Programming Languages: Theory and Practice

(Working Draft of January 14, 2003)

Robert Harper
Carnegie Mellon University

Spring Semester, 2002
Preface

This is a collection of lecture notes for Computer Science 15–312 Programming Languages. This course has been taught by the author in the Spring of 1999 and 2000 at Carnegie Mellon University, and by Andrew Appel in the Fall of 1999, 2000, and 2001 at Princeton University. I am grateful to Andrew for his advice and suggestions, and to our students at both Carnegie Mellon and Princeton whose enthusiasm (and patience!) was instrumental in helping to create the course and this text.

What follows is a working draft of a planned book that seeks to strike a careful balance between developing the theoretical foundations of programming languages and explaining the pragmatic issues involved in their design and implementation. Many considerations come into play in the design of a programming language. I seek here to demonstrate the central role of type theory and operational semantics in helping to define a language and to understand its properties.

Comments and suggestions are most welcome. Please send any you may have to me by electronic mail.

Enjoy!
Contents

Preface iii

I Preliminaries 1

1 Inductive Definitions 3
  1.1 Informal Overview ........................................... 3
    1.1.1 Judgements and Rules .................................. 3
    1.1.2 Rule Induction ......................................... 5
    1.1.3 Defining Functions by Rule Induction .................. 7
    1.1.4 Admissible and Derivable Rules ......................... 7
  1.2 A More Rigorous Development ............................... 9
    1.2.1 Universes .............................................. 9
    1.2.2 Inference Rules ....................................... 10
    1.2.3 Rule Induction ....................................... 11
    1.2.4 Admissibility and Derivability ....................... 13
  1.3 Exercises .................................................. 14

2 Transition Systems 15
  2.1 Transition Systems ......................................... 15
  2.2 Exercises .................................................. 16

II Defining a Language 17

3 Concrete Syntax 19
  3.1 Context-Free Grammars ..................................... 19
  3.2 Ambiguity .................................................. 21
  3.3 Exercises .................................................. 23
4 First-Order Abstract Syntax 25
  4.1 Abstract Syntax Trees .................................. 25
  4.2 Structural Induction .................................... 26
  4.3 Parsing .................................................. 27
  4.4 Exercises ............................................... 29

5 Higher-Order Abstract Syntax 31
  5.1 Variables, Binding, and Scope ......................... 31
  5.2 Higher-Order Terms .................................... 35
  5.3 Renaming and Substitution .............................. 36
  5.4 Exercises ............................................... 37

6 Static Semantics 39
  6.1 Well-Formed Arithmetic Expressions .................. 39
  6.2 Exercises ............................................... 40

7 Dynamic Semantics 41
  7.1 Structured Operational Semantics ..................... 41
  7.2 Evaluation Semantics .................................. 44
  7.3 Relating Transition and Evaluation Semantics ........ 45
  7.4 Exercises ............................................... 46

III A Functional Language 47

8 MinML, A Minimal Functional Language 49
  8.1 Syntax .................................................. 49
    8.1.1 Concrete Syntax ................................... 49
    8.1.2 Abstract Syntax .................................. 50
  8.2 Static Semantics ...................................... 50
  8.3 Properties of Typing ................................... 52
  8.4 Dynamic Semantics .................................... 55
  8.5 Properties of the Dynamic Semantics .................. 57
  8.6 Exercises ............................................... 57

9 Type Safety for MinML 59
  9.1 Defining Type Safety ................................... 59
  9.2 Type Safety of MinML .................................. 60
  9.3 Run-Time Errors and Safety ............................ 63
16.1 Evaluation Semantics ........................................ 135
16.2 Relating Evaluation Semantics to Transition Semantics ..... 136
16.3 Cost Semantics ............................................... 137
16.4 Relating Cost Semantics to Transition Semantics .......... 138
16.5 Exercises ..................................................... 139

17 Implicit Parallelism ................................. 141
17.1 Tuple Parallelism ........................................ 141
17.2 Work and Depth .......................................... 143
17.3 Vector Parallelism ........................................ 146

18 A Parallel Abstract Machine ................. 151
18.1 A Simple Parallel Language .......................... 151
18.2 A Parallel Abstract Machine ....................... 153
18.3 Cost Semantics, Revisited ............................ 155
18.4 Provable Implementations (Summary) ............... 156

VIII Data Structures and Abstraction ............. 159
19 Aggregate Data Structures ....................... 161
19.1 Products ............................................... 161
19.2 Sums ................................................... 163
19.3 Recursive Types .................................... 164

20 Polymorphism ............................................ 167
20.1 Polymorphic MinML .................................... 168
20.2 ML-style Type Inference .............................. 174
20.3 Parametricity .......................................... 175
20.3.1 Informal Discussion ............................... 176
20.3.2 Relational Parametricity ......................... 179

21 Data Abstraction ........................................ 185
21.1 Existential Types .................................... 186
21.1.1 Abstract Syntax ................................ 186
21.1.2 Correspondence With ML ....................... 186
21.1.3 Static Semantics ................................ 188
21.1.4 Dynamic Semantics .............................. 189
21.1.5 Safety ........................................... 190
21.2 Representation Independence .................... 190
Part I

Preliminaries
Chapter 1

Inductive Definitions

Inductive definitions are an indispensable tool in the study of programming languages. In this chapter we will develop the basic framework of inductive definitions, and give some examples of their use.

1.1 Informal Overview

In this section we give an informal overview of inductive definitions, with an emphasis on how they are used in practice.

1.1.1 Judgements and Rules

An inductive definition consists of a collection of inference rules defining one or more judgements. A judgement is an assertion stating that a property holds of some object. For example, the judgement \( x \in \text{nat} \) might state that \( x \) is a natural number, and the judgement \( t \in \text{tree} \) might state that \( t \) is a binary tree.

The inference rules determine the conditions under which a judgement may be inferred, or derived. An inference rule has the form of an implication, stating that a judgement is inferrable whenever some other judgements (possibly none) are inferrable. Rules are written in the form

\[
\frac{J_1 \ldots J_n}{J}
\]

where \( J \) and each \( J_i \) (\( 1 \leq i \leq n \)) are judgements. The judgement \( J \) is called the conclusion of the rule, and the judgements \( J_1, \ldots, J_n \) are its premises. If a rule has no premises (i.e., \( n = 0 \)), the rule is called an axiom.
A rule of this form states that the judgement $J$ is inferrable, provided that each of the judgements $J_1, \ldots, J_n$ is inferrable. Thus axioms state that a judgement is inferrable unconditionally, whereas rules with premises state the conditional inferrability of a judgement. For example, the following set of rules, $R_N$, constitute an inductive definition of the judgement $x$ \text{nat}:

\[
\begin{array}{c}
\text{zero} \text{nat} \\
\text{succ}(x) \text{nat}
\end{array}
\]

The first rule states that \text{zero} is a natural number. The second states that if $x$ is a natural number, then so is $\text{succ}(x)$.

Rules may be composed to form a derivation of a judgement $J$ from premises $J_1, \ldots, J_n$. A derivation is a tree whose nodes are judgements such that the children of a node are the premises of some rule ending with the judgement at that node. Such a tree is a derivation of a judgement $J$ from premises $J_1, \ldots, J_n$ iff the root of the tree is $J$ and its leaves are the judgements $J_1, \ldots, J_n$.

Derivation trees are normally depicted as “stacked” inference rules. For example, here is a derivation of the judgement $\text{succ}(\text{succ}(\text{zero}))$ \text{nat}:

\[
\begin{array}{c}
\text{zero} \text{nat} \\
\text{succ}(\text{zero}) \text{nat} \\
\text{succ}(\text{succ}(\text{zero})) \text{nat}
\end{array}
\]

To take another example, here is an inductive definition of the judgement $t$ \text{tree}, stating that $t$ is a binary tree:

\[
\begin{array}{c}
\text{empty} \text{tree} \\
\text{x} \text{tree} \quad \text{y} \text{tree} \\
\text{node}(x, y) \text{tree}
\end{array}
\]

Using these rules, we may construct a derivation of the judgement

$$\text{node}\left(\text{empty}, \text{node}\left(\text{empty}, \text{empty}\right)\right) \text{tree}$$

as follows:

\[
\begin{array}{c}
\text{empty} \text{tree} \\
\text{empty} \text{tree} \\
\text{empty} \text{tree} \\
\text{node}(\text{empty}, \text{empty}) \text{tree} \\
\text{node}(\text{empty}, \text{node}(\text{empty}, \text{empty})) \text{tree}
\end{array}
\]

In practice, we find a derivation of a judgement $J$ by starting with $J$ and working “backwards”, looking for a rule ending with $J$ with premises $J_1, \ldots, J_n$, then finding derivations of each of the $J_i$’s by the same procedure. This process is called goal-directed search; the judgement $J$ is the goal,
1.1 Informal Overview

and each of the \( J_i \)'s are subgoals. Note that there may be many rules ending with \( J \); if we fail to find a derivation by using one rule, we may have to abandon the attempt, and try another rule instead. If \( J \) is, in fact, derivable, then this process will eventually find a derivation, but if not, there is no guarantee that it will terminate! We may, instead, futilely apply rules forever, introducing more sub-goals each time, and never completing the derivation.

Often we give a simultaneous inductive definition of several judgements at once. For example, here is a simultaneous inductive definition of the judgements \( t \) tree, stating that \( t \) is a variadic tree, and \( f \) forest, stating that \( f \) is a variadic forest. By “variadic” we mean that the number of children of any given node in a tree varies with each node.

\[
\begin{array}{c}
\text{\( f \) forest} \\
\text{\( \text{node}(f) \) tree} \\
\text{\( \text{nil} \) forest} \\
\text{\( \text{cons}(t, f) \) forest}
\end{array}
\]

1.1.2 Rule Induction

What makes an inductive definition inductive is that the rules are exhaustive in the sense that a judgement is defined to hold iff it can be inferred by these rules. This means that if a judgement \( J \) is inferrable from a rule set \( \mathcal{R} \), then there must be a rule in \( \mathcal{R} \) ending with \( J \) such that each of its premises are also inferrable. For example, if \( n \) nat is inferrable according to the rules \( \mathcal{R}_N \), then either it is inferrable by the first rule, in which case \( n = \text{zero} \), or by the second, in which case \( n = \text{succ}(m) \) and \( m \) nat is itself inferrable. Similarly, if \( t \) tree is inferrable according to the rules \( \mathcal{R}_T \) given above, then either \( t = \text{empty} \) or \( t = \text{node}(t_1, t_2) \), where \( t_1 \) tree and \( t_2 \) tree are both inferrable.

This observation provides the basis for reasoning about derivable judgements by rule induction (also known as induction on derivations). For any set of rules, \( \mathcal{R} \), to show that a property \( P \) holds of every inferrable judgement, it is enough to show that for every rule

\[
\frac{J_1 \cdots J_n}{J}
\]

in \( \mathcal{R} \), if \( J_1, \ldots, J_n \) all have property \( P \), then \( J \) also has property \( P \). By doing this for every rule in \( \mathcal{R} \), we cover all the cases, and establish that \( P \) holds for every inferrable judgement.

The assumption that \( P \) holds for each premise of a rule is called the inductive hypothesis. The proof that \( P \) holds for the conclusion, under these
assumptions, is called the inductive step. In the case of axioms the inductive hypothesis is vacuous; we must simply establish the conclusion outright, with no further assumptions to help us. If we can carry out the inductive step for each rule in $\mathcal{R}$, we thereby establish that $P$ holds for every inferrable judgement, since the inference must arise by the application of some rule whose premises are derivable (and hence, by inductive hypothesis, have the property $P$).

For example, consider again the rule set $\mathcal{R}_N$. The principle of rule induction for $\mathcal{R}_N$ states that to show $P(n \text{ nat})$, it is enough to show

1. $P(\text{zero } \text{nat})$;
2. if $P(n \text{ nat})$, then $P(\text{succ}(n) \text{ nat})$.

This is, of course, the familiar principle of mathematical induction.

Similarly, the principle of rule induction for $\mathcal{R}_T$ states that if we are to show that $P(t \text{ tree})$, it is enough to show

1. $P(\text{empty } \text{tree})$;
2. if $P(t_1 \text{ tree})$ and $P(t_2 \text{ tree})$, then $P(\text{node}(t_1, t_2) \text{ tree})$.

This is called the principle of tree induction, or induction on the structure of a tree.

As a notational convenience, when the judgements in question are all of the form $x \ l$ for $x$ an object and $l$ is a property of the object $x$, we often write $P_l(x)$, rather than the more cumbersome $P(x \ l)$. If there is only one form of judgement, $x \ l$, then we often drop the subscript entirely, writing just $P(x \ l)$. Thus, instead of writing $P(n \text{ nat})$, we may write $P_{\text{nat}} n$, or just $P(n)$, when it is clear from context that we are working the $\mathcal{R}_N$. Similarly $P(t \text{ tree})$ is often written $P_{\text{tree}}(t)$, or just $P(t)$.

Rule sets that define more than one judgement give rise to proofs by simultaneous induction. For example, if we wish to show $P_{\text{tree}}(t)$ for all $t$ such that $t \text{ tree}$ and $P_{\text{forest}}(f)$ for all $f$ such that $f \text{ forest}$, then it is enough to show

1. if $P_{\text{forest}}(f)$, then $P_{\text{tree}}(\text{node}(f))$.
2. $P_{\text{forest}}(\text{nil})$.
3. if $P_{\text{tree}}(t)$ and $P_{\text{forest}}(f)$, then $P_{\text{forest}}(\text{cons}(t, f))$.

It is easy to check that this induction principle follows from the general principle of rule induction by simply working through the rules $\mathcal{R}_{TF}$, taking account of the notational conventions just mentioned.
1.1 Informal Overview

1.1.3 Defining Functions by Rule Induction

A common use of rule induction is to justify the definition of a function by a set of equations. For example, consider the following recursion equations:

\[
\begin{align*}
\text{hgt}_{\text{tree}}(\text{empty}) & = 0 \\
\text{hgt}_{\text{tree}}(\text{node}(t_1, t_2)) & = 1 + \max(\text{hgt}_{\text{tree}}(t_1), \text{hgt}_{\text{tree}}(t_2))
\end{align*}
\]

We prove by rule induction that if \( t \) is a tree then there exists a unique \( n \geq 0 \) such that \( \text{hgt}_{\text{tree}}(t) = n \). In other words, the above equations determine a function, \( \text{hgt} \).

We consider each rule in \( \mathcal{R}_T \) in turn. The first rule, stating that \( \text{empty} \) is covered by the first equation. For the second rule, we may assume that \( \text{hgt}_{\text{tree}} \) assigns a unique height to \( t_1 \) and \( t_2 \). But then the second equation assigns a unique height to \( t = \text{node}(t_1, t_2) \).

Similarly, we may prove by simultaneous induction that the following equations define the height of a variadic tree and a variadic forest:

\[
\begin{align*}
\text{hgt}_{\text{tree}}(\text{node}(f)) & = 1 + \text{hgt}_{\text{forest}}(f) \\
\text{hgt}_{\text{forest}}(\text{nil}) & = 0 \\
\text{hgt}_{\text{forest}}(\text{cons}(t, f)) & = \max(\text{hgt}_{\text{tree}}(t), \text{hgt}_{\text{forest}}(f)).
\end{align*}
\]

It is easy to show by simultaneous induction that these equations determine two functions, \( \text{hgt}_{\text{tree}} \) and \( \text{hgt}_{\text{forest}} \).

1.1.4 Admissible and Derivable Rules

Given an inductive definition consisting of a set of rules \( \mathcal{R} \), there are two senses in which a rule

\[
\begin{array}{c}
J_1 \\
\vdots \\
J_n \\
\hline
J
\end{array}
\]

may be thought of as being redundant.

Such a rule is said to be derivable if there is a derivation of \( J \) from premises \( J_1, \ldots, J_n \). This means that there is a composition of rules starting with the \( J_i \)'s and ending with \( J \). For example, the following rule is derivable in \( \mathcal{R}_N \):

\[
\begin{array}{c}
n \text{nat} \\
\hline
\text{succ}(\text{succ}(\text{succ}(n))) \text{ nat.}
\end{array}
\]
Inductive Definitions

Its derivation is as follows:

\[
\begin{array}{c}
\text{n nat} \\
\hline
\text{succ(n) nat} \\
\hline
\text{succ(succ(n)) nat} \\
\hline
\text{succ(succ(succ(n))) nat.}
\end{array}
\]

Such a rule is said to be \textit{admissible} iff its conclusion is derivable from no premises whenever its premises are derivable from no premises. For example, the following rule is \textit{admissible} in \( \mathcal{R}_N \):

\[
\begin{array}{c}
\text{succ(n) nat} \\
\hline
\text{n nat}.
\end{array}
\]

First, note that this rule is not derivable for any choice of \( n \). For if \( n = \text{zero} \), then the only rule that applies has no premises, and if \( n = \text{succ}(m) \), then the only rule that applies has as premise \( m \text{ nat} \), rather than \( n \text{ nat} \). However, this rule is admissible! We may prove this by induction on the derivation of the premise of the rule. For if \( \text{succ}(n) \text{ nat} \) is derivable from no premises, it can only be by second rule, which means that \( n \text{ nat} \) is also derivable, as required.

While this example shows that not every admissible rule is derivable, the converse holds. For a rule to be derivable means precisely that if its premises are derivable, then so is its conclusion!

The distinction between admissible and derivable rules can be hard to grasp at first. One way to gain intuition is to note that if a rule is derivable in a rule set \( \mathcal{R} \), then it remains derivable in any rule set \( \mathcal{R}' \supseteq \mathcal{R} \). This is because the derivation of that rule depends only on what rules are available, and is not sensitive to whether any other rules are also available. In contrast a rule can be admissible in \( \mathcal{R} \), but inadmissible in some extension \( \mathcal{R}' \supseteq \mathcal{R} \)! For example, suppose that we add to \( \mathcal{R}_N \) the rule

\[
\text{succ(junk) nat.}
\]

Now it is no longer the case that the rule

\[
\begin{array}{c}
\text{succ(n) nat} \\
\hline
\text{n nat}.
\end{array}
\]

is admissible, because if the premise were derived by the additional rule, there is no way to obtain a derivation of \textit{junk nat}!
1.2 A More Rigorous Development

Since admissibility is sensitive to which rules are absent, as well as to which are present, a proof of admissibility almost always proceeds by induction on one or more of its premises. This constitutes an exhaustive analysis of how the premises might have been derived, and concludes that in each case the conclusion must also have been derived. Adding an additional rule requires that we add an additional case to the proof, and there is no assurance (as we have just illustrated) that this will go through.

1.2 A More Rigorous Development

In this section we will give a more rigorous account of inductive definitions of a subset of a given set. This will include as a special case the foregoing treatment of inductive definitions of judgements, and will make clear the mathematical underpinnings of the principle of rule induction.

1.2.1 Universes

We will consider inductive definitions of subsets of some fixed universe of objects. In principle we may consider inductive definitions over any set of objects we like, but in practice we confine ourselves to sets of finitary objects, which can be put into one-to-one correspondence with the natural numbers. Given such a correspondence, it suffices to make all inductive definitions over the set of natural numbers. However, doing so requires that we explicitly define the encoding of each object of interest as a natural number, called its Gödel number. To avoid this complication we take a more liberal approach in which we admit inductive definitions over any specified set of objects.

For example, we will make use of the set of (finite) strings over a given alphabet as a universe for inductive definitions. Let \( \Sigma \) be a countable set of symbols, or letters, or characters. For example, \( \Sigma \) might be the set of ASCII or Unicode characters. The set of strings over \( \Sigma \), written \( \Sigma^* \), consists of the finite sequences of symbols from \( \Sigma \). We write \( s_1 \ s_2 \) for the concatenation of the string \( s_1 \) followed by the string \( s_2 \), write \( \varepsilon \) for the null string, and treat every \( a \in \Sigma \) as string of length 1.

Another example is the set of (first-order) terms over a given set of operators. Let \( \mathcal{O} \) be a countable set of operators, and let \( \alpha : \mathcal{O} \to \mathbb{N} \) be an assignment of arities to each of the operators. An operator \( o \in \mathcal{O} \) of arity \( n \) (i.e., for which \( \alpha(o) = n \)) is said to be \( n \)-ary; the 0-ary operators are called constants. The set \( T \) of abstract syntax trees (ast’s), or terms, consists of
all expressions of the form \(o(t_1, \ldots, t_n)\), where \(o\) is an \(n\)-ary operator, and 
\(t_1, \ldots, t_n\) are themselves ast’s. Such a term may be depicted as an ordered 
Tree with root labelled by the operator \(o\), and with \(n\) children corresponding 
to the terms \(t_1, \ldots, t_n\).

We often work with combinations of these basic universes. For exam-
ple, we may consider inductive subsets of \(T \times T\), the set of ordered pairs of 
ast’s, and so forth. Generally we will leave implicit the exact choice of the 
universe for a particular inductive definition.

1.2.2 Inference Rules

An inductive definition of a subset of a universe \(U\) consists of a collection 
of rules over \(U\). A rule over \(U\) has the form

\[
\frac{x_1 \ldots x_n}{x}
\]

where \(x \in U\) and each \(x_i \in U\) \((1 \leq i \leq n)\). Thus a rule consists of a finite 
subset of \(U\) and an element of \(U\). The element \(x\) is called the conclusion 
of the rule; the elements \(x_1, \ldots, x_n\) are called the premises of the rule. A rule 
set is, quite obviously, a set of rules.

A subset \(A \subseteq U\) is closed under \(\mathcal{R}\), or \(\mathcal{R}\)-closed, iff \(x \in A\) whenever

\[
\frac{x_1 \ldots x_n}{x}
\]

is a rule in \(\mathcal{R}\) and each \(x_i \in A\) for every \(1 \leq i \leq n\).

The subset \(I = I(\mathcal{R})\) inductively defined by \(\mathcal{R}\) is given by the equation

\[
I(\mathcal{R}) = \bigcap \{ A \subseteq U \mid A \text{ is } \mathcal{R}\text{-closed} \}.
\]

As we shall see shortly, this is the smallest set closed under \(\mathcal{R}\).

For example, here is a set, \(\mathcal{R}_P\), of rules for deriving strings that are, as 
we shall prove later, are palindromes:

\[
\varepsilon \quad \overline{a} \quad \frac{8}{a \overline{a}}
\]

The set of rules \(\mathcal{R}_P\) just given has \(2 \times |\Sigma| + 1\) rules, where \(|\Sigma|\) is the 
cardinality of the alphabet \(\Sigma\). In particular, if \(\Sigma\) is infinite, then there are in-
finitely many rules! Since we cannot expect to write down infinitely many 
rules, we need some means of defining large (or even infinite) rule sets. 
Here we have specified these using rule schemes. A rule scheme is a rule
1.2 A More Rigorous Development

involving one or more parameters ranging over a specified set (by default, the universe). For example, the third rule above is a rule scheme with two parameters, a and s. The rule scheme determines one rule for each possible choice of character $a \in \Sigma$ and $s \in \Sigma^*$.

A simultaneous inductive definition of one or more judgements can be considered a single inductive definition of a subset of a suitable universe by a simple “labelling” device. A simultaneous inductive definition of the judgements $x_1 l_1, \ldots, x_n l_n$, where each $x_i$ ranges over a universe $\mathcal{U}$, may be thought of as a simple inductive definition of a subset of the disjoint union of $n$ copies of $\mathcal{U}$, namely

$$\mathcal{U} \times \{ l_1, \ldots, l_n \} = \{ x l_i \mid x \in \mathcal{U}, \ 1 \leq i \leq n \},$$

where we write $x l_i$ for the ordered pair $(x, l_i)$. The rules defining these judgements emerge as rules over this enlarged universe. Thus the rules $R_{TF}$ given above may be seen as defining an inductive subset of the universe $\mathcal{T} \times \{ \text{tree, forest} \}$.

1.2.3 Rule Induction

As we mentioned earlier, the set $I(\mathcal{R})$ is the least set closed under $\mathcal{R}$.

**Theorem 1.1**

Let $\mathcal{R}$ be a rule set over $\mathcal{U}$, and let $I = I(\mathcal{R})$.

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

**Proof:**

1. Suppose that

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

is a rule in $\mathcal{R}$, and that

$$X = \{ x_1, \ldots, x_n \} \subseteq I.$$

Since $I$ is the intersection of all $\mathcal{R}$-closed sets, $X \subseteq A$ for each $\mathcal{R}$-closed set $A$. But then $x \in A$ for each such $A$, by the definition of $\mathcal{R}$-closure, and hence $x$ is an element of their intersection, $I$.

2. If $A$ is $\mathcal{R}$-closed, then it is among the sets in the intersection defining $I$. So $I \subseteq A$.  

**Working Draft** January 14, 2003
The importance of this theorem is that it licenses the principle of *proof by rule induction* for a rule set \( \mathcal{R} \):

To show that \( I(\mathcal{R}) \subseteq X \), it suffices to show that \( X \) is \( \mathcal{R} \)-closed.

That is, if we wish to show that \( x \in X \) for every \( x \in I(\mathcal{R}) \), it is enough to show that \( X \) is closed under the rules \( \mathcal{R} \).

Returning to the inductively defined set \( P \) above, suppose we wish to show that every \( s \in P \) is in fact a palindrome. That is, we wish to show that

\[
P \subseteq \{ s \in \Sigma^* \mid s = s^R \}.
\]

For this to hold, it is enough to show that the set of palindromes is closed under the rules \( \mathcal{R} \). We consider each rule in turn, showing that if the premises are palindromes, then so is the conclusion.

1. \( \varepsilon = \varepsilon^R \), so \( \varepsilon \) is a palindrome.
2. \( a = a^R \) for every \( a \in \Sigma \), so \( a \) is a palindrome.
3. Assume that \( s = s^R \). Observe that

\[
(a s a)^R = a s^R a = a s a.
\]

This completes the proof.\(^1\)

The *parity* of a palindrome is either 0 or 1, according to whether its length is either even or odd. Now that we know that the set of palindromes is inductively defined by the rules given earlier, we may define the parity function by the following equations:

\[
\begin{align*}
\text{parity}(\varepsilon) & = 0 \\
\text{parity}(a) & = 1 \\
\text{parity}(a s a) & = \text{parity}(s)
\end{align*}
\]

Notice that we include one clause of the function definition for each rule defining the domain of the function.

Why does this define a function? We must prove that if \( s \) is a palindrome, then there exists a unique \( x \in \{ 0, 1 \} \) such that \( \text{parity}(s) = x \). This

---

\(^1\)You might also like to prove that every palindrome is a member of \( P \). This can be achieved by (strong) induction over the length of palindrome \( s \).
1.2 A More Rigorous Development

may be proved by rule induction by showing that the property \( P(s) \) given by the formula

\[
\exists! x \in \{0, 1\} \; \text{parity}(s) = x
\]

is closed under the rules \( \mathcal{R}_P \). The first two rules, for the null string and the single-letter strings, are covered by the first two clauses of the definition of \text{parity}. For the third rule, we assume that \text{parity} is well-defined for \( s \) (i.e., there exists a unique \( x \) such that \( \text{parity}(s) = x \)). But then it follows directly from the third clause of the definition of \text{parity} that it is uniquely defined for \( asa \).

1.2.4 Admissibility and Derivability

Using this machinery we may shed additional light on admissibility and derivability of inference rules. Fix a rule set \( \mathcal{R} \) over some universe \( U \). A rule

\[
\frac{x_1 \ldots x_n}{x}
\]

is derivable iff \( x \in I(\mathcal{R} \cup \{x_1, \ldots, x_n\}) \). That is, we take \( x_1, \ldots, x_n \) as new axioms, and ask whether \( x \) is derivable according to this expansion of the rule set. The same rule is admissible iff \( x \in I(\mathcal{R}) \) whenever \( x_i \in I(\mathcal{R}) \) for each \( 1 \leq i \leq n \). That is, we check whether \( x \) is in the set inductively defined by \( \mathcal{R} \), whenever the \( x_i \)'s are in that same set.

For example, consider the rule set \( \mathcal{R}_P \) defining the palindromic strings. It is easy to see that the rule

\[
\frac{s}{aba}\]

is derivable, since if we add \( s \) as a new axiom, then we can apply the third rule of \( \mathcal{R}_P \) to this axiom twice to obtain a derivation of \( aba \). On the other hand, the rule

\[
\frac{asa}{s}
\]

is admissible, since if \( asa \in I(\mathcal{R}) \), then so we must also have \( s \in I(\mathcal{R}) \). This is easily proved by rule induction, showing that the set

\[
\{ t \mid t = asa \text{ and } s \in I(\mathcal{R}) \} \cup \Sigma \cup \{ \varepsilon \}
\]

is \( \mathcal{R}_P \)-closed.
1.3 Exercises

1. Give (simultaneous) inductive definitions of various languages.

2. Prove properties of these languages, including well-definedness of certain functions over them.

3. Fill in missing proofs.
Chapter 2

Transition Systems

Transition systems are fundamental to the study of programming languages. They are used to describe the execution behavior of programs by defining an abstract computing device with a set, $S$, of states that are related by a transition relation, $\rightarrow$. The transition relation describes how the state of the machine evolves during execution.

2.1 Transition Systems

A transition system consists of a set $S$ of states, a subset $I \subseteq S$ of initial states, a subset $F \subseteq S$ of final states, and a binary transition relation $\rightarrow \subseteq S \times S$. We write $s \rightarrow s'$ to indicate that $(s, s') \in \rightarrow$. It is convenient to require that $s \not\rightarrow$ in the case that $s \in F$.

An execution sequence is a sequence of states $s_0, \ldots, s_n$ such that $s_0 \in I$, and $s_i \rightarrow s_{i+1}$ for every $0 \leq i < n$. An execution sequence is maximal iff $s_n \not\rightarrow$; it is complete iff it is maximal and, in addition, $s_n \in F$. Thus every complete execution sequence is maximal, but maximal sequences are not necessarily complete.

A state $s \in S$ for which there is no $s' \in S$ such that $s \rightarrow s'$ is said to be stuck. We require that all final states are stuck: if $s \in F$, then $s \not\rightarrow$. But not all stuck states are final; these correspond to “run-time errors”, states for which there is no well-defined next state.

A transition system is deterministic iff for every $s \in S$ there exists at most one $s' \in S$ such that $s \rightarrow s'$. Most of the transition systems we will consider in this book are deterministic, the notable exceptions being those used to model concurrency.
The reflexive, transitive closure, $\rightarrow^*$, of the transition relation $\rightarrow$ is inductively defined by the following rules:

$$
\frac{s \rightarrow^* s}{s \rightarrow^* s} \quad \frac{s \rightarrow s' \quad s' \rightarrow^* s''}{s \rightarrow^* s''}
$$

It is easy to prove by rule induction that $\rightarrow^*$ is indeed reflexive and transitive.

The complete transition relation, $\rightarrow^1$, is the restriction to $\rightarrow^*$ to $S \times F$. That is, $s \rightarrow^1 s'$ iff $s \rightarrow^* s'$ and $s' \in F$.

The multistep transition relation, $\rightarrow^n$, is defined by induction on $n \geq 0$ as follows:

$$
\frac{s \rightarrow^0 s}{s \rightarrow^0 s} \quad \frac{s \rightarrow s' \quad s' \rightarrow^n s''}{s \rightarrow^{n+1} s''}
$$

It is easy to show that $s \rightarrow^* s'$ iff $s \rightarrow^n s'$ for some $n \geq 0$.

Since the multistep transition is inductively defined, we may prove that $P(e, e')$ holds whenever $e \rightarrow^* e'$ by showing

1. $P(e, e)$.
2. if $e \rightarrow e'$ and $P(e', e'')$, then $P(e, e'')$.

The first requirement is to show that $P$ is reflexive. The second is often described as showing that $P$ is closed under head expansion, or closed under reverse evaluation.

### 2.2 Exercises

1. Prove that $s \rightarrow^* s'$ iff there exists $n \geq 0$ such that $s \rightarrow^n s'$. 

---

**Working Draft** January 14, 2003
Part II

Defining a Language
Chapter 3

Concrete Syntax

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, improving readability is, of course, a matter of taste about which reasonable people may disagree. Techniques for eliminating ambiguity include precedence conventions for binary operators and various forms of parentheses for grouping sub-expressions. Techniques for enhancing readability include the use of suggestive key words and phrases, and establishment of punctuation and layout conventions.

3.1 Context-Free Grammars

The standard method for defining concrete syntax is by giving a context-free grammar (CFG) for the language. A grammar consists of three things:

1. An alphabet $\Sigma$ of terminals, or letters.

2. A finite set $\mathcal{N}$ of non-terminals that stand for the syntactic categories.

3. A set $\mathcal{P}$ of productions of the form $A ::= \alpha$, where $A$ is a non-terminal and $\alpha$ is a string of terminals and non-terminals.
Whenever there is a set of productions

\[
A ::= \alpha_1
\]

\[
\vdots
\]

\[
A ::= \alpha_n.
\]

all with the same left-hand side, we often abbreviate it as follows:

\[
A ::= \alpha_1 \mid \cdots \mid \alpha_n.
\]

A context-free grammar is essentially a simultaneous inductive definition of its syntactic categories. Specifically, we may associate a rule set \( R \) with a grammar according to the following procedure. First, we treat each non-terminal as a label of its syntactic category. Second, for each production

\[
A ::= s_1 A_1 s_2 \cdots s_{n-1} A_n s_n
\]

of the grammar, where \( A_1, \ldots, A_n \) are all of the non-terminals on the right-hand side of that production, and \( s_1, \ldots, s_n \) are strings of terminals, add a rule

\[
\frac{t_1 A_1 \cdots t_n A_n}{s_1 t_1 s_2 \cdots s_{n-1} t_n s_n A}
\]

to the rule set \( R \). For each non-terminal \( A \), we say that \( s \) is a string of syntactic category \( A \), written \( s \in L(A) \), iff \( s \in I(R)_A \) (i.e., \( s A \in I(R) \)).

An example will make these ideas clear. Let us give a grammar defining the syntax of a simple language of arithmetic expressions extended with a variable-binding construct.

\[
\text{Digits} \quad d ::= 0 \mid 1 \mid \cdots \mid 9
\]

\[
\text{Numbers} \quad n ::= d | n d
\]

\[
\text{Expressions} \quad e ::= n | e + e | e * e
\]

A number \( n \) is a non-empty sequence of decimal digits. An expression \( e \) is either a number \( n \), or the sum or product of two expressions.

Here is this grammar presented as a simultaneous inductive definition:

\[
\begin{align*}
0 \text{ digit} & \cdots 9 \text{ digit} \\
\frac{d \text{ digit} \quad n \text{ number} \quad d \text{ digit}}{d \text{ number} \quad n d \text{ number}}
\end{align*}
\]
3.2 Ambiguity

Let $R$ be the above set of rules, and let $I = I(R)$. The syntactic categories of the grammar are the sections of $I$ by the non-terminal standing for that category. For example, the set of expressions is $I_{\text{expr}}$, and so forth.

### 3.2 Ambiguity

Apart from subjective matters of readability, a principal goal of concrete syntax design is to eliminate ambiguity. The grammar of arithmetic expressions given above is ambiguous in the sense that some strings may be thought of as arising in several different ways. For example, the string $1+2\times3$ may be thought of as arising by applying the rule for multiplication first, then the rule for addition, or vice versa. The former interpretation corresponds to the expression $(1+2)\times3$; the latter corresponds to the expression $1+(2\times3)$.

The trouble is that we cannot simply tell from the generated string which reading is intended. This causes numerous problems. For example, suppose that we wish to define a function $\text{eval}$ that assigns to each arithmetic expression $e$ its value $n \in N$. A natural approach is to use rule induction on the rules determined by the grammar of expressions.

We will define three functions simultaneously, as follows:

\[
\begin{align*}
\text{eval}_{\text{dig}}(0) & = 0 \\
\vdots & \\
\text{eval}_{\text{dig}}(9) & = 9 \\
\text{eval}_{\text{num}}(d) & = \text{eval}_{\text{dig}}(d) \\
\text{eval}_{\text{num}}(n\,d) & = 10 \times \text{eval}_{\text{num}}(n) + \text{eval}_{\text{dig}}(d) \\
\text{eval}_{\text{exp}}(n) & = \text{eval}_{\text{num}}(n) \\
\text{eval}_{\text{exp}}(e_1+e_2) & = \text{eval}_{\text{exp}}(e_1) + \text{eval}_{\text{exp}}(e_2) \\
\text{eval}_{\text{exp}}(e_1\times e_2) & = \text{eval}_{\text{exp}}(e_1) \times \text{eval}_{\text{exp}}(e_2)
\end{align*}
\]
The all-important question is: *are these functions well-defined?* The answer is *no!* The reason is that a string such as $1+2*3$ arises in two different ways, using either the rule for addition expressions (thereby reading it as $1+(2*3)$) or the rule for multiplication (thereby reading it as $(1+2)*3$). Since these have different values, it is impossible to prove that there exists a unique value for every string of the appropriate grammatical class. (It is true for digits and numbers, but not for expressions.)

What do we do about ambiguity? The two most common methods to eliminate this kind of ambiguity are these:

1. Introduce parenthesization into the grammar so that the person writing the expression can choose the intended interpretation.

2. Introduce precedence relationships that resolve ambiguities between distinct operations (*e.g.*, by stipulating that multiplication takes precedence over addition).

3. Introduce associativity conventions that determine how to resolve ambiguities between operators of the same precedence (*e.g.*, by stipulating that addition is right-associative).

Using these techniques, we arrive at the following revised grammar for arithmetic expressions.

\[
\begin{align*}
\text{Digits} & \quad d ::= \ 0 \ | \ 1 \ | \cdots \ | \ 9 \\
\text{Numbers} & \quad n ::= \ d \ | \ nd \\
\text{Expressions} & \quad e ::= \ t \ | \ t+e \\
\text{Terms} & \quad t ::= \ f \ | \ f*t \\
\text{Factors} & \quad f ::= \ n \ | \ (e)
\end{align*}
\]

We have made two significant changes. The grammar has been “layered” to express the precedence of multiplication over addition and to express right-associativity of each, and an additional form of expression, parenthesization, has been introduced.

It is a straightforward exercise to translate this grammar into an inductive definition. Having done so, it is also straightforward to revise the definition of the evaluation functions so that are well-defined. The revised definitions are given by rule induction; they require additional clauses for
the new syntactic categories.

\[
\begin{align*}
\text{eval}_{\text{dig}}(0) &= 0 \\
\vdots \\
\text{eval}_{\text{dig}}(9) &= 9 \\
\text{eval}_{\text{num}}(d) &= \text{eval}_{\text{dig}}(d) \\
\text{eval}_{\text{num}}(n\ d) &= 10 \times \text{eval}_{\text{num}}(n) + \text{eval}_{\text{dig}}(d) \\
\text{eval}_{\text{exp}}(t) &= \text{eval}_{\text{trm}}(t) \\
\text{eval}_{\text{exp}}(t+e) &= \text{eval}_{\text{trm}}(t) + \text{eval}_{\text{exp}}(e) \\
\text{eval}_{\text{trm}}(f) &= \text{eval}_{\text{trm}}(f) \\
\text{eval}_{\text{trm}}(f*t) &= \text{eval}_{\text{trm}}(f) \times \text{eval}_{\text{trm}}(t) \\
\text{eval}_{\text{trm}}(n) &= \text{eval}_{\text{num}}(n) \\
\text{eval}_{\text{trm}}(e) &= \text{eval}_{\text{exp}}(e)
\end{align*}
\]

A straightforward proof by rule induction shows that these functions are well-defined.

### 3.3 Exercises

1. Give context-free grammars for various languages.

2. Ensure that a grammar is parseable using the techniques described here.
Chapter 4

First-Order Abstract Syntax

The concrete syntax of a language is an inductively-defined set of strings over a given alphabet. Its first-order abstract syntax is an inductively-defined set of first-order terms, or ast’s, over a set of operators. Abstract syntax avoids the ambiguities of concrete syntax by employing operators that determine the outermost form of any given expression, rather than relying on parsing conventions to disambiguate strings. The reason to call this representation “first-order” will become apparent in Chapter 5, where we introduce mechanisms to account for the binding and scope of variables.

4.1 Abstract Syntax Trees

To specify the first-order abstract syntax of a language, it is necessary to specify in advance the set of operators, and their arities, used to build ast’s. For example, to specify the abstract syntax of a language of arithmetic expressions, we may specify the following operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Arity</th>
</tr>
</thead>
<tbody>
<tr>
<td>num(n)</td>
<td>0</td>
</tr>
<tr>
<td>+</td>
<td>2</td>
</tr>
<tr>
<td>*</td>
<td>2</td>
</tr>
</tbody>
</table>

Here $n$ ranges over the natural numbers; the operator $\text{num}(n)$ is the $n$th numeral, which takes no arguments. The operators $+$ and $*$ take two arguments each, as might be expected.

Having specified the operators, we then give an inductive definition of the various syntactic categories of the language. For example, in the
case of arithmetic expressions there is only one syntactic category, that of expressions, which may be specified as follows:

\[
\begin{align*}
\text{num}(n) & \quad \text{expr} \\
\text{plus}(e_1, e_2) & \quad \text{expr} \\
\times & \quad \text{expr} \\
\end{align*}
\]

Notice that the conclusion of each rule is an ast whose outermost constructor uniquely identifies the rule used to construct it.

As an alternative to rules, we often use a notation similar to context-free grammars to specify the abstract syntax. The difference compared to similar specifications of concrete syntax lies in how we interpret the grammar. In the case of concrete syntax we interpret the grammar as a simultaneous inductive definition of sets of strings, whereas in the case of (first-order) abstract syntax, we interpret it as a simultaneous inductive definition of sets of ast’s. For example, the abstract syntax of the language of arithmetic expressions introduced in Chapter 3 may be defined by the following grammar:

\[
\text{Expressions} \quad e \ ::= \quad \text{num}(n) \mid \text{plus}(e_1, e_2) \mid \times(e_1, e_2)
\]

This grammar, understood as a specification of abstract syntax, has the same meaning as the rules just given for the same language.

In practice we do not explicitly declare the operators and their arities in advance of giving an inductive definition of the abstract syntax of a language. Instead we leave it to the reader to infer the set of operators and their arities required for the definition to make sense.

### 4.2 Structural Induction

When applied to the rules defining the abstract syntax of a language, the principle of rule induction is called \textit{structural induction}. We say that a proposition is proved “by induction on the structure of …” or “by structural induction on …” to indicate that we are applying the general principle of rule induction to the rules defining the abstract syntax of some expression.
In the case of the abstract syntax of arithmetic expressions just given, the principle of structural induction is as follows. To prove that a property $P$ holds of every expression $e$ of the abstract syntax, it is enough to show that $P$ is closed under the rules defining the abstract syntax. Specifically,

1. Show that $P$ holds of $\text{num}(n)$ for any number $n$.

2. Assuming that $P$ holds of $e_1$ and $e_2$, show that $P$ holds of $\text{plus}(e_1, e_2)$.

3. Assuming that $P$ holds of $e_1$ and $e_2$, show that $P$ holds of $\text{times}(e_1, e_2)$.

For example, we may prove that the equations

\[
\begin{align*}
\text{eval}(\text{num}(n)) &= n \\
\text{eval}(\text{plus}(e_1, e_2)) &= \text{eval}(e_1) + \text{eval}(e_2) \\
\text{eval}(\text{times}(e_1, e_2)) &= \text{eval}(e_1) \times \text{eval}(e_2)
\end{align*}
\]

determine a function $\text{eval}$ from the abstract syntax of expressions to numbers. That is, we may show by induction on the structure of $e$ that there is a unique $n$ such that $\text{eval}(e) = n$.

In practice we often (somewhat sloppily) define both the concrete and abstract syntax of a language by a single grammar. The idea is that the same grammar can be read as a (possibly ambiguous) specification of the concrete syntax, and as an (unambiguous) specification of the abstract syntax. Since the ambiguities in the concrete syntax can, presumably, be resolved using standard methods, we do not bother to specify them, but rather rely on the reader’s experience to fill in the details. It takes a little experience to get used to this approach, but it greatly simplifies the presentation of languages for which we are not concerned to design a “pretty” concrete syntax.

### 4.3 Parsing

The process of translation from concrete to abstract syntax is called parsing. If $C$ is the concrete syntax of a language (an inductively-defined set of strings), and $A$ is its abstract syntax (an inductively-defined set of ast’s), then a parser is a function $\text{parse} : C \to A$ mapping strings to ast’s. Since $C$ is inductively defined, it is natural to formulate the definition of $\text{parse}$ by induction on the rules defining the concrete syntax.

For example, consider the language of arithmetic expressions discussed in Chapter 3. Since we wish to define a function on the concrete syntax, it
should be clear from the discussion in Section 3.2 that we should work with the disambiguated grammar that makes explicit the precedence and associativity of addition and multiplication. With the rules of this grammar in mind, we may define simultaneously a family of parsing functions for each syntactic category by the following equations:

\[
\begin{align*}
\text{parse}_{\text{dig}}(0) &= 0 \\
&\vdots \\
\text{parse}_{\text{dig}}(9) &= 9 \\
\text{parse}_{\text{num}}(d) &= \text{num}(\text{parse}_{\text{dig}}(d)) \\
\text{parse}_{\text{num}}(n\ d) &= \text{num}(10 \times k + \text{parse}_{\text{dig}}(d)) \\
&\text{where } \text{parse}_{\text{num}} n = \text{num}(k) \\
\text{parse}_{\text{exp}}(t) &= \text{parse}_{\text{trm}}(t) \\
\text{parse}_{\text{exp}}(t+e) &= \text{plus}(\text{parse}_{\text{trm}}(t), \text{parse}_{\text{exp}}(e)) \\
\text{parse}_{\text{trm}}(f) &= \text{parse}_{\text{fct}}(f) \\
\text{parse}_{\text{trm}}(f \ast t) &= \text{times}(\text{parse}_{\text{fct}}(f), \text{parse}_{\text{trm}}(t)) \\
\text{parse}_{\text{fct}}(n) &= \text{parse}_{\text{num}}(n) \\
\text{parse}_{\text{fct}}(e) &= \text{parse}_{\text{exp}}(e)
\end{align*}
\]

It is a simple matter to prove by rule induction that these functions are all well-defined.

There is one remaining issue about this specification of the parsing function that requires further remedy. Look closely at the definition of the function \(\text{parse}_{\text{num}}\). It relies on a decomposition of the input string into two parts: a string, which is parsed as a number, followed a character, which is parsed as a digit. This is quite unrealistic, at least if we expect to process the input “on the fly”, since it requires us to work from the end of the input, rather than the beginning. To remedy this, we modify the grammatical clauses for numbers to be right recursive, rather than left recursive, as follows:

\[
\text{Numbers } n ::= d \mid d\ n
\]

This re-formulation ensures that we may process the input from left-to-right, one character at a time. It is a simple matter to re-define the parser to reflect this change in the grammar, and to check that it is well-defined.

An implementation of a parser that obeys this left-to-right discipline and is defined by induction on the rules of the grammar is called a recursive
descent parser. This is the method of choice for hand-coded parsers. Parser generators, which automatically create parsers from grammars, make use of a different technique that is more efficient, but much harder to implement by hand.

4.4 Exercises

1. Give a concrete and (first-order) abstract syntax for a language.

2. Write a parser for that language.
Chapter 5

Higher-Order Abstract Syntax

First-order abstract syntax captures the “deep structure” of an expression in the sense that it makes explicit the hierarchical relationships among the components of an expression. For example, in the case of arithmetic expressions the rules of abstract syntax make clear whether a given expression is an addition, one of whose arguments is a multiplication, or vice-versa.

Higher-order abstract syntax takes this process one step further to take account of the binding and scope of variables. The binding of a variable is the point at which it is introduced; its scope is its range of significance. The names of bound variables are not significant; this is captured by the notion of \(\alpha\)-conversion. Variables may be replaced by other terms by a process called substitution. Both of these notions are captured through the mechanism of higher-order terms.

5.1 Variables, Binding, and Scope

Variables are place-holders. They may be replaced by other terms (possibly involving other variables) to obtain a new term. For example, if we extend the syntax of arithmetic expressions to include variables, we obtain (integer) polynomials.

Variables \( x \) ::= any identifier

Expressions \( e \) ::= var\( (x) \) | num\( (n) \) | plus\( (e_1, e_2) \) | times\( (e_1, e_2) \)

We will not be specific about what counts as an identifier; typically we admit any string over some specified alphabet.

For example, \( \text{plus}(\text{times}(3, \text{var}(x)), 1) \) is an expression involving the variable \( \text{var}(x) \), written in abstract syntax notation. Using standard con-
crete syntax conventions, this expression would be written $3 \times x + 1$, relying on precedence to disambiguate. We may replace $\text{var}(x)$ by another expression, say $\text{plus}(2, y)$ to obtain the expression $\text{plus}(\text{times}(3, \text{plus}(2, y)), 1)$. Written in terms of concrete syntax, this is $3 \times (2 + y) + 1$; parentheses are required to disambiguate.

As far as first-order abstract syntax is concerned there is nothing else to be said. Variables are just a form of abstract syntax, no different from any other piece of abstract syntax in the language. But something interesting happens when we introduce operators that introduce, or bind, variables. For example, we might extend the language of arithmetic expressions to include a “let” statement that introduces a variable and gives it a definition. The syntax is extended as follows:

\[
\begin{align*}
\text{Variables} & \quad x ::= \text{any identifier} \\
\text{Expressions} & \quad e ::= \text{var}(x) | \text{num}(n) | \text{plus}(e_1, e_2) | \text{times}(e_1, e_2) | \text{let}(x, e_1, e_2)
\end{align*}
\]

The ast $\text{let}(x, e_1, e_2)$ might be written in concrete syntax as $\text{let } x \text{ be } e_1 \text{ in } e_2$ to make it a little easier to read.

