} \,\texttt{=>}\, e' \,\texttt{|}\, x_1\,\texttt{::}\,x_2 \,\texttt{=>}\, e'' \,\texttt{end} \cong \Omega_{\tau'} : \tau'}$$

Most importantly, we may prove equivalences by induction on the structure of a list. Suppose that $\Gamma[x{:}\tau\,\texttt{list}] \vdash e_i : \tau'$ $(i = 1, 2)$. To prove that

$$\Gamma[x{:}\tau\,\texttt{list}] \vdash e_1 \cong e_2 : \tau',$$

it suffices to show the following two facts:

1. $\Gamma \vdash [\texttt{nil}/x]e_1 \cong [\texttt{nil}/x]e_2 : \tau'$
2. For every $\Gamma \vdash v_h : \tau$ and $\Gamma \vdash v_t : \tau\,\texttt{list}$, if $\Gamma \vdash [v_t/x]e_1 \cong [v_t/x]e_2 : \tau'$, then $\Gamma \vdash [v_h\,\texttt{::}\,v_t/x]e_1 \cong [v_h\,\texttt{::}\,v_t/x]e_2 : \tau'$.

**Exercise 64**
*The list append and reversal functions are defined as follows:*[2]

```
fun app(l:τ list, m:τ list):τ list is
  listcase l of nil => m | h::t => h :: app (t, m) end
end

fun rev (l:τ list) is
  listcase l of nil => nil | h::t => app (rev(t), [h]) end
end
```

[2]Officially, the two-argument append function is written using bind as follows:

```
fun app(lm:τ list * τ list):τ list is bind (l:τ list, m:τ list) to lm
in ...end end.
```

We use infix notation for the append function for the sake of clarity.

*Use list induction and the laws of expression equivalence to prove the following two facts:*

1. $x{:}\tau\,\texttt{list}, y{:}\tau\,\texttt{list} \vdash \texttt{rev}(\texttt{app}(x, y)) \cong \texttt{app}(\texttt{rev}(y))\texttt{rev}(x) : \tau\,\texttt{list}$
2. $x{:}\tau\,\texttt{list} \vdash \texttt{rev}(\texttt{rev}(x)) \cong x : \tau\,\texttt{list}$