What is important about the $\text{let}$ expression is that the variable $x$ is introduced, or bound, for use within its scope, the expression $e_2$. Ordinarily we would interpret this as defining $x$ to be the expression $e_1$ for use within $e_2$, but that interpretation is not especially important for the present purposes. All that matters is that the $\text{let}$ binds a variable within a specified scope.

All occurrences of a variable within the scope of a binding are treated as references to its binding site, the point at which the variable is bound. Whenever we see a variable, we consider it to be a reference to the nearest enclosing binding occurrence of that variable. Thus the occurrences of $x$ in the expression $\text{let } x \text{ be } 2 \text{ in } x + x$ refer to the binding introduced by the $\text{let}$. Similarly, in the expression

\[\text{let } x \text{ be } 2 \text{ in let } y \text{ be } 3 \text{ in } x + y,\]

the occurrence of $x$ refers to the outermost $\text{let}$, and the occurrence of $y$ refers to the innermost. Finally, in the expression

\[\text{let } x \text{ be } 2 \text{ in let } x \text{ be } 3 \text{ in } x + x,\]

both occurrences of $x$ in the addition refer to the innermost binding occurrence, since it is the nearest enclosing binding for the variable $x$. There is no way to refer to the outermost $\text{let}$; the inner binding for $x$ is said to shadow the outer.
Determining the binding occurrence corresponding to a use of a variable is called \textit{scope resolution}. The convention of treating a variable occurrence as a reference to the nearest enclosing binding of that variable is called \textit{lexical}, or \textit{static}, scope. The adjectives \textit{lexical} and \textit{static} indicate that scope resolution is determined by the \textit{program text}, rather than its execution behavior. In Chapter 11 we will consider an alternative, called \textit{dynamic} scope, in which bindings are left unresolved until execution time.

Not all variables in an expression refer to a binding. For example, consider the expression

\begin{verbatim}
let x be 2 in x + y;
\end{verbatim}

The variable \(x\) is bound by the \texttt{let}, according to the rules just given; it is said to be a \textit{bound} variable of the expression. On the other hand the variable \(y\) is not bound anywhere in this expression; it is said to be a \textit{free} variable of the expression. An expression containing free variables is said to be \textit{open}, whereas one that does not contain any free variables is said to be \textit{closed}. Note that expressions with no variables at all are closed, as are expressions all of whose variables are bound.

Since bound variables are used only to refer to their binding site, the choice of names of bound variable does not matter. Thus the expression

\begin{verbatim}
let x be 2 in x + x
\end{verbatim}

is not materially different from the expression

\begin{verbatim}
let y be 2 in y + y,
\end{verbatim}

since their binding structure is the same. That is, the variable \(x\) is used to refer to the outermost binding in the first expression, whereas the variable \(y\) is used for the same purpose in the second. Of course we typically choose mnemonic identifiers, but for the purposes of scope resolution the choice does not matter.

Two expressions that differ only in the choice of bound variable names are said (for historical reasons) to be \textit{\(\alpha\)-equivalent}. We write \(e_1 \equiv e_2\) to indicate that \(e_1\) and \(e_2\) are \(\alpha\)-equivalent. This is clearly an equivalence relation. Moreover, it is a \textit{congruence}, which means that if we replace any sub-expression by an \(\alpha\)-equivalent sub-expression, the result is \(\alpha\)-equivalent to the original.

The fundamental principle of higher-order abstract syntax is that we \textit{identify} \(\alpha\)-equivalent expressions. Put in other terms, higher-order abstract
syntax is the quotient of first-order abstract syntax by $\alpha$-equivalence. Elements of the quotient are equivalence classes of first-order ast’s under $\alpha$-equivalence.

The main consequence of working with higher-order abstract syntax is that we do not distinguish between ast’s that differ only in the names of their bound variables, because we are really working with equivalence classes. However, to write down an equivalence class requires that we choose a representative. That is, we must make an arbitrary choice of names for the bound variables. The beauty of higher-order abstract syntax is that we may always choose the bound variable name to be different from any given finite set of variable names. Such a choice is said to be fresh, or new, relative to that set of names. Thus, when we write \texttt{let } x \texttt{be } 3 \texttt{in } x + x, we implicitly choose $x$ to be a “new” variable, different from all others currently in use. This completely avoids the problem of shadowing, since we may always choose another representative that avoids re-use of variable names. Provided that we make such a choice, the variable name uniquely determines its binding occurrence.

We will often wish to replace all occurrences of a free variable $x$ in an expression $e'$ by another expression $e$. This process is called substitution. While substitution may, at first glance, seem like a simple process of replacement (changing all $x$’s into $e$’s), there is a subtle difficulty that must be avoided, called capture. Since the variable $x$ might occur within the scope of a binding occurrence of some variable $y$ within $e'$, if $y$ also occurs free in $e$, then simple replacement would incur capture, thereby changing the meaning of the expression.

For example, suppose that $e'$ is the expression \texttt{let } y \texttt{be } 7 \texttt{in } x + y, and let $e$ be the expression $y*2$. The result of simply replacing $x$ by $y*2$ yields the expression

\begin{equation*}
\texttt{let } y \texttt{be } 7 \texttt{in } y*2 + y.
\end{equation*}

The binding for $y$ in $e'$ is said to capture the free occurrence of $y$ in $e$. Capture is to be avoided is because it is inappropriately sensitive to the choice of bound variable names. If we had chosen a different representative for $e'$, say

\begin{equation*}
\texttt{let } z \texttt{be } 7 \texttt{in } x + z,
\end{equation*}

then the result of substitution would be

\begin{equation*}
\texttt{let } z \texttt{be } 7 \texttt{in } y*2 + z,
\end{equation*}

which is a different expression!
Thus, substitution is well-defined on $\alpha$-equivalence classes provided that we use renaming to avoid capture. This can always be achieved by simply choosing all bound variable names in the target of the the substitution to be different from those that occur free in the substituting expression. Since there are only finitely many such variables, this requirement can always be met by a suitable choice of representatives. In the above example we would choose the bound variable in $e'$ to be $z$, rather than $y$, since $y$ occurs free in $e$.

5.2 Higher-Order Terms

To make all of this more precise and systematic, we will introduce higher-order (more precisely, second-order) terms. Recall that a first-order term has the form $o(t_1, \ldots, t_n)$ where $o$ is an operator of arity $n$, and $t_1, \ldots, t_n$ are first-order terms. To make the step to higher-order abstract syntax we generalize the notion of the arity of an operator, revise the treatment of variables, and extend the class of terms with a new form, called an abstraction.

As in the first-order case each term-forming operator in higher-order abstract syntax is assigned an arity, but now an arity is a list of natural numbers, written $[n_1, \ldots, n_k]$. Such an operator, $o$, takes $k \geq 0$ higher-order terms $t_1, \ldots, t_k$ as arguments, with $t_i$ of arity $n_i$ for each $1 \leq i \leq k$.

What is a term of arity $n$? The term $o(t_1, \ldots, t_k)$, where $o$ and the $t_i$’s are as just described is a term of arity 0 (called a saturated term). A term of arity $n > 0$ is an abstraction of the form $x_1, \ldots, x_k.t$, where each $x_i$ is a variable of arity 0, and $t$ is a term of arity 0. Variables are a primitive notion in higher-order abstract syntax, rather than a designated set of identifiers as they are in the first-order case. The reason for this is that the concepts of binding and scope are handled by the general notion of higher-order abstract syntax, rather than on a per-language basis. Specifically, in an abstraction of the form $x_1, \ldots, x_k.t$, the variables $x_i$ are bound in the term $t$, and may be consistently renamed without changing the meaning of the term. The concept of $\alpha$-conversion is built in as a primitive notion of higher-order abstract syntax.

Higher-order abstract syntax generalizes first-order abstract syntax in the sense that a $k$-ary first-order operator is a higher-order operator of arity $[0, \ldots, 0]$, with $k$ occurrences of 0. For example, the operator $+$ has arity $[0, 0]$.

\footnote{Note that we are using the term “arity” in two senses here. The arity of an operator is a list of natural numbers; the arity of a term is a natural number.}
indicating that it takes two saturated terms as arguments. The operator \( \texttt{let} \) has arity \([0, 1]\), which indicates that it takes two arguments, one a saturated term, the other an abstraction of arity 1. The saturated higher-order term \( \texttt{let}(e_1, x.e_2) \) corresponds to the first-order term

\[
\texttt{let}(\texttt{var}(x), e_1, e_2),
\]

but also indicates that the variable \( x \) is bound within \( e_2 \).

It is important to observe that \( \alpha \)-equivalence preserves the structure of a term. That is, a term of the form \( o(t_1, \ldots, t_n) \) is \( \alpha \)-equivalent only to terms of the same form. This means that structural induction can be extended from first-order to higher-order abstract syntax without special mention. Note, however, that a proof by structural induction on higher-order abstract syntax must respect \( \alpha \)-conversion in the sense that the validity of the proof cannot depend upon the exact choice of bound variable names.

### 5.3 Renaming and Substitution

In this section we will give a more rigorous account of variable renaming and substitution for higher-order terms. First, we will define variable renaming as a function on raw higher-order terms. Second, we will use this to define \( \alpha \)-equivalence. Third, we will define capture-avoiding substitution on \( \alpha \)-equivalence classes of higher-order terms.

The set of free variables, \( \text{FV}(t) \), in a raw higher-order term \( t \) is inductively defined by the following equations:

\[
\begin{align*}
\text{FV}(x) &= \{ x \} \\
\text{FV}(o(t_1, \ldots, t_n)) &= \bigcup_{1 \leq i \leq n} \text{FV}(t_i) \\
\text{FV}(x_1, \ldots, x_n.t) &= \text{FV}(t) \setminus \{ x_1, \ldots, x_n \}
\end{align*}
\]

Let \( \bar{x} = x_1, \ldots, x_k \) and \( \bar{z} = z_1, \ldots, z_k \). The simultaneous renaming of variables \( \bar{x} \) to \( \bar{z} \) in \( t \), written \( \{ \bar{z}/\bar{x} \}t \), is defined by induction on the structure of \( t \) as follows:

\[
\begin{align*}
\{ \bar{z}/\bar{x} \}y &= \begin{cases} 
z_i & \text{if } y = x_i \\
y & \text{otherwise}
\end{cases} \\
\{ \bar{z}/\bar{x} \}o(t_1, \ldots, t_n) &= o(\{ \bar{z}/\bar{x} \}t_1, \ldots, \{ \bar{z}/\bar{x} \}t_n) \\
\{ \bar{z}/\bar{x} \}(y_1, \ldots, y_n.t) &= y_1, \ldots, y_n, \{ \bar{z}/\bar{x} \}t
\end{align*}
\]

The last clause is defined only in case that no \( y_i \) \((1 \leq i \leq n)\) is among the \( \bar{z} \) (so as to preclude capture) or among the \( \bar{x} \) (so as to avoid confusion).
the first clause $y$ cannot be among the $\bar{z}$, by the assumption that no variable in $\bar{z}$ is free in the target term.

We may now define $\alpha$-equivalence as the least congruence containing all instances of the axiom

$$x_1, \ldots, x_n.t \equiv y_1, \ldots, y_n.\{y_1, \ldots, y_n/x_1, \ldots, x_n\}t$$

provided that no $y_i$ occurs free in $t$ (so as to avoid capture). By the “least congruence”, we mean that $\equiv$ is the least equivalence relation closed under the following congruence principles:

$$\frac{t_1 \equiv t'_1 \cdots t_n \equiv t'_n}{o(t_1, \ldots, t_n) \equiv o(t'_1, \ldots, t'_n)} \quad \frac{t \equiv t'}{x_1, \ldots, x_n.t \equiv x_1, \ldots, x_n.t'}$$

Thus we may replace any sub-term of a term by an $\alpha$-equivalent one, and obtain a term that is $\alpha$-equivalent to the one we started with. Moreover, $\alpha$-equivalence preserves the structure of a higher-order term. In particular, an abstraction of degree $n$ is $\alpha$-equivalent only to other abstractions of the same degree.

Finally, we may define simultaneous substitution of a sequence $\bar{t} = t_1, \ldots, t_k$ of terms for a sequence $\bar{x} = x_1, \ldots, x_k$ in a term $t$, written $\{\bar{t}/\bar{x}\}t$. We will define substitution only up to $\alpha$-equivalence, relying on implicit renaming of bound variables to ensure that capture is avoided. Simultaneous substitution is defined by induction on the structure of $t$ as follows:

$$\{\bar{t}/\bar{x}\}y = \begin{cases} t_i & \text{if } y = x_i \\ y & \text{otherwise} \end{cases}$$

$$\{\bar{t}/\bar{x}\}o(t'_1, \ldots, t'_n) = o(\{\bar{t}/\bar{x}\}t'_1, \ldots, \{\bar{t}/\bar{x}\}t'_n)$$

$$\{\bar{t}/\bar{x}\}(y_1, \ldots, y_n; t') = y_1, \ldots, y_n; \{\bar{t}/\bar{x}\}t'$$

The last clause is defined only if no $y_i$ occurs free in $\bar{t}$, and the $y_i$'s are distinct from all $x_i$'s. This restriction may always be met by a suitable choice of bound variable names. However, this means that the result of substitution is only defined up to $\alpha$-equivalence.

5.4 Exercises

1. Give the higher-order abstract syntax for an interesting language.

2. Implement named and name-free representations of hoas.
3. Develop the theory of deBruijn indices, including substitution.

4. Prove some properties of substitution and renaming.
Chapter 6

Static Semantics

The static semantics of a language isolates a subset of the abstract syntax of the language that is deemed well-formed. In simple cases every ast is well-formed, but in most cases additional constraints must be imposed. Chief among these are type constraints that ensure that the constructors of the language are combined in a sensible manner. An inductive definition of the type constraints governing a language is called a (static) type system, or static semantics.

6.1 Well-Formed Arithmetic Expressions

Since it is difficult to give a fully general account of static semantics, we will instead illustrate the main ideas by example. We will give a formal definition of well-formedness of arithmetic expressions that ensures that there are no unbound variables in a complete expression. Of course we could simply define $e$ to be well-formed in this sense iff $FV(e) = \emptyset$, but we will instead give a direct axiomatization of well-formedness.

A well-formedness judgement, or well-formedness assertion, has the form $\Gamma \vdash e\,\text{ok}$, where $\Gamma$ is a finite set of variables. The intended meaning of this assertion is that $e$ is an arithmetic expression all of whose free variables are in the set $\Gamma$. In particular, if $\emptyset \vdash e\,\text{ok}$ (often abbreviated to $e\,\text{ok}$), then $e$ has no unbound (free) variables, and is therefore suitable for evaluation to an integer.
Formally, well-formedness is inductively defined by the following rules:

\[
\begin{align*}
(x \in \Gamma) & \quad \Gamma \vdash \text{var}(x) \text{ ok} \\
(n \geq 0) & \quad \Gamma \vdash \text{num}(n) \text{ ok} \\
\Gamma \vdash e_1 \text{ ok} & \quad \Gamma \vdash e_2 \text{ ok} \quad \Gamma \vdash \text{plus}(e_1, e_2) \text{ ok} \\
\Gamma \vdash e_1 \text{ ok} & \quad \Gamma \vdash e_2 \text{ ok} \quad \Gamma \vdash \text{times}(e_1, e_2) \text{ ok} \\
\Gamma \vdash e_1 \text{ ok} & \quad \Gamma \cup \{ x \} \vdash e_2 \text{ ok} \quad (x \notin \Gamma) \\
\Gamma \vdash \text{let}(e_1, x.e_2) \text{ ok}
\end{align*}
\]

Frequently well-formedness rules are stated using concrete syntax for the sake of readability, but it is understood that we are really referring to the abstract syntax of the language.

There are a few things to notice about these rules. First, a variable is well-formed iff it is in \( \Gamma \). This is consistent with the informal reading of the judgement. Second, a \texttt{let} expression adds a new variable to \( \Gamma \) for use within \( e_2 \). The "newness" of the variable is captured by the requirement that \( x \notin \Gamma \). By the conventions of higher-order abstract syntax, this condition can always be met by a suitable renaming prior to application of the rule. Third, the rules are syntax-directed in the sense that there is one rule for each form of expression; as we will see later, this is not necessarily the case.

### 6.2 Exercises

1. Show that \( \Gamma \vdash e \text{ ok} \iff \text{FV}(e) \subseteq \Gamma \). From left to right, proceed by rule induction. From right to left, proceed by induction on the structure of \( e \).

2. Integers and floats. Add types to variable declarations.
Chapter 7

Dynamic Semantics

The *dynamic semantics* of a language specifies how programs are to be executed. There are two popular methods for specifying dynamic semantics. One method, called *structured operational semantics (SOS)*, or *transition semantics*, presents the dynamic semantics of a language as a transition system specifying the step-by-step execution of programs. Another, called *evaluation semantics*, or *ES*, presents the dynamic semantics as a binary relation specifying the result of a complete execution of a program.

7.1 Structured Operational Semantics

A structured operational semantics for a language consists of a transition system whose states are programs and whose transition relation is defined by induction over the structure of programs. We will illustrate SOS for the simple language of arithmetic expressions (including *let* expressions) presented in Chapter 5.

The set of states is the set of well-formed arithmetic expressions:

\[ S = \{ e \mid \exists \Gamma \Gamma \vdash e \text{ ok} \}. \]

The set of initial states, \( I \subseteq S \), is the set of closed expressions:

\[ I = \{ e \mid \emptyset \vdash e \text{ ok} \}. \]

The set of final states, \( F \subseteq S \), is just the set of numerals for natural numbers:

\[ F = \{ \text{num}(n) \mid n \geq 0 \}. \]
42 Dynamic Semantics

The transition relation \( \rightarrow \subseteq S \times S \) is inductively defined by the following rules:

\[
\begin{align*}
(p = m + n) & \quad \Rightarrow \quad \text{plus}(\text{num}(m), \text{num}(n)) \rightarrow \text{num}(p) \\
(p = m \times n) & \quad \Rightarrow \quad \text{times}(\text{num}(m), \text{num}(n)) \rightarrow \text{num}(p) \\
\text{let}(\text{num}(n), x.e) & \quad \Rightarrow \quad \{\text{num}(n)/x\}e
\end{align*}
\]

\[
\begin{align*}
e_1 \rightarrow e'_1 & \quad \Rightarrow \quad \text{plus}(e_1, e_2) \rightarrow \text{plus}(e'_1, e_2) \\
e_2 \rightarrow e'_2 & \quad \Rightarrow \quad \text{plus}(\text{num}(n_1), e_2) \rightarrow \text{plus}(\text{num}(n_1), e'_2) \\
e_1 \rightarrow e'_1 & \quad \Rightarrow \quad \text{times}(e_1, e_2) \rightarrow \text{times}(e'_1, e_2) \\
e_2 \rightarrow e'_2 & \quad \Rightarrow \quad \text{times}(\text{num}(n_1), e_2) \rightarrow \text{times}(\text{num}(n_1), e'_2) \\
\text{let}(e_1, x.e_2) & \quad \Rightarrow \quad \text{let}(e'_1, x.e_2)
\end{align*}
\]

Observe that variables are stuck states, but they are not final. Free variables have no binding, and hence cannot be evaluated to a number.

To enhance readability we often write SOS rules using concrete syntax, as follows:

\[
\begin{align*}
(p = m + n) & \quad \Rightarrow \quad m + n \rightarrow p \\
(p = m \times n) & \quad \Rightarrow \quad m \times n \rightarrow p \\
\text{let } x \text{ be } n \text{ in } e & \quad \Rightarrow \quad \{n/x\}e
\end{align*}
\]

\[
\begin{align*}
e_1 \rightarrow e'_1 & \quad \Rightarrow \quad e_1 + e_2 \rightarrow e'_1 + e_2 \\
e_2 \rightarrow e'_2 & \quad \Rightarrow \quad n_1 + e_2 \rightarrow n_1 + e'_2 \\
e_1 \rightarrow e'_1 & \quad \Rightarrow \quad e_1 \times e_2 \rightarrow e'_1 \times e_2 \\
e_2 \rightarrow e'_2 & \quad \Rightarrow \quad n_1 \times e_2 \rightarrow n_1 \times e'_2 \\
\text{let } x \text{ be } e_1 \text{ in } e_2 & \quad \Rightarrow \quad \text{let } x \text{ be } e'_1 \text{ in } e_2
\end{align*}
\]

The intended meaning is the same, the only difference is the presentation.

The first three rules defining the transition relation are sometimes called *instructions*, since they correspond to the primitive execution steps of the machine. Addition and multiplication are evaluated by adding and multiplying; let bindings are evaluated by substituting the definition for the
variable in the body. In all three cases the principal arguments of the constructor are required to be numbers. Both arguments of an addition or multiplication are principal, but only the binding of the variable in a let expression is principal. We say that these primitives are evaluated by value, because the instructions apply only when the principal arguments have been fully evaluated.

What if the principal arguments have not (yet) been fully evaluated? Then we must evaluate them! In the case of arithmetic expressions we arbitrarily choose a left-to-right evaluation order. First we evaluate the first argument, then the second. Once both have been evaluated, the instruction rule applies. In the case of let expressions we first evaluate the binding, after which the instruction step applies. Note that evaluation of an argument can take multiple steps. The transition relation is defined so that one step of evaluation is made at a time, reconstructing the entire expression as necessary.

For example, consider the following evaluation sequence:

\[
\begin{align*}
\text{let } x & \text{ be } 1 + 2 \text{ in } (x + 3) \times 4 \\
& \quad \rightarrow \text{ let } x \text{ be } 3 \text{ in } (x + 3) \times 4 \\
& \quad \rightarrow (3 + 3) \times 4 \\
& \quad \rightarrow 6 \times 4 \\
& \quad \rightarrow 24
\end{align*}
\]

Each step is justified by a rule defining the transition relation. Instruction rules are axioms, and hence have no premises, but all other rules are justified by a subsidiary deduction of another transition. For example, the first transition is justified by a subsidiary deduction of \(1 + 2 \rightarrow 3\), which is justified by the first instruction rule defining the transition relation. Each of the subsequent steps is justified similarly.

Since the transition relation in SOS is inductively defined, we may reason about it using rule induction. Specifically, to show that \(P(e, e')\) holds whenever \(e \rightarrow e'\), it is sufficient to show that \(P\) is closed under the rules defining the transition relation. For example, it is a simple matter to show by rule induction that the transition relation for evaluation of arithmetic expressions is deterministic: if \(e \rightarrow e'\) and \(e \rightarrow e''\), then \(e' = e''\). This may be proved by simultaneous rule induction over the definition of the transition relation.
7.2 Evaluation Semantics

Another method for defining the dynamic semantics of a language, called evaluation semantics, consists of a direct inductive definition of the evaluation relation, written \( e \Downarrow v \), specifying the value, \( v \), of an expression, \( e \). More precisely, an evaluation semantics consists of a set \( E \) of evaluatable expressions, a set \( V \) of values, and a binary relation \( \Downarrow \subseteq E \times V \). In contrast to SOS the set of values need not be a subset of the set of expressions; we are free to choose values as we like. However, it is often advantageous to choose \( V \subseteq E \).

We will give an evaluation semantics for arithmetic expressions as an example. The set of evaluatable expressions is defined by

\[
E = \{ e \mid \emptyset \vdash e \in \text{ok} \}.
\]

The set of values is defined by

\[
V = \{ \text{num}(n) \mid n \geq 0 \}.
\]

The evaluation relation for arithmetic expressions is inductively defined by the following rules:

\[
\begin{align*}
\text{num}(n) & \Downarrow \text{num}(n) \\
\text{num}(n_1) & \Downarrow \text{num}(n_2) \quad (n = n_1 + n_2) \\
\text{num}(n_1) & \Downarrow \text{num}(n_2) \quad (n = n_1 \times n_2) \\
\text{let}(e_1, x: e_2) & \Downarrow v
\end{align*}
\]

Notice that the rules for evaluation semantics are not syntax-directed! The value of a let expression is determined by the value of its binding, and the value of the corresponding substitution instance of its body. Since the substitution instance is not a sub-expression of the let, the rules are not syntax-directed.

Since the evaluation relation is inductively defined, it has associated with it a principle of proof by rule induction. Specifically, to show that \( P(e, v) \) holds for some property \( P \subseteq E \times V \), it is enough to show that \( P \) is closed under the rules given above. Specifically,
1. Show that $P(\text{num}(n), \text{num}(n))$.

2. Assume that $P(e_1, \text{num}(n_1))$ and $P(e_2, \text{num}(n_2))$. Show that $P(\text{plus}(e_1, e_2), \text{num}(n_1 + n_2))$ and that $P(\text{times}(e_1, e_2), \text{num}(n_1 \times n_2))$.

3. Assume that $P(e_1, v_1)$ and $P(\{v_1/x\}e_2, v_2)$. Show that $P(\text{let}(e_1, x, e_2), v_2)$.

### 7.3 Relating Transition and Evaluation Semantics

We have given two different forms of dynamic semantics for the same language. It is natural to ask whether they are equivalent, but to do so first requires that we consider carefully what we mean by equivalence. The transition semantics describes a step-by-step process of execution, whereas the evaluation semantics suppresses the intermediate states, focusing attention on the initial and final states alone. This suggests that the appropriate correspondence is between complete execution sequences in the transition semantics and the evaluation relation in the evaluation semantics.

**Theorem 7.1**

For all well-formed, closed arithmetic expressions $e$ and all natural numbers $n$, $e \mapsto^1 \text{num}(n)$ iff $e \Downarrow \text{num}(n)$.

How might we prove such a theorem? We will consider each direction separately. We consider the easier case first.

**Lemma 7.2**

If $e \Downarrow \text{num}(n)$, then $e \mapsto^1 \text{num}(n)$.

**Proof:** By induction on the definition of the evaluation relation. For example, suppose that $\text{plus}(e_1, e_2) \Downarrow \text{num}(n)$ by the rule for evaluating additions. By induction we know that $e_1 \mapsto^1 \text{num}(n_1)$ and $e_2 \mapsto^1 \text{num}(n_2)$. We reason as follows:

\[
\begin{align*}
\text{plus}(e_1, e_2) & \mapsto^* \text{plus}(\text{num}(n_1), e_2) \\
& \mapsto^* \text{plus}(\text{num}(n_1), \text{num}(n_2)) \\
& \mapsto \text{num}(n_1 + n_2)
\end{align*}
\]

Therefore $\text{plus}(e_1, e_2) \mapsto^1 \text{num}(n_1 + n_2)$, as required. The other cases are handled similarly.

What about the converse? Recall from Chapter 2 that the complete evaluation relation, $\mapsto^1$, is the restriction of the multi-step evaluation relation,
\[ \rightarrow^* \] to initial and final states (here closed expressions and numerals). Recall also that multi-step evaluation is inductively defined by two rules, reflexivity and closure under head expansion. By definition \( \text{num}(n) \downarrow \text{num}(n) \), so it suffices to show closure under head expansion.

**Lemma 7.3**

*If \( e \rightarrow e' \) and \( e' \downarrow \text{num}(n) \), then \( e \downarrow \text{num}(n) \).*

**Proof:** By induction on the definition of the transition relation. For example, suppose that \( \text{plus}(e_1, e_2) \rightarrow \text{plus}(e'_1, e_2) \), where \( e_1 \rightarrow e'_1 \). Suppose further that \( \text{plus}(e'_1, e_2) \downarrow \text{num}(n) \), so that \( e'_1 \downarrow \text{num}(n_1) \), and \( e_2 \downarrow \text{num}(n_2) \) and \( n = n_1 + n_2 \). By induction \( e_1 \downarrow \text{num}(n_1) \), and hence \( \text{plus}(e_1, e_2) \downarrow n \), as required.

### 7.4 Exercises

1. Prove that if \( e \rightarrow e_1 \) and \( e \rightarrow e_2 \), then \( e_1 \equiv e_2 \).

2. Prove that if \( e \in I \) and \( e \rightarrow e' \), then \( e' \in I \). Proceed by induction on the definition of the transition relation.

3. Prove that if \( e \in I \setminus F \), then there exists \( e' \) such that \( e \rightarrow e' \). Proceed by induction on the rules defining well-formedness given in Chapter 6.

4. Prove that if \( e \downarrow v_1 \) and \( e \downarrow v_2 \), then \( v_1 \equiv v_2 \).

5. Complete the proof of equivalence of evaluation and transition semantics.
Part III

A Functional Language
Chapter 8

MinML, A Minimal Functional Language

The language MinML will serve as the jumping-off point for much of our study of programming language concepts. MinML is a call-by-value, effect-free language with integers, booleans, and a (partial) function type.

8.1 Syntax

8.1.1 Concrete Syntax

The concrete syntax of MinML is divided into three main syntactic categories, types, expressions, and programs. Their definition involves some auxiliary syntactic categories, namely variables, numbers, and operators.

These categories are defined by the following grammar:

Variables \( x ::= \ldots \)

Numbers \( n ::= \ldots \)

Operators \( o ::= + \mid * \mid - \mid = \mid < \)

Types \( \tau ::= \text{int} \mid \text{bool} \mid \tau_1 \rightarrow \tau_2 \)

Expr's \( e ::= x \mid n \mid o(e_1, \ldots, e_n) \mid \text{true} \mid \text{false} \)

\( \mid \text{if} e_1 \text{then} e_2 \text{else} e_3 \text{fi} \)

\( \mid \text{fun} f(x: \tau_1) : \tau_2 \text{is} e_4 \text{end} \)

\( \mid \text{apply}(e_1, e_2) \)

Programs \( p ::= e \)

We do not specify precisely the sets of numbers or variables. We generally write \( x, y, \text{etc.} \) for variables, and we write numbers in ordinary decimal
notation. As usual we do not bother to specify such niceties as parenthe-
sization or the use of infix syntax for binary operators, both of which would
be necessary in practice.

8.1.2 Abstract Syntax

The intended first-order abstract syntax of MinML may be read off from its
concrete syntax by interpreting the preceding grammar as a specification
of a set of first-order terms, rather than as a set of strings. To specify the
higher-order abstract syntax we need only state the binding and scope con-
ventions. Specifically, in the function expression \texttt{fun f (x:}\tau_1):\tau_2 \texttt{is e end}
the variables \(f\) and \(x\) are both bound within the \textit{body} of the function, \(e\).

The first- and higher-order abstract syntax of MinML expressions may
be specified succinctly by defining the set of type- and expression-forming
operators and their arities.

\begin{center}
\begin{tabular}{|c|c|}
\hline
\textit{Operator} & \textit{Arity} \\
\hline
\texttt{int} & \[\] \\
\texttt{bool} & \[0, 0\] \\
\rightarrow & \[0, 0\] \\
\hline
\texttt{n} & \[0, \ldots, 0\] \\
\texttt{o} & \[0, \ldots, 0\] \\
\hline
\texttt{fun} & \[0, 0, 2\] \\
\texttt{apply} & \[0, 0\] \\
\texttt{true} & \[0, 0\] \\
\texttt{false} & \[0, 0\] \\
\texttt{if} & \[0, 0, 0\] \\
\hline
\end{tabular}
\end{center}

In the above specification \(o\) is an \(n\)-argument primitive operator.

Using higher-order terms the function \texttt{fun f (x:}\tau_1):\tau_2 \texttt{is e end}\ would
be written \texttt{fun(}\tau_1, \tau_2, f, x, e\texttt{)}. This makes explicit that \(f\) and \(x\) are bound
within \(e\), and that the argument and result types of the function are part of
the syntax.

8.2 Static Semantics

Not all expressions in MinML are well-formed. For example, the expression
\texttt{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 $\text{FV}(e) \subseteq \text{dom}(\Gamma)$. 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 $x \not\in \text{dom}(\Gamma)$, is defined by the following equation

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

The typing relation is inductively defined by the following rules:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \quad (8.1)$$

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

$$\frac{}{\Gamma \vdash n : \text{int}} \quad (8.2)$$

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}} \quad (8.3)$$

$$\frac{}{\Gamma \vdash \text{false} : \text{bool}} \quad (8.4)$$

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

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash +(e_1, e_2) : \text{int}} \quad (8.5)$$
8.3 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.
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
\[ \Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \text{ fi} : \tau, \]
it suffices to show that \( \Gamma \vdash e : \text{bool} \), \( \Gamma \vdash e_1 : \tau \), and \( \Gamma \vdash e_2 : \tau \), because
of Rule 8.10. Since there is exactly one typing rule for each expression, the
typing rules also express necessary conditions for typing. For example, if
\( \Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \text{ fi} : \tau \), then \( \Gamma \vdash e : \text{bool} \), \( \Gamma \vdash e_1 : \tau \) 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 8.1 (Inversion)**

1. If \( \Gamma \vdash x : \tau \), then \( \Gamma(x) = \tau \).
2. If \( \Gamma \vdash n : \tau \), then \( \tau = \text{int} \).
3. If \( \Gamma \vdash \text{true} : \tau \), then \( \tau = \text{bool} \), and similarly for \( \text{false} \).
4. If \( \Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \text{ fi} : \tau \), then \( \Gamma \vdash e : \text{bool} \), \( \Gamma \vdash e_1 : \tau \) and
   \( \Gamma \vdash e_2 : \tau \).
5. If \( \Gamma \vdash \text{fun } f(x:\tau_1) : \tau_2 \text{ ise } \text{ end} : \tau \), then \( \Gamma[f:\tau_1 \rightarrow \tau_2][x:\tau_1] \vdash e : \tau_2 \)
   and \( \tau = \tau_1 \rightarrow \tau_2 \).
6. If \( \Gamma \vdash \text{apply}(e_1, e_2) : \tau \), then there exists \( \tau_2 \) such that \( \Gamma \vdash e_1 : \tau_2 \rightarrow \tau \)
   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.

**Lemma 8.2**

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 $\Gamma' \vdash e_1 : \tau_2 \rightarrow \tau$ and if $\Gamma' \supseteq \Gamma$, then $\Gamma' \vdash e_2 : \tau_2$. Consequently, if $\Gamma' \supseteq \Gamma$, then $\Gamma' \vdash \text{apply} (e_1, e_2) : \tau$, 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.

(Rule 8.1) 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.

(Rule 8.11) We have that $e' = \text{fun } f (y : \tau_1) : \tau_2 \text{ is } e_2 \text{ end}$ and $\tau' = \tau_1 \rightarrow \tau_2$. We may assume that $f$ and $y$ are chosen so that

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

By definition of substitution,

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

Applying the inductive hypothesis to the premise of Rule 8.11,

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

it follows that

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

Hence

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

as required.
8.4 Dynamic Semantics

The dynamic semantics of MinML is given by 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 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 \( v \), which is either 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.\(^1\)

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:

\[
+ (m, n) \mapsto m + n \tag{8.13}
\]

The other primitive operations are defined similarly.

The primitive instructions for conditional expressions are as follows:

\[
\text{if true then } e_1 \text{ else } e_2 \text{ fi } \mapsto e_1 \tag{8.14}
\]
\[
\text{if false then } e_1 \text{ else } e_2 \text{ fi } \mapsto e_2 \tag{8.15}
\]

The primitive instruction for application is as follows:

\[
(\text{apply}(v, v_1) = \{v, v_1/f, x\}e) \tag{8.16}
\]

To apply the function \( v = \text{fun } f (x: \tau_1): \tau_2 \text{ is } e \text{ end} \) to an argument \( v_1 \) (which must be a value!), we substitute the function itself, \( v \), for \( f \), and the

\(^1\)Some languages are, by contrast, non-deterministic, notably those involving concurrent interaction. We’ll come back to those later.
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:

\[
\begin{align*}
\text{e}_1 & \rightarrow e'_1 \\
+ (e_1, e_2) & \rightarrow + (e'_1, e_2)
\end{align*}
\] (8.17)

\[
\begin{align*}
\text{e}_2 & \rightarrow e'_2 \\
+ (v_1, e_2) & \rightarrow + (v_1, e'_2)
\end{align*}
\] (8.18)

The other primitive operations are handled similarly.

For the conditional, we evaluate the test expression.

\[
\begin{align*}
e & \rightarrow e' \\
\text{if } e \text{ then } e_1 \text{ else } e_2 \text{ fi} & \rightarrow \text{if } e' \text{ then } e_1 \text{ else } e_2 \text{ fi}
\end{align*}
\] (8.19)

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

\[
\begin{align*}
\text{e}_1 & \rightarrow e'_1 \\
\text{apply} (e_1, e_2) & \rightarrow \text{apply} (e'_1, e_2)
\end{align*}
\] (8.20)

\[
\begin{align*}
\text{e}_2 & \rightarrow e'_2 \\
\text{apply} (v_1, e_2) & \rightarrow \text{apply} (v_1, e'_2)
\end{align*}
\] (8.21)

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

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

\[
\begin{align*}
e & \rightarrow^* e \\
e \rightarrow e' & \rightarrow^* e'' & \rightarrow^* e''
\end{align*}
\] (8.22)

(8.23)

In words: \( e \rightarrow^* e' \) iff performing zero or more steps of evaluation starting from the expression \( e \) yields the expression \( e' \). The relation \( \rightarrow^* \) is sometimes called the Kleene closure, or reflexive-transitive closure, of the relation \( \rightarrow \).
8.5 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 \( v \not\rightarrow e \). 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 8.3**

For every closed expression \( e \), there exists at most one \( e' \) such that \( e \rightarrow e' \). In other words, the relation \( \rightarrow \) 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 8.4**

For every closed expression \( e \), there exists at most one value \( v \) such that \( e \rightarrow^* v \).

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

8.6 Exercises

1. 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.

2. 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.
Chapter 9

Type Safety for MinML

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.

9.1 Defining Type Safety

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 \( e \rightarrow e' \), 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 \( e \rightarrow e' \).
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 \( \text{apply}(3, 4) \) — 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 quotient, that are undefined for some arguments. We’ll come back to this shortly.

### 9.2 Type Safety of MinML

**Theorem 9.1 (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.

**(Rule 8.13)** Here \( e = +(m, n) \), \( \tau = \text{int} \), and \( e' = m + n \). Clearly \( e' : \text{int} \), as required. The other primitive operations are handled similarly.

**(Rule 8.14)** Here \( e = \text{if true then } e_1 \text{ else } e_2 \text{ fi} \) and \( e' = e_1 \). Since \( e : \tau \), by inversion \( e_1 : \tau \), as required.

**(Rule 8.15)** Here \( e = \text{if false then } e_1 \text{ else } e_2 \text{ fi} \) and \( e' = e_2 \). Since \( e : \tau \), by inversion \( e_2 : \tau \), as required.

**(Rule 8.16)** Here \( e = \text{apply}(v_1, v_2) \), where \( v_1 = \text{fun } f (x : \tau_2) : \tau \text{ is } e_2 \text{ end} \), and \( e' = \{v_1, v_2/f, x\}e_2 \). By inversion applied to \( e \) we have \( v_1 : \tau_2 \rightarrow \tau \) and
9.2 Type Safety of MinML

$v_2 : \tau_2$. 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.

(Rule 8.17) Here $e = +(e_1, e_2)$, $e' = +(e'_1, e_2)$, and $e_1 \mapsto e'_1$. By inversion $e_1 : \text{int}$, so that by induction $e'_1 : \text{int}$, and hence $e' : \text{int}$, as required.

(Rule 8.18) Here $e = +(v_1, e_2)$, $e' = +(v_1, e'_2)$, and $e_2 \mapsto e'_2$. By inversion $e_2 : \text{int}$, so that by induction $e'_2 : \text{int}$, and hence $e' : \text{int}$, as required.

The other primitive operations are handled similarly.

(Rule 8.19) Here $e = \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ fi}$ and $e' = \text{if } e'_1 \text{ then } e_2 \text{ else } e_3 \text{ fi}$. By inversion we have that $e_1 : \text{bool}$, $e_2 : \tau$ and $e_3 : \tau$. By inductive hypothesis $e'_1 : \text{bool}$, and hence $e' : \tau$.

(Rule 8.20) Here $e = \text{apply}(e_1, e_2)$ and $e' = \text{apply}(e'_1, e_2)$. 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 $e' : \tau$.

(Rule 8.21) Here $e = \text{apply}(v_1, e_2)$ and $e' = \text{apply}(v_1, e'_2)$. 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 9.2 (Canonical Forms)
Suppose that $v : \tau$ is a closed, well-formed value.

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

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

Exercise 9.3
Give a proof of the canonical forms lemma.
Theorem 9.4 (Progress)
If \( e : \tau \), then either \( e \) is a value, or there exists \( e' \) such that \( e \rightarrow e' \).

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

(Rule 8.1) Cannot occur, since \( e \) is closed.

(Rules 8.2, 8.3, 8.4, 8.11) In each case \( e \) is a value, which completes the proof.

(Rule 8.5) Here \( e = +(e_1, e_2) \) and \( \tau = \text{int} \), with \( e_1 : \text{int} \) and \( e_2 : \text{int} \). 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 \( e' = +(e_1', e_2) \). 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 \( e' = +(v_1, e_2') \), as desired.

(Rule 8.10) Here \( e = \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ fi} \), with \( e_1 : \text{bool} \), \( e_2 : \tau \), 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 \( e_1 = \text{true} \) or \( e_1 = \text{false} \). 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 \( e' = \text{if } e_1' \text{ then } e_2 \text{ else } e_3 \text{ fi} \), by Rule 8.19.

(Rule 8.12) Here \( e = \text{apply } (e_1, e_2) \), 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 \( e \mapsto \text{apply } (e_1', e_2) \) by Rule 8.20, 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 \( e' = \text{apply } (e_1, e_2') \), as required. Finally, if both \( e_1 \) and \( e_2 \) are values, then by the Canonical Forms Lemma, \( e_1 = \text{fun } f (x : \tau_2) : \tau \text{ is } e'' \text{ end} \), and \( e \mapsto e' \), where \( e' = \{e_1, e_2/f, x\}e'' \), by Rule 8.16.

Theorem 9.5 (Safety)
If \( e \) is closed and well-typed, then evaluation of \( e \) can only terminate with a value of the same type. In particular, evaluation cannot “get stuck” in an
9.3 Run-Time Errors and Safety

Stuck states correspond to ill-defined programs that attempt to, say, treat an integer as a pointer to a function, or that move a pointer beyond the limits of a region of memory. In an unsafe language there are no stuck states — every program will do something — but it may be impossible to predict how the program will behave in certain situations. It may “dump core”, or it may allow the programmer to access private data, or it may compute a “random” result.

The best-known example of an unsafe language is C. It’s lack of safety manifests itself in numerous ways, notably in that computer viruses nearly always rely on overrunning a region of memory as a critical step in an attack. Another symptom is lack of portability: an unsafe program may execute sensibly on one platform, but behave entirely differently on another. To avoid this behavior, standards bodies have defined portable subsets of C that are guaranteed to have predictable behavior on all platforms. But there is no good way to ensure that a programmer, whether through malice or neglect, adheres to this subset.\(^1\)

Safe languages, in contrast, avoid ill-defined states entirely, by imposing typing restrictions that ensure that well-typed programs have well-defined behavior. MinML is a good example of a safe language. It is inherently portable, because its dynamic semantics is specified in an implementation-independent manner, and because its static semantics ensures that well-typed programs never “get stuck”. Stated contrapositively, the type safety theorem for MinML assures us that stuck states are ill-typed.

But suppose that we add to MinML a primitive operation, such as quotient, that is undefined for certain arguments. An expression such as 3/0 would most-assuredly be “stuck”, yet would be well-typed, at least if we take the natural typing rule for it:

\[
\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \\
\Gamma \vdash e_1 / e_2 : \text{int}
\]

What are we to make of this? Is the extension of MinML with quotient unsafe?

To recover safety, we have two options:

\(^1\)It should be easy to convince yourself that it is undecidable whether a given C program can reach an implementation-dependent state.
Type Safety for MinML

1. Enhance the type system so that no well-typed program can ever divide by zero.

2. Modify the dynamic semantics so that division by zero is not “stuck”, but rather incurs a run-time error.

The first option amounts to requiring that the type checker prove that the denominator of a quotient is non-zero in order for it to be well-typed. But this means that the type system would, in general, be undecidable, for we can easily arrange for the denominator of some expression to be non-zero exactly when some Turing machine halts on blank tape. It is the subject of ongoing research to devise conservative type checkers that are sufficiently expressive to be useful in practice, but we shall not pursue this approach any further here.

The second option is widely used. It is based on distinguishing checked from unchecked errors. A checked error is one that is detected at execution time by an explicit test for ill-defined situations. For example, the quotient operation tests whether its denominator is zero, incurring an error if so. An unchecked error is one that is not detected at execution time, but rather is regarded as “stuck” or “ill-defined”. Type errors in MinML are unchecked errors, precisely because the static semantics ensures that they can never occur.

The point of introducing checked errors is that they ensure well-defined behavior even for ill-defined programs. Thus 3/0 evaluates to error, rather than simply “getting stuck” or behaving unpredictably. The essence of type safety is that well-typed programs should have well-defined behavior, even if that behavior is to signal an error. That way we can predict how the program will behave simply by looking at the program itself, without regard to the implementation or platform. In this sense safe languages are inherently portable, which explains the recent resurgence in interest in them.

How might checked errors be added to MinML? The main idea is to add to MinML a special expression, error, that designates a run-time fault in an expression. Its typing rule is as follows:

$$\Gamma \vdash \text{error} : \tau$$ (9.1)

Note that a run-time error can have any type at all. The reasons for this will become clear once we re-state the safety theorem.

The dynamic semantics is augmented in two ways. First, we add new transitions for the checked errors. For example, the following rule checks
for a zero denominator in a quotient:

\[ v_1 / 0 \rightarrow \text{error} \]  \hspace{1cm} (9.2)

Second, we add rules to propagate errors; once an error has arisen, it aborts the rest of the computation. Here are two representative error propagation rules:

\[ \text{error}(v_2) \rightarrow \text{error} \]  \hspace{1cm} (9.3)

\[ v_1(\text{error}) \rightarrow \text{error} \]  \hspace{1cm} (9.4)

These rules state that if the function or argument position of an application incur an error, then so does the entire application.

With these changes, the type safety theorem may be stated as follows:

**Theorem 9.6 (Safety With Errors)**

*If an expression is well-typed, it can only evaluate to a value or evaluate to error. It cannot “get stuck” in an ill-defined state.*

As before, safety follows from preservation and progress. The preservation theorem states that types are preserved by evaluation. We have already proved this for MinML; we need only consider error transitions. But for these preservation is trivial, since error has any type whatsoever. The canonical forms lemma carries over without change. The progress theorem is proved as before, relying on checked errors to ensure that progress can be made, even in ill-defined states such as division by zero.
Part IV

An Imperative Language
Chapter 10

TinyC, An Imperative Language

We consider a tiny fragment of the C language consisting of the following rudimentary features:

- Global variable and function declarations.
- Local variable declarations.
- Assignment, iteration, conditional statements.
- A minimal language of integer expressions.

The goal is to demonstrate the formalization of an imperative language, one whose execution model is based on assignment to variables.
10.1 Syntax

The syntax of TinyC is given by the following grammar.\(^1\)

\[
\begin{align*}
prog & \quad p ::= g \ s \\
stmt & \quad s ::= e ; \mid \text{if} \ (e) \ s_1 \ \text{else} \ s_2 \mid \text{return} \ e ; \mid \text{while} \ (e) \ s \ \{ \ l \ ss \} \\
nseq & \quad ss ::= o \mid ss \\
exp & \quad e ::= v \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid x = e \mid f(es) \\
exps & \quad es ::= e \mid e, es \\
gdec & \quad g ::= o \mid vd g \mid fd g \\
ldec & \quad l ::= o \mid vd l \\
vdec & \quad vd ::= \text{int} \ x = v ; \\
fdec & \quad fd ::= \text{int} \ f(as) \ s \\
arg & \quad a ::= \text{int} \ x \\
args & \quad as ::= a \mid a , as \\
value & \quad v ::= n
\end{align*}
\]

This grammar certainly makes clear the “stripped down” nature of TinyC syntax! Following C syntax conventions, we use “=” for variable assignment (a form of expression!), and curly braces to delimit blocks.

A statement is either an expression, executed for its effect and value, a conditional statement, a return statement, a while statement, or a block. A statement sequence is a sequence of statements. An expression is either a numeric literal, a variable, an arithmetic operation, an assignment, or a function call. A program consists of a sequence of global declarations together with a statement (usually a block).\(^2\) A global declaration is either a variable declaration or a function declaration; local declarations are limited

\(^1\)The notation “o” stands for the empty sequence.

\(^2\)The statement part is executed when the program is initiated. This generalizes the startup convention for C programs since the statement could simply be a call to a function named main.
to declare variables. All variables must be initialized to a given constant value.

It is important to realize that variables in TinyC do not correspond to variables in MinML (or in ordinary mathematical practice)! In MinML, as in mathematics, variables do not vary in the sense that once a binding is established for a variable, it retains that binding within its scope. In contrast, TinyC is based on assignment, which changes the bindings of variables during execution. Thus TinyC lacks “true” variables, and hence there is no role for higher-order abstract syntax in its formulation. Lacking the implicit notion of variable renaming provided by higher-order abstract syntax, we make the blanket assumption that no variables or function names are re-used within their scope. This restriction is not as severe as it may seem at first glance; we can, at the expense of some complication to the static semantics, take proper account of shadowing of variables as part of the static semantics.

10.2 Static Semantics

There being only one type, int, the static semantics of TinyC is quite simple. The only thing that needs to be enforced is that functions are called with the correct number of arguments. We can also, en passant, ensure that all functions and variables are declared before they are used, so we include that in the static semantics as well.

Let V range over finite sets of variables, and let F range over finite functions assigning a non-negative integer to some finite set of function variables. The set V will record the set of active variables; the function F will record the active functions and their arities (number of arguments).

The rules of the static semantics define judgements of the following
forms:

\[
\begin{align*}
&\vdash p \text{ ok} & p \text{ is well-formed} \\
&F, V \vdash s \text{ ok} & s \text{ is well-formed wrt } F \text{ and } V \\
&F, V \vdash ss \text{ ok} & ss \text{ is well-formed wrt } F \text{ and } V \\
&F, V \vdash e \text{ ok} & e \text{ is well-formed wrt } F \text{ and } V \\
&F, V \vdash es \text{ ok } n & es \text{ is well-formed wrt } F \text{ and } V \text{ and is of length } n \\
&F, V \vdash g \text{ decs } F', V' & g \text{ declares functions } F' \text{ and variables } V' \\
&F, V \vdash l \text{ decs } V' & l \text{ declares variables } V' \\
&F, V \vdash vd \text{ decs } x & vd \text{ declares variable } x \\
&F, V \vdash fd \text{ decs } f : n & fd \text{ declares function } f \text{ with arity } n \\
&F, V \vdash a \text{ decs } x & a \text{ declares variable } x \\
&F, V \vdash as \text{ decs } V' & as \text{ declares variables } V' \\
\end{align*}
\]

The rules defining these judgements follow.

A program is well-formed if its declarations are well-formed, and its body is well-formed with respect to those declarations.

\[
\frac{
\emptyset, \emptyset \vdash g \text{ decs } F, V \\
F, V \vdash s \text{ ok}}{
\vdash g; s \text{ ok}} 
\]  \hspace{1cm} (10.1)

A statement is well-formed if all of its constituent expressions are well-formed.

\[
\frac{
F, V \vdash e \text{ ok}}{
F, V \vdash e; \text{ ok}} 
\]  \hspace{1cm} (10.2)

\[
\frac{
F, V \vdash e \text{ ok} \quad F, V \vdash s_1 \text{ ok} \quad F, V \vdash s_2 \text{ ok} \quad F, V \vdash \text{ if } (e) \ s_1 \text{ else } s_2 \text{ ok}}{
F, V \vdash \text{ if } (e) \ s_1 \text{ else } s_2 \text{ ok}} 
\]  \hspace{1cm} (10.3)

\[
\frac{
F, V \vdash e \text{ ok} \quad F, V \vdash s \text{ ok}}{
F, V \vdash \text{ while } (e) \ s \text{ ok}} 
\]  \hspace{1cm} (10.4)

\[
\frac{
F, V \vdash e \text{ ok}}{
F, V \vdash \text{ return } e; \text{ ok}} 
\]  \hspace{1cm} (10.5)

\[
\frac{
F, V \vdash l \text{ decs } V' \quad F, V \cup V' \vdash ss \text{ ok} \quad (V \cap V' = \emptyset)}{
F, V \vdash \{ l \ ss \} \text{ ok}} 
\]  \hspace{1cm} (10.6)
A statement sequence is well-formed if its constituent statements are:

\[ F, V \vdash o \text{ok} \] (10.7)

\[ F, V \vdash s \text{ok} \quad F, V \vdash ss \text{ok} \]
\[ F, V \vdash s ss \text{ok} \] (10.8)

An expression is well-formed if all of its functions and variables are declared, and its functions are called with the correct number of arguments.

\[ F, V \vdash v \text{ok} \] (10.9)

\[ (x \in V) \]
\[ F, V \vdash x \text{ok} \] (10.10)

\[ F, V \vdash e_1 \text{ok} \quad F, V \vdash e_2 \text{ok} \]
\[ F, V \vdash e_1 + e_2 \text{ok} \] (10.11)

\[ F, V \vdash e_1 \text{ok} \quad F, V \vdash e_2 \text{ok} \]
\[ F, V \vdash e_1 - e_2 \text{ok} \] (10.12)

\[ F, V \vdash e \text{ok} \quad (x \in V) \]
\[ F, V \vdash x = e \text{ok} \] (10.13)

\[ F, V \vdash es \text{ok} \quad (F(f) = n) \]
\[ F, V \vdash f(es) \text{ok} \] (10.14)

Sequences of expressions are considered from left to right.

\[ F, V \vdash e \text{ok} \]
\[ F, V \vdash e \text{ok} 1 \] (10.15)

\[ F, V \vdash e \text{ok} \quad F, V \vdash es \text{ok} n \]
\[ F, V \vdash e, es \text{ok} n + 1 \] (10.16)
A sequence of global declarations may not re-declare a variable or function.

\[
F, V \vdash \circ \text{dec} \emptyset
\]  
(10.17)

\[
F, V \vdash vd \text{dec} x \\ F, V \cup \{x\} \vdash g \text{dec} F', V' \quad (x \notin V')
\]
\[
F, V \vdash vd g \text{dec} F', \{x\} \cup V'
\]  
(10.18)

\[
F, V \vdash fd \text{dec} f : n \\ F \cup \{f : n\}, V \vdash g \text{dec} F', V' \quad (f \notin F')
\]
\[
F, V \vdash fd g \text{dec} \{f : n\} \cup F', V'
\]  
(10.19)

A sequence of local declarations may not re-declare a variable.

\[
F, V \vdash \circ \text{dec} \emptyset
\]  
(10.20)

\[
F, V \vdash vd \text{dec} x \\ F, V \cup \{x\} \vdash l \text{dec} V' \quad (x \notin V')
\]
\[
F, V \vdash vd l \text{dec} \{x\} \cup V'
\]  
(10.21)

A variable declaration is well-formed if it is not already declared.

\[
(x \notin V)
\]
\[
F, V \vdash \text{int} \; x = v ; \text{dec} \; x
\]  
(10.22)

A function declaration is well-formed if it is not already declared, and its body is well-formed relative to its arguments and the assumption that the function itself is well-formed (to allow for recursion!).

\[
F \cup \{f : n\}, V \cup V' \vdash \text{ok} \\ F, V \vdash \text{as} \text{dec} V' \quad (|V'| = n, f \notin F)
\]
\[
F, V \vdash \text{int} \; f (as) \; s \text{dec} \; f : n
\]  
(10.23)

An argument must not already be a declared variable. A sequence of arguments may not re-declare a variable.

\[
(x \notin V)
\]
\[
F, V \vdash \text{int} \; x \text{dec} \; x
\]  
(10.24)

\[
F, V \vdash a \text{dec} \; x
\]
\[
F, V \vdash a \text{dec} \; \{x\}
\]  
(10.25)
10.3 Dynamic Semantics

The dynamic semantics for TinyC is given by a transition system, following along lines similar to those used for the semantics of MinML. However, since TinyC is an imperative language — one based on assignment to variables — it is necessary to take care to maintain the current values of the variables and to take account of changes to their values during execution. This is achieved by maintaining two pieces of information during evaluation, the current environment of function and variable declarations, and the current expression (or current statement) being evaluated. Each step of execution may modify the variables in the environment because of assignments embedded within expressions.

The dynamic semantics of TinyC is given by a simultaneous inductive definition of the following relations:

\[
\begin{align*}
&\text{Program execution} & p &\rightarrow p' \\
&\text{Statement execution} & (g, s) &\rightarrow (g', s') \\
&\text{Expression execution} & (g, e) &\rightarrow (g', e') \\
&\text{Expression list execution} & (g, es) &\rightarrow (g', es')
\end{align*}
\]

The rules defining these relations follow.

A program is finished if it has the form \((g, v ; )\), for some value \(v\).

A program is executed by executing its statement relative to its global declarations.

\[
\begin{align*}
(g, s) &\rightarrow (g', s') \\
g s &\rightarrow g' s'
\end{align*}
\]

An expression is executed for its effect on the environment.

\[
\begin{align*}
(g, e) &\rightarrow (g', e') \\
(g, e ; ; ) &\rightarrow (g', e' ; ; )
\end{align*}
\]
A conditional is executed by evaluating the test expression, then branching on whether the result is non-zero.

\[(g,e) \mapsto (g',e')\]
\[(g,\text{if} (e) \; s_1 \; \text{else} \; s_2) \mapsto (g',\text{if} (e') \; s_1 \; \text{else} \; s_2)\] (10.29)

\[(g,\text{if} (0) \; s_1 \; \text{else} \; s_2) \mapsto (g, s_2)\] (10.30)

\[(v \neq 0)\]
\[(g,\text{if} (v) \; s_1 \; \text{else} \; s_2) \mapsto (g, s_1)\] (10.31)

A while statement is executed by conditionally unrolling the body.

\[(g,\text{while} (e) \; s) \mapsto (g,\text{if} (e) \; \{ \; s \; \text{while} (e) \; s \; \} \; \text{else} \; \{ \; 0 \; ; \} \; )\] (10.32)

The statement \(0\; ;\) is a convenient “null statement” signalling loop termination.

A return statement is executed by evaluating its argument to a number.

\[(g,e) \mapsto (g',e')\]
\[(g,\text{return} \; e; \; ) \mapsto (g',\text{return} \; e'; \; )\] (10.33)

A block is executed by extending the environment with the local declarations, then executing the body to completion. The multi-step evaluation in the premise is a reflection of the block structure.\(^3\)

\[(g \cdot l, ss) \mapsto^* (g' \cdot l', v;\; )\]
\[(g, \{ \; l \; ss \; \}) \mapsto (g', v; \; )\] (10.34)

\[(g \cdot l, ss) \mapsto^* (g' \cdot l', \text{return} \; v;\; )\]
\[(g, \{ \; l \; ss \; \}) \mapsto (g', \text{return} \; v; \; )\] (10.35)

We assume that \(g'\) and \(g\) declare the same variables and functions, but differ only in the bindings of their variables, and that \(l'\) and \(l\) declare the same variables, and differ only in the bindings of their variables.

\(^3\)The notation \(g \cdot l\) stands for the result of concatenating the local declarations \(l\) onto the global declaration sequence \(g\). This may be defined by induction on the structure of \(l\) in the obvious manner.
A sequence of statements is executed by executing the first statement to completion, then evaluating the remaining sequence. Return statements stop the execution of a sequence. An expression executed for effect is discarded once completed. The null sequence is equivalent to returning 0.

\[
(g; \circ) \mapsto (g; 0; ) \quad (10.36)
\]

\[
(g; s) \mapsto (g'; s')
(g; s ss) \mapsto (g'; s' ss) \quad (10.37)
\]

\[
(g; v; ss) \mapsto (g; ss) \quad (10.38)
\]

\[
(g; \text{return } v; ss) \mapsto (g; \text{return } v; ) \quad (10.39)
\]

Variables in expressions evaluate to their current binding in the environment.

\[
g @ x = v \\
(g, x) \mapsto (g, v) \quad (10.40)
\]

The notation \( g @ x = v \) means that the result of looking up the binding of \( x \) in \( g \) is \( n \).

Arithmetic is performed by evaluating arguments, then performing the specified operation on them.

\[
(g, e_1) \mapsto (g', e'_1) \\
(g, e_1 + e_2) \mapsto (g', e'_1 + e_2) \quad (10.41)
\]

\[
(g, e_2) \mapsto (g', e'_2) \\
(g, n_1 + e_2) \mapsto (g', n_1 + e'_2) \quad (10.42)
\]

\[
(n = n_1 + n_2) \\
(g, n_1 + n_2) \mapsto (g, n) \quad (10.43)
\]

The rules for subtraction — and other arithmetic operations — are similar.

Assignments are evaluated by evaluating the right-hand side, then updating the environment.

\[
(g, e) \mapsto (g', e') \\
(g, x = e) \mapsto (g', x = e') \quad (10.44)
\]
\[
g \odot x \leftarrow v = g'
\]
\[
(g, x = v) \mapsto (g', v)
\]  
(10.45)

The notation \( g \odot x \leftarrow v = g' \) means that \( g' \) is the result of updating \( g \) by replacing the binding of \( x \) by \( n \).

Function calls are evaluated by evaluating the arguments, then creating a new block consisting of local bindings of the parameters to the corresponding arguments together with the body of the function as the body of the block.

\[
(g, es) \mapsto (g', es')
\]
\[
(g, f(es)) \mapsto (g', f(es'))
\]  
(10.46)

\[
g \odot f = \text{int } f(\text{int } x_1, \ldots \text{int } x_k, \circ ) \ s
\]
\[
(g, \{ \text{int } x_1 = v_1; \ldots \text{int } x_k = v_k; \circ s \}) \mapsto (g', \text{return } v;)
\]  
(10.47)

\[
g \odot f = \text{int } f(\text{int } x_1, \ldots \text{int } x_k, \circ ) \ s
\]
\[
(g, \{ \text{int } x_1 = v_1; \ldots \text{int } x_k = v_k; \circ s \}) \mapsto (g', v;)
\]  
(10.48)

Finally, sequences of expressions are evaluated from left to right.

\[
(g, e) \mapsto (g', e')
\]
\[
(g, e) \mapsto (g', e')
\]  
(10.49)

The premise of the rule refers to the evaluation of a single expression; the conclusion defines evaluation of a length-1 sequence of expressions.

\[
(g, e) \mapsto (g', e')
\]
\[
(g, e, es) \mapsto (g', e', es)
\]  
(10.50)

\[
(g, es) \mapsto (g', es')
\]
\[
(g, v, es) \mapsto (g', v, es')
\]  
(10.51)
10.4 Block Structure

TinyC is carefully designed to exhibit block structure, which means that its local declarations are allocated and deallocated in a stack-like manner. When entering a block or calling a function, we allocate its local variables or arguments, and when exiting a block or returning from a function we deallocate those variables.

This is achieved in the dynamic semantics by a “trick” embodied in Rules 10.47 and 10.48. The premise of this rule specifies that the body of the block must be evaluated to completion before the block is exited. In effect we are compressing the evaluation of a block of statements into a single step of evaluation. This ensures that the local variables, which are adjoined to the global declarations while processing the body of the block, are allocated only for the duration of the evaluation of the body, and are deallocated immediately afterwards.

Why does this work? The crucial insight is that the ultimate value of a sequence of statements is an integer, which can be safely “exported” from the block. This property, which ensures that the language may be implemented on a stack, would be violated if the returned value of the body of the block could depend on its locally declared variables in some manner. This is possible in C, but not in TinyC. In C it is possible to return the address of a local variable from a procedure. The execution behavior is ill-defined, because local variables are de-allocated on exit from a block. This is admitted, despite its ill-defined semantics; such is the nature of C.

An alternative might be to give up on the idea of stack allocation of local variables entirely, in favor of a more sophisticated memory management system. The idea here is that only unreachable storage is ever deallocated. If you return a reference to a local variable, then that local variable cannot be deallocated (and should not, therefore, be allocated on a stack). Instead, the storage for that variable must persist beyond the lifetime of that block to be recovered only if it is no longer needed for execution. Storage management based on reachability, called automatic storage management, is increasingly seen as an important tool, largely because it inherently avoids ill-defined situations such as those that can arise in C.

10.5 Type Safety

The safety of TinyC is stated as follows.
Theorem 10.1

If \( p \) is ok, then either \( p \) is finished, or there exists \( p' \) such that \( p \rightarrow p' \)

The only way the dynamic semantics can “get stuck” is to call a function with the wrong number of arguments, which is ruled out by the static semantics.

10.6 Exercises
Part V

Control and Data Flow
Chapter 11

Abstract Machines

Long considered to be a topic of primarily academic interest, abstract, or virtual, machines are now attracting renewed attention, especially by 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 state transformation using only physically realizable methods.\(^2\) Execution

\(^1\)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.

\(^2\)For example, consider the instruction that, given the representation of a program, sets register zero to one iff there is an input on which that program halts when executed, and
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 $e \xrightarrow{M} e'$ 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.

### 11.1 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:

1. stack

\[ \text{ stack} \]

2. frame stack

\[ f \text{ frame } k \text{ stack} \]

\[ f \triangleright k \text{ stack} \]

sets it to zero otherwise. This instruction 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:

\[
\begin{align*}
    e_2 \text{ expr} & \quad (11.3) \\
    + (\square, e_2) \text{ frame} & \\

    v_1 \text{ value} & \quad (11.4) \\
    + (v_1, \square) \text{ frame} \end{align*}
\]

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

\[
\begin{align*}
    e_1 \text{ expr} & \quad e_2 \text{ expr} \quad (11.5) \\
    \text{if} \square \text{then} e_1 \text{ else} e_2 \text{ fi} \text{ frame} \end{align*}
\]

\[
\begin{align*}
    e_2 \text{ expr} & \quad (11.6) \\
    \text{apply} (\square, e_2) \text{ frame} \end{align*}
\]

\[
\begin{align*}
    v_1 \text{ value} & \quad (11.7) \\
    \text{apply} (v_1, \square) \text{ frame} \end{align*}
\]

Thus a control stack is a sequence of frames \( f_1 \triangleright \cdots \triangleright f_n \triangleright \bullet \) (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 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.

\[
\begin{align*}
    (k, +(e_1, e_2)) \mapsto_C & \quad (+ (\square, e_2) \triangleright k, e_1) \quad (11.8) \\
    (+ (\square, e_2) \triangleright k, v_1) \mapsto_C & \quad (+ (v_1, \square) \triangleright k, e_2) \quad (11.9) \\
    (+ (n_1, \square) \triangleright k, n_2) \mapsto_C & \quad (k, n_1 + n_2) \quad (11.10) 
\end{align*}
\]

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 booleans.

\[
(k, \text{if} \square \text{then} e_1 \text{ else} e_2 \text{ fi}) \mapsto_C (\text{if} \square \text{then} e_1 \text{ else} e_2 \text{ fi} \triangleright k, e) \quad (11.11)
\]
Abstract Machines

\[(\text{if } \text{then } e_1 \text{ else } e_2 \text{ fi } \triangleright k, \text{true}) \mapsto_C (k, e_1) \quad (11.12)\]

\[(\text{if } \text{then } e_1 \text{ else } e_2 \text{ fi } \triangleright k, \text{false}) \mapsto_C (k, e_2) \quad (11.13)\]

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.

\[(k, \text{apply}(e_1, e_2)) \mapsto_C (\text{apply}(\Box, e_2) \triangleright k, e_1) \quad (11.14)\]

\[(\text{apply}(\Box, e_2) \triangleright k, v_1) \mapsto_C (\text{apply}(v_1, \Box) \triangleright k, e_2) \quad (11.15)\]

\[(\text{apply}(v_1, \Box) \triangleright k, v_2) \mapsto_C (k, \{v_1, v_2/f, x\} e) \quad (11.16)\]

The last rule applies in the case that \(v_1 = \text{fun } f \ (x : \tau_1) : \tau_2 \text{ is } e \text{ end}\). 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 \((v, \bullet)\) 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 machine by introducing a stack of pending sub-goals that specifies the order in which they are to be considered. In this sense the C machine 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 machine 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 machine states and M machine states. Intuitively the control stack in the C machine corresponds to the “surrounding context” of an expression, which is saved for consideration once the expression has been evaluated. Thus a C machine state may be thought of as representing the M machine 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:

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

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

**Theorem 11.1**

1. If $(k, e) \mapsto_C (k', e')$, then either $k @ e = k' @ e'$, or $k @ e \mapsto_M k' @ e'$.

2. If $e \mapsto_M e'$ and $(k, e') \mapsto_C^* (\bullet, v)$, then $(k, e) \mapsto_C^* (\bullet, v)$.

The first part of the Theorem states that the C machine 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 $k$.

**Proof:**

1. By induction on the definition of the C machine. We will do the cases for application here; the remainder follow a similar pattern.

---

[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.
(a) Consider the transition 

\[(k, \text{apply}(e_1, e_2)) \rightsquigarrow_{\mathcal{C}} (\text{apply}(\square, e_2) \triangleright k, e_1).\]

Here \(e = \text{apply}(e_1, e_2), k' = \text{apply}(\square, e_2) \triangleright k,\) and \(e' = e_1.\) It is easy to check that \(k \circ e = k' \circ e'.\)

(b) Consider the transition 

\[(\text{apply}(\square, e_2) \triangleright k'', v_1) \rightsquigarrow_{\mathcal{C}} (\text{apply}(v_1, \square) \triangleright k'', e_2).\]

Here \(e = v_1, k = \text{apply}(\square, e_2) \triangleright k'', e' = e_2,\) and \(k' = \text{apply}(v_1, \square) \triangleright k''.\) It is easy to check that \(k \circ e = k' \circ e'.\)

(c) Consider the transition 

\[(\text{apply}(v_1, \square) \triangleright k', v_2) \rightsquigarrow_{\mathcal{C}} (k', \{v_1, v_2/f, x\}e),\]

where \(v_1 = \text{fun } f (x : \tau_2) : \tau \text{ is } e \text{ end.} \) Here \(k = \text{apply}(v_1, \square) \triangleright k', e = v_2,\) and \(e' = \{v_1, v_2/f, x\}e.\) We have

\[k \circ e = k' \circ \text{apply}(v_1, v_2) \rightsquigarrow k' \circ e'\]

as desired. The second step follows from the observation that stacks are defined so that the \(M\) search rules “glide over” \(k'\) — the next instruction to execute in \(k' \circ \text{apply}(v_1, v_2)\) must be the application \(\text{apply}(v_1, v_2).\)

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

(a) \(e = \text{apply}(v_1, v_2) \rightsquigarrow_{M} \{v_1, v_2/f, x\}e_2 = e',\) where the value \(v_1 = \text{fun } f (x : \tau_2) : \tau \text{ is } e_2 \text{ end.} \) Suppose that \((k, e') \rightsquigarrow_{\mathcal{C}}^* (\bullet, v).\)

By the definition of the \(C\) machine transition relation,

\[
(k, e) \rightsquigarrow_{\mathcal{C}} (\text{apply}(\square, v_2) \triangleright k, v_1) \\
\rightsquigarrow_{\mathcal{C}} (\text{apply}(v_1, \square) \triangleright k, v_2) \\
\rightsquigarrow_{\mathcal{C}} (k, e')
\]

From this, the result follows immediately.

(b) \(e = \text{apply}(e_1, e_2) \rightsquigarrow_{M} \text{apply}(e_1', e_2) = e',\) where \(e_1 \rightsquigarrow_{M} e_1'.\)

Suppose that \((k, e') \rightsquigarrow_{\mathcal{C}}^* (\bullet, v).\) Since \(e' = \text{apply}(e_1', e_2),\) and
since the C machine is deterministic, this transition sequence must have the form

$$(k, e') = (k, \text{apply}(e_1, e_2)) \rightarrow_C (\text{apply}(\square, e_2) \triangleright k, e_1') \rightarrow_C^* (\bullet, v).$$

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

$$(\text{apply}(\square, e_2) \triangleright k, e_1) \rightarrow_C^* (\bullet, v).$$

Now since

$$(k, e) = (k, \text{apply}(e_1, e_2)) \rightarrow_C (\text{apply}(\square, e_2) \triangleright k, e_1)$$

the result follows immediately.

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

$$(k, e') = (k, \text{apply}(v_1, e_2')) \rightarrow_C (\text{apply}(v_1, \square) \triangleright k, e_2') \rightarrow_C^* (\bullet, v).$$

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

$$(\text{apply}(v_1, \square) \triangleright k, e_2) \rightarrow_C^* (\bullet, v).$$

Now since

$$(k, e) = (k, \text{apply}(v_1, e_2)) \rightarrow_C (\text{apply}(v_1, \square) \triangleright k, e_1)$$

the result follows immediately.

Exercise 11.2

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

Corollary 11.3

1. If $(k, e) \rightarrow_C^* (\bullet, v)$, then $k \rightarrow e \rightarrow_M^* v$. Hence if $(\bullet, e) \rightarrow_C^* (\bullet, v)$, then $e \rightarrow_M^* v$.

2. If $e \rightarrow_M^* e'$ and $(k, e') \rightarrow_C^* (\bullet, v)$, then $(k, e) \rightarrow_C^* (\bullet, v)$. Hence if $e \rightarrow_M^* v$, then $(\bullet, e) \rightarrow_C^* (\bullet, v)$. 

Working Draft  January 14, 2003
Proof:

1. By induction on the transition sequence, making use of part (1) of the theorem, then taking \( k = \bullet \). For the induction we have two cases to consider, one for each rule defining multi-step transition:

   (a) Reflexivity. In this case \( k = \bullet \) and \( e = v \). It follows that \( k \oplus e = e^{*} v \), as required.

   (b) Reverse execution. Here we have \((k', e') \mapsto (k, v)\), \((k, e) \mapsto (\bullet, v)\). By induction \( k \oplus e \mapsto_{M} v \), and by Theorem 11.1 \( k' \oplus e' \mapsto_{E} k \oplus e \), so \( k' \oplus e' \mapsto_{M} v \).

2. By induction on transition sequence, making use of part (2) of the theorem, then taking \( e' = v \) and \( k = \bullet \). We have two cases:

   (a) Reflexivity. In this case \( e = e' \) and the result is immediate.

   (b) Reverse execution. Here \( e \mapsto_{M} e'' \mapsto^{*} e' \) and \((k, e') \mapsto^{*} (\bullet, v)\). By induction \((k, e'') \mapsto^{*} (\bullet, v)\) and by Theorem 11.1 we have \((k, e) \mapsto^{*} (\bullet, v)\), as required.

To facilitate comparison with the E machine described below, it is useful to restructure the C machine in the following manner. First, we introduce an “auxiliary” state of the form \((v, k)\), which represents the process of passing the value \( v \) to the stack \( k \). Second, we “link” these two states by the transition rule

\[
(k, v) \mapsto_{C} (v, k).
\]  

(11.17)

That is, when encountering a value, pass it to the stack. Finally, we modify the transition relation so that all analysis of the stack is performed using the auxiliary state. Note that transitions now have one of four forms:

\[
\begin{align*}
(k, e) & \mapsto_{C} (k', e') & \text{process expression} \\
(k, v) & \mapsto_{C} (v, k) & \text{pass value to stack} \\
(v, k) & \mapsto_{C} (v', k') & \text{pass value up stack} \\
(v, k) & \mapsto_{C} (k', e') & \text{process pending expression}
\end{align*}
\]

Exercise 11.4

Complete the suggested re-formulation of the C machine, and show that it is equivalent to the original formulation.
11.2 Environments

The C machine 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. 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, \(E\) is an environment, a finite function mapping variables to machine values, and \(e\) is an open expression such that \(\text{FV}(e) \subseteq \text{dom}(E)\). 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 machine, however, is to avoid substitution by maintaining an environment that records

\[\text{In fact, the notion of “dynamic scope” arose as a result of an error in the original Lisp interpreter (circa 1960) that confused the scopes of variables.}\]
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 \( \text{int} \rightarrow \text{int} \rightarrow \text{int} \). When applied to an integer \( n \), the result \( \text{apply}(f, n) \) yields a function of type \( \text{int} \rightarrow \text{int} \). For example, \( f \) might be the following function:

\[
\text{fun } (x : \text{int}) : \text{int} \rightarrow \text{int} \text{is fun } (y : \text{int}) : \text{int} \text{is } x \text{ end end,}
\]

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

\[
\text{fun } (y : \text{int}) : \text{int} \rightarrow \text{int} \text{is } x \text{ end,}
\]

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.

This problem is avoided by the use of closures. The value returned by the application \( \text{apply}(f, 1) \) is the closure\(^6\)

\[
\text{fun } (y : \text{int}) : \text{int} \rightarrow \text{int} \text{is } [E[x \mapsto 1]]\end
\]

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

\(^5\)This is the error in the original implementation of Lisp referred to earlier.

\(^6\)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.
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 and machine values. The latter are inductively defined by the following rules:

\[
\begin{align*}
\text{n mvalue} & \\
\text{true mvalue} & \\
\text{false mvalue} & \\
\text{fun x (y:}\tau_1):\tau_2\text{ is end}[E] \text{ mvalue}
\end{align*}
\] (11.21)

An environment, \( E \), is a finite function mapping variables to machine values.

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

\[
\begin{align*}
\bullet \text{ mstack}
\end{align*}
\] (11.22)

\[
\frac{F \text{ mframe} \ K \text{ mstack}}{F \triangleright K \text{ mstack}}.\] (11.23)

Here \( F \) is a machine frame. The set of machine frames is inductively defined by these rules:

\[
\frac{e_2 \text{ expr}}{+(\Box, e_2)[E] \text{ mframe}}
\] (11.24)

\[
\frac{V_1 \text{ mvalue}}{+(V_1, \Box) \text{ mframe}}
\] (11.25)

\[
\frac{e_1 \text{ expr} \ e_2 \text{ expr}}{\text{if} \Box \text{ then} e_1 \text{ else} e_2 \text{ fi}[E] \text{ mframe}}
\] (11.26)

\[
\frac{e_2 \text{ expr}}{\text{apply}(\Box, e_2)[E] \text{ mframe}}
\] (11.27)

\[
\frac{V_1 \text{ mvalue}}{\text{apply}(V_1, \Box) \text{ mframe}}
\] (11.28)
The notation for $E$ machine frames is deceptively similar to the notation for $C$ machine frames. Note, however, that $E$ machine 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 $E$ machine has two kinds of states: $(K, E, e)$, described earlier, and “auxiliary” states of the form $(V, K)$, where $K$ is a machine stack and $V$ is a machine value. The auxiliary state represents the passage of a machine value to the top frame of the machine stack. (In the $C$ machine this is accomplished by simply filling the hole in the stack frame, but here a bit more work is required.)

The $E$ machine is inductively defined by a set of rules for transitions of one of the following four forms:

- $(K, E, e) \rightarrow_E (K', E', e')$ process expression
- $(K, E, v) \rightarrow_E (V', K')$ pass value to stack
- $(V, K) \rightarrow_E (V', K')$ pass value up stack
- $(V, K) \rightarrow_E (K', E', e')$ process pending expression

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.

$$(K, E, x) \rightarrow_E (E(x), K) \quad (11.29)$$

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

$$(K, E, n) \rightarrow_E (n, K) \quad (11.30)$$

$$(K, E, \text{true}) \rightarrow_E (\text{true}, K) \quad (11.31)$$

$$(K, E, \text{false}) \rightarrow_E (\text{false}, K) \quad (11.32)$$

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.

\[(K, E, \text{fun } (x: \tau_1): \tau_2 \text{ is } \text{end}) \mapsto_E (\text{fun } (x: \tau_1): \tau_2 \text{ is } \text{end}[E], K)\]

(11.33)

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.

\[(K, E, + (e_1, e_2)) \mapsto_E (+ (\Box, e_2)[E] \triangleright K, E, e_1)\]

(11.34)

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.

\[(K, E, \text{if } e \text{ then } e_1 \text{ else } e_2 \text{ fi}) \mapsto_E (\text{if } \Box \text{ then } e_1 \text{ else } e_2 \text{ fi}[E] \triangleright K, E, e)\]

(11.35)

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.

\[(K, E, \text{apply } (e_1, e_2)) \mapsto_E (\text{apply } (\Box, e_2)[E] \triangleright K, E, e_1)\]

(11.36)

To complete the definition of the \(E\) 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.

\[(V_1, + (\Box, e_2)[E] \triangleright K) \mapsto_E (+ (V_1, \Box) \triangleright K, E, e_2)\]

(11.37)

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.

\[(n_2, + (n_1, \Box) \triangleright K) \mapsto_E (n_1 + n_2, K)\]

(11.38)
Pending conditional branches are handled in the obvious manner.

\[
\text{true, if } \Box \text{ then } e_1 \text{ else } e_2 \text{ fi}[E] \triangleright K \mapsto_E (K, E, e_1) \quad (11.39)
\]

\[
\text{false, if } \Box \text{ then } e_1 \text{ else } e_2 \text{ fi}[E] \triangleright K \mapsto_E (K, E, e_2) \quad (11.40)
\]

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

Pending function applications are handled as follows.

\[
(V, \text{apply}(\Box, e_2)[E] \triangleright K) \mapsto_E (\text{apply}(V, \Box) \triangleright K, E, e_2) \quad (11.41)
\]

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

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

\[
(V_2, \text{apply}(V, \Box) \triangleright K) \mapsto_E (K, E[f \mapsto V][x \mapsto V_2], e) \quad (11.42)
\]

where

\[
V = \text{fun } f : \tau_1 : \tau_2 \text{ is } e \text{ end}[E].
\]

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

A correspondence between the \(E\) and the \(C\) machine along the lines of the correspondence between the \(C\) machine and the \(M\) 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 \text{MinML} program of base type (\text{int} or \text{bool}), then \((\bullet, e) \mapsto^*_C (\bullet, v) \iff (\bullet, \emptyset, e) \mapsto^*_E (v, \bullet)\). (The restriction to base type is necessary if we are to claim that both machines return the same value.)
Chapter 12

Continuations

The semantics of many control constructs (such as exceptions and co-routines) can be expressed in terms of reified control stacks, a representation of a control stack as an ordinary value. This is achieved by allowing a stack to be passed as a value 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. 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. In Section 12.3 we will show how to build a threads package for concurrent programming using continuations.

12.1 Informal Overview of Continuations

We will extend MinML with the type \( \tau \) 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\( x \)ine and throw\( e_1 \)to\( e_2 \).

Informally, evaluation of letcc\( x \)ine binds the current continuation\(^2\) to \( x \) and evaluates \( e \). The current continuation is, 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

\(^1\)Close relatives of these primitives are available in SML/NJ in the following forms: for letcc\( x \)ine, write SMLofNJ.Cont.callcc \((\text{fn } x = > e)\), and for throw\( e_1 \)to\( e_2 \), write SMLofNJ.Cont.throw \( e_2 \) \( e_1 \).

\(^2\)Hence the name “letcc”.

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 \( \tau \text{cont} \). Note that this is the only way to obtain a value of type \( \tau \text{cont} \); there are no expressions that evaluate to continuations. (This is similar to our treatment of references — values of type \( \tau \text{ref} \) are locations, but locations can only be obtained by evaluating a \( \text{ref} \) expression.)

We may “jump” to a saved control point by throwing a value to a continuation, written \( \text{throw} e_1 \text{to} e_2 \). The expression \( e_2 \) must evaluate to a \( \tau_1 \text{cont} \), 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:

```ml
fun mult_list (l:int list):int = 
  letcc ret in 
  let fun mult nil = 1 
  | mult (0::_) = throw 0 to ret 
  | mult (n::l) = n * mult l 
  in mult l end 
```

Ignoring the \( \text{letcc} \) for the moment, the body of \( \text{mult_list} \) is a let expression that defines a recursive procedure \( \text{mult} \), and applies it to the argument of \( \text{mult_list} \). The job of \( \text{mult} \) is to return the product of the elements of the list. Ignoring the second line of \( \text{mult} \), it should be clear why and how this code works.

Now let’s consider the second line of \( \text{mult} \), and the outer use of \( \text{letcc} \). Intuitively, the purpose of the second line of \( \text{mult} \) 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 \( \text{ret} \). This variable is bound by \( \text{letcc} \) surrounding the body of \( \text{mult_list} \). What continuation is it? It’s the continuation that runs upon completion of the body of \( \text{mult_list} \). 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 another formulation of the same function:
fun mult_list l =  
    let fun mult nil ret = 1  
        | mult (0::l) ret = throw 0 to ret  
        | mult (n::l) ret = n * mult l ret  
    in letcc ret in (mult l) ret 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.

Let’s look at another example: given a continuation \( k \) of type \( \tau \text{ cont} \) and a function \( f \) of type \( \tau' \rightarrow \tau \), return a continuation \( k' \) of type \( \tau' \text{ cont} \) 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:

\[
\text{fun compose } (f: \tau' \rightarrow \tau, k: \tau \text{ cont}): \tau' \text{ cont} = \ldots
\]

The function \( \text{compose} \) will have type

\[
((\tau' \rightarrow \tau) \times \tau \text{ cont}) \rightarrow \tau' \text{ cont}
\]

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 \( \text{throw } f(...) \) 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

\[
\text{throw } f(\text{letcc } x: \tau' \text{ cont in } \ldots) \text{ to } k
\]

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:

\[
\text{fun compose } (f, k) =
\text{letcc ret in}
\text{throw } (f \text{ (letcc } r \text{ in throw ret } r)) \text{ to } k
\]

The type of \( \text{ret} \) is \( \tau' \text{ cont cont} \), 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:

```haskell
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:

```haskell
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 l (fn p => k (n*p))
in cps_mult l k k end
```

The continuation k0 never changes; it is always the return continuation for cps_mult_list. The argument continuation to cps_mult_list is duplicated on the call to cps_mult.

Observe that the type of the first version of cps_mult becomes

\[ \text{int list} \to (\text{int} \to \alpha) \to \alpha, \]

and that the type of the second version becomes

\[ \text{int list} \to (\text{int} \to \alpha) \to (\text{int} \to \alpha) \to \alpha, \]

These transformations are representative of the general case.

### 12.2 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 surprisingly simple and intuitive.

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

$$e ::= \ldots | \text{letcc$x$in } e_1 \text{ to } e_2 | K$$

In the expression $\text{letcc$x$in } e$ 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 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:

$$K_{stack} \quad K_{value} \quad (12.1)$$

Stacks are as defined for the C machine, extended with these additional frames:

$$e_2 \text{ expr} \quad \text{throw} \square \text{ to } e_2 \text{ frame} \quad (12.2)$$

$$v_1 \text{ value} \quad \text{throw} v_1 \text{ to } \square \text{ frame} \quad (12.3)$$

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

$$\Gamma[x: \tau_{cont}] \vdash e : \tau \quad \Gamma \vdash \text{letcc$x$in } e : \tau \quad (12.4)$$

$$\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_1_{cont} \quad \Gamma \vdash \text{throw} e_1 \text{ to } e_2 : \tau' \quad (12.5)$$

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:

$$\vdash K : \tau_{stack} \quad \Gamma \vdash K : \tau_{cont} \quad (12.6)$$

That is, a continuation value $K$ has type $\tau_{cont}$ exactly if it is a stack accepting values of type $\tau$. This relation is defined below, when we consider type safety of this extension to MinML.
Finally, the dynamic semantics. We use the 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:

$$ (K, \text{letcc$x$ine}) \rightarrow (K, \{K/x\}e) \quad (12.7) $$

$$ \frac{}{(\text{throw$v$to} \square \triangleright K, K') \rightarrow (K', v)} \quad (12.8) $$

In addition we specify the order of evaluation of arguments to $\text{throw}$:

$$ (K, \text{throw$e_1$to$e_2$}) \rightarrow (\text{throw}$\square$to$e_2 \triangleright K, e_1) \quad (12.9) $$

$$ \frac{}{(\text{throw}$\square$to$e_2 \triangleright K, v_1) \rightarrow (\text{throw}$v_1$to$\square \triangleright K, e_2)} \quad (12.10) $$

Notice that evaluation of $\text{letcc}$ duplicates the control stack, and that evaluation of $\text{throw}$ eliminates the current control stack.

The safety of this extension of MinML may be established by proving a preservation and progress theorem for the abstract machine. The well-formedness of a machine state is defined by the following rule:

$$ \frac{\vdash K : \tau\text{stack} \quad \vdash e : \tau}{(K, e) \text{ ok}} \quad (12.11) $$

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.

To define the judgement $\vdash K : \tau\text{stack}$, we must first fix the type of the “ultimate answer” of a program, the value returned when evaluation is completed. The particular choice of answer type is not important, but it is important that it be a fixed type, $\tau_{\text{ans}}$.

$$ \vdash \bullet : \tau_{\text{ans}}\text{stack} \quad (12.12) $$

$$ \frac{\vdash F : (\tau, \tau')\text{frame} \quad \vdash K : \tau'\text{stack}}{\vdash F \triangleright K : \tau\text{stack}} \quad (12.13) $$

Thus a stack is well-typed iff its frames compose properly. The typing rules for frames as as follows:

$$ \frac{\vdash e_2 : \text{int}}{\vdash (\square, e_2) : (\text{int}, \text{int})\text{frame}} \quad (12.14) $$
\[
\begin{align*}
&v_1 \text{ value} \vdash v_1 : \text{int} \\
&\vdash + (v_1, \Box) : (\text{int, int}) \text{ frame} \tag{12.15} \\
&\vdash e_1 : \tau \quad \vdash e_2 : \tau \\
&\vdash \text{if } \Box \text{ then } e_1 \text{ else } e_2 \text{ fi} : (\text{bool, } \tau) \text{ frame} \tag{12.16} \\
&\vdash e_2 : \tau_2 \\
&\vdash \text{apply} (\Box, e_2) : (\tau_2 \rightarrow \tau, \tau) \text{ frame} \tag{12.17} \\
&v_1 \text{ value} \vdash v_1 : \tau_2 \rightarrow \tau \\
&\vdash \text{apply} (v_1, \Box) : (\tau_2, \tau) \text{ frame} \tag{12.18} \\
&\vdash e_2 : \tau \\
&\vdash \text{throw } \Box \text{ to } e_2 : (\tau, \tau') \text{ frame} \tag{12.19}
\end{align*}
\]

Intuitively, a frame of type \((\tau_1, \tau_2)\) 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 hand, we may state the preservation theorem as follows:

**Theorem 12.1 (Preservation)**

If \((K, e)\) ok and \((K, e) \leftrightarrow (K', e')\), 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 12.2 (Canonical Forms)**

If \(\vdash v : \tau \text{ cont}\), then \(v = K\) for some control stack \(K\) such that \(\vdash K : \tau \text{ stack}\).

Finally, progress is stated as follows:

**Theorem 12.3 (Progress)**

If \((K, e)\) ok then either \(K = \bullet\) and \(e\) value, or there exists \(K'\) and \(e'\) such that \((K, e) \leftrightarrow (K', e')\).

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

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, writting in Standard ML.

```ml
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 in terms of the SML/NJ continuation primitives:
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 \texttt{fork} is called to create a new thread executing the body of
the given function. The function \texttt{yield} is called to cede control to another
thread, selected by the thread scheduler. The function \texttt{exit} is called to
terminate a thread.

User-level threads are naturally implemented as continuations. A thread
is a value of type \texttt{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 imple-
mented 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 succes-
sion 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.

### 12.4 Exercises

1. Study the short-circuit multiplication example carefully to be sure you understand why it works!

2. Attempt to solve the problem of composing a continuation with a function yourself, before reading the solution.

3. Simulate the evaluation of \( \text{compose} \left(f, k\right) \) on the empty stack. Observe that the control stack substituted for \( x \) is

   \[
   \text{apply} \left(f, \Box\right) \triangleright \text{throw} \Box \text{to} \ k \triangleright \bullet \tag{12.20}
   \]

   This stack is returned from \( \text{compose} \). Next, simulate the behavior of throwing a value \( \nu' \) to this continuation. Observe that the above stack is reinstated and that \( \nu' \) is passed to it.
Chapter 13

Exceptions

Exceptions effect a non-local transfer of control from the point at which the exception is raised to a dynamically enclosing handler for that exception. This transfer interrupts the normal flow of control in a program in response to unusual conditions. For example, exceptions can be used to signal an error condition, or to indicate the need for special handling in certain circumstances that arise only rarely. To be sure, one could use explicit conditionals to check for and process errors or unusual conditions, but using exceptions is often more convenient, particularly since the transfer to the handler is direct and immediate, rather than indirect via a series of explicit checks. All too often explicit checks are omitted (by design or neglect), whereas exceptions cannot be ignored.

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:

\[ e ::= \ldots | \text{fail} | \text{try} \ v_1 \ow \ v_2 \]

The expression \text{fail} raises an exception. The expression \text{try} \ v_1 \ow \ v_2 evaluates \( v_1 \). If it terminates normally, we return its value; otherwise, if it fails, we continue by evaluating \( v_2 \).

The static semantics of exceptions is quite straightforward:

\[ \Gamma \vdash \text{fail} : \tau \]  

\[ (13.1) \]
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:

\[
\frac{e_2 \text{ expr}}{\text{try} \square \text{ow} e_2 \text{ frame}} \quad (13.3)
\]

The evaluation rules are extended as follows:

\[
(K, \text{try} e_1 \text{ow} e_2) \mapsto (\text{try} \square \text{ow} e_2 \triangleright K, e_1) \quad (13.4)
\]

\[
(\text{try} \square \text{ow} e_2 \triangleright K, v) \mapsto (K, v) \quad (13.5)
\]

\[
(\text{try} \square \text{ow} e_2 \triangleright K, \text{fail}) \mapsto (K, e_2) \quad (13.6)
\]

\[
(F \neq \text{try} \square \text{ow} e_2) \quad (F \triangleright K, \text{fail}) \mapsto (K, \text{fail}) \quad (13.7)
\]

To evaluate \( \text{try} e_1 \text{ow} e_2 \) 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 \((\bullet, \text{fail})\) as a final state of computation, corresponding to an uncaught exception.

Using the definition of stack typing given in 12, we can state and prove safety of the exception mechanism.

**Theorem 13.1 (Preservation)**

If \((K, e) \text{ ok and } (K, e) \mapsto (K', e')\), then \((K, e) \text{ ok}\).

**Proof:** By induction on evaluation. \[\square]\
Theorem 13.2 (Progress)
If \((K, e)\) ok then either

1. \(K = \bullet\) and \(e\) value, or
2. \(K = \bullet\) and \(e = \text{fail}\), or
3. there exists \(K'\) and \(e'\) such that \((K, e) \mapsto (K', e')\).

Proof: By induction on typing.

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:

\[
\begin{array}{l}
\text{hstack} \\
\end{array}
\]

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:

\[ ((K', e') \triangleright H, K, \text{fail}) \mapsto (H, K', e') \]  \hfill (13.10)

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:

\[ (\bullet, K, \text{fail}) \mapsto (\bullet, \bullet, \text{fail}) \]  \hfill (13.11)

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

\[ (H, K, \text{try } e_1 \circ e_2) \mapsto ((K, e_2) \triangleright H, \text{try } \Box \circ e_2 \triangleright K, e_1) \]  \hfill (13.12)

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 \( e_1 \):

\[ ((K, e_2) \triangleright H, \text{try } \Box \circ e_2 \triangleright K, v_1) \mapsto (H, K, v_1) \]  \hfill (13.13)

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.

The prefix property may be taken as a formal justification of an implementation based on the \texttt{setjmp} and \texttt{longjmp} constructs of the C language. Unlike \texttt{setjmp} and \texttt{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.

Finally, let us consider value-passing exceptions such as are found in Standard ML. The main idea is to replace the failure expression, \texttt{fail}, by a more general \texttt{raise} expression, \texttt{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:

\[
\frac{\Gamma \vdash e : \tau\text{exn}}{\Gamma \vdash \text{raise}(e) : \tau} \quad (13.14)
\]

\[
\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau\text{exn} \rightarrow \tau}{\Gamma \vdash \text{try}\ e_1\ \text{ow}\ e_2 : \tau} \quad (13.15)
\]

These rules are parameterized by the type of values associated with exceptions, \(\tau\text{exn}\).

The question is: what should be the type \(\tau\text{exn}\)? 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 \(\tau\text{exn}\) that supports a flexible programming style.

For example, we might choose, say, \text{string}, for \(\tau\text{exn}\), with the idea that the value associated with an exception is a description of the cause of the exception. For example, we might write

\[
\text{fun div} \ (m, 0) = \text{raise "Division by zero attempted."} \\
| \text{div} \ (m, n) = \ldots \text{raise "Arithmetic overflow occurred."} \ldots
\]

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 \(\tau\text{exn}\) to be, say, \text{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 \(\tau\text{exn}\) to be a given datatype \(\text{exc}\). For example, we might have the declaration

\[
\text{datatype exc = Div | Overflow | Match | Bind}
\]

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

\[
\text{fun div} \ (m, 0) = \text{raise Div} \\
| \text{div} \ (m, n) = \ldots \text{raise Overflow} \ldots
\]
Now the handler can easily dispatch on \texttt{Div} or \texttt{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 \texttt{exc} datatype.

To allow extensibility Standard ML includes a special extensible datatype called \texttt{exn}. Values of type \texttt{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 \texttt{exn} in the usual way. But, in addition, we may introduce new constructors of type \texttt{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:

\begin{verbatim}
  exception Div
  exception Overflow
\end{verbatim}

Now \texttt{Div} and \texttt{Overflow} are constructors of type \texttt{exn}, and may be used in a \texttt{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 \texttt{Error}, then these are different exceptions; no confusion is possible.

The interesting thing about the \texttt{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 \texttt{exn} type for other purposes; indeed, Java has an analogue of the type \texttt{exn}, called \texttt{Object}. This is the basis for downcasting and so-called typecase in Java.

\section{13.1 Exercises}

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


4. 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:

(a) As in ML, mutations are irrevocable, even in the face of exceptions that “backtrack” to a surrounding handler.

(b) 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.

5. State and prove the safety of the formulation of exceptions using a handler stack.

6. Prove that the prefix property is preserved by every step of evaluation.
Part VI

Imperative Functional Programming
Chapter 14

Mutable Storage

MinML is said to be a pure language because the execution model consists entirely of evaluating an expression for its value. ML is an impure language because its execution model also includes effects, specifically, control effects and store effects. Control effects are non-local transfers of control; these were studied in Chapters 12 and 13. Store effects are dynamic modifications to mutable storage. This chapter is concerned with store effects.

14.1 References

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

\[
e ::= l \mid \text{ref}(e) \mid !e \mid e_1 := e_2
\]

As in Standard ML, \( \text{ref}(e) \) allocates a “new” reference cell, \( !e \) retrieves the contents of the cell \( e \), and \( e_1 := e_2 \) 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:

\[
\frac{(\Lambda(l) = \tau)}{\Lambda; \Gamma \vdash l : \tau \text{ ref}} \quad (14.1)
\]

\[
\frac{\Lambda; \Gamma \vdash e : \tau}{\Lambda; \Gamma \vdash \text{ref}(e) : \tau \text{ ref}} \quad (14.2)
\]

\[
\frac{\Lambda; \Gamma \vdash e : \tau \text{ ref}}{\Lambda; \Gamma \vdash !e : \tau} \quad (14.3)
\]

\[
\frac{\Lambda; \Gamma \vdash e_1 : \tau_2 \text{ ref} \quad \Lambda; \Gamma \vdash e_2 : \tau_2}{\Lambda; \Gamma \vdash e_1 := e_2 : \tau_2} \quad (14.4)
\]

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 \(\text{dom}(M)\) 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:

\[
(M, e) \rightarrow (M', e')
\]

\[
(M, \text{ref}(e)) \rightarrow (M', \text{ref}(e'))
\]

\[
(l \notin \text{dom}(M))
\]

\[
(M, \text{ref}(v)) \rightarrow (M[l=v], l)
\]

\[
(M, e) \rightarrow (M', e')
\]

\[
(M, !e) \rightarrow (M', !e')
\]
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 $\updownarrow M : \Lambda$, is inductively defined by the following rule:

\[
\begin{align*}
(l \in \text{dom}(M)) & \quad \frac{(M, l) \mapsto (M, M(l))}{(M, l) : \uparrow \Lambda(l)} \\
(M, e_1) & \mapsto (M', e'_1) \\
(M, e_1 := e_2) & \mapsto (M', e'_1 := e_2) \\
(M, e_2) & \mapsto (M', e'_2) \\
(M, v_1 := e_2) & \mapsto (M', v_1 := e'_2) \\
(M, l := v) & \mapsto (M[l := v], v)
\end{align*}
\]

\[(14.8)\]

\[(14.9)\]

\[(14.10)\]

\[(14.11)\]

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 \textit{backpatching}. It is used in some compilers to implement recursive functions (and other forms of looping construct).

\textbf{Exercise 14.1}

1. Sketch the contents of the memory after each step in the above example. Observe that after the assignment to \texttt{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 $\models M : \Lambda$.

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

\[
\begin{array}{c}
\models M : \Lambda \\
\Lambda; \cdot \vdash e : \tau
\end{array} \quad \rightarrow \quad (M, e) \text{ ok}
\]

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

\textbf{Theorem 14.2 (Preservation)}

\textit{If $(M, e)$ ok and $(M, e) \rightarrow (M', e')$, then $(M', e')$ ok.}

\textbf{Proof:} The trick is to prove a stronger result by induction on evaluation: if $(M, e) \rightarrow (M', e')$, $\models M : \Lambda$, and $\Lambda; \cdot \vdash e : \tau$, then there exists $\Lambda' \supseteq \Lambda$ such that $\models M' : \Lambda'$ and $\Lambda'; \cdot \vdash e' : \tau$.

\textbf{Exercise 14.3}

\textit{Prove Theorem 14.2. 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 14.4 (Progress)**

If $(M, e)$ is ok then either $(M, e)$ is a final state or there exists $(M', e')$ such that $(M, e) \rightarrow (M', e')$.

**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 $(M, e) \rightarrow (M', e')$.

**Exercise 14.5**

Prove Theorem 14.4 by induction on typing of machine states.
Chapter 15

Monads

As we saw in Chapter 14 one way to combine functional and imperative programming is to add a type of reference cells to MinML. This approach works well for call-by-value languages, because we can easily predict where expressions are evaluated, and hence where references are allocated and assigned. For call-by-name languages this approach is problematic, because in such languages it is much harder to predict when (and how often) expressions are evaluated.

Enriching ML with a type of references has an additional consequence that one can no longer determine from the type alone whether an expression mutates storage. For example, a function of type int→int must taken an integer as argument and yield an integer as result, but may or may not allocate new reference cells or mutate existing reference cells. The expressive power of the type system is thereby weakened, because we cannot distinguish \textit{pure} (effect-free) expressions from \textit{impure} (effect-ful) expressions.

Another approach to introducing effects in a purely functional language is to make the use of effects explicit in the type system. Several methods have been proposed, but the most elegant and widely used is the concept of a \textit{monad}. Roughly speaking, we distinguish between \textit{pure} and \textit{impure} expressions, and make a corresponding distinction between \textit{pure} and \textit{impure} function types. Then a function of type int→int is a pure function (has no effects when evaluated), whereas a function of type int → int may have an effect when applied. The monadic approach is more popular for call-by-name languages, but is equally sensible for call-by-value languages.

\footnote{We need to introduce cbv and cbn earlier, say in Chapter 8.}
15.1 Monadic MinML

A monadic variant of MinML is obtained by separating pure from impure expressions. The pure expressions are those of MinML. The impure expressions consist of any pure expression (vacuously impure), plus a new primitive expression, called \textit{bind}, for sequencing evaluation of impure expressions. In addition the impure expressions include primitives for allocating, mutating, and accessing storage; these are “impure” because they depend on the store for their execution.

The abstract syntax of monadic MinML is given by the following grammar:

\begin{align*}
\text{Types} & \quad \tau ::= \text{int} | \text{bool} | \tau_1 \rightarrow \tau_2 | \tau_1 
\text{Pure} & \quad e ::= x | n | o(e_1 \ldots , e_n) | 
true | false | \text{if } e \text{ then } e_1 \text{ else } e_2 \text{ fi } | 
\text{fun } f(x: \tau_1) : \tau_2 \text{ is } e \text{ end } | \text{apply}(e_1, e_2) 
\text{fun } f(x: \tau_1) : \tau_2 \text{ is } m \text{ end } \\
\text{Impure} & \quad m ::= \text{return } e | \text{bind } x: \tau \leftarrow m_1 \text{ in } m_2 \\
& \quad \text{if } e \text{ then } m_1 \text{ else } m_2 \text{ fi } | \text{apply}(e_1, e_2)
\end{align*}

Monadic MinML is a general framework for computing with effects. Note that there are two forms of function, one whose body is pure, and one whose body is impure. Correspondingly, there are two forms of application, one for pure functions, one for impure functions. There are also two forms of conditional, according to whether the arms are pure or impure. (We will discuss methods for eliminating some of this redundancy below.)

The static semantics of monadic MinML consists of two typing judgements, \( \Gamma \vdash e : \tau \) for pure expressions, and \( \Gamma \vdash m : \tau \) for impure expressions.
Most of the rules are as for MinML; the main differences are given below.

\[
\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash m : \tau_2 \\
\Gamma \vdash \text{fun } f (x : \tau_1) : \tau_2 \text{ is } m \text{ end} : \tau_1 \to \tau_2
\]

\[
\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash \text{apply}(e_1, e_2) : \tau
\]

\[
\Gamma \vdash e : \tau \\
\Gamma \vdash \text{return } e : \tau
\]

\[
\Gamma \vdash m_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash m_2 : \tau_2 \\
\Gamma \vdash \text{bind } x : \tau \leftarrow m_1 \text{ in } m_2 : \tau_2
\]

\[
\Gamma \vdash e : \text{bool} \quad \Gamma \vdash m_1 : \tau \quad \Gamma \vdash m_2 : \tau \\
\Gamma \vdash \text{if } e \text{ then } m_1 \text{ else } m_2 \text{ fi} : \tau
\]

So far we have not presented any mechanisms for engendering effects! Monadic MinML is rather a framework for a wide variety of effects that we will instantiate to the case of mutable storage. This is achieved by adding the following forms of impure expression to the language:

\[
\text{Impure } m ::= \text{ref}(e) \mid !e \mid e_1 := e_2
\]

Their typing rules are as follows:

\[
\Gamma \vdash e : \tau \\
\Gamma \vdash \text{ref}(e) : \tau \text{ ref}
\]

\[
\Gamma \vdash e : \tau \text{ ref} \\
\Gamma \vdash !e : \tau
\]

\[
\Gamma \vdash e_1 : \tau \text{ ref} \quad \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash e_1 := e_2 : \tau_2
\]

In addition we include locations as pure expressions, with typing rule

\[
\Gamma(l) = \tau \\
\Gamma \vdash l : \tau \text{ ref}
\]

(For convenience we merge the location and variable typings.)

The dynamic semantics of monadic MinML is an extension to that of MinML. Evaluation of pure expressions does not change, but we must
add rules governing evaluation of impure expressions. For the purposes of describing mutable storage, we must consider transitions of the form 
\((M, m) \mapsto (M', m')\), where \(M\) and \(M'\) are memories, as in Chapter 14.

\[
\begin{align*}
    e & \mapsto e' \\
    (M, \text{return } e) & \mapsto (M, \text{return } e') \\
    (M, m_1) & \mapsto (M', m'_1) \\
    (M, \text{bind } x:\tau \leftarrow m_1 \text{ in } m_2) & \mapsto (M', \text{bind } x:\tau \leftarrow m'_1 \text{ in } m_2) \\
    (M, \text{bind } x:\tau \leftarrow \text{return } v \text{ in } m_2) & \mapsto (M, \{v/x\}m_2)
\end{align*}
\]

The evaluation rules for the reference primitives are as in Chapter 14.

### 15.2 Reifying Effects

The need for pure and impure function spaces in monadic MinML is somewhat unpleasant because of the duplication of constructs. One way to avoid this is to introduce a new type constructor, \(!\tau\), whose elements are unevaluated impure expressions. The computation embodied by the expression is said to be reified (turned into a “thing”).

The syntax required for this extension is as follows:

**Types**

\(\tau ::= !\tau\)

**Pure**

\(e ::= \text{box}(m)\)

**Impure**

\(m ::= \text{unbox}(e)\)

Informally, the pure expression \(\text{box}(m)\) is a value that contains an *unevaluated* impure expression \(m\); the expression \(m\) is said to be boxed. Boxed expressions can be used as ordinary values without restriction. The expression \(\text{unbox}(e)\) “opens the box” and evaluates the impure expression inside; it is therefore itself an impure expression.

The static semantics of this extension is given by the following rules:

\[
\begin{align*}
    \Gamma \vdash m : \tau & \\
    \Gamma \vdash \text{box}(m) : !\tau \\
    \Gamma \vdash e : !\tau & \\
    \Gamma \vdash \text{unbox}(e) : \tau
\end{align*}
\]
The dynamic semantics is given by the following transition rules:

\[
\begin{align*}
(M, \text{unbox}(\text{box}(m))) & \Rightarrow (M, m) \\
(e \mapsto e') & \Rightarrow (M, \text{unbox}(e'))
\end{align*}
\]

The expression \( \text{box}(m) \) is a value, for any choice of \( m \).

One use for reifying effects is to replace the impure function space, \( \tau_1 \rightarrow \tau_2 \), with the pure function space \( \tau_1 \rightarrow! \tau_2 \). The idea is that an impure function is a pure function that yields a suspended computation that must be unboxed to be executed. The impure function expression

\[
\text{fun } f \ (x : \tau_1) : \tau_2 \text{ is } m \text{ end}
\]

is replaced by the pure function expression

\[
\text{fun } f \ (x : \tau_1) : \tau_2 \text{ is } \text{box}(m) \text{ end.}
\]

The impure application,

\[
\text{apply} (e_1, e_2),
\]

is replaced by

\[
\text{unbox} (\text{apply} (e_1, e_2)),
\]

which unboxes, hence executes, the suspended computation.

15.3 Exercises

1. Consider other forms of effect such as I/O.
2. Check type safety.
3. Problems with multiple monads to distinguish multiple effects.
Part VII

Cost Semantics and Parallelism
Chapter 16

Cost Semantics

The dynamic semantics of MinML is given by a transition relation $e \rightarrow e'$ defined using Plotkin’s method of Structured Operational Semantics (SOS). One benefit of a transition semantics is that it provides a natural measure of the time complexity of an expression, namely the number of steps required to reach a value.

An evaluation semantics, on the other hand, has an appealing simplicity, since it defines directly the value of an expression, suppressing the details of the process of execution. However, by doing so, we no longer obtain a direct account of the cost of evaluation as we do in the transition semantics.

The purpose of a cost semantics is to enrich evaluation semantics to record not only the value of each expression, but also the cost of evaluating it. One natural notion of cost is the number of instructions required to evaluate the expression to a value. The assignment of costs in the cost semantics can be justified by relating it to the transition semantics.

16.1 Evaluation Semantics

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

$$
\begin{align*}
\frac{}{n \Downarrow n} \quad (16.1) \\
\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{+(e_1, e_2) \Downarrow n_1 + n_2} \quad (16.2)
\end{align*}
$$
(and similarly for the other primitive operations).

\[
\begin{array}{ll}
\text{true} \downarrow \text{true} & \text{false} \downarrow \text{false} \\
\hline
\end{array}
\]  

(16.3)

\[
\frac{e \downarrow \text{true} \quad e_1 \downarrow v}{\text{if} \ e_1 \ \text{else} e_2 \ \text{fi} \downarrow v}
\]  

(16.4)

\[
\frac{e \downarrow \text{false} \quad e_2 \downarrow v}{\text{if} \ e_1 \ \text{else} e_2 \ \text{fi} \downarrow v}
\]  

(16.5)

\[
\frac{\text{fun} \ f \ (x: \tau_1) : \tau_2 \ \text{is} \ \text{end} \downarrow \text{fun} \ f \ (x: \tau_1) : \tau_2 \ \text{is} \ \text{end}}
\]

(16.6)

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

(16.7)

(where \(v_1 = \text{fun} \ f \ (x: \tau_1) : \tau_2 \ \text{is} \ \text{end}\).

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

### 16.2 Relating Evaluation Semantics to Transition Semantics

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

**Theorem 16.1**

1. If \(e \downarrow v\), then \(e \rightarrow^* v\).
2. If \(e \rightarrow e'\) and \(e' \downarrow v\), then \(e \downarrow v\). Consequently, if \(e \rightarrow^* v\), then \(e \downarrow v\).

**Proof:**

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

\[\boxed{\text{16.3 Cost Semantics}}\]

In this section we will give a cost semantics for MinML that reflects the num-
ber of steps required to complete evaluation according to the structured 
operational semantics given in Chapter 8.

Evaluation judgements have the form \( e \Downarrow^n v \), with the informal mean-
ing that \( e \) evaluates to \( v \) in \( n \) steps. The rules for deriving these judgements 
are easily defined.

\begin{equation}
\frac{\text{n \Downarrow^0 n}}{}
\end{equation}

\begin{equation}
\frac{e_1 \Downarrow^{k_1} n_1 \quad e_2 \Downarrow^{k_2} n_2}{e_1 \Downarrow e_2 \downarrow^{k_1+k_2+1} n_1 + n_2}
\end{equation}

(and similarly for the other primitive operations).

\begin{equation}
\frac{\text{true \Downarrow^0 true} \quad \text{false \Downarrow^0 false}}{}
\end{equation}

\begin{equation}
\frac{e \Downarrow^k \text{true} \quad e_1 \Downarrow^{k_1} v}{\text{if} \text{then} e_1 \text{else} e_2 \text{fi} \Downarrow^{k+k_1+1} v}
\end{equation}

\begin{equation}
\frac{e \Downarrow^k \text{false} \quad e_2 \Downarrow^{k_2} v}{\text{if} \text{then} e_1 \text{else} e_2 \text{fi} \Downarrow^{k+k_2+1} v}
\end{equation}

Working Draft January 14, 2003
\[ \text{fun } f (x: \tau_1): \tau_2 \text{ is end} \Downarrow^0 \text{fun } f (x: \tau_1): \tau_2 \text{ is end} \]  
\[ \begin{array}{c}
e_1 \Downarrow^k v_1 \\
 e_2 \Downarrow^{k_2} v_2 \\
 \{v_1, v_2/f, x\} e \Downarrow^k v \\
 \text{apply}(e_1, e_2) \Downarrow^{k_1 + k_2 + k + 1} v \end{array} \]  
\[ \text{(where } v_1 = \text{fun } f (x: \tau_1): \tau_2 \text{ is end).} \]

This completes the definition of the cost semantics for MinML.

### 16.4 Relating Cost Semantics to Transition Semantics

What is it that makes the cost semantics given above “correct”? Informally, we expect that if \( e \Downarrow^k v \), then \( e \) should evaluate to \( v \) in \( k \) steps. Moreover, we also expect the converse to hold — the cost semantics should be completely faithful to the underlying execution model. This is captured by the following theorem.

To state the theorem we need one additional bit of notation. Define \( e \rightarrow^k e' \) by induction on \( k \) as follows. For the basis, we define \( e \rightarrow^0 e' \) iff \( e = e' \); if \( k = k' + 1 \), we define \( e \rightarrow^k e' \) to hold iff \( e \rightarrow^{k'} e' \).

**Theorem 16.2**

For any closed expression \( e \) and closed value \( v \) of the same type, \( e \Downarrow^k v \) iff \( e \rightarrow^k v \).

**Proof:** From left to right we proceed by induction on the definition of the cost semantics. For example, consider the rule for function application. We have \( e = \text{apply}(e_1, e_2) \) and \( k = k_1 + k_2 + k + 1 \), where

1. \( e_1 \Downarrow^{k_1} v_1 \),
2. \( e_2 \Downarrow^{k_2} v_2 \),
3. \( v_1 = \text{fun } f (x: \tau_1): \tau_2 \text{ is end} \),
4. \( \{v_1, v_2/f, x\} e \Downarrow^k v \).

By induction we have

1. \( e_1 \rightarrow^{k_1} v_1 \),
2. \( e_2 \rightarrow^{k_2} v_2 \),
3. \( \{v_1, v_2/f, x\} e \rightarrow^k v \),
and hence
\[
\begin{align*}
e_1 (e_2) & \rightarrow^{k_1} v_1 (e_2) \\
& \rightarrow^{k_2} v_1 (v_2) \\
& \rightarrow \{v_1, v_2/f, x\} e \\
& \rightarrow^k v
\end{align*}
\]
which is enough for the result.

From right to left we proceed by induction on \(k\). For \(k = 0\), we must have \(e = v\). By inspection of the cost evaluation rules we may check that \(v \downarrow^0 v\) for every value \(v\). For \(k = k' + 1\), we must show that if \(e \rightarrow e'\) and \(e' \downarrow^{k'} v\), then \(e \downarrow^k v\). This is proved by a subsidiary induction on the transition rules. For example, suppose that \(e = e_1 (e_2) \rightarrow e'_1 (e_2) = e'\), with \(e_1 \rightarrow e'_1\). By hypothesis \(e'_1 (e_2) \downarrow^k v\), so \(k = k_1 + k_2 + k_3 + 1\), where

1. \(e'_1 \downarrow^{k_1} v_1\),
2. \(e_2 \downarrow^{k_2} v_2\),
3. \(v_1 = \text{fun } f (x: \tau_1) : \tau_2 \text{ is e end}\),
4. \(\{v_1, v_2/f, x\} e \downarrow^{k_3} v\).

By induction \(e_1 \downarrow^{k_1 + 1} v_1\), hence \(e \downarrow^{k+1} v\), as required.

\[\square\]
Chapter 17

Implicit Parallelism

In this chapter we study the extension of MinML with implicit data parallelism, a means of speeding up computations by allowing expressions to be evaluated simultaneously. By “implicit” we mean that the use of parallelism is invisible to the programmer as far as the ultimate results of computation are concerned. By “data parallel” we mean that the parallelism in a program arises from the simultaneous evaluation of the components of a data structure.

Implicit parallelism is very natural in an effect-free language such as MinML. The reason is that in such a language it is not possible to determine the order in which the components of an aggregate data structure are evaluated. They might be evaluated in an arbitrary sequential order, or might even be evaluated simultaneously, without affecting the outcome of the computation. This is in sharp contrast to effect-ful languages, for then the order of evaluation, or the use of parallelism, is visible to the programmer. Indeed, dependence on the evaluation order must be carefully guarded against to ensure that the outcome is determinate.

17.1 Tuple Parallelism

We begin by considering a parallel semantics for tuples according to which all components of a tuple are evaluated simultaneously. For simplicity we consider only pairs, but the ideas generalize in a straightforward manner to tuples of any size. Since the “widths” of tuples are specified statically as part of their type, the amount of parallelism that can be induced in any one step is bounded by a static constant. In Section ?? we will consider a mechanism by which the amount of parallelism generated at each step is
determined dynamically.

To facilitate comparison, we will consider two operational semantics for this extension of MinML, the sequential and the parallel. The sequential semantics is as in Chapter 19. However, we now write \( e \mapsto_{seq} e' \) for the transition relation to stress that this is the sequential semantics. The sequential evaluation rules for pairs are as follows:

\[
\begin{align*}
  (e_1, e_2) & \mapsto_{seq} (e'_1, e_2) \\
  (v_1, e_2) & \mapsto_{seq} (v_1, e'_2) \\
  \text{split } (v_1, v_2) & \text{ as } (x, y) \text{ in } e \end{align*}
\]

(17.1) (17.2) (17.3)

The parallel semantics is similar, except that we evaluate both components of a pair simultaneously whenever this is possible. This leads to the following rules:

\[
\begin{align*}
  (e_1, e_2) & \mapsto_{par} (e'_1, e'_2) \\
  (e_1, v_2) & \mapsto_{par} (e'_1, v_2) \\
  (v_1, e_2) & \mapsto_{par} (v_1, e'_2)
\end{align*}
\]

(17.5) (17.6) (17.7)

Three rules are required to account for the possibility that evaluation of one component may complete before the other.

When presented two semantics for the same language, it is natural to ask whether they are equivalent. They are, in the sense that both semantics deliver the same value for any expression. This is the precise statement of what we mean by “implicit parallelism”.

\[1\] It might be preferable to admit progress on either \( e_1 \) or \( e_2 \) alone, without requiring the other to be a value.
Theorem 17.1

For every closed, well-typed expression $e$, $e \xrightarrow{*_{\text{seq}}} v \iff e \xrightarrow{*_{\text{par}}} v$.

Proof: For the implication from left to right, it suffices to show that if $e \xrightarrow{*_{\text{seq}}} e' \xrightarrow{*_{\text{par}}} v$, then $e \xrightarrow{*_{\text{par}}} v$. This is proved by induction on the sequential evaluation relation. For example, suppose that $(e_1, e_2) \xrightarrow{*_{\text{seq}}} (e'_1, e'_2) \xrightarrow{*_{\text{par}}} (v_1, v_2)$,

where $e_1 \xrightarrow{*_{\text{seq}}} e'_1$. By inversion of the parallel evaluation sequence, we have $e'_1 \xrightarrow{*_{\text{par}}} v_1$ and $e_2 \xrightarrow{*_{\text{par}}} v_2$. Hence, by induction, $e_1 \xrightarrow{*_{\text{par}}} v_1$, from which it follows immediately that $(e_1, e_2) \xrightarrow{*_{\text{par}}} (v_1, v_2)$. The other case of sequential evaluation for pairs is handled similarly. All other cases are immediate since the sequential and parallel semantics agree on all other constructs.

For the other direction, it suffices to show that if $e \xrightarrow{*_{\text{par}}} e' \xrightarrow{*_{\text{seq}}} v$, then $e \xrightarrow{*_{\text{seq}}} v$. We proceed by induction on the definition of the parallel evaluation relation. For example, suppose that we have

$(e_1, e_2) \xrightarrow{*_{\text{par}}} (e'_1, e'_2) \xrightarrow{*_{\text{seq}}} (v_1, v_2)$

with $e_1 \xrightarrow{*_{\text{par}}} e'_1$ and $e_2 \xrightarrow{*_{\text{par}}} e'_2$. We are to show that $(e_1, e_2) \xrightarrow{*_{\text{seq}}} (v_1, v_2)$. Since $(e'_1, e'_2) \xrightarrow{*_{\text{seq}}} (v_1, v_2)$, it follows that $e'_1 \xrightarrow{*_{\text{seq}}} v_1$ and $e'_2 \xrightarrow{*_{\text{seq}}} v_2$. By induction $e_1 \xrightarrow{*_{\text{seq}}} v_1$ and $e_2 \xrightarrow{*_{\text{seq}}} v_2$, which is enough for the result. The other cases of evaluation for pairs are handled similarly.

One important consequence of this theorem is that parallelism is semantically invisible: whether we use parallel or sequential evaluation of pairs, the result is the same. Consequently, parallelism may safely be left implicit, at least as far as correctness is concerned. However, as one might expect, parallelism effects the efficiency of programs.

17.2 Work and Depth

An operational semantics for a language induces a measure of time complexity for expressions, namely the number of steps required to evaluate that expression to a value. The sequential complexity of an expression is its time complexity relative to the sequential semantics; the parallel complexity is its time complexity relative to the parallel semantics. These can, in general, be quite different. Consider, for example, the following naïve implementation of the Fibonacci sequence in MinML with products:
fun fib (n:int):int is
  if n=0 then 1
  else if n=1 then 1
  else plus(fib(n-1),fib(n-2)) fi fi

where plus is the following function on ordered pairs:

fun plus (p:int*int):int is
  split p as (m:int,n:int) in m+n

The sequential complexity of fib n is $O(2^n)$, whereas the parallel complexity of the same expression is $O(n)$. The reason is that each recursive call spawns two further recursive calls which, if evaluated sequentially, lead to an exponential number of steps to complete. However, if the two recursive calls are evaluated in parallel, then the number of parallel steps to completion is bounded by $n$, since $n$ is decreased by 1 or 2 on each call. Note that the same number of arithmetic operations is performed in each case! The difference is only in whether they are performed simultaneously.

This leads naturally to the concepts of work and depth. The work of an expression is the total number of primitive instruction steps required to complete evaluation. Since the sequential semantics has the property that each rule has at most one premise, each step of the sequential semantics amounts to the execution of exactly one instruction. Therefore the sequential complexity coincides with the work required. (Indeed, work and sequential complexity are often taken to be synonymous.) The work required to evaluate fib n is $O(2^n)$.

On the other hand the depth of an expression is the length of the longest chain of sequential dependencies in a complete evaluation of that expression. A sequential dependency is induced whenever the value of one expression depends on the value of another, forcing a sequential evaluation ordering between them. In the Fibonacci example the two recursive calls have no sequential dependency among them, but the function itself sequentially depends on both recursive calls — it cannot return until both calls have returned. Since the parallel semantics evaluates both components of an ordered pair simultaneously, it exactly captures the independence of the two calls from each, but the dependence of the result on both. Thus the parallel complexity coincides with the depth of the computation. (Indeed, they are often taken to be synonymous.) The depth of the expression fib n is $O(n)$.

With this in mind, the cost semantics introduced in Chapter 16 may be extended to account for parallelism by specifying both the work and the
depth of evaluation. The judgements of the parallel cost semantics have
the form \( e \Downarrow_{w,d} v \), where \( w \) is the work and \( d \) the depth. For all cases but
evaluation of pairs the work and the depth track one another. The rule for
pairs is as follows:

\[
\frac{e_1 \Downarrow_{w_1,d_1} v_1 \quad e_2 \Downarrow_{w_2,d_2} v_2}{(e_1, e_2) \Downarrow_{w_1+w_2, \max(d_1,d_2)} (v_1, v_2)}
\]  

(17.8)

The remaining rules are easily derived from the sequential cost semantics,
with both work and depth being additively combined at each step.\(^2\)

The correctness of the cost semantics states that the work and depth
costs are consistent with the sequential and parallel complexity, respec-
tively, of the expression.

**Theorem 17.2**

For any closed, well-typed expression \( e \), \( e \Downarrow_{w,d} v \) iff \( e \Downarrow_{\text{seq}} v \) and \( e \Downarrow_{\text{par}} v \).

**Proof:** From left to right, we proceed by induction on the cost semantics.
For example, we must show that if \( e_1 \Downarrow_{d_1} v_1 \) and \( e_2 \Downarrow_{d_2} v_2 \), then

\[
(e_1, e_2) \Downarrow_{d} (v_1, v_2),
\]

where \( d = \max(d_1, d_2) \). Suppose that \( d = d_2 \), and let \( d' = d - d_1 \) (the case
\( d = d_1 \) is handled similarly). We have \( e_1 \Downarrow_{d_1} v_1 \) and \( e_2 \Downarrow_{d_2} v_2 \).
It follows that

\[
(e_1, e_2) \Downarrow_{d_1} (v_1, e'_2) \\
\Downarrow_{d_2} (v_1, v_2).
\]

For the converse, we proceed by considering work and depth costs sep-
arily. For work, we proceed as in Chapter 16. For depth, it suffices
to show that if \( e \Downarrow_{\text{par}} e' \) and \( e' \Downarrow_{d} v \), then \( e \Downarrow_{d+1} v \).\(^3\) For example,
suppose that \( (e_1, e_2) \Downarrow_{\text{par}} (e'_1, e'_2) \), with \( e_1 \Downarrow_{\text{par}} e'_1 \) and \( e_2 \Downarrow_{\text{par}} e'_2 \).
Since \( (e'_1, e'_2) \Downarrow_{d} v \), we must have \( v = (v_1, v_2) \), \( d = \max(d_1, d_2) \) with
\( e'_1 \Downarrow_{d_1} v_1 \) and \( e'_2 \Downarrow_{d_2} v_2 \). By induction \( e_1 \Downarrow_{d_1+1} v_1 \) and \( e_2 \Downarrow_{d_2+1} v_2 \) and hence
\( (e_1, e_2) \Downarrow_{d+1} (v_1, v_2) \), as desired. \(\blacksquare\)

\(^2\)If we choose, we might evaluate arguments of primop’s in parallel, in which case the
depth complexity would be calculated as one more than the maximum of the depths of its
arguments. We will not do this here since it would only complicate the development.

\(^3\)The work component of the cost is suppressed here for the sake of clarity.
17.3 Vector Parallelism

To support vector parallelism we will extend MinML with a type of vectors, which are finite sequences of values of a given type whose length is not determined until execution time. The primitive operations on vectors are chosen so that they may be executed in parallel on a shared memory multiprocessor, or SMP, in constant depth for an arbitrary vector.

The following primitives are added to MinML to support vectors:

\[
\begin{align*}
\text{Types} & \quad \tau ::= \tau \text{ vector} \\
\text{Expr's} & \quad e ::= [e_0, \ldots, e_{n-1}] \mid \text{elt}(e_1, e_2) \mid \text{size}(e) \mid \text{index}(e) \mid \text{map}(e_1, e_2) \mid \text{update}(e_1, e_2) \\
\text{Values} & \quad v ::= [v_0, \ldots, v_{n-1}] 
\end{align*}
\]

These expressions may be informally described as follows. The expression \([e_0, \ldots, e_{n-1}]\) evaluates to an \(n\)-vector whose elements are given by the expressions \(e_i, 0 \leq i < n\). The operation \(\text{elt}(e_1, e_2)\) retrieves the element of the vector given by \(e_1\) at the index given by \(e_2\). The operation \(\text{size}(e)\) returns the number of elements in the vector given by \(e\). The operation \(\text{index}(e)\) creates a vector of length \(n\) (given by \(e\)) whose elements are \(0, \ldots, n - 1\). The operation \(\text{map}(e_1, e_2)\) applies the function given by \(e_1\) to every element of \(e_2\) in parallel. Finally, the operation \(\text{update}(e_1, e_2)\) yields a new vector of the same size, \(n\), as the vector \(v\) given by \(e_1\), but whose elements are updated according to the vector \(v'\) given by \(e_2\). The elements of \(e_2\) are triples of the form \((b, i, x)\), where \(b\) is a boolean flag, \(i\) is a non-negative integer less than or equal to \(n\), and \(x\) is a value, specifying that the \(i\)th element of \(v\) should be replaced by \(x\), provided that \(b = \text{true}\).

The static semantics of these primitives is given by the following typing rules:

\[
\begin{align*}
\Gamma \vdash e_1 : \tau & \quad \ldots \quad \Gamma \vdash e_n : \tau \\
\Gamma \vdash [e_0, \ldots, e_{n-1}] : \tau \text{ vector} \\
\Gamma \vdash e_1 : \tau \text{ vector} & \quad \Gamma \vdash e_2 : \text{int} \\
\Gamma \vdash \text{elt}(e_1, e_2) : \tau \\
\Gamma \vdash \text{size}(e) : \text{int} \\
\Gamma \vdash \text{index}(e) : \text{int} \text{ vector} \\
\end{align*}
\]
The parallel dynamic semantics is given by the following rules. The most important is the parallel evaluation rule for vector expressions, since this is the sole source of parallelism:

\[ \forall i \in I \left( e_i \mapsto_{\text{par}} e_i' \right) \quad \forall i \notin I \left( e_i' = e_i \land e_i \text{ value} \right) \]

\[ [e_0, \ldots, e_{n-1}] \mapsto_{\text{par}} [e'_0, \ldots, e'_{n-1}] \]  \hspace{1cm} (17.15)

where \( \emptyset \neq I \subseteq \{0, \ldots, n-1\} \). This allows for the parallel evaluation of all components of the vector that have not yet been evaluated.

For each of the primitive operations of the language there is a rule specifying that its arguments are evaluated in left-to-right order. We omit these rules here for the sake of brevity. The primitive instructions are as follows:

\[ \text{elt}([v_0, \ldots, v_{n-1}], i) \mapsto_{\text{par}} v_i \]  \hspace{1cm} (17.16)

\[ \text{size}([v_0, \ldots, v_{n-1}]) \mapsto_{\text{par}} n \]  \hspace{1cm} (17.17)

\[ \text{index}(n) \mapsto_{\text{par}} [0, \ldots, n-1] \]  \hspace{1cm} (17.18)

\[ \text{map}(v, [v_0, \ldots, v_{n-1}]) \mapsto_{\text{par}} [\text{apply}(v, v_0), \ldots, \text{apply}(v, v_{n-1})] \]  \hspace{1cm} (17.19)

\[ \text{update}([v_0, \ldots, v_{n-1}], [(b_0, i_0, x_0), \ldots, (b_{k-1}, i_{k-1}, x_{k-1})]) \mapsto_{\text{par}} [v'_0, \ldots, v'_{n-1}] \]  \hspace{1cm} (17.20)

where for each \( i \in \{i_0, \ldots, i_{k-1}\} \), if \( b_i \) is \text{true}, then \( v'_i = x_i \), and otherwise \( v'_i = v_i \). If an index \( i \) appears more than once, the rightmost occurrence takes precedence over the others.
The sequential dynamic semantics of vectors is defined similarly to the parallel semantics. The only difference is that vector expressions are evaluated in left-to-right order, rather than in parallel. This is expressed by the following rule:

$$e_i \pto_{seq} e'_i$$

$$[v_0, \ldots, v_{i-1}, e_i, e_{i+1}, \ldots, e_{n-1}] \mapsto [v_0, \ldots, v_{i-1}, e'_i, e_{i+1}, \ldots, e_{n-1}]$$

(17.21)

We write $e \pto_{seq} e'$ to indicate that $e$ steps to $e'$ under the sequential semantics.

With these two basic semantics in mind, we may also derive a cost semantics for MinML with vectors, where the work corresponds to the number of steps required in the sequential semantics, and the depth corresponds to the number of steps required in the parallel semantics. The rules are as follows.

Vector expressions are evaluated in parallel.

$$\forall 0 \leq i < n (e_i \downarrow^{w_i,d_i} v_i)$$

$$[e_0, \ldots, e_{n-1}] \downarrow^{w,d} [v_0, \ldots, v_{n-1}]$$

(17.22)

where $w = \sum_{i=0}^{n-1} w_i$ and $d = \max_{i=0}^{n-1} d_i$.

Retrieving an element of a vector takes constant work and depth.

$$e_1 \downarrow^{w_1,d_1} [v_0, \ldots, v_{n-1}] \quad e_2 \downarrow^{w_2,d_2} i \quad (0 \leq i < n)$$

$$\text{elt}(e_1, e_2) \downarrow^{w_1+w_2+1,d_1+d_2+1} v_i$$

(17.23)

Retrieving the size of a vector takes constant work and depth.

$$e \downarrow^{w,d} [v_0, \ldots, v_{n-1}]$$

$$\text{size}(e) \downarrow^{w+1,d+1} n$$

(17.24)

Creating an index vector takes linear work and constant depth.

$$e \downarrow^{w,d} n$$

$$\text{index}(e) \downarrow^{w+n,d+1} [0, \ldots, n-1]$$

(17.25)
17.3 Vector Parallelism

Mapping a function across a vector takes constant work and depth beyond the cost of the function applications.

\[
\begin{align*}
\text{map}(e_1, e_2) & \Downarrow^{w_1+w_2+w+1,d_1+d_2+d+1}[v'_0, \ldots, v'_{n-1}] \\
& \Downarrow^{w_1,d_1}[\text{apply}(v, v_0), \ldots, \text{apply}(v, v_{n-1})] \\
& \Downarrow^{w_2,d_2}[v_0, \ldots, v_{n-1}] \\
& \Downarrow^{w_1,d_1}[v, v_0, \ldots, v_{n-1}] \\
& \Downarrow^{w_2,d_2}[v_0, \ldots, v_{n-1}] \\
\end{align*}
\]

(17.26)

Updating a vector takes linear work and constant depth.

\[
\begin{align*}
\text{update}(e_1, e_2) & \Downarrow^{w_1+w_2+w+1,d_1+d_2+d+1}[v'_0, \ldots, v'_{n-1}] \\
& \Downarrow^{w_1,d_1}[v_0, \ldots, v_{n-1}] \\
& \Downarrow^{w_2,d_2}[v_0, \ldots, v_{n-1}] \\
& \Downarrow^{w_1+d_1}[v, v_0, \ldots, v_{n-1}] \\
\end{align*}
\]

(17.27)

where for each \(i \in \{i_1, \ldots, i_k\}\), if \(b_i\) is true, then \(v'_i = x_i\), and otherwise \(v'_i = v_i\). If an index \(i\) appears more than once, the rightmost occurrence takes precedence over the others.

Theorem 17.3

For the extension of MinML with vectors, \(e \Downarrow^w v\) iff \(e \rightarrow^d_{\text{par}} v\) and \(e \rightarrow^w_{\text{seq}} v\).
The parallel operational semantics described in Chapter 17 abstracts away some important aspects of the implementation of parallelism. For example, the parallel evaluation rule for ordered pairs

\[
\frac{e_1 \rightarrow_{\text{par}} e'_1 \quad e_2 \rightarrow_{\text{par}} e'_2}{(e_1, e_2) \rightarrow_{\text{par}} (e'_1, e'_2)}
\]

does not account for the overhead of allocating \( e_1 \) and \( e_2 \) to two (physical or virtual) processors, or for synchronizing with those two processors to obtain their results. In this chapter we will discuss a more realistic operational semantics that accounts for this overhead.

### 18.1 A Simple Parallel Language

Rather than specify which primitives, such as pairing, are to be evaluated in parallel, we instead introduce a “parallel let” construct that allows the programmer to specify the simultaneous evaluation of two expressions. Moreover, we restrict the language so that the arguments to all primitive operations must be values. This forces the programmer to decide for herself which constructs are to be evaluated in parallel, and which are to be evaluated sequentially.
The binding conventions are as for MinML with product types, with the additional specification that the variables \(x_1\) and \(x_2\) are bound within the body of a let expression. Note that variables are regarded as values only for the purpose of defining the syntax of the language; evaluation is, as ever, defined only on closed terms.

As will become apparent when we specify the dynamic semantics, the “sequential let” is definable from the “parallel let”:

\[
\text{let } x_1 : \tau_1 \text{ be } e_1 \text{ in } e_2 \text{ end} := \text{let } x_1 : \tau_1 \text{ be } e_1 \text{ and } x : \text{unit be } () \text{ in } e_2 \text{ end}
\]

where \(x\) does not occur free in \(e_2\). Using these, the “parallel pair” is definable by the equation

\[
(e_1, e_2)_{\text{par}} := \text{let } x_1 : \tau_1 \text{ be } e_1 \text{ and } x_2 : \tau_2 \text{ be } e_2 \text{ in } (x_1, x_2) \text{ end}
\]

whereas the “(left-to-right) sequential pair” is definable by the equation

\[
(e_1, e_2)_{\text{seq}} := \text{let } x_1 : \tau_1 \text{ be } e_1 \text{ in } \text{let } x_2 : \tau_2 \text{ be } e_2 \text{ in } (x_1, x_2) \text{ end} \text{ end}.
\]

The static semantics of this language is essentially that of MinML with product types, with the addition of the following typing rule for the parallel let construct:

\[
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e : \tau}{\Gamma \vdash \text{let } x_1 : \tau_1 \text{ be } e_1 \text{ and } x_2 : \tau_2 \text{ be } e_2 \text{ in } e \text{ end} : \tau}
\] (18.1)

It is a simple exercise to give a parallel structured operational semantics to this language in the style of Chapter 17. In particular, it would employ the following rules for the parallel let construct.

\[
\begin{array}{c}
\text{let } x_1 : \tau_1 \text{ be } e_1 \text{ and } x_2 : \tau_2 \text{ be } e_2 \text{ in } e \text{ end} \\
\hline
\hline
\text{let } x_1 : \tau_1 \text{ be } e'_1 \text{ and } x_2 : \tau_2 \text{ be } e'_2 \text{ in } e \text{ end}
\end{array}
\]

(18.2)

\[
\begin{array}{c}
\text{let } x_1 : \tau_1 \text{ be } v_1 \text{ and } x_2 : \tau_2 \text{ be } e_2 \text{ in } e \text{ end} \\
\hline
\hline
\text{let } x_1 : \tau_1 \text{ be } e'_1 \text{ and } x_2 : \tau_2 \text{ be } v_2 \text{ in } e \text{ end}
\end{array}
\]

(18.3)

\[
\begin{array}{c}
\text{let } x_1 : \tau_1 \text{ be } v_1 \text{ and } x_2 : \tau_2 \text{ be } e_2 \text{ in } e \text{ end} \\
\hline
\hline
\text{let } x_1 : \tau_1 \text{ be } e'_1 \text{ and } x_2 : \tau_2 \text{ be } v'_2 \text{ in } e \text{ end}
\end{array}
\]

(18.4)
However, these rules ignore the overhead associated with allocating the sub-expression to processors. In the next section we will consider an abstract machine that accounts for this overhead.

Exercise 18.1

Prove preservation and progress for the static and dynamic semantics just given.

18.2 A Parallel Abstract Machine

The essence of parallelism is the simultaneous execution of several programs. Each execution is called a thread of control, or thread, for short. The problem of devising a parallel abstract machine is how to represent multiple threads of control, in particular how to represent the creation of new threads and synchronization between threads. The P-machine is designed to represent a parallel computer with an unbounded number of processors in a simple and elegant manner.

The main idea of the P-machine is to represent the state of a parallel computer by a nested composition of parallel let statements representing the active threads in a program. Each step of the machine consists of executing all of the active instructions in the program, resulting in a new P-state.

In order to account for the activation of threads and the synchronization of their results we make explicit the process of activating an expression, which corresponds to assigning it to a processor for execution. Execution of a parallel let instruction whose constituent expressions have not yet been activated consists of the activation of these expressions. Execution of a parallel let whose constituents are completely evaluated consists of substituting the values of these expressions into the body of the let, which is itself then activated. Execution of all other instructions is exactly as before, with the result being made active in each case.

This can be formalized using parallelism contexts, which capture the tree structure of nested parallel computations. Let $l$ and variants range over a countable set of labels. These will serve to identify the abstract processors assigned to the execution of an active expression. The set of parallelism contexts $\mathcal{L}$ is defined by the following grammar:

$$
\mathcal{L} ::= l: \Box | l:\text{let } x_1 : \tau_1 \text{ be } \mathcal{L}_1 \text{ and } x_2 : \tau_2 \text{ be } \mathcal{L}_2 \text{ in } e \text{ end}
$$

$$
l:\text{let } x_1 : \tau_1 \text{ be } v_1 \text{ and } x_2 : \tau_2 \text{ be } \mathcal{L}_2 \text{ in } e \text{ end}
$$
A parallelism context is well-formed only if all labels occurring within it are distinct; hereafter we will consider only well-formed parallelism contexts.

A labelled “hole” in a parallelism context represents an active computation site; a labelled let expression represents a pending computation that is awaiting completion of its child threads. We have arranged things so that all active sites are children of pending sites, reflecting the intuition that an active site must have been spawned by some (now pending) site.

The arity of a context is defined to be the number of “holes” occurring within it. The arity is therefore the number of active threads within the context. If } \mathcal{L} \text{ is a context with arity } n, \text{ then the expression } \mathcal{L}[l=e_i^n] \text{ represents the result of “filling” the hole labelled } l_i \text{ with the expression } e_i, \text{ for each } 1 \leq i \leq n. \text{ Thus the } e_i's \text{ represent the active expressions within the context; the label } l_i \text{ represents the “name” of the processor assigned to execute } e_i.

Each step of the P-machine consists of executing all of the active instructions in the current state. This is captured by the following evaluation rule:

\[
e_1 \rightarrow e'_1 \quad \cdots \quad e_n \rightarrow e'_n \quad \frac{\mathcal{L}[l=e_i^n] \mapsto \mathcal{L}[l=e'_i^n]}{}
\]

The relation } e \rightarrow e' \text{ defines the atomic instruction steps of the P-machine. These are defined by a set of axioms. The first is the fork axiom, which initiates execution of a parallel let statement:

\[
\text{let } x_1: \tau_1 \text{ be } e_1 \text{ and } x_2: \tau_2 \text{ be } e_2 \text{ in } e \text{ end} \rightarrow \text{let } x_1: l_1: e_1 \text{ and } x_2: l_2: e_2 \text{ in } e \text{ end} \tag{18.5}
\]

Here } l_1 \text{ and } l_2 \text{ are “new” labels that do not otherwise occur in the computation. They serve as the labels of the processors assigned to execute } e_1 \text{ and } e_2, \text{ respectively.

The second instruction is the join axiom, which completes execution of a parallel let:

\[
\text{let } x_1: \tau_1 \text{ be } l_1: v_1 \text{ and } x_2: \tau_2 \text{ be } l_2: v_2 \text{ in } e \text{ end} \rightarrow \{v_1, v_2/x_1, x_2\} e \tag{18.6}
\]

The other instructions are inherited from the M-machine. For example, function application is defined by the following instruction:

\[
\text{let } x_1: \tau_1 \text{ be } l_1: v_1 \text{ and } x_2: \tau_2 \text{ be } l_2: v_2 \text{ in } e \text{ end} \rightarrow \{v_1, v_2/f, x\} e \tag{18.7}
\]
This completes the definition of the P-machine.

**Exercise 18.2**

*State and prove preservation and progress relative to the P-machine.*

### 18.3 Cost Semantics, Revisited

A primary motivation for introducing the P-machine was to achieve a proper accounting for the cost of creating and synchronizing threads. In the simplified model of Chapter 17 we ignored these costs, but here we seek to take them into account. This is accomplished by taking the following rule for the cost semantics of the parallel \texttt{let} construct:

\[
\begin{align*}
\frac{e_1 \parallel^{w_1,d_1} v_1 \quad e_2 \parallel^{w_2,d_2} v_2 \quad \{v_1,v_2/x_1,x_2\} e \parallel^{w,d} v}{\text{let } x_1 : \tau_1 \text{ be } e_1 \text{ and } x_2 : \tau_2 \text{ be } e_2 \text{ in end } \parallel^{w',d'} v}
\end{align*}
\]

where \(w' = w_1 + w_2 + w + 2\) and \(d' = \max(d_1,d_2) + d + 2\). Since the remaining expression forms are all limited to values, they have unit cost for both work and depth.

The calculation of work and depth for the parallel \texttt{let} construct is justified by relating the cost semantics to the P-machine. The work performed in an evaluation sequence \(e \mapsto_P^* v\) is the total number of primitive instruction steps performed in the sequence; it is the sequential cost of executing the expression \(e\).

**Theorem 18.3**

*If \(e \parallel^{w,d} v\), then \(l : e \mapsto_P^* l : v\) with work \(w\).*

**Proof:** The proof from left to right proceeds by induction on the cost semantics. For example, consider the cost semantics of the parallel \texttt{let} construct. By induction we have

1. \(l_1 : e_1 \mapsto_P^{d_1} l_1 : v_1\) with work \(w_1\);

2. \(l_2 : e_2 \mapsto_P^{d_2} l_2 : v_2\) with work \(w_2\);

3. \(l : \{v_1,v_2/x_1,x_2\} e \mapsto_P^* l : v\) with work \(w\).
We therefore have the following P-machine evaluation sequence:

\[
\begin{align*}
& l: \text{let } x_1 : \tau_1 \text{ be } e_1 \text{ and } x_2 : \tau_2 \text{ be } e_2 \text{ in } e \text{ end} \quad \mapsto_p \max(d_1, d_2) \\
& l: \text{let } x_1 : \tau_1 \text{ be } \nu_1 \text{ and } x_2 : \tau_2 \text{ be } \nu_2 \text{ in } e \text{ end} \quad \mapsto_p d \\
& l: \{\nu_1, \nu_2 / x_1, x_2\} e \quad \mapsto_d \\
& l: v
\end{align*}
\]

The total length of the evaluation sequence is \(\max(d_1, d_2) + d + 2\), as required by the depth cost, and the total work is \(w_1 + w_2 + w + 2\), as required by the work cost.

18.4 Provable Implementations (Summary)

The semantics of parallelism given above is based on an idealized parallel computer with an unlimited number of processors. In practice this idealization must be simulated using some fixed number, \(p\), of physical processors. In practice \(p\) is on the order of 10’s of processors, but may even rise (at the time of this writing) into the 100’s. In any case \(p\) does not vary with input size, but is rather a fixed parameter of the implementation platform. The important question is how efficiently can one simulate unbounded parallelism using only \(p\) processors? That is, how realistic are the costs assigned to the language by our semantics? Can we make accurate predictions about the running time of a program on a real parallel computer based on the idealized cost assigned to it by our semantics?

The answer is yes, through the notion of a provably efficient implementation. While a full treatment of these ideas is beyond the scope of this book, it is worthwhile to summarize the main ideas.

**Theorem 18.4 (Blelloch and Greiner)**

If \(e \not\equiv_{w,d} v\), then \(e\) can be evaluated on an SMP with \(p\)-processors in time \(O(w/p + d \lg p)\).

For our purposes, an SMP is any of a wide range of parallel computers, including a CRCW PRAM, a hypercube, or a butterfly network. Observe that for \(p = 1\), the stated bound simplifies to \(O(w)\), as would be expected.

To understand the significance of this theorem, observe that the definition of work and depth yields a lower bound of \(\Omega(\max(w/p, d))\) on the execution time on \(p\) processors. We can never complete execution in fewer
than $d$ steps, and can, at best, divide the total work evenly among the $p$ processors. The theorem tells us that we can come within a constant factor of this lower bound. The constant factor, $\lg p$, represents the overhead of scheduling parallel computations on $p$ processors.

The goal of parallel programming is to maximize the use of parallelism so as to minimize the execution time. By the theorem this will occur if the term $w/p$ dominates, which occurs if the ratio $w/d$ of work to depth is at least $p \lg p$. This ratio is sometimes called the parallelizability of the program. For highly sequential programs, $d$ is directly proportional to $w$, yielding a low parallelizability — increasing the number of processors will not speed up the computation. For highly parallel programs, $d$ might be constant or proportional to $\lg w$, resulting in a large parallelizability, and good utilization of the available computing resources. It is important to keep in mind that it is not known whether there are inherently sequential problems (for which no parallelizable solution is possible), or whether, instead, all problems can benefit from parallelism. The best that we can say at the time of this writing is that there are problems for which no parallelizable solution is known.

To get a sense of what is involved in the proof of Blelloch and Greiner’s theorem, let us consider the assumption that the index operation on vectors (given in Chapter 17) has constant depth. The theorem implies that index is implementable on an SMP in time $O(n/p + \lg p)$. We will briefly sketch a proof for this one case. The main idea is that we may assume that every processor is assigned a unique number from 0 to $p - 1$. To implement index, we simply allocate, but do not initialize, a region of memory of the appropriate size, and ask each processor to simultaneously store its identifying number $i$ into the $i$th element of the allocated array. This works directly if the size of the vector is no more than the number of processors. Otherwise, we may divide the problem in half, and recursively build two index vectors of half the size, one starting with zero, the other with $n/2$. This process need proceed at most $\lg p$ times before the vectors are small enough, leaving $n/p$ sub-problems of size at most $p$ to be solved. Thus the total time required is $O(n/p + \lg p)$, as required by the theorem.

The other primitive operations are handled by similar arguments, justifying the cost assignments made to them in the operational semantics. To complete the proof of Blelloch and Greiner’s theorem, we need only argue that the total work $w$ can indeed be allocated to $p$ processors with a cost of only $\lg p$ for the overhead. This is a consequence of Brent’s Theorem, which states that a total workload $w$ divided into $d$ parallel steps may be
implemented on $p$ processors in $O(n/p + d \log p)$ time. The argument relies on certain assumptions about the SMP, including the ability to perform a parallel fetch-and-add operation in constant time.
Part VIII

Data Structures and Abstraction
Chapter 19

Aggregate Data Structures

It is interesting to add to MinML support for programming with aggregate data structures such as \( n \)-tuples, lists, and tree structures. We will decompose these familiar data structures into three types:

1. **Product (or tuple) types.** In general these are types whose values are \( n \)-tuples of values, with each component of a specified type. We will study two special cases that are sufficient to cover the general case: 0-tuples (also known as the unit type) and 2-tuples (also known as ordered pairs).

2. **Sum (or variant or union) types.** These are types whose values are values of one of \( n \) specified types, with an explicit “tag” indicating which of the \( n \) choices is made.

3. **Recursive types.** These are “self-referential” types whose values may have as constituents values of the recursive type itself. Familiar examples include lists and trees. A non-empty list consists of a value at the head of the list together with another value of list type.

### 19.1 Products

The first-order abstract syntax associated with nullary and binary product types is given by the following grammar:

\[
\begin{align*}
\text{Types} & \quad \tau ::= \text{unit} | \tau_1 \times \tau_2 \\
\text{Expressions} & \quad e ::= () | \check{e_1 \text{ is } e_2 \text{ end}} | (e_1, e_2) | \text{split } e_1 \text{ as } (x, y) \text{ in } e_2 \text{ end}
\end{align*}
\]

\[
\begin{align*}
\text{Values} & \quad v ::= () | (v_1, v_2)
\end{align*}
\]
The higher-order abstract syntax is given by stipulating that in the expression \( \text{split} \ e_1 \text{ as } (x, y) \text{ in } e_2 \text{ end} \) the variables \( x \) and \( y \) are bound within \( e_2 \), and hence may be renamed (consistently, avoiding capture) at will without changing the interpretation of the expression.

The static semantics of these constructs is given by the following typing rules:

\[
\Gamma \vdash () : \text{unit} \tag{19.1}
\]

\[
\Gamma \vdash e_1 : \text{unit} \quad \Gamma \vdash e_2 : \tau_2
\]

\[
\Gamma \vdash \text{check} \ e_1 \text{ is } () \text{ in } e_2 \text{ end} : \tau_2 \tag{19.2}
\]

\[
\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2
\]

\[
\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2 \tag{19.3}
\]

\[
\Gamma \vdash e_1 : \tau_1 * \tau_2 \quad \Gamma, x: \tau_1, y: \tau_2 \vdash e_2 : \tau
\]

\[
\Gamma \vdash \text{split} \ e_1 \text{ as } (x, y) \text{ in } e_2 \text{ end} : \tau \tag{19.4}
\]

The dynamic semantics is given by these rules:

\[
\text{check } () \text{ is } () \text{ in } e \text{ end} \mapsto e \tag{19.5}
\]

\[
e_1 \mapsto e_1' \quad \text{check } e_1 \text{ is } () \text{ in } e_2 \text{ end} \mapsto \text{check } e_1' \text{ is } () \text{ in } e_2 \text{ end} \tag{19.6}
\]

\[
e_1 \mapsto e_1' \quad (e_1, e_2) \mapsto (e_1', e_2) \tag{19.7}
\]

\[
e_2 \mapsto e_2' \quad (v_1, e_2) \mapsto (v_1, e_2') \tag{19.8}
\]

\[
\text{split } (v_1, v_2) \text{ as } (x, y) \text{ in } e \text{ end} \mapsto \{v_1, v_2/x, y\} e \tag{19.9}
\]

\[
e_1 \mapsto e_1' \quad \text{split } e_1 \text{ as } (x, y) \text{ in } e_2 \text{ end} \mapsto \text{split } e_1' \text{ as } (x, y) \text{ in } e_2 \text{ end} \tag{19.10}
\]

**Exercise 19.1**

State and prove the soundness of this extension to MinML.
Exercise 19.2
A variation is to treat any pair \((e_1, e_2)\) as a value, regardless of whether or not \(e_1\) or \(e_2\) are values. Give a precise formulation of this variant, and prove it sound.

Exercise 19.3
It is also possible to formulate a direct treatment of \(n\)-ary product types (for \(n \geq 0\)), rather than to derive them from binary and nullary products. Give a direct formalization of \(n\)-ary products. Be careful to get the cases \(n = 0\) and \(n = 1\) right!

Exercise 19.4
Another variation is to consider labelled products in which the components are accessed directly by referring to their labels (in a manner similar to C’s struct’s). Formalize this notion.

19.2 Sums

The first-order abstract syntax of nullary and binary sums is given by the following grammar:

**Types**
\[ \tau ::= \tau_1 + \tau_2 \]

**Expressions**
\[ e ::= \mathrm{inl}_{\tau_1 + \tau_2}(e_1) \mid \mathrm{inr}_{\tau_1 + \tau_2}(e_2) \mid \]
\[ \text{case}_\tau e_0 \text{ of } \mathrm{inl}(x: \tau_1) = e_1 \mid \mathrm{inr}(y: \tau_2) = e_2 \text{ end} \]

**Values**
\[ v ::= \mathrm{inl}_{\tau_1 + \tau_2}(v_1) \mid \mathrm{inr}_{\tau_1 + \tau_2}(v_2) \]

The higher-order abstract syntax is given by noting that in the expression
\[ \text{case}_\tau e_0 \text{ of } \mathrm{inl}(x: \tau_1) = e_1 \mid \mathrm{inr}(y: \tau_2) = e_2 \text{ end}, \]
the variable \(x\) is bound in \(e_1\) and the variable \(y\) is bound in \(e_2\).

The typing rules governing these constructs are given as follows:

\[ \Gamma \vdash e_1 : \tau_1 \]
\[ \Gamma \vdash \mathrm{inl}_{\tau_1 + \tau_2}(e_1) : \tau_1 + \tau_2 \] (19.11)

\[ \Gamma \vdash e_2 : \tau_2 \]
\[ \Gamma \vdash \mathrm{inl}_{\tau_1 + \tau_2}(e_2) : \tau_1 + \tau_2 \] (19.12)

\[ \Gamma \vdash e_0 : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau \]
\[ \Gamma \vdash \text{case}_\tau e_0 \text{ of } \mathrm{inl}(x_1 : \tau_1) = e_1 \mid \mathrm{inr}(x_2 : \tau_2) = e_2 \text{ end} : \tau \] (19.13)
The evaluation rules are as follows:

\[ e \mapsto e' \]
\[ \text{inl}_{\tau_1 + \tau_2}(e) \mapsto \text{inl}_{\tau_1 + \tau_2}(e') \] (19.14)

\[ e \mapsto e' \]
\[ \text{inr}_{\tau_1 + \tau_2}(e) \mapsto \text{inr}_{\tau_1 + \tau_2}(e') \] (19.15)

\[
\text{case}_{\tau} \text{inl}_{\tau_1 + \tau_2}(v) \text{ of } \text{inl}(x_1 : \tau_1) = \to e_1 \upharpoonright \text{inr}(x_2 : \tau_2) = \to e_2 \text{ end } \mapsto \{v/x_1\}e_1
\] (19.16)

\[
\text{case}_{\tau} \text{inr}_{\tau_1 + \tau_2}(v) \text{ of } \text{inl}(x_1 : \tau_1) = \to e_1 \upharpoonright \text{inr}(x_2 : \tau_2) = \to e_2 \text{ end } \mapsto \{v/x_2\}e_2
\] (19.17)

Exercise 19.5
State and prove the soundness of this extension.

Exercise 19.6
Consider these variants: \( \text{inl}_{\tau_1 + \tau_2}(e) \) and \( \text{inr}_{\tau_1 + \tau_2}(e) \) are values, regardless of whether or not \( e \) is a value; \( n \)-ary sums; labelled sums.

19.3 Recursive Types

Recursive types are somewhat less familiar than products and sums. Few well-known languages provide direct support for these. Instead the programmer is expected to simulate them using pointers and similar low-level representations. Here instead we’ll present them as a fundamental concept.

As mentioned in the introduction, the main idea of a recursive type is similar to that of a recursive function — self-reference. The idea is easily illustrated by example. Informally, a list of integers may be thought of as either the empty list, \( \text{nil} \), or a non-empty list, \( \text{cons}(h, t) \), where \( h \) is an integer and \( t \) is another list of integers. The operations \( \text{nil} \) and \( \text{cons}(-, -) \) are \textit{value constructors} for the type \( \text{ilist} \) of integer lists. We may program with lists using a form of case analysis, written

\[
\text{listcase } e \text{ of } \text{nil} = \to e_1 \upharpoonright \text{cons}(x, y) = \to e_2 \text{ end},
\]

where \( x \) and \( y \) are bound in \( e_2 \). This construct analyses whether \( e \) is the empty list, in which case it evaluates \( e_1 \), or a non-empty list, with head \( x \) and tail \( y \), in which case it evaluates \( e_2 \) with the head and tail bound to these variables.
Exercise 19.7
Give a formal definition of the type \(\text{ilist}\).

Rather than take lists as a primitive notion, we may define them from a combination of sums, products, and a new concept, recursive types. The essential idea is that the types \(\text{ilist}\) and \(\text{unit} + (\text{int} \times \text{ilist})\) are isomorphic, meaning that there is a one-to-one correspondence between values of type \(\text{ilist}\) and values of the foregoing sum type. In implementation terms we may think of the correspondence “pointer chasing” — every list is a pointer to a tagged value indicating whether or not the list is empty and, if not, a pair consisting of its head and tail. (Formally, there is also a value associated with the empty list, namely the sole value of \(\text{unit}\) type. Since its value is predictable from the type, we can safely ignore it.) This interpretation of values of recursive type as pointers is consistent with the typical low-level implementation strategy for data structures such as lists, namely as pointers to cells allocated on the heap. However, by sticking to the more abstract viewpoint we are not committed to this representation, however suggestive it may be, but can choose from a variety of programming tricks for the sake of efficiency.

Exercise 19.8
Consider the type of binary trees with integers at the nodes. To what sum type would such a type be isomorphic?

This motivates the following general definition of recursive types. The first-order abstract syntax is given by the following grammar:

\[
\begin{align*}
\text{Types} & \quad \tau ::= t \mid \text{rectis}\tau \\
\text{Expressions} & \quad e ::= \text{roll}(e) \mid \text{unroll}(e) \\
\text{Values} & \quad v ::= \text{roll}(v)
\end{align*}
\]

Here \(t\) ranges over a set of type variables, which are used to stand for the recursive type itself, in much the same way that we give a name to recursive functions to stand for the function itself. For the present we will insist that type variables are used only for this purpose; they may occur only inside of a recursive type, where they are bound by the recursive type constructor itself.

For example, the type \(\tau = \text{rectisunit} + (\text{int} \times t)\) is the recursive type of lists of integers. It is isomorphic to its unrolling, the type

\(\text{unit} + (\text{int} \times \tau)\).

This is the isomorphism described informally above.
The abstract “pointers” witnessing the isomorphism are written \( \text{roll}(e) \), which “allocates” a pointer to (the value of) \( e \), and \( \text{unroll}(e) \), which “chases” the pointer given by (the value of) \( e \) to recover its underlying value. This interpretation will become clearer once we have given the static and dynamic semantics of these constructs.

The static semantics of these constructs is given by the following rules:

\[
\Gamma \vdash e : \{\text{rectis} \tau / t\} \tau \\
\Gamma \vdash \text{roll}(e) : \text{rectis} \tau
\]  
(19.18)

\[
\Gamma \vdash e : \text{rectis} \tau \\
\Gamma \vdash \text{unroll}(e) : \{\text{rectis} \tau / t\} \tau
\]  
(19.19)

These primitive operations move back and forth between a recursive type and its unrolling.

The dynamic semantics is given by the following rules:

\[\text{unroll(roll(v))} \leftrightarrow v\]  
(19.20)

\[e \leftrightarrow e' \implies \text{unroll}(e) \leftrightarrow \text{unroll}(e')\]  
(19.21)

\[e \leftrightarrow e' \implies \text{roll}(e) \leftrightarrow \text{roll}(e')\]  
(19.22)

**Exercise 19.9**
State and prove the soundness of this extension of MinML.

**Exercise 19.10**
Consider the definition of the type \( \text{ilist} \) as a recursive type given above. Give definitions of \( \text{nil}, \text{cons}, \) and \( \text{listcase} \) in terms of the operations on recursive types, sums, and products.
Chapter 20

Polymorphism

MinML is an explicitly typed language. The abstract syntax is defined to have sufficient type information to ensure that all expressions have a unique type. In particular the types of the parameters of a function must be chosen when the function is defined.

While this is not itself a serious problem, it does expose a significant weakness in the MinML type system. For example, there is no way to define a generic procedure for composing two functions whose domain and range match up appropriately. Instead we must define a separate composition operation for each choice of types for the functions being composed. Here is one composition function

\[
\begin{align*}
\text{fun} & \quad (f:\text{string->int}):(\text{char->string})\rightarrow(\text{string->int}) \text{ is} \\
& \quad \text{fun} & \quad (g:\text{char->string}):\text{string->int} \text{ is} \\
& \quad \quad \text{fun} & \quad (x:\text{string}):\text{int} \text{ is } \text{apply}(f, \text{apply}(g, x)),
\end{align*}
\]

and here is another

\[
\begin{align*}
\text{fun} & \quad (f:\text{float->double}):(\text{int->float})\rightarrow(\text{int->double}) \text{ is} \\
& \quad \text{fun} & \quad (g:\text{int->float}):\text{int->double} \text{ is} \\
& \quad \quad \text{fun} & \quad (x:\text{int}):\text{double} \text{ is } \text{apply}(f, \text{apply}(g, x)).
\end{align*}
\]

The annoying thing is that both versions of function composition execute the same way; they differ only in the choice of types of the functions being composed. This is rather irksome, and very quickly gets out of hand in practice. Statically typed languages have long been criticized for precisely this reason. Fortunately this inflexibility is not an inherent limitation of statically typed languages, but rather a limitation of the particular type system we have given to MinML. A rather straightforward extension
is sufficient to provide the kind of flexibility that is essential for a practical language. This extension is called *polymorphism*.

While ML has had such a type system from its inception (circa 1978), few other languages have followed suit. Notably the Java language suffers from this limitation (but the difficulty is mitigated somewhat in the presence of subtyping). Plans are in the works, however, for adding polymorphism (called *generics*) to the Java language. A compiler for this extension, called Generic Java, is already available.

### 20.1 Polymorphic MinML

*Polymorphic MinML*, or *PolyMinML*, is an extension of MinML with the ability to define *polymorphic functions*. Informally, a polymorphic function is a function that takes a *type* as argument and yields a *value* as result. The type parameter to a polymorphic function represents an *unknown*, or *generic*, type, which can be instantiated by applying the function to a specific type. The types of polymorphic functions are called *polymorphic types*, or *polytypes*.

A significant design decision is whether to regard polymorphic types as “first-class” types, or whether they are, instead, “second-class” citizens. Polymorphic functions in ML are second-class — they cannot be passed as arguments, returned as results, or stored in data structures. The only thing we may do with polymorphic values is to bind them to identifiers with a `val` or `fun` binding. Uses of such identifiers are automatically instantiated by an implicit polymorphic instantiation. The alternative is to treat polymorphic functions as first-class values, which can be used like any other value in the language. Here there are no restrictions on how they can be used, but you should be warned that doing so precludes using type inference to perform polymorphic abstraction and instantiation automatically.

We’ll set things up for second-class polymorphism by explicitly distinguishing polymorphic types from monomorphic types. The first-class case can then be recovered by simply conflating polytypes and monotypes.
Abstract Syntax

The abstract syntax of PolyMinML is defined by the following extension to the MinML grammar:

- **Polytypes**
  \[ \sigma ::= \tau \mid \forall t(\sigma) \]
- **Monotypes**
  \[ \tau ::= \ldots \mid t \]
- **Expressions**
  \[ e ::= \ldots \mid \text{Fun in e end} \mid \text{inst}(e, \tau) \]
- **Values**
  \[ v ::= \ldots \mid \text{Fun in e end} \]

The variable \( t \) ranges over a set of type variables, which are written ML-style \('a, 'b, and so on\) in examples. In the polytype \( \forall t(\sigma) \) the type variable \( t \) is bound in \( \sigma \); we do not distinguish between polytypes that differ only in the names of bound variables. Since the quantifier can occur only at the outermost level, in ML it is left implicit. An expression of the form \( \text{Fun in e end} \) is a polymorphic function with parameter \( t \) and body \( e \). The variable \( t \) is bound within \( e \). An expression of the form \( \text{inst}(e, \tau) \) is a polymorphic instantiation of the polymorphic function \( e \) at monotype \( \tau \). Notice that we may only instantiate polymorphic functions with monotypes. In examples we write \( f[\tau] \) for polymorphic instantiation, rather than the more verbose \( \text{inst}(f, \tau) \).

We write \( \text{FTV}(\tau) \) (respectively, \( \text{FTV}(\sigma), \text{FTV}(e) \)) for the set of free type variables occurring in \( \tau \) (respectively, \( \sigma, e \)). Capture-avoiding substitution of a monotype \( \tau \) for free occurrences of a type variable \( t \) in a polytype \( \sigma \) (resp., monotype \( \tau' \), expression \( e \)) is written \( \{ \tau/t \} \sigma \) (resp., \( \{ \tau/t \} \tau', \{ \tau/t \} e \)).

Static Semantics

The static semantics of PolyMinML is a straightforward extension to that of MinML. One significant change, however, is that we must now keep track of the scopes of type variables, as well as ordinary variables. In the static semantics of MinML a typing judgement had the form \( \Gamma \vdash e : \tau \), where \( \Gamma \) is a context assigning types to ordinary variables. Only those variables in \( \text{dom} \Gamma \) may legally occur in \( e \). For PolyMinML we must introduce an additional context, \( \Delta \), which is a set of type variables, those that may legally occur in the types and expression of the judgement.

The static semantics consists of rules for deriving the following two judgements:

- \( \Delta \vdash \tau \ok \quad \tau \text{ is a well-formed type in } \Delta \)
- \( \Gamma \vdash \Delta e : \sigma \quad e \text{ is a well-formed expression of type } \sigma \text{ in } \Gamma \text{ and } \Delta \)
The rules for validity of types are as follows:

\[
\frac{t \in \Delta}{\Delta \vdash t \text{ ok}} \tag{20.1}
\]

\[
\Delta \vdash \text{int ok} \tag{20.2}
\]

\[
\Delta \vdash \text{bool ok} \tag{20.3}
\]

\[
\frac{\Delta \vdash \tau_1 \text{ ok} \quad \Delta \vdash \tau_2 \text{ ok}}{\Delta \vdash \tau_1 \rightarrow \tau_2 \text{ ok}} \tag{20.4}
\]

\[
\frac{\Delta \cup \{ t \} \vdash \sigma \text{ ok} \quad t \notin \Delta}{\Delta \vdash \forall \tau(\sigma) \text{ ok}} \tag{20.5}
\]

The auxiliary judgement \( \Delta \vdash \Gamma \) is defined by the following rule:

\[
\frac{\Delta \vdash \Gamma(x) \text{ ok} \quad \forall x \in \text{dom}(\Gamma)}{\Delta \vdash \Gamma \text{ ok}} \tag{20.6}
\]

The rules for deriving typing judgements \( \Gamma \vdash_{\Delta} e : \sigma \) are as follows. We assume that \( \Delta \vdash \Gamma \text{ ok} \), \( \Delta \vdash \sigma \text{ ok} \), \( \text{FV}(e) \subseteq \text{dom}(\Gamma) \), and \( \text{FTV}(e) \subseteq \Delta \). We give only the rules specific to PolyMinML; the remaining rules are those of MinML, augmented with a set \( \Delta \) of type variables.

\[
\frac{\Gamma \vdash_{\Delta \cup \{ \epsilon \} \epsilon : \sigma \quad t \notin \Delta}{\Gamma \vdash_{\Delta} \text{Fun in Fun in Fun in} : \forall \tau(\sigma)} \tag{20.7}
\]

\[
\frac{\Gamma \vdash_{\Delta} e : \forall \tau(\sigma) \quad \Delta \vdash \tau \text{ ok}}{\Gamma \vdash_{\Delta} \text{inst}(e, \tau) : \{ \tau/t \} \sigma} \tag{20.8}
\]

For example, here is the polymorphic composition function in PolyMinML:

```plaintext
Fun t in
    Fun u in
        Fun v in
            fun _ (f:u->v):(t->u)->(t->v) is
                fun _ (g:t->u):t->v is
                    fun _ (x:t):v is apply(f, apply(g, x))
```

WORKING DRAFT

January 14, 2003
It is easy to check that it has type

$$\forall t(\forall u(\forall v((u \rightarrow v) \rightarrow (t \rightarrow u) \rightarrow (t \rightarrow v)))).$$ 

We will need the following technical lemma stating that typing is preserved under instantiation:

**Lemma 20.1 (Instantiation)**
If $\Gamma \vdash \Delta \cup \{t\} e : \sigma$, where $t \notin \Delta$, and $\Delta \vdash \tau$ ok, then $\{\tau/t\} \Gamma \vdash \Delta \{\tau/t\} e : \{\tau/t\} \sigma$.

The proof is by induction on typing, and involves no new ideas beyond what we have already seen.

We will also have need of the following canonical forms lemma:

**Lemma 20.2 (Canonical Forms)**
If $v : \forall t(\sigma)$, then $v = \text{Fun } t \text{ in } e \text{ end}$ for some $t$ and $e$ such that $\emptyset \vdash \{t\} e : \sigma$.

This is proved by a straightforward analysis of the typing rules.

**Dynamic Semantics**

The dynamic semantics of PolyMinML is a simple extension of that of MinML. We need only add the following two SOS rules:

$$\begin{align*}
\text{inst}(\text{Fun } t \text{ in } e \text{ end}, \tau) & \mapsto \{\tau/t\} e \\
\tau & \mapsto e' \\
\text{inst}(e, \tau) & \mapsto \text{inst}(e', \tau)
\end{align*}$$

It is then a simple matter to prove safety for this language.

**Theorem 20.3 (Preservation)**
If $e : \sigma$ and $e \mapsto e'$, then $e' : \sigma$.

The proof is by induction on evaluation.

**Theorem 20.4 (Progress)**
If $e : \sigma$, then either $e$ is a value or there exists $e'$ such that $e \mapsto e'$.

As before, this is proved by induction on typing.
First-Class Polymorphism

The syntax given above describes an ML-like treatment of polymorphism, *albeit* one in which polymorphic abstraction and instantiation is explicit, rather than implicit, as it is in ML. To obtain the first-class variant of PolyMinML, we simply ignore the distinction between poly- and mono-types, regarding them all as simply types. Everything else remains unchanged, including the proofs of progress and preservation.

With first-class polymorphism we may consider types such as

\[ \forall t(t \rightarrow t) \rightarrow \forall t(t \rightarrow t), \]

which cannot be expressed in the ML-like fragment. This is the type of functions that accept a polymorphic function as argument and yield a polymorphic function (of the same type) as result. If \( f \) has the above type, then \( f (\text{fun} \in \text{fun} \,(x : t) : t \rightarrow x \text{ end} \text{end}) \) is well-formed. However, the application \( f (\text{fun} \,(x : \text{int}) : \text{int} \rightarrow (x, 1) \text{ end}) \) is ill-formed, because the successor function does not have type \( \forall t(t \rightarrow t) \). The requirement that the argument be polymorphic is a significant restriction on how \( f \) may be used!

Contrast this with the following type (which does lie within the ML-like fragment):

\[ \forall t((t \rightarrow t) \rightarrow (t \rightarrow t)). \]

This is the type of polymorphic functions that, for each type \( t \), accept a function on \( t \) and yield another function on \( t \). If \( g \) has this type, the expression \( \text{inst} \,(g, \text{int}) \,(\text{succ}) \) is well-formed, since we first instantiate \( g \) at \( \text{int} \), then apply it to the successor function.

The situation gets more interesting in the presence of data structures such as lists and reference cells. It is a worthwhile exercise to consider the difference between the types \( \forall t(\sigma) \text{list} \) and \( \forall t(\sigma \text{list}) \) for various choices of \( \sigma \). Note once again that the former type cannot be expressed in ML, whereas the latter can.

Recall the following counterexample to type soundness for the early version of ML without the so-called value restriction:

```ml
let
  val r : ('a -> 'a) ref = ref (fn x:'a => x)
in
  r := (fn x:int => x+1) ; (!r)(true)
end
```

WORKING DRAFT  
JANUARY 14, 2003
A simple check of the polymorphic typing rules reveals that this is a well-formed expression, provided that the value restriction is suspended. Of course, it “gets stuck” during evaluation by attempting to add 1 to true.

Using the framework of explicit polymorphism, I will argue that the superficial plausibility of this example (which led to the unsoundness in the language) stems from a failure to distinguish between these two types:

1. The type $\forall t(t \rightarrow t \text{ref})$ of polymorphic functions yielding reference cells containing a function from a type to itself.

2. The type $\forall t(t \rightarrow t \text{ref}$ of reference cells containing polymorphic functions yielding a function from a type to itself.

(Notice the similarity to the distinctions discussed above.) For this example to be well-formed, we rely on an inconsistent reading of the example. At the point of the \texttt{val} binding we are treating $r$ as a value of the latter type, namely a reference cell containing a polymorphic function. But in the body of the \texttt{let} we are treating it as a value of the former type, a polymorphic function yielding a reference cell. We cannot have it both ways at once!

To sort out the error let us make the polymorphic instantiation and abstraction explicit. Here’s one rendering:

\begin{verbatim}
let
  val r : All 'a (('a -> 'a) ref) = Fun 'a in ref (fn x:'a => x) end
in
  r[int] := (fn x:int => x+1) ; (!r[bool])(true)
end
\end{verbatim}

Notice that we have made the polymorphic abstraction explicit, and inserted corresponding polymorphic instantiations. This example is type correct, and hence (by the proof of safety above) sound. But notice that it allocates \textit{two} reference cells, not \textit{one}! Recall that polymorphic functions are values, and the binding of $r$ is just such a value. Each of the two instances of $r$ executes the body of this function separately, each time allocating a new reference cell. Hence the unsoundness goes away!

Here’s another rendering that is, in fact, ill-typed (and should be, since it “gets stuck”!).
let
    val r : (All 'a ('a -> 'a)) ref = 
        ref (Fun 'a in fn x:'a => x end)
    in
    r := (fn x:int => x+1) ; (!r)[bool](true)
end

The assignment to \( r \) is ill-typed because the successor is not sufficiently polymorphic. The retrieval and subsequent instantiation and application is type-correct, however. If we change the program to

let
    val r : (All 'a ('a -> 'a)) ref = 
        ref (Fun 'a in fn x:'a => x end)
    in
    r := (Fun 'a in fn x:'a => x end) ; (!r)[bool](true)
end

then the expression is well-typed, and behaves sanely, precisely because we have assigned to \( r \) a sufficiently polymorphic function.

## 20.2 ML-style Type Inference

ML-style type inference may be viewed as a translation from the implicitly typed syntax of ML to the explicitly-typed syntax of PolyMinML. Specifically, the type inference mechanism performs the following tasks:

- Attaching type labels to function arguments and results.
- Inserting polymorphic abstractions for declarations of polymorphic type.
- Inserting polymorphic instantiations whenever a polymorphic declared variable is used.

Thus in ML we may write

\[
\begin{align*}
\text{val} & \ I : \ 'a \to \ 'a = \text{fn} \ x \Rightarrow x \\
\text{val} & \ n : \ \text{int} = I(I)(3)
\end{align*}
\]

This stands for the PolyMinML declarations\(^1\)

\(^1\)We’ve not equipped PolyMinML with a declaration construct, but you can see from the example how this might be done.
20.3 Parametricity

val I : $\forall t(t \to t) = \text{Fun} \text{in} \text{fun} \_ \_ \_ : t \text{is} x \text{end} \text{end}$
val n : int = inst(I, int $\to$ int)(inst(I, int))(3)

Here we apply the polymorphic identity function to itself, then apply the result to 3. The identity function is explicitly abstracted on the type of its argument and result, and its domain and range types are made explicit on the function itself. The two occurrences of I in the ML code are replaced by instantiations of I in the PolyMinML code, first at type int $\to$ int, the second at type int.

With this in mind we can now explain the “value restriction” on polymorphism in ML. Referring to the example of the previous section, the type inference mechanism of ML generates the first rendering of the example give above in which the type of the reference cell is $\forall t((t \to t) \text{ref})$. As we’ve seen, when viewed in this way, the example is not problematic, provided that polymorphic abstractions are seen as values. For in this case the two instances of r generate two distinct reference cells, and no difficulties arise. Unfortunately, ML does not treat polymorphic abstractions as values! Only one reference cell is allocated, which, in the absence of the value restriction, would lead to unsoundness.

Why does the value restriction save the day? In the case that the polymorphic expression is not a value (in the ML sense) the polymorphic abstraction that is inserted by the type inference mechanism changes a non-value into a value! This changes the semantics of the expression (as we’ve seen, from allocating one cell, to allocating two different cells), which violates the semantics of ML itself. However, if we limit ourselves to values in the first place, then the polymorphic abstraction is only ever wrapped around a value, and no change of semantics occurs. Therefore, the insertion of polymorphic abstraction doesn’t change the semantics, and everything is safe. The example above involving reference cells is ruled out, because the expression ref (fn x => x) is not a value, but such is the nature of the value restriction.

20.3 Parametricity

Our original motivation for introducing polymorphism was to enable more programs to be written — those that are “generic” in one or more types, such as the composition function give above. The idea is that if the behavior

---

2 One could argue that the ML semantics is incorrect, which leads to a different language.
3 This would need to be proved, of course.
of a function does not depend on a choice of types, then it is useful to be able
to define such “type oblivious” functions in the language. Once we have
such a mechanism in hand, it can also be used to ensure that a particular
piece of code can not depend on a choice of types by insisting that it be
polymorphic in those types. In this sense polymorphism may be used to
impose restrictions on a program, as well as to allow more programs to be
written.

The restrictions imposed by requiring a program to be polymorphic un-
derlie the often-observed experience when programming in ML that if the
types are correct, then the program is correct. Roughly speaking, since the
ML type system is polymorphic, if a function type checks with a polymor-
phic type, then the strictures of polymorphism vastly cut down the set of
well-typed programs with that type. Since the intended program is one
these (by the hypothesis that its type is “right”), you’re much more likely
to have written it if the set of possibilities is smaller.

The technical foundation for these remarks is called parametricity. The
goal of this section is to give an account of parametricity for PolyMinML.
To keep the technical details under control, we will restrict attention to the
ML-like (prenex) fragment of PolyMinML. It is possibly to generalize to
first-class polymorphism, but at the expense of considerable technical com-
plexity. Nevertheless we will find it necessary to gloss over some technical
details, but wherever a “pedagogic fiction” is required, I will point it out.
To start with, it should be stressed that the following does not apply to lan-
guages with mutable references!

20.3.1 Informal Discussion

We will begin with an informal discussion of parametricity based on a “seat
of the pants” understanding of the set of well-formed programs of a type.

Suppose that a function value \( f \) has the type \( \forall t (t \rightarrow t) \). What function
could it be?

1. It could diverge when instantiated — \( f [\tau] \) goes into an infinite loop.
   Since \( f \) is polymorphic, its behavior cannot depend on the choice of
   \( \tau \), so in fact \( f [\tau'] \) diverges for all \( \tau' \) if it diverges for \( \tau \).

2. It could converge when instantiated at \( \tau \) to a function \( g \) of type \( \tau \rightarrow \tau \)
   that loops when applied to an argument \( v \) of type \( \tau \) — i.e., \( g (v) \) runs
   forever. Since \( f \) is polymorphic, \( g \) must diverge on every argument \( v \)
of type \( \tau \) if it diverges on some argument of type \( \tau \).
3. It could converge when instantiated at \( \tau \) to a function \( g \) of type \( \tau \rightarrow \tau \) that, when applied to a value \( v \) of type \( \tau \) returns a value \( v' \) of type \( \tau \). Since \( f \) is polymorphic, \( g \) cannot depend on the choice of \( v \), so \( v' \) must in fact be \( v \).

Let us call cases (1) and (2) uninteresting. The foregoing discussion suggests that the only interesting function \( f \) of type \( \forall t (t \rightarrow t) \) is the polymorphic identity function.

Suppose that \( f \) is an interesting function of type \( \forall t (t) \). What function could it be? A moment’s thought reveals that it cannot be interesting! That is, every function \( f \) of this type must diverge when instantiated, and hence is uninteresting. In other words, there are no interesting values of this type — it is essentially an “empty” type.

For a final example, suppose that \( f \) is an interesting function of type \( \forall t (t \text{list} \rightarrow t \text{list}) \). What function could it be?

1. The identity function that simply returns its argument.
2. The constantly-nil function that always returns the empty list.
3. A function that drops some elements from the list according to a predetermined (data-independent) algorithm — e.g., always drops the first three elements of its argument.
4. A permutation function that reorganizes the elements of its argument.

The characteristic that these functions have in common is that their behavior is entirely determined by the spine of the list, and is independent of the elements of the list. For example, \( f \) cannot be the function that drops all “even” elements of the list — the elements might not be numbers! The point is that the type of \( f \) is polymorphic in the element type, but reveals that the argument is a list of unspecified elements. Therefore it can only depend on the “list-ness” of its argument, and never on its contents.

In general if a polymorphic function behaves the same at every type instance, we say that it is parametric in that type. In PolyMinML all polymorphic functions are parametric. In Standard ML most functions are, except those that involve equality types. The equality function is not parametric because the equality test depends on the type instance — testing equality of integers is different than testing equality of floating point numbers, and we cannot test equality of functions. Such “pseudo-polymorphic” operations are said to be ad hoc, to contrast them from parametric.
How can parametricity be exploited? As we will see later, parametricity is the foundation for data abstraction in a programming language. To get a sense of the relationship, let us consider a classical example of exploiting parametricity, the polymorphic Church numerals. Let \( N \) be the type \( \forall t (t \rightarrow (t \rightarrow t) \rightarrow t) \). What are the interesting functions of the type \( N \)? Given any type \( \tau \), and values \( z : \tau \) and \( s : \tau \rightarrow \tau \), the expression

\[
f[\tau](z)(s)
\]

must yield a value of type \( \tau \). Moreover, it must behave uniformly with respect to the choice of \( \tau \). What values could it yield? The only way to build a value of type \( \tau \) is by using the element \( z \) and the function \( s \) passed to it. A moment’s thought reveals that the application must amount to the \( n \)-fold composition

\[
s(s(\ldots s(z)\ldots)).
\]

That is, the elements of \( N \) are in 1-to-1 correspondence with the natural numbers.

Let us write \( \pi \) for the polymorphic function of type \( N \) representing the natural number \( n \), namely the function

\[
\begin{align*}
\text{Fun } t \ in \\
fn z : t \ in \\
\quad fn s : t \rightarrow t \ in \\
\quad \quad s(s(\ldots s(z)\ldots)) \\
end \\
end
\end{align*}
\]

where there are \( n \) occurrences of \( s \) in the expression. Observe that if we instantiate \( \pi \) at the built-in type \( \text{int} \) and apply the result to \( 0 \) and \( \text{succ} \), it evaluates to the number \( n \). In general we may think of performing an “experiment” on a value of type \( N \) by instantiating it at a type whose values will constitute the observations, the applying it to operations \( z \) and \( s \) for performing the experiment, and observing the result.

Using this we can calculate with Church numerals. Let us consider how to define the addition function on \( N \). Given \( m \) and \( n \) of type \( N \), we wish to compute their sum \( m + n \), also of type \( N \). That is, the addition function must look as follows:
The question is: how to fill in the missing code? Think in terms of experiments. Given \( m \) and \( n \) of type \( N \), we are to yield a value that when “probed” by supplying a type \( t \), an element \( z \) of that type, and a function \( s \) on that type, must yield the \((m + n)\)-fold composition of \( s \) with \( z \). One way to do this is to “run” \( m \) on \( t \), \( z \), and \( s \), yielding the \( m \)-fold composition of \( s \) with \( z \), then “running” \( n \) on this value and \( s \) again to obtain the \( n \)-fold composition of \( s \) with the \( n \)-fold composition of \( s \) with \( z \)— the desired answer. Here’s the code:

\[
\begin{align*}
  &\text{fun } m \colon N \text{ in} \\
  &\quad \text{fun } n \colon N \text{ in} \\
  &\quad \quad \text{fun } t \text{ in} \\
  &\quad \quad \quad \text{fun } z : t \text{ in} \\
  &\quad \quad \quad \quad \text{fun } s : t \rightarrow t \text{ in} \\
  &\quad \quad \quad \quad \quad \ldots \\
  &\quad \quad \quad \quad \quad \text{end} \\
  &\quad \quad \quad \text{end} \\
  &\quad \quad \text{end} \\
  &\quad \text{end} \\
  &\text{end}
\end{align*}
\]

To see that it works, instantiate the result at \( \tau \), apply it to \( z \) and \( s \), and observe the result.

### 20.3.2 Relational Parametricity

In this section we give a more precise formulation of parametricity. The main idea is that polymorphism implies that certain equations between expressions must hold. For example, if \( f : \forall t(t \rightarrow t) \), then \( f \) must be equal to
the identity function, and if $f : N$, then $f$ must be equal to some Church numeral $\pi$. To make the informal idea of parametricity precise, we must clarify what we mean by equality of expressions.

The main idea is to define equality in terms of “experiments” that we carry out on expressions to “test” whether they are equal. The valid experiments on an expression are determined solely by its type. In general we say that two closed expressions of a type $\tau$ are equal iff either they both diverge, or they both converge to equal values of that type. Equality of closed values is then defined based on their type. For integers and booleans, equality is straightforward: two values are equal iff they are identical. The intuition here is that equality of numbers and booleans is directly observable. Since functions are “infinite” objects (when thought of in terms of their input/output behavior), we define equality in terms of their behavior when applied. Specifically, two functions $f$ and $g$ of type $\tau_1 \rightarrow \tau_2$ are equal iff whenever they are applied to equal arguments of type $\tau_1$, they yield equal results of type $\tau_2$.

More formally, we make the following definitions. First, we define equality of closed expressions of type $\tau$ as follows:

$$e \equiv_{\text{exp}} e' : \tau \iff e \mapsto v \Leftrightarrow e' \mapsto v.$$ 

Notice that if $e$ and $e'$ both diverge, then they are equal expressions in this sense. For closed values, we define equality by induction on the structure of monotypes:

$$v \equiv_{\text{val}} v' : \text{bool} \iff v = v' = \text{true} \lor v = v' = \text{false}$$

$$v \equiv_{\text{val}} v' : \text{int} \iff v = v' = n \text{ for some } n \geq 0$$

$$v \equiv_{\text{val}} v' : \tau_1 \rightarrow \tau_2 \iff \forall v_1 \equiv_{\text{val}} v'_1 : \tau_1 \implies v (v_1) \equiv_{\text{exp}} v' (v'_1) : \tau_2$$

The following lemma states two important properties of this notion of equality.

**Lemma 20.5**

1. Expression and value equivalence are reflexive, symmetric, and transitive.

2. Expression equivalence is a congruence: we may replace any sub-expression of an expression $e$ by an equivalent sub-expression to obtain an equivalent expression.

So far we’ve considered only equality of closed expressions of monomorphic type. The definition is made so that it readily generalizes to the polymorphic case. The idea is that when we quantify over a type, we are not
able to say a priori what we mean by equality at that type, precisely because it is “unknown”. Therefore we also quantify over all possible notions of equality to cover all possible interpretations of that type. Let us write $R : \tau \leftrightarrow \tau'$ to indicate that $R$ is a binary relation between values of type $\tau$ and $\tau'$.

Here is the definition of equality of polymorphic values:

$$v \cong_{\text{val}} v' : \forall t(\sigma) \quad \text{iff} \quad \text{for all } \tau \text{ and } \tau', \text{ and all } R : \tau \leftrightarrow \tau', \text{ and } v[\tau] \cong_{\exp} v'[\tau'] : \sigma$$

where we take equality at the type variable $t$ to be the relation $R$ (i.e., $v \cong_{\text{val}} v' : t \iff v R v'$).

There is one important proviso: when quantifying over relations, we must restrict attention to what are called admissible relations, a sub-class of relations that, in a suitable sense, respects computation. Most natural choices of relation are admissible, but it is possible to contrive examples that are not. The rough-and-ready rule is this: a relation is admissible iff it is closed under “partial computation”. Evaluation of an expression $e$ to a value proceeds through a series of intermediate expressions $e \mapsto e_1 \mapsto e_2 \mapsto \cdots \mapsto e_n$. The expressions $e_i$ may be thought of as “partial computations” of $e$, stopping points along the way to the value of $e$. If a relation relates corresponding partial computations of $e$ and $e'$, then, to be admissible, it must also relate $e$ and $e'$ — it cannot relate all partial computations, and then refuse to relate the complete expressions. We will not develop this idea any further, since to do so would require the formalization of partial computation. I hope that this informal discussion suffices to give the idea.

The following is Reynolds’ Parametricity Theorem:

**Theorem 20.6 (Parametricity)**

If $e : \sigma$ is a closed expression, then $e \cong_{\exp} e : \sigma$.

This may seem obvious, until you consider that the notion of equality between expressions of polymorphic type is very strong, requiring equivalence under all possible relational interpretations of the quantified type.

Using the Parametricity Theorem we may prove a result we stated informally above.

**Theorem 20.7**

If $f : \forall t(\tau \rightarrow t)$ is an interesting value, then $f \cong_{\text{val}} \text{id} : \forall t(\tau \rightarrow t)$, where $\text{id}$ is the polymorphic identity function.

**Proof:** Suppose that $\tau$ and $\tau'$ are monotypes, and that $R : \tau \leftrightarrow \tau'$. We wish to show that

$$f \left[ \tau \right] \cong_{\exp} \text{id} \left[ \tau' \right] : t \rightarrow t,$$
where equality at type $t$ is taken to be the relation $R$.

Since $f$ (and $id$) are interesting, there exists values $f_\tau$ and $id_{\tau'}$ such that

$$f_\tau \xrightarrow{\ast} f_\tau$$

and

$$id_{\tau'} \xrightarrow{\ast} id_{\tau'}.$$  

We wish to show that

$$f_\tau \cong_{val} id_{\tau'} : t \rightarrow t.$$  

Suppose that $v_1 \cong_{val} v_1' : t$, which is to say $v_1 R v_1'$ since equality at type $t$ is taken to be the relation $R$. We are to show that

$$f_\tau (v_1) \cong_{exp} id_{\tau'} (v_1') : t.$$  

By the assumption that $f$ is interesting (and the fact that $id$ is interesting), there exists values $v_2$ and $v_2'$ such that

$$f_\tau (v_1) \xrightarrow{\ast} v_2$$

and

$$id_{\tau'} (v_1') \xrightarrow{\ast} v_2'.$$

By the definition of $id$, it follows that $v_2' = v_1'$ (it’s the identity function!). We must show that $v_2 R v_1'$ to complete the proof.

Now define the relation $R' : \tau \rightarrow \tau$ to be the set $\{ (v, v) \mid v R v' \}$. Since $f : \forall \tau (t \rightarrow t)$, we have by the Parametricity Theorem that $f \cong_{val} f : \forall \tau (t \rightarrow t)$, where equality at type $t$ is taken to be the relation $R'$. Since $v_1 R v_1'$, we have by definition $v_1 R' v_1$. Using the definition of equality of polymorphic type, it follows that

$$f_\tau (v_1) \cong_{exp} id_{\tau'} (v_1) : t.$$  

Hence $v_2 R v_1'$, as required.

You might reasonably wonder, at this point, what the relationship $f \cong_{val} id : \forall \tau (t \rightarrow t)$ has to do with $f$’s execution behavior. It is a general fact, which we will not attempt to prove, that equivalence as we’ve defined it yields results about execution behavior. For example, if $f : \forall \tau (t \rightarrow t)$, we can show that for every $\tau$ and every $v : \tau$, $f [\tau] (v)$ evaluates to $v$. By the preceding theorem $f \cong_{val} id : \forall \tau (t \rightarrow t)$. Suppose that $\tau$ is some monotype and $v : \tau$ is some closed value. Define the relation $R : \tau \rightarrow \tau$ by

$$v_1 R v_2 \iff v_1 = v_2 = v.$$
Then we have by the definition of equality for polymorphic values

\[ f \left[ \tau \right] (v) \equiv_{exp \ id \left[ \tau \right]} (v) : t, \]

where equality at \( t \) is taken to be the relation \( R \). Since the right-hand side terminates, so must the left-hand side, and both must yield values related by \( R \), which is to say that both sides must evaluate to \( v \).
Chapter 21

Data Abstraction

Data abstraction is perhaps the most fundamental technique for structuring programs to ensure their robustness over time and to facilitate team development. The fundamental idea of data abstraction is the separation of the client from the implementor of the abstraction by an interface. The interface is a form of "contract" between the client and implementor. It specifies the operations that may be performed on values of the abstract type by the client and, at the same time, imposes the obligation on the implementor to provide these operations with the specified functionality. By limiting the client’s view of the abstract type to a specified set of operations, the interface protects the client from depending on the details of the implementation of the abstraction, most especially its representation in terms of well-known constructs of the programming language. Doing so ensures that the implementor is free to change the representation (and, correspondingly, the implementation of the operations) of the abstract type without affecting the behavior of a client of the abstraction.

Our intention is to develop a rigorous account of data abstraction in an extension of PolyMinML with existential types. Existential types provide the fundamental linguistic mechanisms for defining interfaces, implementing them, and using the implementation in client code. Using this extension of PolyMinML we will then develop a formal treatment of representation independence based on Reynolds’s Parametricity Theorem for PolyMinML. The representation independence theorem will then serve as the basis for proving the correctness of abstract type implementations using bisimulation relations.
21.1 Existential Types

21.1.1 Abstract Syntax

The syntax of PolyMinML is extended with the following constructs:

\[
\begin{align*}
\text{Polytypes} & \quad \sigma ::= \ldots \\
& \quad | \quad \exists \sigma \\
\text{Expressions} & \quad e ::= \ldots \\
& \quad | \quad \text{pack } \tau \text{ with } e \text{ as } \sigma \text{ end} \\
& \quad | \quad \text{open } e_1 \text{ as } t \text{ with } x : \sigma \text{ in } e_2 \text{ end} \\
\text{Values} & \quad v ::= \ldots \\
& \quad | \quad \text{pack } \tau \text{ with } v \text{ as } \sigma \text{ end}
\end{align*}
\]

The polytype \( \exists \sigma \) is called an existential type. An existential type is the interface of an abstract type. An implementation of the existential type \( \exists \sigma \) is a package value of the form \( \text{pack } \tau \text{ with } v \text{ as } \exists \sigma \text{ end} \) consisting of a monotype \( \tau \) together with a value \( v \) of type \( \{ \tau/t \} \sigma \). The monotype \( \tau \) is the representation type of the implementation; the value \( v \) is the implementation of the operations of the abstract type. A client makes use of an implementation by opening it within a scope, written \( \text{open } e_1 \text{ as } t \text{ with } x : \sigma \text{ in } e_2 \text{ end} \), where \( e_1 \) is an implementation of the interface \( \exists \sigma \), and \( e_2 \) is the client code defined in terms of an unknown type \( t \) (standing for the representation type) and an unknown value \( x \) of type \( \sigma \) (standing for the unknown operations).

In an existential type \( \exists \sigma \) the type variable \( t \) is bound in \( \sigma \), and may be renamed at will to satisfy uniqueness requirements. In an expression of the form \( \text{open } e_1 \text{ as } t \text{ with } x : \sigma \text{ in } e_2 \text{ end} \) the type variable \( t \) and the ordinary variable \( x \) are bound in \( e_2 \) and may also be renamed at will to satisfy non-occurrence requirements. As we will see below, renaming of bound variables is crucial for ensuring that an abstract type is “new” in the sense of being distinct from any other type whenever it is opened for use in a scope. This is sometimes called generativity of abstract types, since each occurrence of \( \text{open} \) “generates” a “new” type for use within the body of the client. In reality this informal notion of generativity comes down to renaming of bound variables to ensure their uniqueness in a context.

21.1.2 Correspondence With ML

To fix ideas, it is worthwhile to draw analogies between the present formalism and (some aspects of) the Standard ML module system. We have the following correspondences:

\[
\begin{align*}
\text{Polytypes} & \quad \sigma ::= \ldots \\
& \quad | \quad \exists \sigma \\
\text{Expressions} & \quad e ::= \ldots \\
& \quad | \quad \text{pack } \tau \text{ with } e \text{ as } \sigma \text{ end} \\
& \quad | \quad \text{open } e_1 \text{ as } t \text{ with } x : \sigma \text{ in } e_2 \text{ end} \\
\text{Values} & \quad v ::= \ldots \\
& \quad | \quad \text{pack } \tau \text{ with } v \text{ as } \sigma \text{ end}
\end{align*}
\]
Here is an example of these correspondences in action. In the sequel we will use ML-like notation with the understanding that it is to be interpreted in *PolyMinML* in the following fashion.

Here is an ML signature for a persistent representation of queues:

```ml
signature QUEUE =
  sig
    type queue
    val empty : queue
    val insert : int * queue -> queue
    val remove : queue -> int * queue
  end
```

This signature is deliberately stripped down to simplify the development. In particular we leave undefined the meaning of *remove* on an empty queue.

The corresponding existential type is $\sigma_q := \exists q(\tau_q)$, where

$$\tau_q := q^{*}((\text{int}^{*}q) \rightarrow q) \rightarrow (\text{int}^{*}q)$$

That is, the operations of the abstraction consist of a three-tuple of values, one for the empty queue, one for the insert function, and one for the remove function.

Here is a straightforward implementation of the *QUEUE* interface in ML:

```ml
structure QL :> QUEUE =
  struct
    type queue = int list
    val empty = nil
    fun insert (x, xs) = x::xs
    fun remove xs =
        let val (x,xs') = rev xs in (x, rev xs') end
  end
```

A queue is a list in reverse enqueue order — the last element to be enqueued is at the head of the list. Notice that we use *opaque* signature ascription to ensure that the type *queue* is hidden from the client!
The corresponding package is \( e_q := \text{pack int list with } v_q \text{ as } \sigma_q \text{ end} \), where
\[ v_q := (\text{nil}, (v_i, v_r)) \]
where \( v_i \) and \( v_r \) are the obvious function abstractions corresponding to the ML code given above.

Finally, a client of an abstraction in ML might typically open it within a scope:

\[
\begin{align*}
\text{local} & \quad \text{open QL} \\
& \quad \text{in} \\
& \quad \ldots \\
\text{end}
\end{align*}
\]

This corresponds to writing

\[
\text{open QL as } q \text{ with } <n, i, r> : \tau_q \text{ in } \ldots \text{ end}
\]

in the existential type formalism, using pattern matching syntax for tuples.

### 21.1.3 Static Semantics

The static semantics is an extension of that of PolyMinML with rules governing the new constructs. The rule of formation for existential types is as follows:

\[
\frac{\Delta \cup \{t\} \vdash \sigma \text{ ok } \quad t \notin \Delta}{\Delta \vdash \exists t(\sigma) \text{ ok}}
\]  
(21.1)

The requirement \( t \notin \Delta \) may always be met by renaming the bound variable.

The typing rule for packages is as follows:

\[
\frac{\Delta \vdash \tau \text{ ok } \quad \Delta \vdash \exists t(\sigma) \text{ ok } \quad \Gamma \vdash_{\Delta} e : \{\tau/t\}\sigma}{\Gamma \vdash_{\Delta} \text{pack } \tau \text{ with } e \text{ as } \exists t(\sigma) \text{ end}}
\]  
(21.2)

The implementation, \( e \), of the operations “knows” the representation type, \( \tau \), of the ADT.

The typing rule for opening a package is as follows:

\[
\frac{\Delta \vdash \tau_c \text{ ok } \quad \Gamma, x:\sigma \vdash_{\Delta\cup\{t\}} e_c : \tau_c \quad \Gamma \vdash_{\Delta} e_i : \exists t(\sigma) \quad t \notin \Delta}{\Gamma \vdash_{\Delta} \text{open } e_i \text{ as } t \text{ with } x:\sigma \text{ in } e_c \text{ end : } \tau_c}
\]  
(21.3)

This is a complex rule, so study it carefully! Two things to note:
1. The type of the client, $\tau_c$, must not involve the abstract type $t$. This prevents the client from attempting to export a value of the abstract type outside of the scope of its definition.

2. The body of the client, $e_c$, is type checked without knowledge of the representation type, $t$. The client is, in effect, polymorphic in $t$.

As usual, the condition $t \notin \Delta$ can always be met by renaming the bound variable $t$ of the open expression to ensure that it is distinct from all other active types $\Delta$. It is in this sense that abstract types are “new”! Whenever a client opens a package, it introduces a local name for the representation type, which is bound within the body of the client. By our general conventions on bound variables, this local name may be chosen to ensure that it is distinct from any other such local name that may be in scope, which ensures that the “new” type is different from any other type currently in scope. At an informal level this ensures that the representation type is “held abstract”; we will make this intuition more precise in Section 21.2 below.

### 21.1.4 Dynamic Semantics

We will use structured operational semantics (SOS) to specify the dynamic semantics of existential types. Here is the rule for evaluating package expressions:

$$
\frac{e \mapsto e'}{\text{pack } \tau \text{ with } e \text{ as } \sigma \text{ end} \mapsto \text{pack } \tau \text{ with } e' \text{ as } \sigma \text{ end}} \quad (21.4)
$$

Opening a package begins by evaluating the package expressions:

$$
\frac{e_i \mapsto e'_i}{\text{open } e_i \text{ as } t \text{ with } x: \sigma \text{ in } e_c \text{ end} \mapsto \text{open } e'_i \text{ as } t \text{ with } x: \sigma \text{ in } e_c \text{ end}} \quad (21.5)
$$

Once the package is fully evaluated, we bind $t$ to the representation type and $x$ to the implementation of the operations within the client code:

$$
\frac{(\sigma = \exists t(\sigma'))}{\text{open (pack } \tau \text{ with } v \text{ as } \sigma \text{ end} \text{ as } t \text{ with } x: \sigma' \text{ in } e_c \text{ end} \mapsto \{\tau, v/t, x\} e_c}} \quad (21.6)
$$

Observe that there are no abstract types at run time! During execution of the client, the representation type is fully exposed. It is held abstract only
during type checking to ensure that the client does not (accidentally or ma-
licularly) depend on the implementation details of the abstraction. Once
the program type checks there is no longer any need to enforce abstraction.
The dynamic semantics reflects this intuition directly.

21.1.5 Safety

The safety of the extension is stated and proved as usual. The argument is
a simple extension of that used for PolyMinML to the new constructs.

Theorem 21.1 (Preservation)
If $e : \tau$ and $e \rightsquigarrow e'$, then $e' : \tau$.

Lemma 21.2 (Canonical Forms)
If $v : \exists t(\sigma)$ is a value, then $v = \text{pack} \tau$ with $v'$ as $\exists t(\sigma)$ end for some mono-
type $\tau$ and some value $v' : \{\tau/t\} \sigma$.

Theorem 21.3 (Progress)
If $e : \tau$ then either $e$ value or there exists $e'$ such that $e \rightsquigarrow e'$.

21.2 Representation Independence

Parametricity is the essence of representation independence. The typing
rules for open given above ensure that the client of an abstract type is poly-
morphic in the representation type. According to our informal understand-
ing of parametricity this means that the client’s behavior is in some sense
“independent” of the representation type.

More formally, we say that an (admissible) relation $R : \tau_1 \leftrightarrow \tau_2$ is a
bisimulation between the packages

$$\text{pack} \tau_1 \text{ with } v_1 \text{ as } \exists t(\sigma) \text{ end}$$

and

$$\text{pack} \tau_2 \text{ with } v_2 \text{ as } \exists t(\sigma) \text{ end}$$
of type $\exists t(\sigma)$ iff $v_1 \equiv_{\text{val}} v_2 : \sigma$, taking equality at type $t$ to be the relation
$R$. The reason for calling such a relation $R$ a bisimulation will become
apparent shortly. Two packages are said to be bisimilar whenever there is a
bisimulation between them.
Since the client $e_c$ of a data abstraction of type $\exists t(\sigma)$ is essentially a polymorphic function of type $\forall t(\sigma \to \tau_c)$, where $t \notin FTV(\tau_c)$, it follows from the Parametricity Theorem that

$$\{\tau_1, v_1/t, x\} e_c \cong_{exp} \{\tau_2, v_2/t, x\} e_c : \tau_c$$

whenever $R$ is such a bisimulation. Consequently,

$$\text{open } e_1 \text{ as } t \text{ with } x : \sigma \text{ in } e_c \text{ end } \cong_{exp} \text{open } e_2 \text{ as } t \text{ with } x : \sigma \text{ in } e_c \text{ end} : \tau_c.$$ 

That is, the two implementations are indistinguishable by any client of the abstraction, and hence may be regarded as equivalent. This is called Representation Independence; it is merely a restatement of the Parametricity Theorem in the context of existential types.

This observation licenses the following technique for proving the correctness of an ADT implementation. Suppose that we have an implementation of an abstract type $\exists t(\sigma)$ that is “clever” in some way. We wish to show that it is a correct implementation of the abstraction. Let us therefore call it a candidate implementation. The Representation Theorem suggests a technique for proving the candidate correct. First, we define a reference implementation of the same abstract type that is “obviously correct”. Then we establish that the reference implementation and the candidate implementation are bisimilar. Consequently, they are equivalent, which is to say that the candidate is “equally correct as” the reference implementation.

Returning to the queues example, let us take as a reference implementation the package determined by representing queues as lists. As a candidate implementation we take the package corresponding to the following ML code:

```ml
structure QFB :> QUEUE =
  struct
    type queue = int list * int list
    val empty = (nil, nil)
    fun insert (x, (bs, fs)) = (x::bs, fs)
    fun remove (bs, nil) = remove (nil, rev bs)
    | remove (bs, f::fs) = (f, (bs, fs))
  end
```

We will show that $QL$ and $QFB$ are bisimilar, and therefore indistinguishable by any client.

Define the relation $R : \text{int list} \leftrightarrow \text{int list} \times \text{int list}$ as follows:

$$R = \{ (l, (b, f)) \mid l \cong_{val} b @ \text{rev}(f) \}$$
We will show that $R$ is a bisimulation by showing that implementations of `empty`, `insert`, and `remove` determined by the structures $QL$ and $QFB$ are equivalent relative to $R$.

To do so, we will establish the following facts:

1. $QL\text{.empty} R QFB\text{.empty}$.

2. Assuming that $m \equiv_{\text{val}} n : \text{int}$ and $l R (b, f)$, show that

   
   $QL\text{.insert}((m, l)) R QFB\text{.insert}((n, (b, f)))$.

3. Assuming that $l R (b, f)$, show that

   
   $QL\text{.remove}(l) \equiv_{\text{exp}} QFB\text{.remove}((b, f)) : \text{int}^* t$,

   taking $t$ equality to be the relation $R$.

Observe that the latter two statements amount to the assertion that the operations preserve the relation $R$ — they map related input queues to related output queues. It is in this sense that we say that $R$ is a bisimulation, for we are showing that the operations from $QL$ simulate, and are simulated by, the operations from $QFB$, up to the relationship $R$ between their representations.

The proofs of these facts are relatively straightforward, given some relatively obvious lemmas about expression equivalence.

1. To show that $QL\text{.empty} R QFB\text{.empty}$, it suffices to show that

   
   $\text{nil} @ \text{rev}(\text{nil}) \equiv_{\text{exp}} \text{nil} : \text{int} \text{list}$,

   which is obvious from the definitions of append and reverse.

2. For `insert`, we assume that $m \equiv_{\text{val}} n : \text{int}$ and $l R (b, f)$, and prove that

   
   $QL\text{.insert}((m, l)) R QFB\text{.insert}((n, (b, f)))$.

   By the definition of $QL\text{.insert}$, the left-hand side is equivalent to $m :: l$, and by the definition of $QR\text{.insert}$, the right-hand side is equivalent to $(n :: b, f)$. It suffices to show that

   
   $m :: l \equiv_{\text{exp}} (n :: b) @ \text{rev}(f) : \text{int} \text{list}$.
Calculating, we obtain
\[(n::b)@\text{rev}(f) \equiv \exp n::(b@\text{rev}(f))\]
\[\equiv \exp n::l\]

since \(l \equiv \exp b@\text{rev}(f)\). Since \(m \equiv \text{val} n : \text{int}\), it follows that \(m = n\), which completes the proof.

3. For \text{remove}, we assume that \(l\) is related by \(R\) to \((b, f)\), which is to say that \(l \equiv \exp b@\text{rev}(f)\). We are to show

\[Q\text{L.remove}(l) \equiv \exp Q\text{F.B.remove}((b, f)) : \text{int}*t,\]

taking \(t\) equality to be the relation \(R\). Assuming that the queue is non-empty, so that the \text{remove} is defined, we have \(l \equiv \exp l'@[m]\) for some \(l'\) and \(m\). We proceed by cases according to whether or not \(f\) is empty. If \(f\) is non-empty, then \(f \equiv \exp n::f'\) for some \(n\) and \(f'\). Then by the definition of \(Q\text{F.B.remove}\),

\[Q\text{F.B.remove}((b, f)) \equiv \exp (n, (b, f')) : \text{int}*t,\]

relative to \(R\). We must show that

\[(m, l') \equiv \exp (n, (b, f')) : \text{int}*t,\]

relative to \(R\). This means that we must show that \(m = n\) and \(l' \equiv \exp b@\text{rev}(f') : \text{int list}\).

Calculating from our assumptions,

\[l = l'@[m]\]
\[= b@\text{rev}(f)\]
\[= b@\text{rev}(n::f')\]
\[= b@(\text{rev}(f')@[n])\]
\[= (b@\text{rev}(f'))@[n]\]

From this the result follows. Finally, if \(f\) is empty, then \(b \equiv \exp b'@[n]\) for some \(b'\) and \(n\). But then \(\text{rev}(b) \equiv \exp n::\text{rev}(b')\), which reduces to the case for \(f\) non-empty.

This completes the proof — by Representation Independence the reference and candidate implementations are equivalent.
Part IX

Laziness
Chapter 22

Lazy Types

The language MinML is an example of an *eager*, or *strict*, functional language. Such languages are characterized by two, separable features of their operational semantics.

1. **Call-by-value.** The argument to a function is evaluated before control is passed to the body of the function. Function parameters are only ever bound to values.

2. **Strict data types.** A value of a data type is constructed, possibly from other values, at the point at which the constructor is used.

Since most familiar languages are eager, this might seem to be the most natural, or even the only possible, choice. The subject of this chapter is to explore an alternative, *lazy evaluation*, that seeks to delay evaluation of expressions as long as possible, until their value is actually required to complete a computation. This strategy is called “lazy” because we perform only the evaluation that is actually required to complete a computation. If the value of an expression is never required, it is never (needlessly) computed. Moreover, the lazy evaluation strategy memoizes delayed computations so that they are never performed more than once. Once (if ever) the value has been determined, it is stored away to be used in case the value is ever needed again.

Lazy languages are characterized by the following features of their operational semantics.

1. **Call-by-need.** The argument to a function is passed to the body of the function without evaluating it. The argument is only evaluated if it is needed in the computation, and then its value is saved for future reference in case it is needed again.
2. **Lazy data types.** An expression yielding a value of a data type is not evaluated until its value is actually required to complete a computation. The value, once obtained, is saved in case it is needed again.

While it might seem, at first glance, that lazy evaluation would lead to more efficient programs (by avoiding unnecessary work), it is not at all obvious that this is the case. In fact it’s not the case. The main issue is that memoization is costly, because of the bookkeeping overhead required to manage the transition from unevaluated expression to evaluated value. A delayed computation must store the code that determines the value of an expression (should it be required), together with some means of triggering its evaluation once it is required. If the value is ever obtained, the value determined by the code must be stored away, and we must somehow ensure that this value is returned on subsequent access. This can slow down many programs. For example, if we know that a function will inspect the value of every element of a list, it is much more efficient to simply evaluate these elements when the list is created, rather than fruitlessly delaying the computation of each element, only to have it be required eventually anyway. **Strictness analysis** is used in an attempt to discover such cases, so that the overhead can be eliminated, but in general it is impossible (for decidability reasons) to determine completely and accurately whether the value of an expression is surely needed in a given program.

The real utility of lazy evaluation lies not in the possible efficiency gains it may afford in some circumstances, but rather in a substantial increase in expressive power that it brings to a language. By delaying evaluation of an expression until it is needed, we can naturally model situations in which the value does not even exist until it is required. A typical example is interactive input. The user can be modelled as a “delayed computation” that produces its values (i.e., enters its input) only upon demand, not all at once before the program begins execution. Lazy evaluation models this scenario quite precisely.

Another example of the use of lazy evaluation is in the representation of infinite data structures, such as the sequence of all natural numbers. Obviously we cannot hope to compute the entire sequence at the time that it is created. Fortunately, only a finite initial segment of the sequence is ever needed to complete execution of a program. Using lazy evaluation we can compute this initial segment on demand, avoiding the need to compute the part we do not require.

Lazy evaluation is an important and useful concept to have at your disposal. The question that we shall explore in this chapter is how best to
provide such a feature in a programming language. Historically, there has been a division between eager and lazy languages, exemplified by ML and Haskell, respectively, which impose one or the other evaluation strategy globally, leaving no room for combining the best of both approaches.

More recently, it has come to be recognized by both communities that it is important to support both forms of evaluation. This has led to two, distinct approaches to supporting laziness:

1. **Lazy types in a strict language.** The idea is to add support for lazy data types to a strict language by providing a means of defining such types, and for creating and destroying values of these types. Constructors are implicitly memoized to avoid redundant re-computation of expressions. The call-by-value evaluation strategy for functions is maintained.

2. **Strict types in a lazy language.** The idea is to add support for constructors that forcibly evaluate their arguments, avoiding the overhead of managing the bookkeeping associated with delayed, memoized computation. The call-by-need evaluation strategy for function calls is maintained.

We will explore both alternatives.

### 22.1 Lazy Types in MinML

We will first explore the addition of lazy data types to a strict functional language. We will focus on a specific example, the type of lazy lists. For the sake of simplicity we’ll consider only lazy lists of integers, but nothing hinges on this assumption.\(^1\) For the rest of this section we’ll drop the modifier “lazy”, and just write “list”, instead of “lazy list”.

The key idea is to treat a computation of a list element as a value of list type, where a computation is simply a memoized, delayed evaluation of an expression. By admitting computations as values we can support lazy lists in a strict language. In particular the call-by-value evaluation strategy is not disrupted. Passing a lazy list to a function does not cause the delayed computation to be evaluated; rather, it is passed in delayed form to the function as a computation of that type. Pattern matching on a value of list type requires that the computation be forced to expose the underlying

---

\(^1\)It simply allows us to avoid forward-referencing the concept of polymorphism.
list element, which is then analyzed and deconstructed. It is very important to keep in mind the distinction between evaluation of an expression of list type, and forcing a value of list type. The former simply yields a computation as value, whereas the latter evaluates and memoizes the delayed computation.

One consequence of laziness is that the tail of a (non-empty) lazy list, need not “exist” at the time the non-empty list is created. Being itself a lazy list, the tail need only be produced “on demand”, by forcing a computation. This is the key to using lazy lists to model interactive input and to represent infinite data structures. For example, we might define the infinite list of natural numbers by the equation

\[ nats = \text{iterate} \ \text{successor} \ 0 \]

where the function iterate is defined (informally) by the equation

\[ \text{iterate} \ f \ x = \text{lcons} \ (x, \ \text{iterate} \ f \ (f \ x)), \]

where lcons creates a non-empty lazy list with the specified head and tail. We must think of nats as being created on demand. Successive elements of nats are created by successive recursive calls to iterate, which are only made as we explore the list.

Another approach to defining the infinite list of natural numbers is to make use of self-reference, as illustrated by the following example. The infinite sequence of natural numbers may be thought as a solution to the recursion equation

\[ nats = \text{lcons} \ (0, \ \text{lmap} \ \text{successor} \ nats), \]

where successor and lmap are the evident functions. Here again we must think of nats as being created on demand. Successive elements of nats are created as follows. When we inspect the first element of nats, it is immediately revealed to be 0, as specified. When we inspect the second element, we apply lmap successor to nats, then inspect the head element of the result. This is successor(0), or 1; it’s tail is the result of mapping successor over that list — that is, the result of adding 2 to every element of the original list, and so on.
22.1 Lazy Types in MinML

22.1.1 Lazy Lists in an Eager Language

The additional constructs required to add lazy lists to MinML are given by the following grammar:

\[
\begin{align*}
\text{Types} & \quad \tau ::= \text{lilist} \\
\text{Expressions} & \quad e ::= \text{lnil} | \text{lcons}(e_1,e_2) | \text{lazy}x\text{is}e \\
& \quad \quad \text{lcase} e\text{of} \text{lnil} => e_0 | \text{lcons}(x,y) => e_1 \text{end}
\end{align*}
\]

In the expression \(\text{lazy}x\text{is}e\) the variable \(x\) is bound within \(e\); in the expression \(\text{lcase} e\text{of} \text{lnil} => e_0 | \text{lcons}(x,y) => e_1 \text{end}\) the variables \(x\) and \(y\) are bound in \(e_1\). As usual we identify expressions that differ only in the names of their bound variables.

Lazy lists may be defined either by explicit construction — using \lilist\ and \text{lcons} — or by a recursion equation — using \(\text{lazy}x\text{is}e\), where \(e\) is a lazy list expression. The idea is that the variable \(x\) stands for the list constructed by \(e\), and may be used within \(e\) to refer to the list itself. For example, the infinite list of 1’s is given by the expression

\[
\text{lazy}x\text{is}\text{lcons}(1,x).
\]

More interesting examples can be expressed using recursive definitions such as the following definition of the list of all natural numbers:

\[
\text{lazy}x\text{is}\text{lcons}(1, \text{lmap successor} x).
\]

To complete this definition we must define \text{lmap}. This raises a subtle issue that is very easy to overlook. A natural choice is as follows:

\[
\begin{align*}
\text{fun map}(f: \text{int->int}): \text{lilist->lilist} \text{ is} \\
\text{fun lmap}(f: \text{llist}) \text{ is} \\
& \quad \text{lcase} l \\
& \quad \quad \text{of} \text{lnil} => \text{lnil} \\
& \quad \quad \text{| lcons}(x,y) => \text{lcons} (f x, \text{lmap} f y).
\end{align*}
\]

Unfortunately this definition doesn’t work as expected! Suppose that \(f\) is a function of type \text{int->int} and that \(l\) is a non-empty lazy list. Consider what happens when we evaluate the expression \text{map} \(f\) \(l\). The \text{lcase} forces evaluation of \(l\), which leads to a recursive call to the internal function \text{lmap}, which forces the evaluation of the tail of \(l\), and so on. If \(l\) is an infinite list, the application diverges.

The problem is that the result of a call to \text{map} \(f\) \(l\) should be represented by a \text{computation} of a list, in which subsequent calls to \text{map} on the tail(s) of that list are delayed until they are needed. This is achieved by the following coding trick:
fun map(f:int->int):llist->llist is
fun lmapf(l:llist) is
    [lazy . is]
    lcase l
    of lnil => lnil
    | lcons(x,y) => lcons (f x, lmapf y).

All we have done is to interpose a lazy constructor (with no name, indicated by writing an underscore) to ensure that the evaluation of the lcase expression is deferred until it is needed. Check for yourself that map f l terminates even if l is an infinite list, precisely because of the insertion of the use of lazy in the body of lmapf. This usage is so idiomatic that we sometimes write instead the following definition:

fun map(f:int->int):llist->llist is
fun [lazy] lmapf(l:llist) is
    lcase l
    of lnil => lnil
    | lcons(x,y) => lcons (f x, lmapf y).

The keyword lazy on the inner fun binding ensures that the body is evaluated lazily.

Exercise 22.1
Give a formal definition of nats in terms of iterate according to the informal equation given earlier. You will need to make use of lazy function definitions.

The static semantics of these lazy list expressions is given by the following typing rules:

$$\Gamma \vdash \text{lnil: llist}$$  (22.1)

$$\Gamma \vdash e_1: \text{int} \quad \Gamma \vdash e_2: \text{llist}$$
$$\Gamma \vdash \text{lcons}(e_1, e_2): \text{llist}$$  (22.2)

$$\Gamma, x: \text{llist} \vdash e: \text{llist}$$
$$\Gamma \vdash \text{lazye} x: \text{llist}$$  (22.3)

$$\Gamma \vdash e: \text{llist} \quad \Gamma \vdash e_0: \tau \quad \Gamma, x: \text{int}, y: \text{llist} \vdash e_1: \tau$$
$$\Gamma \vdash \text{lcase } e \text{ of lnil } => e_0 \mid \text{lcons}(x, y) => e_1 \text{ end } : \tau$$  (22.4)
In Rule 22.2 the body, $e$, of the lazy list expression $\text{lazy} \ x \ is \ e$ is type checked under the assumption that $x$ is a lazy list.

We will consider two forms of dynamic semantics for lazy lists. The first, which exposes the “evaluate on demand” character of lazy evaluation, but neglects the “evaluate at most once” aspect, is given as follows. First, we regard $\text{lnil}$, $\text{lcons}(e_1, e_2)$, and $\text{lazy} \ x \ is \ e$ to be values, independently of whether their constituent expressions are values. Second, we evaluate case analyses according to the following transition rules:

\[
\text{lcase \ lnil of lnil => } e_0 | \text{lcons}(x, y) => e_1 \text{ end} \rightleftharpoons e_0 \tag{22.5}
\]

\[
\text{lcase \ lcons}(e_h, e_t) \text{ of lnil => } e_0 | \text{lcons}(x, y) => e_1 \text{ end} \rightleftharpoons \\
\text{let } x: \text{int be } e_h \text{ in let } y: \text{llist be } e_t \text{ in } e_1 \text{ end end} \tag{22.6}
\]

\[
\text{lcase (lazy z is e) of lnil => } e_0 | \text{lcons}(x, y) => e_1 \text{ end} \rightleftharpoons \\
\text{lcase \{lazy z is e/z\}e of lnil => } e_0 | \text{lcons}(x, y) => e_1 \text{ end} \rightleftharpoons \\
e \rightleftharpoons e' \tag{22.7}
\]

\[
\text{lcase e of lnil => } e_0 | \text{lcons}(x, y) => e_1 \text{ end} \rightleftharpoons \\
\text{lcase e' of lnil => } e_0 | \text{lcons}(x, y) => e_1 \text{ end} \rightleftharpoons \tag{22.8}
\]

Observe that lazy list expressions are evaluated only when they appear as the subject of a case analysis expression. In the case of a non-empty list evaluation proceeds by first evaluating the head and tail of the list, then continuing with the appropriate clause. In the case of a recursively-defined list the expression is “unrolled” once before continuing analysis. This exposes the outermost structure of the list for further analysis.

**Exercise 22.2**

**Define the functions** $\text{lhd: llist -> int}$ and $\text{ltl: llist -> llist}$. **Trace the evaluation of** $\text{lhd(ltl(...(ltl(nats))...))}$, **with** $n$ **iterations of** $\text{ltl}$, **and verify that it evaluates to the number** $n$. 

WORKING DRAFT  

JANUARY 14, 2003
Exercise 22.3
State and prove the soundness of the non-memoizing dynamic semantics with respect to the static semantics given above.

Consider the lazy list value $v = \text{lazy } x \text{ is } x$. It is easy to verify that $e$ is well-typed, with type $\text{llist}$. It is also easy to see that performing a case analysis on $v$ leads to an infinite regress, since $\{v/x\}x = v$. The value $v$ is an example of a “black hole”, a value that, when forced, will lead back to the value itself, and, moreover, is easily seen to lead to divergence. Another example of a black hole is the value

\[
\text{lazy } x \text{ is } (\text{lmap succ } x)
\]

that, when forced, maps the successor function over itself.

What is it that makes the recursive list

\[
\text{lazy } \text{nats is } \text{lcons } (0, \text{lmap succ } \text{nats})
\]

well-defined? This expression is not a black hole because the occurrence of \text{nats} in the body of the recursive list expression is “guarded” by the call to \text{lcons}.

Exercise 22.4
Develop a type discipline that rules out black holes as ill-formed. Hint: Define a judgement $\Gamma \vdash e \downarrow x$, which means that $x$ is guarded within $e$. Ensure that $\text{lazy } x \text{ is } e$ is well-typed only if $x$ is guarded within $e$.

Exercise 22.5
It is often convenient to define several lists simultaneously by mutual recursion. Generalize $\text{lazy } x \text{ is } e$ to admit simultaneous recursive definition of several lists at once.

The foregoing dynamic semantics neglects the “evaluate at most once” aspect of laziness — if a lazy list expression is ever evaluated, its value should be stored so that re-evaluation is avoided should it ever be analyzed again. This can be modeled by introducing a memory that holds delayed computations whenever they are created. The memory is updated if (and only if) the value of that computation is ever required. Thus no evaluation is ever repeated, and some pending evaluations may never occur at all. This is called memoization.

The memoizing dynamic semantics is specified by an abstract machine with states of the form $(M, e)$, where $M$ is a memory, a finite mapping of variables to values, and $e$ is an expression whose free variables are all in...
the domain of \( M \). Free variables are used to stand for the values of list expressions; they are essentially pointers into the memory, which stores the value of the expression. We therefore regard free variables as values; these are in fact the only values of list type in this semantics.

The transition rules for the memoizing dynamic semantics are as follows:

\[
(x \notin \text{dom}(M)) \\
(M, \text{lazy is } e) \mapsto (M[x=\text{lazy is } e], x) \quad (22.9)
\]

\[
(x \notin \text{dom}(M)) \\
(M, \text{lnil}) \mapsto (M[x=\text{lnil}], x) \quad (22.10)
\]

\[
(x \notin \text{dom}(M)) \\
(M, \text{lcons}(e_1, e_2)) \mapsto (M[x=\text{lcons}(e_1, e_2)], x) \quad (22.11)
\]

\[
(M(z) = \text{lnil}) \\
(M, \text{lcase } z \text{ of lnil } => e_0 | \text{lcons}(x, y) => e_1 \text{ end}) \mapsto (M, e_0) \quad (22.12)
\]

\[
(M(z) = \text{lcons}(v_h, v_t)) \\
(M, \text{lcase } z \text{ of lnil } => e_0 | \text{lcons}(x, y) => e_1 \text{ end}) \mapsto (M, \{v_h, v_t/x, y\} e_1) \quad (22.13)
\]

\[
(M(z) = \text{lcons}(e_h, e_t)) \\
(M[z=\bullet, e_h] \mapsto^* (M', v_h) \quad (M'[z=\bullet, e_t] \mapsto^* (M'', v_t) \\
(M, \text{lcase } e \text{ of lnil } => e_0 | \text{lcons}(x, y) => e_1 \text{ end}) \mapsto (M''[z=\text{lcons}(v_h, v_t)], \{v_h, v_t/x, y\} e_1) \quad (22.14)
\]

\[
(M(z) = \text{lazy is } e) \\
(M[z=\bullet, e] \mapsto^* (M', v) \\
(M, \text{lcase } z \text{ of lnil } => e_0 | \text{lcons}(x, y) => e_1 \text{ end}) \mapsto (M'[z=v], \text{lcase } v \text{ of lnil } => e_0 | \text{lcons}(x, y) => e_1 \text{ end}) \\
(M, e) \mapsto (M', e') \quad (22.15)
\]

\[
(M, \text{lcase } e \text{ of lnil } => e_0 | \text{lcons}(x, y) => e_1 \text{ end}) \mapsto (M', \text{lcase } e' \text{ of lnil } => e_0 | \text{lcons}(x, y) => e_1 \text{ end}) \quad (22.16)
\]
Warning: These rules are very subtle! Here are some salient points to keep in mind when studying them.

First, observe that the list-forming constructs are no longer values, but instead have evaluation rules associated with them. These rules simply store a pending computation in the memory and return a “pointer” to it as result. Thus a value of lazy list type is always a variable referring to a pending computation in the store.

Second, observe that the rules for case analysis inspect the contents of memory to determine how to proceed. The case for \texttt{lnil} is entirely straightforward, but the other two cases are more complex. Suppose that location \( z \) contains \texttt{lcons}(e_1, e_2). First, we check whether we’ve already evaluated this list cell. If so, we continue by evaluating \( e_1 \), with \( x \) and \( y \) replaced by the previously-computed values of the head and tail of the list. Otherwise, the time has come to evaluate this cell. We evaluate the head and tail completely to obtain their values, then continue by substituting these values for the appropriate variables in the clause for non-empty lists. Moreover, we update the memory to record the values of the head and tail of the list so that subsequent accesses avoid re-evaluation. Similarly, if \( z \) contains a recursively-defined list, we fully evaluate its body, continuing with the result and updating the memory to reflect the result of evaluation.

Third, we explicitly check for “black holes” by ensuring that a run-time error occurs whenever they are encountered. This is achieved by temporarily setting the contents of a list cell to the special “black hole” symbol, \( \bullet \), during evaluation of a list expression, thereby ensuring the evaluation “gets stuck” (i.e., incurs a run-time error) in the case that evaluation of a list expression requires the value of the list itself.

Exercise 22.6
Convince yourself that the replacement of \( z \) by \( \bullet \) in the second premise of Rule 22.14 is redundant — the location \( z \) is already guaranteed to be bound to \( \bullet \).

Exercise 22.7
State and prove the soundness of the memoizing dynamic semantics with respect to the static semantics given above. Be certain that your treatment of the memory takes account of cyclic dependencies.

Exercise 22.8
Give an evaluation semantics for memoized lazy lists by a set of rules for deriving judgements of the form \((M, e) \Downarrow (M', v)\).
Exercise 22.9
Consider once again the augmented static semantics in which black holes are ruled out. Prove that evaluation never “gets stuck” by accessing a cell that contains the black hole symbol.

Exercise 22.10
Consider again the definition of the natural numbers as the lazy list

\[
\text{lazy nats is } (\text{lcons } (0, \text{lmap succ nats})). 
\]

Prove that, for the non-memoized semantics, that accessing the \( n \)th element requires \( O(n^2) \) time, whereas in the memoized semantics the same computation requires \( O(n) \) time. This shows that memoization can improve the asymptotic complexity of an algorithm (not merely lower the constant factors).

### 22.1.2 Delayed Evaluation and Lazy Data Structures

Another approach to lazy evaluation in the context of a strict language is to isolate the notion of a delayed computation as a separate concept. The crucial idea is that a delayed computation is a \textit{value} that can, for example, appear in a component of a data structure. Evaluation of a delayed computation occurs as a result of an explicit \textit{force} operation. Computations are implicitly memoized in the sense that the first time it is forced, its value is stored and returned immediately should it ever be forced again. Lazy data structures can then be built up using standard means, but with judicious use of delayed computations to ensure laziness.

Since the technical details of delayed computation are very similar to those just outlined for lazy lists, we will go through them only very briefly. Here is a syntactic extension to MinML that supports delayed evaluation:

\[
\begin{align*}
\text{Types} & \quad \tau ::= \tau \text{ computation} \\
\text{Expressions} & \quad e ::= \text{delay } x \text{ is } e | \text{eval } e_1 \text{ as } x \text{ in } e_2 \text{ end}
\end{align*}
\]

In the expression \( \text{delay } x \text{ is } e \) the variable \( x \) is bound within \( e \), and in the expression \( \text{eval } e_1 \text{ as } x \text{ in } e_2 \text{ end} \) the variable \( x \) is bound within \( e_2 \). The expression \( \text{delay } x \text{ is } e \) both delays evaluation of \( e \) and gives it a name that can be used within \( e \) to stand for the computation itself. The expression \( \text{eval } e_1 \text{ as } x \text{ in } e_2 \text{ end} \) forces the evaluation of the delayed computation \( e_1 \), binds that value to \( x \), and continues by evaluating \( e_2 \).

The static semantics is given by the following rules:

\[
\begin{align*}
\Gamma \vdash e : \tau \\
\Gamma \vdash \text{delay } x \text{ is } e : \tau \text{ computation}
\end{align*}
\]
A memoizing dynamic semantics for computations is given as follows. We admit, as before, variables as values; they serve as references to memo cells that contain delayed computations. The evaluation rules are as follows:

\[
\frac{\text{A memoizing dynamic semantics for computations is given as follows. We admit, as before, variables as values; they serve as references to memo cells that contain delayed computations. The evaluation rules are as follows:}}{\Gamma \vdash e_1 : \tau_1 \text{ computation } \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{eval } e_1 \text{ as } x \text{ in } e_2 \text{ end} : \tau_2} (22.18)
\]

Exercise 22.11

State and prove the soundness of this extension to MinML.

One advantage of such a type of memoized, delayed computations is that it isolates the machinery of lazy evaluation into a single type constructor that can be used to define many different lazy data structures. For example, the type `llist` of lazy lists may be defined to be the type `lcell` computation, where `lcell` has the following constructors and destructors:

\[
\frac{}{\Gamma \vdash \text{cnil} : \text{lcell}} (22.23)
\]

\[
\frac{\Gamma \vdash e_h : \text{int} \quad \Gamma \vdash e_l : \text{llist}}{\Gamma \vdash \text{ccons} (e_h, e_l) : \text{lcell}} (22.24)
\]

\[
\frac{\Gamma \vdash e : \text{lcell} \quad \Gamma \vdash e_n : \tau \quad \Gamma, x : \text{int}, y : \text{llist} \vdash e_c : \tau}{\Gamma \vdash \text{ccase } e \text{ of cnil } \Rightarrow e_n | \text{ccons} (x, y) \Rightarrow e_c : \tau} (22.25)
\]

Observe that the “tail” of a `ccons` is of type `llist`, not `lcell`. Using these primitives we may define the lazy list constructors as follows:
lnil = lazy . is cnil
lcons(eh, et) = lazy . is ccons(eh, et)

lcase e of nil => en | cons(x, y) => ec =
force z=e in
  case z of cnil => en | ccons(x, y) => ec

Observe that case analysis on a lazy list forces the computation of that list, then analyzes the form of the outermost lazy list cell.

This “two-stage” construction of lazy lists in terms of lazy cells is often short-circuited by simply identifying llist with lcell. However, this is a mistake! The reason is that according to this definition every lazy list expression must immediately determine whether the list is empty, and, if not, must determine its first element. But this conflicts with the “computation on demand” interpretation of laziness, according to which a lazy list might not even have a first element at the time that the list is defined, but only at the time that the code inspects it. It is therefore imperative to distinguish, as we have done, between the type llist of lazy lists (delayed computations of cells) and the type lcell of lazy cells (which specify emptiness and define the first element of non-empty lists).
Chapter 23

Lazy Languages

So far we’ve been considering the addition of lazy types to eager languages. Now we’ll consider the alternative, the notion of a lazy language and, briefly, the addition of eager types to a lazy language.

As we said in the introduction the main features of a lazy language are the call-by-need argument-passing discipline together with lazy value constructors that construct values of a type from delayed computations. Under call-by-value the arguments to functions and constructors are evaluated before the function is called or the constructor is applied. Variables are only ever bound to fully-evaluated expressions, or values, and constructors build values out of other values. Under call-by-need arguments are passed to functions in delayed, memoized form, without evaluating them until they are needed. Moreover, value constructors build delayed, memoized computations out of other delayed, memoized computations, without evaluation. Variables are, in general, bound to pending computations that are only forced when (and if) that value is required. Once forced, the binding is updated to record the computed value, should it ever be required again.

The interesting thing is that the static typing rules for the lazy variant of MinML are exactly the same as those for the eager version. What is different is how those types are interpreted. In an eager language values of type int are integer values (i.e., numbers); in a lazy language they are integer computations, some of which might not even terminate when evaluated. Similarly, in an eager language values of list type are finite sequences of values of the element type; in a lazy language values of list type are computations of such sequences, which need not be finite. And so on. The important point is that the types have different meanings in lazy languages.
than they do in strict languages.

One symptom of this difference is that lazy languages are very liberal in admitting recursive definitions compared to eager languages. In an eager language it makes no sense to admit recursive definitions such as

```ml
val x : int = 1 + x
```
or

```ml
val x : int list = cons (1, x).
```

Roughly speaking, neither of these recursion equations has a solution. There is no integer value \( x \) satisfying the equation \( x = 1 + x \), nor is there any finite list satisfying the equation \( x = \text{cons}(1, x) \).

However, as we've already seen, equations such as

```ml
val x : int delayed = delay (1 + x)
```
and

```ml
val x : int list delayed = delay (lcons (1, x))
```
do make sense, precisely because they define recursive computations, rather than values. The first example defines a computation of an integer that, when forced, diverges; the second defines a computation of a list that, when forced, computes a non-empty list with 1 as first element and the list itself as tail.

In a lazy language every expression stands for a computation, so it is always sensible to make a recursive definition such as

```ml
val rec x : int = 1 + x.
```

Syntactically this looks like the inadmissible definition discussed above, but, when taken in the context of a lazy interpretation, it makes perfect sense as a definition of a recursive computation — the value of \( x \) is the divergent computation of an integer.

The downside of admitting such a liberal treatment of computations is that it leaves no room in the language for ordinary values! Everything's a computation, with values emerging as those computations that happen to have a trivial evaluation (e.g., numerals are trivial computations in the sense that no work is required to evaluate them). This is often touted as an advantage of lazy languages — the “freedom” to ignore whether something is a value or not. But this appearance of freedom is really bondage. By admitting only computations, you are deprived of the ability to work
with plain values. For example, lazy languages do not have a type of natural numbers, but rather only a type of computations of natural numbers. Consequently, elementary programming techniques such as definition by mathematical induction are precluded. The baby’s been thrown out with the bathwater.

In recognition of this most lazy languages now admit eager types as well as lazy types, moving them closer in spirit to eager languages that admit lazy types, but biased in the opposite direction. This is achieved in a somewhat unsatisfactory manner, by relying on data abstraction mechanisms to ensure that the only values of a type are those that are generated by specified strict functions (those that evaluate their arguments). The reason it is unsatisfactory is that this approach merely limits the possible set of computations of a given type, but still admits, for example, the undefined computation as an element of every type.

### 23.0.3 Call-by-Name and Call-by-Need

To model lazy languages we simply extend MinML with an additional construct for recursively-defined computations, written $\text{rec} \, x : \tau \, \text{is} \, e$. The variable $x$ is bound in $e$, and may be renamed at will. Recursive computations are governed by the following typing rule:

$$
\Gamma, x : \tau \vdash e : \tau \\
\Gamma \vdash \text{rec} \, x : \tau \, \text{is} \, e : \tau
$$

(23.1)

In addition we replace the recursive function expression $\text{fun} \, f \, (x : \tau_1) : \tau_2 \, \text{is} \, e \, \text{end}$ with the non-recursive form $\text{fn} \, \tau : x \, \text{in} \, e \, \text{end}$, since the former may be defined by the expression

$$
\text{rec} \, f : \tau_1 \to \tau_2 \, \text{is} \, \text{fn} \, \tau_1 : x \, \text{in} \, e \, \text{end}.
$$

As before, it is simpler to start with a non-memoizing dynamic semantics to better expose the core ideas. We’ll work with core MinML enriched with recursive computations. Closed values are precisely as for the eager case, as are nearly all of the evaluation rules. The only exception is the rule for function application, which is as follows:

$$
\text{fn} \, \tau : x \, \text{in} \, e \, \text{end} \,(e') \mapsto \{\text{fn} \, \tau : x \, \text{in} \, e \, \text{end}, \, e'/x \} \, e
$$

(23.2)

This is known as the *call-by-name*\(^1\) rule, according to which arguments are

\(^1\)The terminology is well-established, but not especially descriptive. As near as I can tell the idea is that we pass the “name” of the computation (i.e., the expression that engenders it), rather than its value.
passed to functions in unevaluated form, deferring their evaluation until the point at which they are actually used.

The only additional rule required is the one for recursive computations. But this is entirely straightforward:

\[ \text{rec } x : \tau e \mapsto \{ \text{rec } x : \tau e / x \} e \]  

(23.3)

To evaluate a recursive computation, simply unroll the recursion by one step and continue from there.

**Exercise 23.1**

Show that the behavior of the recursive function expression \( \text{fun } f \ (x : \tau_1) : \tau_2 \ is \ e \ end \) is correctly defined by

\[ \text{rec } f : \tau_1 \to \tau_2 \ is \ fn \tau_1 : x \ ine \ end \]

in the sense that an application of the latter mimicks the behavior of the former (under call-by-name).

To model the “at most once” aspect of lazy evaluation we introduce, as before, a memory in which we store computations, initially in their unevaluated, and later, if ever, in their evaluated forms. The difference here is that all expressions define computations that must be stored. Since the main ideas are similar to those used to define lazy lists, we simply give the evaluation rules here.

The state of computation is a pair \((M, e)\) where \(M\) is a finite memory mapping variables to values, and \(e\) is an expression whose free variables lie within the domain of \(M\). Final states have the form \((M, v)\), where \(v\) is a closed value. In particular, \(v\) is not a variable.

Nearly all of the rules of MinML carry over to the present case nearly unchanged, apart from propagating the memory appropriately. For example, the rules for evaluating addition expressions is as follows:

\[ (M, e_1) \mapsto (M', e'_1) \]

\[ (M, +(e_1, e_2)) \mapsto (M', +(e'_1, e_2)) \]  

(23.4)

\[ (M, e_2) \mapsto (M', e'_2) \]

\[ (M, +(v_1, e_2)) \mapsto (M', +(v_1, e'_2)) \]  

(23.5)

\[ (M, +(n_1, n_2)) \mapsto (M, n_1 + n_2) \]  

(23.6)
The main differences are in the rule for function application and the need for additional rules for variables and recursive computations.

\[(x \notin \text{dom}(M)) \quad (M, \text{fn } \tau : x \in e \text{ end}(e')) \mapsto (M[x = e'], e) \quad (23.7)\]

\[(M(x) = v) \quad (M, x) \mapsto (M, v) \quad (23.8)\]

\[(M(x) = e) \quad (M[x = \bullet], e) \mapsto^* (M', v) \quad (M, x) \mapsto (M'[x = v], v) \quad (23.9)\]

\[(x \notin \text{dom}(M)) \quad (M, \text{rec } x : \tau \text{ is } e) \mapsto (M[x = e], e) \quad (23.10)\]

Observe that we employ the “black holing” technique to catch ill-defined recursive definitions.

23.0.4 Strict Types in a Lazy Language

As discussed above, lazy languages are committed to the fundamental principle that the elements of a type are computations, which include values, and not just values themselves. This means, in particular, that every type contains a “divergent” element, the computation that, when evaluated, goes into an infinite loop.\(^2\)

One consequence, alluded to above, is that recursive type equations have overly rich solutions. For example, in this setting the recursive type equation

\[\text{data } llist = \text{lnil} \mid \text{lcons of int } \ast \text{ list} \]

does not correspond to the familiar type of finite integer lists. In fact this type contains as elements both divergent computations of lists and also computations of infinite lists. The reason is that the tail of every list is a

\(^2\)This is often called “bottom”, written \(\bot\), for largely historical reasons. I prefer to avoid this terminology because so much confusion has been caused by it. In particular, it is not always correct to identify the least element of a domain with the divergent computation of that type! The domain of values of partial function type contains a least element, the totally undefined function, but this element does not correspond to the divergent computation of that type.
computation of another list, so we can easily use recursion equations such as

\[
\text{rec ones is } \text{lcons } (1, \text{ones})
\]

to define an infinite element of this type.

The inclusion of divergent expressions in every type is unavoidable in a lazy language, precisely because of the commitment to the interpretation of types as computations. However, we can rule out infinite lists (for example) by insisting that \text{cons} evaluate its tail whenever it is applied. This is called a \textit{strictness} annotation. If \text{cons} is strict in its second argument, then the equation

\[
\text{rec ones is } \text{cons } (1, \text{ones})
\]

denotes the divergent computation, rather than the infinite list of ones.

These informal ideas correspond to different rules for evaluating constructors. We will illustrate this by giving a non-memoizing semantics for lazy MinML extended with eager lists. It is straightforward to adapt this to the memoizing case.

In the fully lazy case the rules for evaluation are these. First, we regard \text{lnil} as a value, and regard \text{lcons}\((e_1,e_2)\) as a value, regardless of whether \(e_1\) or \(e_2\) are values. Then we define the transition rules for case analysis as follows:

\[
\text{lcase } \text{lnil} \text{ of } \text{lnil} \Rightarrow e_n \mid \text{lcons}\((x,y)\) \Rightarrow e_c \text{ end } \mapsto e_n \quad (23.11)
\]

\[
\text{lcase } \text{lcons}\((e_1,e_2)\) \text{ of } \text{lnil} \Rightarrow e_n \mid \text{lcons}\((x,y)\) \Rightarrow e_c \text{ end } \mapsto \{e_1,e_2/x,y\}e_c \quad (23.12)
\]

If instead we wish to rule out infinite lists, then we may choose to regard \text{lcons}\((e_1,e_2)\) to be a value only if \(e_2\) is a value, without changing the rules for case analysis. If we wish the elements of the list to be values, then we consider \text{lcons}\((e_1,e_2)\) to be a value only in the case that \(e_1\) is a value, and so on for all the possible combinations of choices.

As we stated earlier, this cuts down the set of possible computations of, say, list type, but retains the fundamental commitment to the interpretation of all types as types of computations.
Part X

Dynamic Typing
Chapter 24

Dynamic Typing

The formalization of type safety given in Chapter 9 states that a language is type safe iff it satisfies both preservation and progress. According to this account, “stuck” states — non-final states with no transition — must be rejected by the static type system as ill-typed. Although this requirement seems natural for relatively simple languages such as MinML, it is not immediately clear that our formalization of type safety scales to larger languages, nor is it entirely clear that the informal notion of safety is faithfully captured by the preservation and progress theorems.

One issue that we addressed in Chapter 9 was how to handle expressions such as 3 div 0, which are well-typed, yet stuck, in apparent violation of the progress theorem. We discussed two possible ways to handle such a situation. One is to enrich the type system so that such an expression is ill-typed. However, this takes us considerably beyond the capabilities of current type systems for practical programming languages. The alternative is to ensure that such ill-defined states are not “stuck”, but rather make a transition to a designated error state. To do so we introduced the notion of a checked error, which is explicitly detected and signalled during execution. Checked errors are constrained with unchecked errors, which are ruled out by the static semantics.

In this chapter we will concern ourselves with question of why there should unchecked errors at all. Why aren’t all errors, including type errors, checked at run-time? Then we can dispense with the static semantics entirely, and, in the process, execute more programs. Such a language is called dynamically typed, in contrast to MinML, which is statically typed.

One advantage of dynamic typing is that it supports a more flexible treatment of conditionals. For example, the expression
(if true then 7 else "7")+1

is statically ill-typed, yet it executes successfully without getting stuck or incurring a checked error. Why rule it out, simply because the type checker is unable to "prove" that the else branch cannot be taken? Instead we may shift the burden to the programmer, who is required to maintain invariants that ensure that no run-time type errors can occur, even though the program may contain conditionals such as this one.

Another advantage of dynamic typing is that it supports heterogeneous data structures, which may contain elements of many different types. For example, we may wish to form the “list”

\[ \text{[true, 1, 3.4, fn x=>x]} \]

consisting of four values of distinct type. Languages such as ML preclude formation of such a list, insisting instead that all elements have the same type; these are called homogenous lists. The argument for heterogeneity is that there is nothing inherently “wrong” with such a list, particularly since its constructors are insensitive to the types of the components — they simply allocate a new node in the heap, and initialize it appropriately.

Note, however, that the additional flexibility afforded by dynamic typing comes at a cost. Since we cannot accurately predict the outcome of a conditional branch, nor the type of a value extracted from a heterogeneous data structure, we must program defensively to ensure that nothing bad happens, even in the case of a type error. This is achieved by turning type errors into checked errors, thereby ensuring progress and hence safety, even in the absence of a static type discipline. Thus dynamic typing catches type errors as late as possible in the development cycle, whereas static typing catches them as early as possible.

In this chapter we will investigate a dynamically typed variant of MinML in which type errors are treated as checked errors at execution time. Our analysis will reveal that, rather than being opposite viewpoints, dynamic typing is a special case of static typing! In this sense static typing is more expressive than dynamic typing, despite the superficial impression created by the examples given above. This viewpoint illustrates the pay-as-you-go principle of language design, which states that a program should only incur overhead for those language features that it actually uses. By viewing dynamic typing as a special case of static typing, we may avail ourselves of the benefits of dynamic typing whenever it is required, but avoid its costs whenever it is not.
24.1 Dynamic Typing

The fundamental idea of dynamic typing is to regard type clashes as *checked*, rather than *unchecked*, errors. Doing so puts type errors on a par with division by zero and other checked errors. This is achieved by augmenting the dynamic semantics with rules that explicitly check for stuck states. For example, the expression `true+7` is such an ill-typed, stuck state. By checking that the arguments of an addition are integers, we can ensure that progress may be made, namely by making a transition to `error`.

The idea is easily illustrated by example. Consider the rules for function application in MinML given in Chapter 8, which we repeat here for convenience:

$$
\begin{align*}
\text{v value } & \quad \text{v₁ value } \quad (v = \text{fun } f \, (x: \tau₁): \tau₂ \text{ is e end}) \\
& \quad \text{apply}(v, v₁) \mapsto \{v, v₁/f, x\} e \\
& \quad e₁ \mapsto e₁' \\
& \quad \text{apply}(e₁, e₂) \mapsto \text{apply}(e₁', e₂) \\
& \quad v₁ \text{ value } \\
& \quad e₂ \mapsto e₂' \\
& \quad \text{apply}(v₁, e₂) \mapsto \text{apply}(v₁, e₂')
\end{align*}
$$

In addition to these rules, which govern the well-typed case, we add the following rules governing the ill-typed case:

$$
\begin{align*}
\text{v value } & \quad \text{v₁ value } \quad (v \neq \text{fun } f \, (x: \tau₁): \tau₂ \text{ is e end}) \\
& \quad \text{apply}(v, v₁) \mapsto \text{error} \\
& \quad \text{apply}(\text{error}, e₂) \mapsto \text{error} \\
& \quad v₁ \text{ value } \\
& \quad \text{apply}(v₁, \text{error}) \mapsto \text{error}
\end{align*}
$$

The first rule states that a run-time error arises from any attempt to apply a non-function to an argument. The other two define the propagation of such errors through other expressions — once an error occurs, it propagates throughout the entire program.

By entirely analogous means we may augment the rest of the semantics of MinML with rules to check for type errors at run time. Once we have done so, it is safe to eliminate the static semantics in its entirety.¹ Having done

¹We may then simplify the language by omitting type declarations on variables and functions, since these are no longer of any use.
so, every expression is well-formed, and hence preservation holds vacuously. More importantly, the progress theorem also holds because we have augmented the dynamic semantics with transitions from every ill-typed expression to error, ensuring that there are no “stuck” states. Thus, the dynamically typed variant of MinML is safe in same sense as the statically typed variant. The meaning of safety does not change, only the means by which it is achieved.

24.2 Implementing Dynamic Typing

Since both the statically- and the dynamically typed variants of MinML are safe, it is natural to ask which is better. The main difference is in how early errors are detected — at compile time for static languages, at run time for dynamic languages. Is it better to catch errors early, but rule out some useful programs, or catch them late, but admit more programs? Rather than attempt to settle this question, we will sidestep it by showing that the apparent dichotomy between static and dynamic typing is illusory by showing that dynamic typing is a mode of use of static typing. From this point of view static and dynamic typing are matters of design for a particular program (which to use in a given situation), rather than a doctrinal debate about the design of a programming language (which to use in all situations).

To see how this is possible, let us consider what is involved in implementing a dynamically typed language. The dynamically typed variant of MinML sketched above includes rules for run-time type checking. For example, the dynamic semantics includes a rule that explicitly checks for an attempt to apply a non-function to an argument. How might such a check be implemented? The chief problem is that the natural representations of data values on a computer do not support such tests. For example, a function might be represented as a word representing a pointer to a region of memory containing a sequence of machine language instructions. An integer might be represented as a word interpreted as a two’s complement integer. But given a word, you cannot tell, in general, whether it is an integer or a code pointer.

To support run-time type checking, we must adulterate our data representations to ensure that it is possible to implement the required checks. We must be able to tell by looking at the value whether it is an integer, a boolean, or a function. Having done so, we must be able to recover the underlying value (integer, boolean, or function) for direct calculation. When-
24.2 Implementing Dynamic Typing 223

ever a value of a type is created, it must be marked with appropriate information to identify the sort of value it represents.

There are many schemes for doing this, but at a high level they all amount to attaching a tag to a “raw” value that identifies the value as an integer, boolean, or function. Dynamic typing then amounts to checking and stripping tags from data during computation, transitioning to error whenever data values are tagged inappropriately. From this point of view, we see that dynamic typing should not be described as “run-time type checking”, because we are not checking types at run-time, but rather tags. The difference can be seen in the application rule given above: we check only that the first argument of an application is some function, not whether it is well-typed in the sense of the MinML static semantics.

To clarify these points, we will make explicit the manipulation of tags required to support dynamic typing. To begin with, we revise the grammar of MinML to make a distinction between tagged and untagged values, as follows:

\[\begin{align*}
\text{Expressions} & \quad e ::= x | v | o(e_1, \ldots, e_n) | \text{if} \ e_1 \ \text{then} \ e_2 \ \text{else} \ e_1 \ \text{fi} | \text{apply}(e_1, e_2) \\
\text{TaggedValues} & \quad v ::= \text{Int}(n) | \text{Bool}(\text{true}) | \text{Bool}(\text{false}) | \text{Fun}(\text{fun} \ x (y: \tau_1): \tau_2 \text{is} \text{end}) \\
\text{UntaggedValues} & \quad u ::= \text{true} | \text{false} | n | \text{fun} \ x (y: \tau_1): \tau_2 \text{is} \text{end}
\end{align*}\]

Note that only tagged values arise as expressions; untagged values are used strictly for “internal” purposes in the dynamic semantics. Moreover, we do not admit general tagged expressions such as \text{Int}(e), but only explicitly-tagged values.

Second, we introduce tag checking rules that determine whether or not a tagged value has a given tag, and, if so, extracts its underlying untagged value. In the case of functions these are given as rules for deriving judgements of the form \( v \text{ is } \text{fun} \ u \), which checks that \( v \) has the form \( \text{Fun} \ (u) \), and extracts \( u \) from it if so, and for judgements of the form \( v \text{ is } \text{nt} \text{ fun} \), that checks that \( v \) does not have the form \( \text{Fun} \ (u) \) for any untagged value \( u \).

\[\begin{align*}
\text{Fun}(u) & \quad \text{is } \text{fun} \ u \\
\text{Int}(\_)) & \quad \text{is} \text{nt } \text{fun} \quad \text{Bool}(\_)) & \quad \text{is} \text{nt } \text{fun}
\end{align*}\]

Similar judgements and rules are used to identify integers and booleans, and to extract their underlying untagged values.
Finally, the dynamic semantics is re-formulated to make use of these judgement forms. For example, the rules for application are as follows:

\[
\frac{\text{\(v_1\) value \(v\) is fun fun \(f\) \(x: \tau_1\) : \tau_2\) is end}}{\text{apply \((v, v_1)\) \mapsto \{v, v_1/f, x\}e}}
\]

\[
\frac{\text{\(v\) value \(v\) isn't fun}}{\text{apply \((v_1, v_1)\) \mapsto \text{error}}}
\]

Similar rules govern the arithmetic primitives and the conditional expression. For example, here are the rules for addition:

\[
\frac{\text{\(v_1\) value \(v_2\) value \(v_1\) is int \(n_1\) \(v_2\) is int \(n_2\) \(n = n_1 + n_2\)}}{\text{\(+\(v_1, v_2\) \mapsto \text{Int} \(n\)}}
\]

Note that we must explicitly check that the arguments are tagged as integers, and that we must apply the integer tag to the result of the addition.

\[
\frac{\text{\(v_1\) value \(v_2\) value \(v_1\) isn't int}}{\text{\(+\(v_1, v_2\) \mapsto \text{error}}}
\]

\[
\frac{\text{\(v_1\) value \(v_2\) value \(v_1\) is int \(n_1\) \(v_2\) isn't int}}{\text{\(+\(v_1, v_2\) \mapsto \text{error}}}
\]

These rules explicitly check for non-integer arguments to addition.

### 24.3 Dynamic Typing as Static Typing

Once tag checking is made explicit, it is easier to see its hidden costs in both time and space — time to check tags, to apply them, and to extract the underlying untagged values, and space for the tags themselves. This is a significant overhead. Moreover, this overhead is imposed whether or not the original program is statically type correct. That is, even if we can prove that no run-time type error can occur, the dynamic semantics nevertheless dutifully performs tagging and untagging, just as if there were no type system at all.

This violates a basic principle of language design, called the pay-as-you-go principle. This principle states that a language should impose the cost of a feature only to the extent that it is actually used in a program. With dynamic typing we pay for the cost of tag checking, even if the program is statically well-typed! For example, if all of the lists in a program are...
homogeneous, we should not have to pay the overhead of supporting heterogeneous lists. The choice should be in the hands of the programmer, not the language designer.

It turns out that we can eat our cake and have it too! The key is a simple, but powerful, observation: dynamic typing is but a mode of use of static typing, provided that our static type system includes a type of tagged data! Dynamic typing emerges as a particular style of programming with tagged data.

The point is most easily illustrated using ML. The type of tagged data values for MinML may be introduced as follows:

```ml
(* The type of tagged values. *)
datatype tagged =
  Int of int |
  Bool of bool |
  Fun of tagged -> tagged
```

Values of type tagged are marked with a value constructor indicating their outermost form. Tags may be manipulated using pattern matching.

Second, we introduce operations on tagged data values, such as addition or function call, that explicitly check for run-time type errors.

```ml
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
    Fun g => g a
  | _ => raise TypeError
```

Observe that these functions correspond precisely to the instrumented dynamic semantics given above.

Using these operations, we can then build heterogeneous lists as values of tagged list.

```ml
val het_list : tagged list =
  [Int 1, Bool true, Fun (fn x => x)]
val f : tagged = hd(tl(tl het_list))
val x : tagged = checked_apply (f, Int 5)
```
The tags on the elements serve to identify what sort of element it is: an integer, a boolean, or a function.

It is enlightening to consider a dynamically typed version of the factorial function:

```haskell
fun dyn_fact (n : tagged) =
  let fun loop (n, a) =
      case n
      of Int m =>
        (case m
         of 0 => a
         | m => loop (Int (m-1),
             checked_mult (m, a)))
      | _ => raise RuntimeTypeError
    in loop (n, Int 1)
  end

Notice that tags must be manipulated within the loop, even though we can prove (by static typing) that they are not necessary! Ideally, we would like to hoist these checks out of the loop:

```haskell
fun opt_dyn_fact (n : tagged) =
  let fun loop (0, a) = a
      | loop (n, a) = loop (n-1, n*a)
  in case n
      of Int m => Int (loop (m, 1))
      | _ => raise RuntimeTypeError
  end
```

It is very hard for a compiler to do this hoisting reliably. But if you consider dynamic typing to be a special case of static typing, as we do here, there is no obstacle to doing this optimization yourself, as we have illustrated here.
Chapter 25

Featherweight Java

We will consider a tiny subset of the Java language, called Featherweight Java, or FJ, that models subtyping and inheritance in Java. We will then discuss design alternatives in the context of FJ. For example, in FJ, as in Java, the subtype relation is tightly coupled to the subclass relation. Is this necessary? Is it desirable? We will also use FJ as a framework for discussing other aspects of Java, including interfaces, privacy, and arrays.

25.1 Abstract Syntax

The abstract syntax of FJ is given by the following grammar:

- **Classes**  \( C ::= \text{class } c \text{ extends } c \{ c f; k d \} \)
- **Constructors**  \( k ::= c(c x) \{ \text{super}(x); \_this.f=x; \} \)
- **Methods**  \( d ::= c m(c x) \{ \text{return } e; \} \)
- **Types**  \( \tau ::= c \)
- **Expressions**  \( e ::= x | e.f | e.m(e) | \text{new } c(e) | (c) e \)

The variable \( f \) ranges over a set of field names, \( c \) over a set of class names, \( m \) over a set of method names, and \( x \) over a set of variable names. We assume that these sets are countably infinite and pairwise disjoint. We assume that there is a distinguished class name, \texttt{Object}, standing for the root of the class hierarchy. Its role will become clear below. We assume that there is a distinguished variable \texttt{this} that cannot otherwise be declared in a program.

As a notational convenience we use “underbarring” to stand for sequences of phrases. For example, \( d \) stands for a sequence of \( d \)'s, whose
individual elements we designate \( d_1, \ldots, d_k \), where \( k \) is the length of the sequence. We write \( cf \) for the sequence \( c_1 f_1, \ldots, c_k f_k \), where \( k \) is the length of the sequences \( c \) and \( f \). Similar conventions govern the other uses of sequence notation.

The class expression

\[
\text{class } c \text{ extends } c' \{ cf; k \ d \}
\]

declares the class \( c \) to be a subclass of the class \( c' \). The subclass has additional fields \( c f \), single constructor \( k \), and method suite \( d \). The methods of the subclass may override those of the superclass, or may be new methods specific to the subclass.

The constructor expression

\[
c(c', x', x) \{ \text{super}(x'); \text{this.f=x';} \}
\]

declares the constructor for class \( c \) with arguments \( c', x', x \), corresponding to the fields of the superclass followed by those of the subclass. The variables \( x' \) and \( x \) are bound in the body of the constructor. The body of the constructor indicates the initialization of the superclass with the arguments \( x' \) and of the subclass with arguments \( x \).

The method expression

\[
c m (c x) \{ \text{return } e; \}
\]

declares a method \( m \) yielding a value of class \( c \), with arguments \( x \) of class \( c \) and body returning the value of the expression \( e \). The variables \( x \) and \( \text{this} \) are bound in \( e \).

The set of types is, for the time being, limited to the set of class names. That is, the only types are those declared by a class. In Java there are more types than just these, including the primitive types \text{integer} \ and \text{boolean} \ and the array types.

The set of expressions is the minimal “interesting” set sufficient to illustrate subtyping and inheritance. The expression \( e.f \) selects the contents of field \( f \) from instance \( e \). The expression \( e.m(e) \) invokes the method \( m \) of instance \( e \) with arguments \( x \). The expression \( \text{new } c(e) \) creates a new instance of class \( c \), passing arguments \( x \) to the constructor for \( c \). The expression \( (c) \ e \) casts the value of \( e \) to class \( c \).

The methods of a class may invoke one another by sending messages to \text{this}, standing for the instance itself. We may think of \text{this} as a bound variable of the instance, but we will arrange things so that renaming of \text{this} is never necessary to avoid conflicts.
A class table $T$ is a finite function assigning classes to class names. The classes declared in the class table are bound within the table so that all classes may refer to one another via the class table.

A program is a pair $(T, e)$ consisting of a class table $T$ and an expression $e$. We generally suppress explicit mention of the class table, and consider programs to be expressions.

A small example of FJ code is given in Figure 25.1. In this example we assume given a class Object of all objects and make use of types int and color that are not, formally, part of FJ.
25.2 Static Semantics

The static semantics of FJ is defined by a collection of judgments of the following forms:

- \( \tau \subseteq \tau' \) \hspace{1cm} \text{subtyping}
- \( \Gamma \vdash e : \tau \) \hspace{1cm} \text{expression typing}
- \( d \text{ ok in } c \) \hspace{1cm} \text{well-formed method}
- \( C \text{ ok} \) \hspace{1cm} \text{well-formed class}
- \( T \text{ ok} \) \hspace{1cm} \text{well-formed class table}
- \( \text{fields}(c) = f \) \hspace{1cm} \text{field lookup}
- \( \text{type}(m, c) = \rightarrow c \) \hspace{1cm} \text{method type}

The rules defining the static semantics follow.

Every variable must be declared:

\[
\begin{align*}
\Gamma(x) &= \tau \\
\Gamma \vdash x : \tau
\end{align*}
\] (25.1)

The types of fields are defined in the class table.

\[
\begin{align*}
\Gamma \vdash e_0 : c_0 & \quad \text{fields}(c_0) = f \\
\Gamma \vdash e_0.f_i : c_i
\end{align*}
\] (25.2)

The argument and result types of methods are defined in the class table.

\[
\begin{align*}
\Gamma \vdash e_0 : c_0 & \quad \Gamma \vdash e : c \\
\text{type}(m, c_0) &= \rightarrow c \quad \rightarrow c' \\
\Gamma \vdash e_0.m(e) : c
\end{align*}
\] (25.3)

Instantiation must provide values for all instance variables as arguments to the constructor.

\[
\begin{align*}
\Gamma \vdash e : c & \quad \rightarrow c' \quad \text{fields}(c) = f \\
\Gamma \vdash \text{new } e_{0}(e) : c
\end{align*}
\] (25.4)

All casts are statically valid, but must be checked at run-time.

\[
\begin{align*}
\Gamma \vdash e_0 : d \\
\Gamma \vdash (c) e_0 : c
\end{align*}
\] (25.5)
25.2 Static Semantics

The subtyping relation is read directly from the class table. Subtyping is the smallest reflexive, transitive relation containing the subclass relation:

\[ \tau <: \tau \]  
(25.6)

\[ \tau <: \tau' \quad \tau' <: \tau'' \]
\[ \tau <: \tau'' \]  
(25.7)

\[ T(c) = \text{class } c \text{ extends } c' \{ \ldots ; \ldots \} \]
\[ c <: c' \]  
(25.8)

A well-formed class has zero or more fields, a constructor that initializes the superclass and the subclass fields, and zero or more methods. To account for method override, the typing rules for each method are relative to the class in which it is defined.

\[ k = c(c' x', c x) \{ \text{super}(c') ; \text{this.f=x}; \} \]
\[ \text{fields}(c') = c' f' \quad c'' \text{ok in } c \]
\[ \text{class } c \text{ extends } c' \{ c f ; k c'' \} \text{ok} \]  
(25.9)

Method overriding takes account of the type of the method in the superclass. The subclass method must have the same argument types and result type as in the superclass.

\[ T(c) = \text{class } c \text{ extends } c' \{ \ldots ; \ldots \} \]
\[ \text{type}(m, c') = c \rightarrow c_0 \quad c ; c_0 ; \text{this.c} ; c_0 \]
\[ c_0 m(c x) \{ \text{return } c_0 ; \} \text{ok in } c \]  
(25.10)

A method table is well-formed iff all of its classes are well-formed:

\[ \forall c \in \text{dom}(T) \quad T(c) \text{ok} \]
\[ T \text{ok} \]  
(25.11)

Note that well-formedness of a class is relative to the method table!

A program is well-formed iff its method table is well-formed and the expression is well-formed:

\[ T \text{ok} \quad \emptyset \vdash e : \tau \]
\[ (T, e) \text{ok} \]  
(25.12)
The auxiliary lookup judgments determine the types of fields and methods of an object. The types of the fields of an object are determined by the following rules:

\[ \text{fields}(\text{Object}) = \bullet \] (25.13)

\[ T(c) = \text{class}\ c\ \text{extends}\ c' \ \{ f ; \ldots \} \quad \text{fields}(c') = c' f' \]
\[ \text{fields}(c) = c f' c f \] (25.14)

The type of a method is determined by the following rules:

\[ T(c) = \text{class}\ c\ \text{extends}\ c' \ \{ \ldots ; \ldots \ c'' \} \]
\[ c'' = c_i m (c_j e) \ \{ \text{return} \ e; \} \]
\[ \text{type}(m_i; c) = c_i \rightarrow c_i \] (25.15)

\[ T(c) = \text{class}\ c\ \text{extends}\ c' \ \{ \ldots ; \ldots \ c'' \} \]
\[ m \notin c'' \quad \text{type}(m; c') = c_i \rightarrow c_i \]
\[ \text{type}(m; c) = c_i \rightarrow c_i \] (25.16)

### 25.3 Dynamic Semantics

The dynamic semantics of FJ may be specified using SOS rules similar to those for MinML. The transition relation is indexed by a class table \( T \), which governs the semantics of casting and dynamic dispatch (which see below). In the rules below we omit explicit mention of the class table for the sake of brevity.

An instance of a class has the form \( \text{new}\ c (\varepsilon) \), where each \( e_i \) is a value.

\[ \varepsilon \ \text{value} \]
\[ \text{new}\ c (\varepsilon) \ \text{value} \] (25.17)

Since we arrange that there be a one-to-one correspondence between instance variables and constructor arguments, an instance expression of this form carries all of the information required to determine the values of the fields of the instance. This makes clear that an instance is essentially just a labelled collection of fields. Each instance is labelled with its class, which is used to guide method dispatch.
Field selection retrieves the value of the named field from either the subclass or its superclass, as appropriate.

\[
\text{fields}(c) = c'.f'.c.f \quad c'.\text{value} \quad e_0 \quad \text{value} \\
\text{new}\ c(e',e) \cdot f' \mapsto e'_i
\] (25.18)

\[
\text{fields}(c) = c'.f',c.f \quad c'.\text{value} \quad e_0 \quad \text{value} \\
\text{new}\ c(e',e) \cdot f_i \mapsto e_i
\] (25.19)

Message send replaces \texttt{this} by the instance itself, and replaces the method parameters by their values.

\[
\text{body}(m,c) = x \mapsto e_0 \quad e_0 \quad e_0 \quad \text{value} \quad e_0' \quad \text{value} \\
\text{new}\ c(e) \cdot m(e') \mapsto \{e'/x\}\{\text{new}\ c(e)/\texttt{this}\}e_0
\] (25.20)

Casting checks that the instance is of a sub-class of the target class, and yields the instance.

\[
c <: c' \quad \text{value} \\
(c') \quad \text{new}\ c(e) \mapsto \text{new}\ c(e)
\] (25.21)

These rules determine the order of evaluation:

\[
e_0 \mapsto e'_0 \\
e_0 \cdot f \mapsto e'_0 \cdot f
\] (25.22)

\[
e_0 \mapsto e'_0 \\
e_0 \cdot m(e) \mapsto e'_0 \cdot m(e)
\] (25.23)

\[
e_0 \quad \text{value} \quad e \mapsto e' \\
e_0 \cdot m(e) \mapsto e_0 \cdot m(e')
\] (25.24)

\[
e \mapsto e' \\
\text{new}\ c(e) \mapsto \text{new}\ c(e')
\] (25.25)

\[
e_0 \mapsto e'_0 \\
(c)\ e_0 \mapsto (c)\ e'_0
\] (25.26)
Dynamic dispatch makes use of the following auxiliary relation to find the correct method body.

\[
T(c) = \text{class } c \text{ extends } c' \{ \ldots ; \ldots d \} \\
\begin{align*}
&d_i = c_i \{ m(c_i \ x) \ \{ \text{return } e; \} \} \\
&\text{body}(m_i, c) = x \rightarrow e
\end{align*}
\] (25.27)

\[
T(c) = \text{class } c \text{ extends } c' \{ \ldots ; \ldots d \} \\
\begin{align*}
m \notin d \quad \text{type}(m, c') = x \rightarrow e \\
&\text{body}(m, c) = x \rightarrow e
\end{align*}
\] (25.28)

Finally, we require rules for evaluating sequences of expressions from left to right, and correspondingly defining when a sequence is a value (i.e., consists only of values).

\[
\begin{align*}
&\begin{array}{llll}
e_1 \text{ value} & \ldots & e_{i-1} \text{ value} & e_i \mapsto e'_i \\
e_1, \ldots, e_{i-1}, e_i, e_{i+1}, \ldots, e_n \mapsto e_1, \ldots, e_{i-1}, e'_i, e_{i+1}, \ldots, e_n
\end{array}
\end{align*}
\] (25.29)

\[
\begin{align*}
&\begin{array}{llll}
e_1 \text{ value} & \ldots & e_n \text{ value}
\end{array}
\end{align*}
\] (25.30)

This completes the dynamic semantics of FJ.

### 25.4 Type Safety

The safety of FJ is stated in the usual manner by the Preservation and Progress Theorems.

Since the dynamic semantics of casts preserves the “true” type of an instance, the type of an expression may become “smaller” in the subtype ordering during execution.

**Theorem 25.1 (Preservation)**

Assume that \( T \) is a well-formed class table. If \( e : \tau \) and \( e \mapsto e' \), then \( e' : \tau' \) for some \( \tau' \) such that \( \tau' < : \tau \).

The statement of Progress must take account of the possibility that a cast may fail at execution time. Note, however, that field selection or message send can never fail — the required field or method will always be present.
Theorem 25.2 (Progress)
Assume that $T$ is a well-formed class table. If $e : \tau$ then either

1. $v$ value, or

2. $e$ contains an instruction of the form $(c)\text{ new } (e_0)$ with $e_0$ value and $d \not\prec c$, or

3. there exists $e'$ such that $e \mapsto e'$.

It follows that if no casts occur in the source program, then the second case cannot arise. This can be sharpened somewhat to admit source-level casts for which it is known statically that the type of casted expression is a subtype of the target of the cast. However, we cannot predict, in general, statically whether a given cast will succeed or fail dynamically.

Lemma 25.3 (Canonical Forms)
If $e : c$ and $e$ value, then $e$ has the form $\text{ new } d (e_0)$ with $e_0$ value and $d \prec c$.

25.5 Acknowledgement

This chapter is based on “Featherweight Java: A Minimal Core Calculus for Java and GJ” by Atsushi Igarashi, Benjamin Pierce, and Philip Wadler.
Part XI

Subtyping and Inheritance
Chapter 26

Subtyping

A subtype relation is a pre-order\(^1\) on types that validates the subsumption principle: if \(\sigma\) is a subtype of \(\tau\), then a value of type \(\sigma\) may be provided whenever a value of type \(\tau\) is required. This means that a value of the subtype should “act like” a value of the supertype when used in supertype contexts.

26.1 MinML With Subtyping

We will consider two extensions of MinML with subtyping. The first, \textit{MinML with implicit subtyping}, is obtained by adding the following rule of implicit subsumption to the typing rules of MinML:

\[
\Gamma \vdash e : \sigma \\
\sigma < : \tau \\
\Gamma \vdash e : \tau
\]

With implicit subtyping the typing relation is no longer syntax-directed, since the subsumption rule may be applied to any expression \(e\), without regard to its form.

The second, called \textit{MinML with explicit subtyping}, is obtained by adding to the syntax by adding an explicit cast expression, \((\tau)\ e\), with the following typing rule:

\[
\Gamma \vdash e : \sigma \\
\sigma < : \tau \\
\Gamma \vdash (\tau)\ e : \tau
\]

The typing rules remain syntax-directed, but all uses of subtyping must be explicitly indicated.

---

\(^1\) A pre-order is a reflexive and transitive binary relation.
We will refer to either variation as MinML<: when the distinction does not matter. When it does, the implicit version is designated MinML^i<:, the implicit MinML^e<:.

To obtain a complete instance of MinML<: we must specify the subtype relation. This is achieved by giving a set of subtyping axioms, which determine the primitive subtype relationships, and a set of variance rules, which determine how type constructors interact with subtyping. To ensure that the subtype relation is a pre-order, we tacitly include the following rules of reflexivity and transitivity:

\[
\begin{align*}
\tau & <: \tau \\
\sigma & <: \tau \\
\rho & <: \tau \quad \text{(transitivity)}
\end{align*}
\]

Note that pure MinML is obtained as an instance of MinML^i<: by giving no subtyping rules beyond these two, so that \(\sigma <: \tau\) iff \(\sigma = \tau\).

The dynamic semantics of an instance of MinML<: must be careful to take account of subtyping. In the case of implicit subsumption the dynamic semantics must be defined so that the primitive operations of a supertype apply equally well to a value of any subtype. In the case of explicit subsumption we need only ensure that there be a means of casting a value of the subtype into a corresponding value of the supertype.

The type safety of MinML<:, in either formulation, is assured, provided that the following subtyping safety conditions are met:

- For MinML^e<:, if \(\sigma <: \tau\), then casting a value of the subtype \(\sigma\) to the supertype \(\tau\) must yield a value of type \(\tau\).

- For MinML^i<:, the dynamic semantics must ensure that the value of each primitive operation is defined for closed values of any subtype of the expected type of its arguments.

Under these conditions we may prove the Progress and Preservation Theorems for either variant of MinML<:.

**Theorem 26.1 (Preservation)**

For either variant of MinML<:, under the assumption that the subtyping safety conditions hold, if \(e : \tau\) and \(e \Rightarrow e'\), then \(e' : \tau\).

**Proof:** By induction on the dynamic semantics, appealing to the casting condition in the case of the explicit subsumption rule of MinML^e<:.

WORKING DRAFT

JANUARY 14, 2003
26.2 Varieties of Subtyping

Theorem 26.2 (Progress)
For either variant of MinML, under the assumption that the subtyping safety conditions hold, if \( e : \tau \), then either \( e \) is a value or there exists \( e' \) such that \( e \rightarrow e' \).

Proof: By induction on typing, appealing to the subtyping condition on primitive operations in the case of primitive instruction steps.

26.2 Varieties of Subtyping

In this section we will explore several different forms of subtyping in the context of extensions of MinML. To simplify the presentation of the examples, we tacitly assume that the dynamic semantics of casts is defined so that \( (\tau) v \rightarrow v \), unless otherwise specified.

26.2.1 Arithmetic Subtyping

In informal mathematics we tacitly treat integers as real numbers, even though \( \mathbb{Z} \not\subseteq \mathbb{R} \). This is justified by the observation that there is an injection \( \iota : \mathbb{Z} \hookrightarrow \mathbb{R} \) that assigns a canonical representation of an integer as a real number. This injection preserves the ordering, and commutes with the arithmetic operations in the sense that \( \iota(m + n) = \iota(m) + \iota(n) \), where \( m \) and \( n \) are integers, and the relevant addition operation is determined by the types of its arguments.

In most cases the real numbers are (crudely) approximated by floating point numbers. Let us therefore consider an extension of MinML with an additional base type, \( \text{float} \), of floating point numbers. It is not necessary to be very specific about this extension, except to say that we enrich the language with floating point constants and arithmetic operations. We will designate the floating point operations using a decimal point, writing \( . \) for floating point addition, and so forth.\(^2\)

By analogy with mathematical practice, we will consider taking the type \( \text{int} \) to be a subtype of \( \text{float} \). The analogy is inexact, because of the limitations of computer arithmetic, but it is, nevertheless, informative to consider it.

To ensure the safety of explicit subsumption we must define how to cast an integer to a floating point number, written \( (\text{float}) n \). We simply postu-

\(^2\)This convention is borrowed from O'Caml.
late that this is possible, writing \( n.0 \) for the floating point representation of the integer \( n \), and noting that \( n.0 \) has type float.\(^3\)

To ensure the safety of implicit subsumption we must ensure that the floating point arithmetic operations are well-defined for integer arguments. For example, we must ensure that an expression such as \( +. (3, 4) \) has a well-defined value as a floating point number. To achieve this, we simply require that floating point operations implicitly convert any integer arguments to floating point before performing the operation. In the foregoing example evaluation proceeds as follows:

\[
+. (3, 4) \mapsto +. (3.0, 4.0) \mapsto 7.0.
\]

This strategy requires that the floating point operations detect the presence of integer arguments, and that it convert any such arguments to floating point before carrying out the operation. We will have more to say about this inefficiency in Section 26.4 below.

### 26.2.2 Function Subtyping

Suppose that \( \text{int} \lltype \text{float} \). What subtyping relationships, if any, should hold among the following four types?

1. \( \text{int} \to \text{int} \)
2. \( \text{int} \to \text{float} \)
3. \( \text{float} \to \text{int} \)
4. \( \text{float} \to \text{float} \)

To determine the answer, keep in mind the subsumption principle, which says that a value of the subtype should be usable in a supertype context.

Suppose \( f : \text{int} \to \text{int} \). If we apply \( f \) to \( x : \text{int} \), the result has type \( \text{int} \), and hence, by the arithmetic subtyping axiom, has type \( \text{float} \). This suggests that

\[
\text{int} \to \text{int} \lltype \text{int} \to \text{float}
\]

is a valid subtype relationship. By similar reasoning, we may derive that

\[
\text{float} \to \text{int} \lltype \text{float} \to \text{float}
\]

\(^3\)We may handle the limitations of precision by allowing for a cast operation to fail in the case of overflow. We will ignore overflow here, for the sake of simplicity.
is also valid.

Now suppose that \( f : \text{float} \rightarrow \text{int} \). If \( x : \text{int} \), then \( x : \text{float} \) by subsumption, and hence we may apply \( f \) to \( x \) to obtain a result of type \( \text{int} \). This suggests that

\[
\text{float} \rightarrow \text{int} <: \text{int} \rightarrow \text{int}
\]

is a valid subtype relationship. Since \( \text{int} \rightarrow \text{int} <: \text{int} \rightarrow \text{float} \), it follows that

\[
\text{float} \rightarrow \text{int} <: \text{int} \rightarrow \text{float}
\]

is also valid.

Subtyping rules that specify how a type constructor interacts with subtyping are called \textit{variance} principles. If a type constructor \textit{preserves} subtyping in a given argument position, it is said to be \textit{covariant} in that position. If, instead, it \textit{inverts} subtyping in a given position it is said to be \textit{contravariant} in that position. The discussion above suggests that the function space constructor is covariant in the range position and contravariant in the domain position. This is expressed by the following rule:

\[
\frac{\tau_1 <: \sigma_1 \quad \sigma_2 <: \tau_2}{\sigma_1 \rightarrow \sigma_2 <: \tau_1 \rightarrow \tau_2}
\]

Note well the inversion of subtyping in the domain, where the function constructor is contravariant, and the preservation of subtyping in the range, where the function constructor is covariant.

To ensure safety in the explicit case, we define the dynamic semantics of a cast operation by the following rule:

\[
(\tau_1 \rightarrow \tau_2) \ v \mapsto \text{fn } x : \tau_1 \in (\tau_2) \ v ((\sigma_1) \ x) \ \text{end}
\]

Here \( v \) has type \( \sigma_1 \rightarrow \sigma_2 \), \( \tau_1 <: \sigma_1 \), and \( \sigma_2 <: \tau_2 \). The argument is cast to the domain type of the function prior to the call, and its result is cast to the intended type of the application.

To ensure safety in the implicit case, we must ensure that the primitive operation of function application behaves correctly on a function of a subtype of the “expected” type. This amounts to ensuring that a function can be called with an argument of, and yields a result of, a subtype of the intended type. One way is to adopt a semantics of procedure call that is independent of the types of the arguments and results. Another is to introduce explicit run-time checks similar to those suggested for floating point arithmetic to ensure that calling conventions for different types can be met.
26.2.3 Product and Record Subtyping

In Chapter 19 we considered an extension of MinML with product types. In this section we’ll consider equipping this extension with subtyping. We will work with $n$-ary products of the form $\tau_1 \times \ldots \times \tau_n$ and with $n$-ary records of the form $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$. The tuple types have as elements $n$-tuples of the form $<e_1, \ldots, e_n>$ whose $i$th component is accessed by projection, e.g. $e_i$. Similarly, record types have as elements records of the form $\{l_1 : e_1, \ldots, l_n : e_n\}$ whose $i$th component is accessed by field selection, e.g. $l_i$.

Using the subsumption principle as a guide, it is natural to consider a tuple type to be a subtype of any of its prefixes:

$$m > n \quad \Rightarrow \quad \tau_1 \times \ldots \times \tau_m \lessdot \tau_1 \times \ldots \times \tau_n$$

Given a value of type $\tau_1 \times \ldots \times \tau_n$, we can access its $i$th component, for any $1 \leq i \leq n$. If $m > n$, then we can equally well access the $i$th component of an $m$-tuple of type $\tau_1 \times \ldots \times \tau_m$, obtaining the same result. This is called *width subtyping* for tuples.

For records it is natural to consider a record type to be a subtype of any record type with any subset of the fields of the subtype. This may be written as follows:

$$m > n \quad \Rightarrow \quad \{l_1 : \tau_1, \ldots, l_m : \tau_m\} \lessdot \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$$

Bear in mind that the ordering of fields in a record type is immaterial, so this rule allows us to neglect any subset of the fields when passing to a supertype. This is called *width subtyping* for records. The justification for width subtyping is that record components are accessed by label, rather than position, and hence the projection from a supertype value will apply equally well to the subtype.

What variance principles apply to tuples and records? Applying the principle of subsumption, it is easy to see that tuples and records may be regarded as covariant in all their components. That is,

$$\forall 1 \leq i \leq n \quad \sigma_i \lessdot \tau_i \quad \Rightarrow \quad \sigma_1 \times \ldots \times \sigma_n \lessdot \tau_1 \times \ldots \times \tau_n$$

and

$$\forall 1 \leq i \leq n \quad \sigma_i \lessdot \tau_i \quad \Rightarrow \quad \{l_1 : \sigma_1, \ldots, l_n : \sigma_n\} \lessdot \{l_1 : \tau_1, \ldots, l_n : \tau_n\}.$$  

These are called *depth subtyping* rules for tuples and records, respectively.
26.2 Varieties of Subtyping

To ensure safety for explicit subsumption we must define the meaning of casting from a sub- to a super-type. The two forms of casting corresponding to width and depth subtyping may be consolidated into one, as follows:

$$m \geq n$$

$$(\tau_1 \cdots \tau_n) < v_1, \ldots, v_m > \mapsto <(\tau_1) v_1, \ldots, (\tau_n) v_n >.$$

An analogous rule defines the semantics of casting for record types.

To ensure safety for implicit subsumption we must ensure that projection is well-defined on a subtype value. In the case of tuples this means that the operation of accessing the $i$th component from a tuple must be insensitive to the size of the tuple, beyond the basic requirement that it have size at least $i$. This can be expressed schematically as follows:

$$<v_1, \ldots, v_i, \ldots> \cdot i \mapsto v_i.$$

The ellision indicates that fields beyond the $i$th are not relevant to the operation. Similarly, for records we postulate that selection of the $l$th field is insensitive to the presence of any other fields:

$$\{l; v, \ldots\} \cdot l \mapsto v.$$

The ellision expresses the independence of field selection from any “extra” fields.

26.2.4 Reference Subtyping

Finally, let us consider the reference types of Chapter 14. What should be the variance rule for reference types? Suppose that $r$ has type $\sigma \text{ref}$. We can do one of two things with $r$:

1. Retrieve its contents as a value of type $\sigma$.
2. Replace its contents with a value of type $\sigma$.

If $\sigma <: \tau$, then retrieving the contents of $r$ yields a value of type $\tau$, by subsumption. This suggests that references are covariant:

$$\sigma <: \tau \quad \sigma \text{ref} <: \tau \text{ref}.$$
On the other hand, if $\tau <: \sigma$, then we may store a value of type $\tau$ into $r$. This suggests that references are contravariant:

$$
\tau <: \sigma \\
\sigma \text{ref} <: \tau \text{ref}.
$$

Given that we may perform either operation on a reference cell, we must insist that reference types are invariant:

$$
\sigma <: \tau \quad \tau <: \sigma \\
\sigma \text{ref} <: \tau \text{ref}.
$$

The premise of the rule is often strengthened to the requirement that $\sigma$ and $\tau$ be equal:

$$
\sigma = \tau \\
\sigma \text{ref} <: \tau \text{ref}
$$

since there are seldom situations where distinct types are mutual subtypes.

A similar analysis may be applied to any mutable data structure. For example, immutable sequences may be safely taken to be covariant, but mutable sequences (arrays) must be taken to be invariant, lest safety be compromised.

### 26.3 Type Checking With Subtyping

Type checking for MinML$<$:, in either variant, clearly requires an algorithm for deciding subtyping: given $\sigma$ and $\tau$, determine whether or not $\sigma <: \tau$. The difficulty of deciding type checking is dependent on the specific rules under consideration. In this section we will discuss type checking for MinML$<$:, under the assumption that we can check the subtype relation.

Consider first the explicit variant of MinML$<$:. Since the typing rules are syntax directed, we can proceed as for MinML, with one additional case to consider. To check whether $(\sigma) e$ has type $\tau$, we must check two things:

1. Whether $e$ has type $\sigma$.
2. Whether $\sigma <: \tau$.

The former is handled by a recursive call to the type checker, the latter by a call to the subtype checker, which we assume given.

This discussion glosses over an important point. Even in pure MinML it is not possible to determine directly whether or not $\Gamma \vdash e : \tau$. For suppose that $e$ is an application $e_1 (e_2)$. To check whether $\Gamma \vdash e : \tau$, we must find
the domain type of the function, \( e_1 \), against which we must check the type of the argument, \( e_2 \). To do this we define a *type synthesis* function that determines the unique (if it exists) type \( \tau \) of an expression \( e \) in a context \( \Gamma \), written \( \Gamma \vdash e \Rightarrow \tau \). To check whether \( e \) has type \( \tau \), we synthesize the unique type for \( e \) and check that it is \( \tau \).

This methodology applies directly to \( \text{MinML}^< \) by using the following rule to synthesize a type for a cast:

\[
\Gamma \vdash e \Rightarrow \sigma \quad \sigma : \tau \\
\Gamma \vdash (\tau) e \Rightarrow \tau
\]

Extending this method to \( \text{MinML}^i \) is a bit harder, because expressions no longer have unique types! The rule of subsumption allows us to weaken the type of an expression at will, yielding many different types for the same expression. A standard approach is define a type synthesis function that determines the principal type, rather than the unique type, of an expression in a given context. The principal type of an expression \( e \) in context \( \Gamma \) is the *least* type (in the subtyping pre-order) for \( e \) in \( \Gamma \). Not every subtype system admits principal types. But we usually strive to ensure that this is the case whenever possible in order to employ this simple type checking method.

The rules synthesizing principal types for expressions of \( \text{MinML}^i \) are as follows:

\[
\begin{align*}
&\Gamma(x) = \tau \\
&\Gamma \vdash x \Rightarrow \tau \\
&\Gamma \vdash n \Rightarrow \text{int} \\
&\Gamma \vdash \text{true} \Rightarrow \text{bool} \\
&\Gamma \vdash \text{false} \Rightarrow \text{bool} \\
&\Gamma \vdash e_1 \Rightarrow \sigma_1 \quad \sigma_1 : \tau_1 \\
&\cdots \\
&\Gamma \vdash e_n \Rightarrow \sigma_n \quad \sigma_n : \tau_n \\
&\Gamma \vdash o(e_1, \ldots, e_n) \Rightarrow \tau
\end{align*}
\]

where \( o \) is an \( n \)-ary primitive operation with arguments of type \( \tau_1, \ldots, \tau_n \), and result type \( \tau \). We use subsumption to ensure that the argument types are subtypes of the required types.

\[
\begin{align*}
&\Gamma \vdash e \Rightarrow \sigma \quad \sigma : \text{bool} \\
&\Gamma \vdash e_1 \Rightarrow \tau_1 \\
&\Gamma \vdash e_2 \Rightarrow \tau_2 \\
&\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \text{ fi} \Rightarrow \tau
\end{align*}
\]

We use subsumption to ensure that the type of the test is a subtype of \( \text{bool} \). Moreover, we rely on explicit specification of the type of the two clauses of the conditional.\(^4\)

\[
\begin{align*}
&\Gamma \vdash f : \tau_1 \rightarrow \tau_2 | x : \tau_1 | e \Rightarrow \tau_2 \\
&\Gamma \vdash \text{fun } f(x : \tau_1) : \tau_2 \text{ is } e \text{ end} \Rightarrow \tau_1 \rightarrow \tau_2
\end{align*}
\]

\(^4\)This may be avoided by requiring that the subtype relation have least upper bounds “whenever necessary”; we will not pursue this topic here.
We use subsumption to check that the argument type is a subtype of the domain type of the function.

**Theorem 26.3**

1. If $\Gamma \vdash e \Rightarrow \sigma$, then $\Gamma \vdash e : \sigma$.

2. If $\Gamma \vdash e : \tau$, then there exists $\sigma$ such that $\Gamma \vdash e \Rightarrow \sigma$ and $\sigma < : \tau$.

**Proof:**

1. By a straightforward induction on the definition of the type synthesis relation.

2. By induction on the typing relation.

---

### 26.4 Implementation of Subtyping

#### 26.4.1 Coercions

The dynamic semantics of subtyping sketched above suffices to ensure type safety, but is in most cases rather impractical. Specifically,

- Arithmetic subtyping relies on run-time type recognition and conversion.
- Tuple projection depends on the insensitivity of projection to the existence of components after the point of projection.
- Record field selection depends on being able to identify the $l$th field in a record with numerous fields.
- Function subtyping may require run-time checks and conversions to match up calling conventions.

These costs are significant. Fortunately they can be avoided by taking a slightly different approach to the implementation of subtyping. Consider, for example, arithmetic subtyping. In order for a mixed-mode expression such as $+. (3, 4)$ to be well-formed, we must use subsumption to weaken
26.4 Implementation of Subtyping

the types of 3 and 4 from int to float. This means that type conversions are required exactly insofar as subsumption is used during type checking — a use of subsumption corresponds to a type conversion.

Since the subsumption rule is part of the static semantics, we can insert the appropriate conversions during type checking, and omit entirely the need to check for mixed-mode expressions during execution. This is called a coercion interpretation of subsumption. It is expressed formally by augmenting each subtype relation \( \sigma <: \tau \) with a function value \( v \) of type \( \sigma \rightarrow \tau \) (in pure MinML) that coerces values of type \( \sigma \) to values of type \( \tau \). The augmented subtype relation is written \( \sigma <: \tau \rightarrow v \).

Here are the rules for arithmetic subtyping augmented with coercions:

\[
\begin{array}{c}
\tau <: \tau \rightarrow \text{id}_\tau \\
\text{int} <: \text{float} \rightarrow \text{to_float}
\end{array}
\]

\[
\begin{array}{c}
\rho <: \sigma \rightarrow v \quad \sigma <: \tau \rightarrow v' \\
\rho <: \tau \rightarrow v; v'
\end{array}
\]

These rules make use of the following auxiliary functions:

1. Primitive conversion: to_float.
2. Identity: \( \text{id}_\tau = \text{fn} \, x: \tau \rightarrow x \).
3. Composition: \( v; v' = \text{fn} \, x: \tau \rightarrow v'(v(x)) \).
4. Functions: \( v_1 \rightarrow v_2 = \text{fn} \, f: \sigma_1 \rightarrow \sigma_2 \rightarrow \text{fn} \, x: \tau_1 \rightarrow v_2(f(v_1(x))) \).

The coercion interpretation is type correct. Moreover, there is at most one coercion between any two types:

**Theorem 26.4**

1. If \( \sigma <: \tau \rightarrow v \), then \( \vdash v : \sigma \rightarrow \tau \).
2. If \( \sigma <: \tau \rightarrow v_1 \) and \( \sigma <: \tau \rightarrow v_2 \), then \( \vdash v_1 \equiv v_2 : \sigma \rightarrow \tau \).

**Proof:**

1. By a simple induction on the rules defining the augmented subtyping relation.
2. Follows from these equations:

(a) Composition is associative with \( \text{id} \) as left- and right-unit element.
\[(b) \text{id} \to \text{id} \cong \text{id}.\]
\[(c) \ (v_1 \to v_2); (v'_1 \to v'_2) \cong (v'_1; v_1) \to (v_2; v'_2).\]

The type checking relation is augmented with a translation from \(\text{MinML}^\prec\) to pure \(\text{MinML}\) that eliminates uses of subsumption by introducing coercions:

\[
\Gamma \vdash e : \sigma \leadsto e' \quad \sigma \leadsto \tau \leadsto v \quad \Gamma \vdash e : \tau \leadsto v(e')
\]

The remaining rules simply commute with the translation. For example, the rule for function application becomes

\[
\Gamma \vdash e_1 : \tau_2 \to \tau \leadsto e'_1 \quad \Gamma \vdash e_2 : \tau_2 \leadsto e'_2 \quad \Gamma \vdash e_1(e_2) : \tau \leadsto e'_1(e'_2)
\]

**Theorem 26.5**

1. If \(\Gamma \vdash e : \tau \leadsto e'\), then \(\Gamma \vdash e' : \tau\) in pure \(\text{MinML}\).

2. If \(\Gamma \vdash e : \tau \leadsto e_1\) and \(\Gamma \vdash e : \tau \leadsto e_2\), then \(\Gamma \vdash e_1 \cong e_2 : \tau\) in pure \(\text{MinML}\).

3. If \(e : \text{int} \leadsto e'\) is a complete program, then \(e \Downarrow n\) iff \(e' \Downarrow n\).

The coercion interpretation also applies to record subtyping. Here the problem is how to implement field selection efficiently in the presence of subsumption. Observe that in the absence of subtyping the type of a record value reveals the exact set of fields of a record (and their types). We can therefore implement selection efficiently by ordering the fields in some canonical manner (say, alphabetically), and compiling field selection as a projection from an offset determined statically by the field’s label.

In the presence of record subtyping this simple technique breaks down, because the type no longer reveals the fields of a record, not their types. For example, every expression of record type has the record type \(\text{fg}\) with no fields whatsoever! This makes it difficult to predict statically the position of the field labelled \(l\) in a record. However, we may restore this important property by using coercions. Whenever the type of a record is weakened using subsumption, insert a function that creates a new record that exactly matches the supertype. Then use the efficient record field selection method just described.

WORKING DRAFT

January 14, 2003
Here, then, are the augmented rules for width and depth subtyping for records:

\[
\begin{align*}
m > n & \quad \{l_1:\tau_1, \ldots, l_m:\tau_m\} < \{l_1:\tau_1, \ldots, l_n:\tau_n\} \leadsto \text{drop}_{m,n,l,\tau} \\
\sigma_1 <: \tau_1 \leadsto v_1 \ldots \sigma_n <: \tau_n \leadsto v_n & \quad \{l_1:\sigma_1, \ldots, l_n:\sigma_n\} <: \{l_1:\tau_1, \ldots, l_n:\tau_n\} \leadsto \text{copy}_{n,l,\sigma,v}
\end{align*}
\]

These rules make use of the following coercion functions:

\[
\begin{align*}
\text{drop}_{m,n,l,\sigma} &= \text{fn } x.\{l_1:%n_1, \ldots, l_m:%n_m\} \text{ in } \{l_1:%x_1, \ldots, l_n:%x_n\} \text{ end} \\
\text{copy}_{n,l,\sigma,v} &= \text{fn } x.\{l_1:%n_1, \ldots, l_n:%n_n\} \text{ in } \{l_1:%v_1(x%l_1), \ldots, l_n:%v_n(x%l_n)\} \text{ end}
\end{align*}
\]

In essence this approach represents a trade-off between the cost of subsumption and the cost of field selection. By creating a new record whenever subsumption is used, we make field selection cheap. On the other hand, we can make subsumption free, provided that we are willing to pay the cost of a search whenever a field is selected from a record.

But what if record fields are mutable? This approach to coercion is out of the question, because of aliasing. Suppose that a mutable record value \(v\) is bound to two variables, \(x\) and \(y\). If coercion is applied to the binding of \(x\), creating a new record, then future changes to \(y\) will not affect the new record, nor vice versa. In other words, uses of coercion changes the semantics of a program, which is unreasonable.

One widely-used approach is to increase slightly the cost of field selection (by a constant factor) by separating the “view” of a record from its “contents”. The view determines the fields and their types that are present for each use of a record, whereas the contents is shared among all uses. In essence we represent the record type \(\{l_1:\tau_1, \ldots, l_n:\tau_n\}\) by the product type

\[
\{l_1:\text{int}, \ldots, l_n:\text{int}\}^*(\tau \text{ array}).
\]

The field selection \(l.e\) becomes a two-stage process:

\[
\text{snd}(e)[\text{fst}(e).l]
\]

Finally, coercions copy the view, without modifying the contents. If \(\sigma = \{l_1:%n_1, \ldots, l_n:%n_n\}\) and \(\tau = \{l_1:%n_1, \ldots, l_n:%n_n\}\), then

\[
\text{drop}_{m,n,l,\sigma} = \text{fn } x.\text{ in } (\text{drop}_{m,n,l,\tau}(\text{fst}(x)), \text{snd}(x)) \text{ end.}
\]
Chapter 27

Inheritance and Subtyping in Java

In this note we discuss the closely-related, but conceptually distinct, notions of inheritance, or subclassing, and subtyping as exemplified in the Java language. Inheritance is a mechanism for supporting code re-use through incremental extension and modification. Subtyping is a mechanism for expressing behavioral relationships between types that allow values of a subtype to be provided whenever a value of a supertype is required.

In Java inheritance relationships give rise to subtype relationships, but not every subtype relationship arises via inheritance. Moreover, there are languages (including some extensions of Java) for which subclasses do not give rise to subtypes, and there are languages with no classes at all, but with a rich notion of subtyping. For these reasons it is best to keep a clear distinction between subclassing and subtyping.

27.1 Inheritance Mechanisms in Java

27.1.1 Classes and Instances

The fundamental unit of inheritance in Java is the class. A class consists of a collection of fields and a collection of methods. Fields are assignable variables; methods are procedures acting on these variables. Fields and methods can be either static (per-class) or dynamic (per-instance).¹ Static fields are per-class data. Static methods are just ordinary functions acting on static fields.

¹Fields and methods are assumed dynamic unless explicitly declared to be static.
Classes give rise to *instances*, or *objects*, that consist of the dynamic methods of the class together with fresh copies (or instances) of its dynamic fields. Instances of classes are created by a *constructor*, whose role is to allocate and initialize fresh copies of the dynamic fields (which are also known as *instance variables*). Constructors have the same name as their class, and are invoked by writing `new C(e1, . . . , en)`, where $C$ is a class and $e1, . . . , en$ are arguments to the constructor.\(^2\) Static methods have access only to the static fields (and methods) of its class; dynamic methods have access to both the static and dynamic fields and methods of the class.

The components of a class have a designated *visibility* attribute, either *public*, *private*, or *protected*. The public components are those that are accessible by all clients of the class. Public static components are accessible to any client with access to the class. Public dynamic components are visible to any client of any instance of the class. Protected components are “semi-private; we’ll have more to say about protected components later.

The components of a class also have a *finality* attribute. Final fields are not assignable — they are read-only attributes of the class or instance. Actually, final dynamic fields can be assigned exactly once, by a constructor of the class, to initialize their values. Final methods are of interest in connection with inheritance, to which we’ll return below.

The components of a class have *types*. The type of a field is the type of its binding as a (possibly assignable) variable. The type of a method specifies the types of its arguments (if any) and the type of its results. The type of a constructor specifies the types of its arguments (if any); its “result type” is the instance type of the class itself, and may not be specified explicitly. (We will say more about the type structure of Java below.)

The public static fields and methods of a class $C$ are accessed using “dot notation”. If $f$ is a static field of $C$, a client may refer to it by writing $C.f$. Similarly, if $m$ is a static method of $C$, a client may invoke it by writing $C.m(e1, . . . , en)$, where $e1, . . . , en$ are the argument expressions of the method. The expected type checking rules govern access to fields and invocations of methods.

The public dynamic fields and methods of an instance $c$ of a class $C$ are similarly accessed using “dot notation”, *albeit* from the instance, rather than the class. That is, if $f$ is a public dynamic field of $C$, then $c.f$ refers to the $f$ field of the instance $c$. Since distinct instances have distinct fields, there is no essential connection between $c.f$ and $c’.f$ when $c$ and $c’$ are

\(^2\)Classes can have multiple constructors that are distinguished by overloading. We will not discuss overloading here.
distinct instances of class \( C \). If \( m \) is a public dynamic method of \( C \), then \( c.m(e_1, \ldots, e_n) \) invokes the method \( m \) of the instance \( c \) with the specified arguments. This is sometimes called sending a message \( m \) to instance \( c \) with arguments \( e_1, \ldots, e_n \).

Within a dynamic method one may refer to the dynamic fields and methods of the class via the pseudo-variable \( \text{this} \), which is bound to the instance itself. The methods of an instance may call one another (or themselves) by sending a message to \( \text{this} \). Although Java defines conventions whereby explicit reference to \( \text{this} \) may be omitted, it is useful to eschew these conveniences and always use \( \text{this} \) to refer to the components of an instance from within code for that instance. We may think of \( \text{this} \) as an implicit argument to all methods that allows the method to access to object itself.

### 27.1.2 Subclasses

A class may be defined by inheriting the visible fields and methods of another class. The new class is said to be a subclass of the old class, the superclass. Consequently, inheritance is sometimes known as subclassing. Java supports single inheritance — every class has at most one superclass. That is, one can only inherit from a single class; one cannot combine two classes by inheritance to form a third. In Java the subclass is said to extend the superclass.

There are two forms of inheritance available in Java:

1. **Enrichment.** The subclass enriches the superclass by providing additional fields and methods not present in the superclass.

2. **Overriding.** The subclass may re-define a method in the superclass by giving it a new implementation in the subclass.

Enrichment is a relatively innocuous aspect of inheritance. The true power of inheritance lies in the ability to override methods.

Overriding, which is also known as method specialization, is used to “specialize” the implementation of a superclass method to suit the needs of the subclass. This is particularly important when the other methods of the class invoke the overridden method by sending a message to \( \text{this} \). If a method \( m \) is overridden in a subclass \( D \) of a class \( C \), then all methods of \( D \) that invoke \( m \) via \( \text{this} \) will refer to the “new” version of \( m \) defined by the override. The “old” version can still be accessed explicitly from the subclass by referring to \( \text{super}.m \). The keyword \( \text{super} \) is a pseudo-variable that may be used to refer to the overridden methods.
Inheritance can be controlled using visibility constraints. A sub-class \( D \) of a class \( C \) automatically inherits the private fields and methods of \( C \) without the possibility of overriding, or otherwise accessing, them. The public fields and methods of the superclass are accessible to the subclass without restriction, and retain their public attribute in the subclass, unless overridden. A protected component is “semi-private” — accessible to the subclass, but not otherwise publicly visible.\(^3\)

Inheritance can also be limited using finality constraints. If a method is declared final, it may not be overridden in any subclass — it must be inherited as-is, without further modification. However, if a final method invokes, via this, a non-final method, then the behavior of the final method can still be changed by the sub-class by overriding the non-final method. By declaring an entire class to be final, no class can inherit from it. This serves to ensure that any instance of this class invokes the code from this class, and not from any subclass of it.

Instantiation of a subclass of a class proceeds in three phases:

1. The instance variables of the subclass, which include those of the superclass, are allocated.
2. The constructor of the superclass is invoked to initialize the superclass’s instance variables.
3. The constructor of the subclass is invoked to initialize the subclass’s instance variables.

The superclass constructor can be explicitly invoked by a subclass constructor by writing \texttt{super(e\_1, \ldots, e\_n)}\(^\text{, but only as the very first statement of the subclass’s constructor. This ensures proper initialization order, and avoids certain anomalies and insecurities that arise if this restriction is relaxed.}\)

\textbf{27.1.3 Abstract Classes and Interfaces}

An abstract class is a class in which one or more methods are declared, but left unimplemented. Abstract methods may be invoked by the other methods of an abstract class by sending a message to this, but since their implementation is not provided, abstract classes do not themselves have instances. Instead the obligation is imposed on a subclass of the abstract

\(^3\)Actually, Java assigns protected components “package scope”, but since we are not discussing packages here, we will ignore this issue.
class to provide implementations of the abstract methods to obtain a concrete class, which does have instances. Abstract classes are useful for setting up “code templates” that are instantiated by inheritance. The abstract class becomes the locus of code sharing for all concretions of that class, which inherit the shared code and provide the missing non-shared code.

Taking this idea to the extreme, an interface is a “fully abstract” class, which is to say that

- All its fields are public static final (i.e., they are constants).
- All its methods are abstract public; they must be implemented by a subclass.

Since interfaces are a special form of abstract class, they have no instances.

The utility of interfaces stems from their role in implements declarations. As we mentioned above, a class may be declared to extend a single class to inherit from it. A class may also be declared to implement one or more interfaces, meaning that the class provides the public methods of the interface, with their specified types. Since interfaces are special kinds of classes, Java is sometimes said to provide multiple inheritance of interfaces, but only single inheritance of implementation. For similar reasons an interface may be declared to extend multiple interfaces, provided that the result types of their common methods coincide.

The purpose of declaring an interface for a class is to support writing generic code that works with any instance providing the methods specified in the interface, without requiring that instance to arise from any particular position in the inheritance hierarchy. For example, we may have two unrelated classes in the class hierarchy providing a method m. If both classes are declared to implement an interface that mentions m, then code programmed against this interface will work for an instance of either class.

The literature on Java emphasizes that interfaces are descriptive of behavior (to the extend that types alone allow), whereas classes are prescriptive of implementation. While this is surely a noble purpose, it is curious that interfaces are classes in Java, rather than types. In particular interfaces are unable to specify the public fields of an instance by simply stating their types, which would be natural were interfaces a form of type. Instead fields in interfaces are forced to be constants (public, static, final), precluding their use for describing the public instance variables of an object.

---

4Classes that do not specify a superclass implicitly extend the class Object of all objects.
27.2 Subtyping in Java

The Java type system consists of the following types:

1. **Base types**, including `int`, `float`, `void`, and `boolean`.
2. **Class types** $C$, which classify the instances of a class $C$.
3. **Array types** of the form $\tau[\ ]$, where $\tau$ is a type, representing mutable arrays of values of type $\tau$.

The basic types behave essentially as one would expect, based on previous experience with languages such as C and C++. Unlike C or C++ Java has true array types, with operations for creating and initializing an array and for accessing and assigning elements of an array. All array operations are safe in the sense that any attempt to exceed the bounds of the array results in a checked error at run-time.

Every class, whether abstract or concrete, including interfaces, has associated with it the type of its instances, called (oddly enough) the *instance type* of the class. Java blurs the distinction between the class as a program structure and the instance type determined by the class — class names serve not only to identify the class but also the instance type of that class. It may seem odd that abstract classes, and interfaces, all define instance types, even though they don’t have instances. However, as will become clear below, even abstract classes have instances, indirectly through their concrete subclasses. Similarly, interfaces may be thought of as possessing instances, namely the instances of concrete classes that implement that interface.

27.2.1 Subtyping

To define the Java subtype relation we need two auxiliary relations. The *subclass* relation, $C \triangleleft C'$, is the reflexive and transitive closure of the *extends* relation among classes, which holds precisely when one class is declared to extend another. In other words, $C \triangleleft C'$ iff $C$ either coincides with $C'$, inherits directly from $C'$, or inherits from a subclass of $C'$. Since interfaces are classes, the subclass relation also applies to interfaces, but note that multiple inheritance of interfaces means that an interface can be a subinterface (subclass) of more than one interface. The *implementation* relation, $C \triangleleft I$, is defined to hold exactly when a class $C$ is declared to implement an interface that inherits from $I$. 
27.2 Subtyping in Java

The Java subtype relation is inductively defined by the following rules. Subtyping is reflexive and transitive:

\[ \tau <: \tau \]  
\[ \tau <: \tau' \quad \tau' <: \tau'' \quad \frac{}{\tau <: \tau''} \]  

(27.1)  
(27.2)

Arrays are covariant type constructors, in the sense of this rule:

\[ \tau <: \tau' \quad \frac{}{\tau[\cdot] <: \tau'[\cdot]} \]  

(27.3)

Inheritance implies subtyping:

\[ C <: C'' \quad \frac{}{C <: C'} \]  

(27.4)

Implementation implies subtyping:

\[ C \triangleq I \quad \frac{}{C <: I} \]  

(27.5)

Every class is a subclass of the distinguished "root" class `Object`:

\[ \tau <: \text{Object} \]  

(27.6)

The array subtyping rule is a structural subtyping principle — one need not explicitly declare subtyping relationships between array types for them to hold. On the other hand, the inheritance and implementation rules of subtyping are examples of nominal subtyping — they hold when they are declared to hold at the point of definition (or are implied by further subtyping relations).

27.2.2 Subsumption

The subsumption principle tells us that if \( e \) is an expression of type \( \tau \) and \( \tau <: \tau' \), then \( e \) is also an expression of type \( \tau' \). In particular, if a method is declared with a parameter of type \( \tau \), then it makes sense to provide an argument of any type \( \tau' \) such that \( \tau' <: \tau \). Similarly, if a constructor takes a parameter of a type, then it is legitimate to provide an argument of a subtype of that type. Finally, if a method is declared to return a value of type \( \tau \), then it is legitimate to return a value of any subtype of \( \tau \).
This brings up an awkward issue in the Java type system. What should be the type of a conditional expression \( e ? e_1 : e_2 \)? Clearly \( e \) should have type boolean, and \( e_1 \) and \( e_2 \) should have the same type, since we cannot in general predict the outcome of the condition \( e \). In the presence of subtyping, this amounts to the requirement that the types of \( e_1 \) and \( e_2 \) have an upper bound in the subtype ordering. To avoid assigning an excessively weak type, and to ensure that there is a unique choice of type for the conditional, it would make sense to assign the conditional the least upper bound of the types of \( e_1 \) and \( e_2 \). Unfortunately, two types need not have a least upper bound! For example, if an interface \( I \) extends incomparable interfaces \( K \) and \( L \), and \( J \) extends both \( K \) and \( L \), then \( I \) and \( J \) do not have a least upper bound — both \( K \) and \( L \) are upper bounds of both, but neither is smaller than the other. To deal with this Java imposes the rather ad hoc requirement that either the type of \( e_1 \) be a subtype of the type of \( e_2 \), or vice versa, to avoid the difficulty.

A more serious difficulty with the Java type system is that the array subtyping rule, which states that the array type constructor is covariant in the type of the array elements, violates the subsumption principle. To understand why, recall that we can do one of two things with an array: retrieve an element, or assign to an element. If \( \tau <: \tau' \) and \( A \) is an array of type \( \tau[\ ] \), then retrieving an element of \( A \) yields a value of type \( \tau \), which is by hypothesis an element of type \( \tau' \). So we are OK with respect to retrieval. Now consider array assignment. Suppose once again that \( \tau <: \tau' \) and that \( A \) is an array of type \( \tau[\ ] \). Then \( A \) is also an array of type \( \tau'[\ ] \), according to the Java rule for array subtyping. This means we can assign a value \( x \) of type \( \tau' \) to an element of \( A \). But this violates the assumption that \( A \) is an array of type \( \tau[\ ] \) — one of its elements is of type \( \tau' \).

With no further provisions the language would not be type safe. It is a simple matter to contrive an example involving arrays that incurs a runtime type error (“gets stuck”). Java avoids this by a simple, but expensive, device — every array assignment incurs a “run-time type check” that ensures that the assignment does not create an unsafe situation. In the next subsection we explain how this is achieved.

### 27.2.3 Dynamic Dispatch

According to Java typing rules, if \( C \) is a sub-class of \( D \), then \( C \) is a sub-type of \( D \). Since the instances of a class \( C \) have type \( C \), they also, by subsump- tion, have type \( D \), as do the instances of class \( D \) itself. In other words, if the static type of an instance is \( D \), it might be an instance of class \( C \) or an
instance of class $D$. In this sense the static type of an instance is at best an approximation of its dynamic type, the class of which it is an instance.

The distinction between the static and the dynamic type of an object is fundamental to object-oriented programming. In particular method specialization is based on the dynamic type of an object, not its static type. Specifically, if $C$ is a sub-class of $D$ that overrides a method $m$, then invoking the method $m$ of a $C$ instance $o$ will always refer to the overriding code in $C$, even if the static type of $o$ is $D$. That is, method dispatch is based on the dynamic type of the instance, not on its static type. For this reason method specialization is sometimes called \textit{dynamic dispatch}, or, less perspicuously, \textit{late binding}.

How is this achieved? Essentially, every object is tagged with the class that created it, and this tag is used to determine which method to invoke when a message is sent to that object. The constructors of a class $C$ “label” the objects they create with $C$. The method dispatch mechanism consults this label when determining which method to invoke.\footnote{In practice the label is a pointer to the vector of methods of the class, and the method is accessed by indexing into this vector. But we can just as easily imagine this to be achieved by a case analysis on the class name to determine the appropriate method vector.}

The same mechanism is used to ensure that array assignments do not lead to type insecurities. Suppose that the static type of $A$ is $C[]$, and that the static type of instance $o$ is $C$. By covariance of array types the dynamic type of $A$ might be $D[]$ for some sub-class $D$ of $C$. But unless the dynamic type of $o$ is also $D$, the assignment of $o$ to an element of $A$ should be prohibited. This is ensured by an explicit run-time check. In Java \textit{every single array assignment incurs a run-time check} whenever the array contains objects.\footnote{Arrays of integers and floats do not incur this overhead, because numbers are not objects.}

### 27.2.4 Casting

A \textit{container class} is one whose instances “contain” instances of another class. For example, a class of lists or trees or sets would be a container class in this sense. Since the operations on containers are largely (or entirely) independent of the type of their elements, it makes sense to define containers generally, rather than defining one for each element type. In Java this is achieved by exploiting subsumption. Since every object has type $\text{Object}$, a general container is essentially a container whose elements are of type $\text{Object}$. This allows the container operations to be defined once for all

---

\textsc{Working Draft} \hspace{1cm} \textsc{January 14, 2003}
element types. However, when retrieving an element from a container its static type is `Object`; we lost track of its dynamic type during type checking. If we wish to use such an object in any meaningful way, we must recover its dynamic type so that message sends are not rejected at compile time.

Java supports a safe form of casting, or change of type. A cast is written `(\tau) e`. The expression `e` is called the subject of the cast, and the type `\tau` is the target type of the cast. The type of the cast is `\tau`, provided that the cast makes sense, and its value is that of `e`. In general we cannot determine whether the cast makes sense until execution time, when the dynamic type of the expression is available for comparison with the target type. For example, every instance in Java has type `Object`, but its true type will usually be some class further down the type hierarchy. Therefore a cast applied to an expression of type `Object` cannot be validated until execution time.

Since the static type is an attenuated version of the dynamic type of an object, we can classify casts into three varieties:

1. **Up casts**, in which the static type of the expression is a subtype of the target type of the cast. The type checker accepts the cast, and no run-time check is required.

2. **Down casts**, in which the static type of the expression is a supertype of the target type. The true type may or may not be a subtype of the target, so a run-time check is required.

3. **Stupid casts**, in which the static type of the expression rules out the possibility of its dynamic type matching the target of the cast. The cast is rejected.

Similar checks are performed to ensure that array assignments are safe.

Note that it is up to the programmer to maintain a sufficiently strong invariant to ensure that down casts do not fail. For example, if a container is intended to contain objects of a class `C`, then retrieved elements of that class will typically be down cast to a sub-class of `C`. It is entirely up to the programmer to ensure that these casts do not fail at execution time. That is, the programmer must maintain the invariant that the retrieved element really contains an instance of the target class of the cast.
27.3 Methodology

With this in hand we can (briefly) discuss the methodology of inheritance in object-oriented languages. As we just noted, in Java subclassing entails subtyping — the instance type of a subclass is a subtype of the instance type of the superclass. It is important to recognize that this is a methodological commitment to certain uses of inheritance.

Recall that a subtype relationship is intended to express a form of behavioral equivalence. This is expressed by the subsumption principle, which states that subtype values may be provided whenever a supertype value is required. In terms of a class hierarchy this means that a value of the subclass can be provided whenever a value of the superclass is required. For this to make good sense the values of the subclass should “behave properly” in superclass contexts — they should not be distinguishable from them.

But this isn’t necessarily so! Since inheritance admits overriding of methods, we can make almost arbitrary changes to the behavior of the superclass when defining the subclass. For example, we can turn a stack-like object into a queue-like object (replacing a LIFO discipline by a FIFO discipline) by inheritance, thereby changing the behavior drastically. If we are to pass off a subclass instance as a superclass instance using subtyping, then we should refrain from making such drastic behavioral changes.

The Java type system provides only weak tools for ensuring a behavioral subtyping relationship between a subclass and its superclass. Fundamentally, the type system is not strong enough to express the desired constraints. To compensate for this Java provides the finality mechanism to limit inheritance. Final classes cannot be inherited from at all, ensuring that values of its instance type are indeed instances of that class (rather than an arbitrary subclass). Final methods cannot be overridden, ensuring that certain aspects of behavior are “frozen” by the class definition.

Nominal subtyping may also be seen as a tool for enforcing behavioral subtyping relationships. For unless a class extends a given class or is declared to implement a given interface, no subtyping relationship holds. This helps to ensure that the programmer explicitly considers the behavioral subtyping obligations that are implied by such declarations, and is therefore an aid to controlling inheritance.

---

7 Limited only by finality declarations in the superclass.
8 Nor is the type system of any other language that I am aware of, including ML.
Part XII

Concurrency
Part XIII

Type Checking
Chapter 29

Type Checking

The type system of MinML is given by an inductive definition of the typing judgement $\Gamma \vdash e : \tau$ stating that the expression $e$ has type $\tau$ in typing context $\Gamma$. These rules are said to be *declarative* because they only specify what programs are type correct, without regard to how to determine whether or not this is the case. This chapter is concerned with the latter problem. The central idea is to devise *algorithmic* presentations of the static semantics that serve as a guide to implementation.

29.1 Type Synthesis

The *type checking problem* for MinML is to determine, given $\Gamma$, $e$, and $\tau$, whether or not $\Gamma \vdash e : \tau$ according to the typing rules of MinML. The *type synthesis problem* for MinML is to find, given $\Gamma$ and $e$, all possible $\tau$ such that $\Gamma \vdash e : \tau$. Type checking may be reduced to type synthesis, provided that we can mechanically check whether a given type is a possible type for a given expression and context.

The abstract syntax of MinML has been carefully designed so that there is at most one type $\tau$ for any given $\Gamma$ and $e$; MinML satisfies the *unique typing* property. This means that we may check whether an expression $e$ has type $\tau$ in a typing context $\Gamma$ by synthesizing a type $\tau'$ for $e$ and checking whether or not $\tau' = \tau$. If there is no such $\tau'$, then $e$ is ill-typed (in $\Gamma$), and hence does not type check.

The type synthesis relation for MinML is written $\Gamma \vdash e \uparrow \tau$. This judgement states that the unique (if it exists) type of $e$ in $\Gamma$ is $\tau$. The rules for
deriving this judgement are as follows:

\[(\Gamma(x) = \tau) \quad \frac{}{\Gamma \vdash x : \tau} \]

\[\Gamma \vdash n : \text{int} \]

\[\Gamma \vdash \text{true} : \text{bool} \]

\[\Gamma \vdash \text{false} : \text{bool} \]

\[\begin{align*}
\Gamma \vdash e_1 : \tau_1 & \quad \Gamma \vdash e_2 : \tau_2 \\
\tau_1 = \text{int} & \quad \tau_2 = \text{int}
\end{align*} \frac{}{\Gamma \vdash + (e_1, e_2) : \text{int}} \]

\[\begin{align*}
\Gamma \vdash e : \tau & \quad \Gamma \vdash e_1 : \tau_1 \\
\Gamma \vdash e_2 : \tau_2 & \quad \tau = \text{bool} \\
\tau_1 = \tau_2
\end{align*} \frac{}{\Gamma \vdash \text{if} \ e \ \text{then} \ e_1 \ \text{else} \ e_2 \ \text{fi} : \tau_1} \]

\[\begin{align*}
\Gamma, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash e : \tau & \quad \tau = \tau_2 \\
\Gamma \vdash \text{fun} \ f \ (x : \tau_1) : \tau_2 \ \text{is} \ e \ \text{end} : \tau_1 \rightarrow \tau_2
\end{align*} \]

\[\begin{align*}
\Gamma \vdash e_1 : \tau_1 & \quad \Gamma \vdash e_2 : \tau_2 \\
\tau_1 = \tau_{1,1} \rightarrow \tau_{1,2} & \quad \tau_2 = \tau_{1,1}
\end{align*} \frac{}{\Gamma \vdash \text{apply} (e_1, e_2) : \tau_{1,2}} \]

Note that these rules are formulated with explicit equality checks at those positions where a type constraint must be satisfied. In the declarative formulation we simply exploit the “magic of non-determinism” to specify that certain types must coincide. But from an algorithmic viewpoint we must actually check that certain equations hold. For MinML it is trivial to check type equality by a simple recursive comparison, since two types are equal iff they are identical. However, in richer languages, such as those that support type definitions, the problem of checking type equality is more complex.

**Theorem 29.1**

1. **[Soundness]** If \(\Gamma \vdash e : \tau\), then \(\Gamma \vdash e : \tau\).

2. **[Completeness]** If \(\Gamma \vdash e : \tau\), then there exists \(\tau'\) such that \(\Gamma \vdash e : \tau'\) and \(\tau = \tau'\).

**Proof:**

1. By induction on the definition of the type synthesis relation.
29.2 Synthesis and Analysis

2. By induction on the definition of the typing relation.

Theorem 29.2
Given \( \Gamma \) and \( e \), there is at most one \( \tau \) such that \( \Gamma \vdash e \uparrow \tau \).

Proof: By induction on the definition of the type synthesis relation.

Theorem 29.3
Given \( \Gamma \) and \( e \) we may effectively compute \( \tau \) (if it exists) such that \( \Gamma \vdash e \uparrow \tau \).

Proof: By induction on the structure of \( e \), making use of the syntax-directed nature of the type synthesis rules.

Theorem 29.4
It is effectively decidable whether or not \( \Gamma \vdash e : \tau \).

Proof: Effectively find \( \tau' \) (if it exists) such that \( \Gamma \vdash e \uparrow \tau' \). Then effectively check whether or not \( \tau = \tau' \).

29.2 Synthesis and Analysis

The type synthesis procedure just described is said to be \textit{bottom up} because it synthesizes a type \( \tau \) for an expression \( e \) by recursively synthesizing types for the subterms of \( e \), and using these to synthesize the type of \( e \) (if any exists). The process of synthesizing a type, \( \tau' \), for \( e \) and checking whether the desired type, \( \tau \), is equal to \( \tau' \), is called \textit{type analysis}, since it analyzes whether or not an expression has a given type.

This description reveals an asymmetry between synthesis and analysis that can be profitably eliminated. Rather than simply reduce type analysis to type synthesis and type equality checking, we can instead give a direct definition of the relation \( \Gamma \vdash e \downarrow \tau \), which directly defines whether or not a given expression \( e \) has a given type \( \tau \) in a given context \( \Gamma \). The rules for doing so are said to be \textit{top down}, because they propagate type information from \( e \) into its subterms as part of the type analysis procedure.

The rules for type analysis are essentially just the typing rules for MinML. Since these rules are syntax-directed, it is straightforward to prove by induction on the structure of \( e \) that it is effectively decidable whether or not
This procedure is the analysis procedure alluded to in the previous paragraph.

While there appears to be no great benefit to distinguishing synthesis and analysis rules for MinML, the distinction pays dividends in the form of a simple, yet flexible, form of type inference. As remarked earlier, the syntax of MinML has been carefully designed to support unique type synthesis. In particular, function expressions must be labelled with both their domain and range types so that we can easily synthesize the type of the overall function. (It might be thought that the range type need not be provided, because we can synthesize it from the body, but this is not so obvious, because to synthesize a type for the body, we must assume that the function itself, \( f \), has the type \( \tau_1 \to \tau_2 \) ultimately ascribed to it. It is not immediately clear how to recover \( \tau_2 \), given only \( \tau_1 \). We will discuss how to do this in Chapter 2.)

While it may seem like good practice to indicate the domain and range of a function explicitly, it can be tiresome to do so. For example, if \( F \) is a function whose domain type is \( \text{int} \to \text{int} \), and we are to apply \( F \) to some function expression \( e \), it is tedious to have to specify the domain and range of \( e \) explicitly, since we know from context that they must both be \( \text{int} \), if \( F(e) \) is to be well-typed. In other words if we know the type of a function, why bother repeating that information on the function expression itself?

The process of discovering missing type information is called type inference, since we must infer, or deduce, type information that has been omitted by the programmer. The preceding discussion leads to a weak, but nevertheless useful, form of type inference for MinML. This form of type inference is less powerful than that provided by, say, Standard ML, but requires substantially less machinery to implement. The main idea is to segregate the abstract syntax of MinML into two categories, synthesis expressions and analysis expressions. Roughly speaking, a synthesis expression is one that has enough explicit type information to ensure that we may synthesize a unique (if any exists) type for it. An analysis expression, on the other hand, is one that can only be type checked if the intended type is given, rather than read off the expression itself. This allows us to omit type labels on functions that arise as analysis expressions.

The two categories of expression interact in several ways. Principal among them is the application expression \( e_1(e_2) \). If the application is synthesis, then \( e_1 \) must also be synthesis, but \( e_2 \) is analysis. That is, to synthesize the type of the application, we must synthesize the function type, and use that to analyze the argument to ensure that it has the domain type of the function. If so, then the synthesized type is the range type of the function. On the other hand if the application is analysis, then, since we are
29.2 Synthesis and Analysis

Given the overall type of the application, the argument must be synthesis, so that we can recover the domain type, and then the function position is analysis (since we now have both the domain and range).

This leads to the following refined abstract syntax for MinML in which we make explicit the distinction between synthesis and analysis expressions.

**Synthesis**

\[ s ::= x \mid n \mid o(a_1, \ldots, a_n) \mid \text{true} \mid \text{false} \mid \text{if} \ a \ \text{then} \ s_1 \ \text{else} \ s_2 \ \text{fi} \mid \text{fun} \ f \ (x : \tau_1) : \tau_2 \ \text{is} \ a \ \text{end} \mid \text{apply} (s_1, a_2) \]

**Analysis**

\[ a ::= s \mid \text{if} \ a \ \text{then} \ a_1 \ \text{else} \ a_2 \ \text{fi} \mid \text{fun} \ f (x) : \tau \ \text{is} \ a \ \text{end} \mid \text{apply} (a_1, s_2) \]

Note that any synthesis term is an analysis term (since we can just check type equality after synthesis). There are now two forms of application and two forms of conditional, one of each for each category of expression.

We may now give a simultaneous inductive definition of the synthesis and analysis typing relations, written \( \Gamma \vdash s \uparrow \tau \) and \( \Gamma \vdash a \downarrow \tau \).

\[
\begin{align*}
(\Gamma (x) = \tau) & \quad \Gamma \vdash x \uparrow \tau \\
\Gamma \vdash n \uparrow \text{int} & \\
\Gamma \vdash \text{true} \uparrow \text{bool} & \\
\Gamma \vdash \text{false} \uparrow \text{bool} & \\
\Gamma \vdash a_1 \downarrow \text{int} \quad \Gamma \vdash a_2 \downarrow \text{int} & \quad \Gamma \vdash (+ (a_1, a_2)) \uparrow \text{int} \\
\Gamma \vdash a \downarrow \text{bool} \quad \Gamma \vdash s_1 \uparrow \tau_1 \quad \Gamma \vdash s_2 \uparrow \tau_2 & \quad \tau_1 = \tau_2 \\
\Gamma \vdash \text{if} \ a \ \text{then} \ s_1 \ \text{else} \ s_2 \ \text{fi} \uparrow \tau_1 & \\
\Gamma, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash s \uparrow \tau & \quad \tau = \tau_2 \\
\Gamma \vdash \text{fun} \ f (x : \tau_1) : \tau_2 \ \text{is} \ s \ \text{end} \uparrow \tau_1 \rightarrow \tau_2 & \\
\Gamma \vdash s_1 \uparrow \tau_1 \quad \Gamma \vdash a_2 \downarrow \tau_2 & \quad \tau_1 = \tau_2 \rightarrow \tau \\
\Gamma \vdash \text{apply} (s_1, a_2) \uparrow \tau 
\end{align*}
\]
These rules are sound and complete for MinML type checking.

**Theorem 29.5 (Soundness)**
1. If $\Gamma \vdash s \uparrow \tau'$ and $\tau' = \tau$, then $\Gamma \vdash s : \tau$.
2. If $\Gamma \vdash a \downarrow \tau$, then $\Gamma \vdash a : \tau$.

**Theorem 29.6 (Completeness)**
1. If $\Gamma \vdash s : \tau$, then $\Gamma \vdash s \uparrow \tau'$ and $\tau' = \tau$.
2. If $\Gamma \vdash a : \tau$, then $\Gamma \vdash a \downarrow \tau$.

**Theorem 29.7 (Effectiveness)**
1. Given $\Gamma$ and $s$, we may effectively determine whether or not there exists $\tau$ such that $\Gamma \vdash e \uparrow \tau$, computing $\tau$ in the case that it exists.
2. Given $\Gamma$, $a$, and $\tau$, we may effectively determine whether or not $\Gamma \vdash e \downarrow \tau$.

**29.3 Exercises**
1. Prove soundness, completeness, and effectiveness of both algorithms.
2. Extend both algorithms to MinML enriched with type definitions, with products and sums, with recursive types.
Chapter 30

Type Reconstruction

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.

30.1 Informal Overview

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:

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

WORKING DRAFT

JANUARY 14, 2003
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:

```haskell
fun map (f : $\alpha$, l : $\beta$) : $\gamma$ 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 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 suggests, a type expression that involves type variables acting as placeholders for types. For example, $\alpha \rightarrow \alpha$ list is a type scheme, with scheme variable $\alpha$. It stands for all types of the form $\tau \rightarrow \tau$ 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 $\beta = \beta'$ 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 $\delta$ 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 $\beta = \beta''$ list and $\beta = \beta'''$ list, where $\beta''$ and $\beta'''$ are fresh type variables. The type of `hd(l)` is therefore $\beta''$ and the type of `tl(l)` is $\beta'''$ list.
4. The expression \( \text{map} (f, \text{tl}(l)) \) induces the requirements that \( \alpha = \alpha \) and \( \beta = \beta'' \text{list} \), which arise from equating corresponding argument and parameter types. The type of the expression is therefore \( \gamma \), the result type of \( \text{map} \).

5. The expression \( f \ (\text{hd}(l)) \) 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 \( f \ (\text{hd}(l)) \) is therefore \( \alpha' \).

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

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

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

This completes the generation of constraints.

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

\[
\begin{align*}
\beta &= \beta' \text{list} \\
\beta' &= \beta'' \text{list} \\
\beta'' &= \beta''' \text{list} \\
\alpha &= \alpha \\
\alpha' &= \beta'' \rightarrow \alpha' \\
\gamma &= \gamma' \text{list} \\
\gamma' &= \gamma' \\
\delta \text{list} &= \gamma' \text{list} \\
\gamma &= \delta \text{list}
\end{align*}
\]

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

\[ \alpha = \beta' \rightarrow \gamma' \]
\[ \beta = \beta' \text{list} \]
\[ \gamma = \gamma' \text{list} \]

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

```haskell
fun \text{map} \ (f:\alpha \rightarrow \beta, x:\alpha \text{list}) : \beta \text{list} is
  if \text{null}(l) then
    \text{nil}
  else
    \text{cons} \ (f \ (\text{hd}(l)), \ \text{map} \ (f, \ \text{tl}(l)))
  fi
end
```

### 30.2 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

```haskell
fun \text{f} (x) is if \text{x=0} then 1 else \text{x * f} (\text{x-1}) fi end
```

rather than in the fully explicit form

```haskell
fun \text{f} (x:int):int is if \text{x=0} then 1 else \text{x * f} (\text{x-1}) fi end
```

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

```haskell
fun \text{f} (x) is \text{fun} \ g(y) is \text{x+y} \ end \ end
```
In explicit form we would have to write

\[
\text{fun } f(x:\text{int}):\text{int} \rightarrow \text{int} \text{ is fun } g(y:\text{int}):\text{int} \text{ is } x+y \text{ end end}
\]

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:

\[
\begin{align*}
\overline{x} & \quad \overline{n} \quad \text{(30.1)} \\
\overline{\text{true}} & \quad \overline{\text{false}} \quad \text{(30.2)} \\
\overline{u_1 \ldots u_n} & \quad \overline{o(u_1, \ldots, u_n)} \quad \text{(30.3)} \\
\overline{u \ u_1 \ u_2} & \quad \overline{\text{if } u \text{ then } u_1 \text{ else } u_2 \text{ fi}} \quad \text{(30.4)} \\
\overline{\text{fun } f(x) \text{ is } u \text{ end}} & \quad \text{(30.5)} \\
\overline{u_1 \ u_2} & \quad \overline{\text{apply}(u_1, u_2)} \quad \text{(30.6)}
\end{align*}
\]

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:

\[
\begin{align*}
x^- & = x \\
n^- & = n \\
\text{true}^- & = \text{true} \\
\text{false}^- & = \text{false} \\
o(e_1, \ldots, e_n)^- & = o(e_1^-, \ldots, e_n^-) \\
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ fi}^- & = \text{if } e^- \text{ then } e_1^- \text{ else } e_2^- \text{ fi} \\
\text{fun } f(x:\tau_1):\tau_2 \text{ is } e \text{ end}^- & = \text{fun } f(x) \text{ is } e^- \text{ end}
\end{align*}
\]

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 \( \text{fun } f(x) \text{ is } x \text{ end} \) can be reconstructed to any typed expression of the form \( \text{fun } f(x: \tau) : \tau \text{ is } x \text{ end} \), 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.

\[
\begin{align*}
\Gamma(x) &= \tau \\
\Gamma \vdash x &\rightsquigarrow x : \tau
\end{align*}
\]  

(30.7)

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

\[
\begin{align*}
\Gamma \vdash n &\rightsquigarrow n : \text{int} \\
\Gamma \vdash \text{true} &\rightsquigarrow \text{true} : \text{bool} \\
\Gamma \vdash \text{false} &\rightsquigarrow \text{false} : \text{bool}
\end{align*}
\]  

(30.8) (30.9) (30.10)

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

\[
\begin{align*}
\Gamma \vdash u_1 &\rightsquigarrow e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash u_n &\rightsquigarrow e_n : \tau_n \\
\Gamma &\vdash o(u_1, \ldots, u_n) \rightsquigarrow o(e_1, \ldots, e_n) : \tau
\end{align*}
\]  

(30.11)

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 \( \text{bool} \).

\[
\begin{align*}
\Gamma \vdash u &\rightsquigarrow e : \text{bool} \\
\Gamma \vdash u_1 &\rightsquigarrow e_1 : \tau \\
\Gamma \vdash u_2 &\rightsquigarrow e_2 : \tau \\
\Gamma &\vdash \text{if } u \text{ then } u_1 \text{ else } u_2 \text{ fi} \rightsquigarrow \text{if } e \text{ then } e_1 \text{ else } e_2 \text{ fi} : \tau
\end{align*}
\]  

(30.12)
Functions are reconstructed by “guessing” domain and range types, which are then assigned to the function name and parameter while reconstructing the body.

\[
\Gamma[f: \tau_1 \rightarrow \tau_2][x: \tau_1] \vdash u \rightsquigarrow e : \tau_2 \\
\Gamma \vdash \text{fun } f(x) \text{ is } u \text{ end } \rightsquigarrow \text{fun } f(x: \tau_1): \tau_2 \text{ is } e \text{ end } : \tau_1 \rightarrow \tau_2
\]

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.

\[
\Gamma \vdash u_1 \rightsquigarrow e_1 : \tau_2 \rightarrow \tau \\
\Gamma \vdash u_2 \rightsquigarrow e_2 : \tau_2 \\
\Gamma \vdash \text{apply } (u_1, u_2) \rightsquigarrow \text{apply } (e_1, e_2) : \tau
\]

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 30.1
Using the elaboration rules, show that the following untyped term can be reconstructed as a term of type \( \text{int} \rightarrow \text{int} \). That is, find \( e \) such that \( \vdash u \rightsquigarrow e : \text{int} \rightarrow \text{int} \), where \( u \) is the following untyped term:

\[
(fn f \text{ in } (fn x \text{ in } f(f(x)) \text{ end}) \text{ end})(fn x \text{ in } x \text{ end})
\]

(Here \( fnxieend \) stands for \( \text{fun } f(x) \text{ is } e \text{ end } \), where \( f \not\in \text{FV}(e) \).) 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 30.2**

1. The elaboration relation is **sound**: if \( \Gamma \vdash u \rightsquigarrow e : \tau \), then \( \Gamma \vdash e : \tau \) and \( e^- = u \).

2. The elaboration relation is **complete**: if \( \Gamma \vdash e : \tau \) and \( e^- = u \), then \( \Gamma \vdash u \rightsquigarrow e : \tau \).

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

\[
\Gamma \vdash \text{fun } f(x) \text{ is } u \text{ end } \rightsquigarrow \text{fun } f(x: \tau_1) : \tau_2 \text{ is } e \text{ end } : \tau_1 \rightarrow \tau_2,
\]

where

\[
\Gamma[f: \tau_1 \rightarrow \tau_2][x: \tau_1] \vdash u \rightsquigarrow e : \tau_2.
\]

By induction we know that

\[
\Gamma[f: \tau_1 \rightarrow \tau_2][x: \tau_1] \vdash e : \tau_2.
\]

Therefore

\[
\Gamma \vdash \text{fun } f(x: \tau_1) : \tau_2 \text{ is } e \text{ end } : \tau_1 \rightarrow \tau_2,
\]

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 \( e = \text{fun } f(x: \tau_1) : \tau_2 \text{ is } e_2 \text{ end } \) and \( \tau = \tau_1 \rightarrow \tau_2 \), where

\[
\Gamma \vdash \text{fun } f(x: \tau_1) : \tau_2 \text{ is } e_2 \text{ end } : \tau_1 \rightarrow \tau_2,
\]

by

\[
\Gamma[f: \tau_1 \rightarrow \tau_2][x: \tau_1] \vdash e_2 : \tau_2.
\]
and that $\text{fun } f (x : \tau_1) : \tau_2 \text{ is } e_2 \text{ end} = u$. We are to show that $\Gamma \vdash u \rightsquigarrow e : \tau$. By the definition of the erasure function, $u = \text{fun } f (x) \text{ is } u_2 \text{ end}$, where $u_2 = e_2$. Since $\tau = \tau_1 \rightarrow \tau_2$, it suffices to show that 

$$
\Gamma[f : \tau][x : \tau_1] \vdash u_2 \rightsquigarrow e_2 : \tau_2,
$$

for then it follows immediately that

$$
\Gamma \vdash u \rightsquigarrow e : \tau,
$$

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 30.3

Complete the proof of soundness and completeness of the elaboration relation.

30.3 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 $\text{hd} (l)$, the choice of a type for $l$ is constrained by the requirement that it be a list type. If this expression occurs as $\text{hd} (l) + 2$, then the choice of type for $l$ is further constrained to $\text{intlist}$ by the fact that $\text{hd} (l)$ 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:

\[
\begin{align*}
\alpha & \quad \text{int} \quad \text{bool} \\
\tau_1^* & \quad \tau_2^* \quad \tau_1^* \rightarrow \tau_2^*
\end{align*}
\]

(30.15)

(30.16)

The set \(TV(\tau^*)\) 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:

\[
\begin{align*}
\top & \quad \tau_1^* = \tau_2^* \\
\phi_1 & \quad \phi_2 \quad \phi_1 \land \phi_2
\end{align*}
\]

(30.17)

(30.18)

The constraint generation relation is written \(\Gamma^* \vdash u \leadsto e^* : \tau^* [\phi]\). 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 30.4 below.) This relation will be total function of \(\Gamma^*\) and \(u\), provided that \(\text{FV}(u) \subseteq \text{dom}(\Gamma)\) — that is, for every \(\Gamma^*\)
and \( u \) there will be a unique \( e^* \) and \( \tau^* \) such that \( \Gamma^* \vdash u \leadsto e^* : \tau^* \phi \), 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.

\[
\Gamma^* \vdash x \leadsto x : \Gamma^*(x)[\top]
\] (30.19)

\[
\Gamma^* \vdash n \leadsto n : \text{int}[\top]
\] (30.20)

\[
\Gamma^* \vdash \text{true} \leadsto \text{true} : \text{bool}[\top]
\] (30.21)

\[
\Gamma^* \vdash \text{false} \leadsto \text{false} : \text{bool}[\top]
\] (30.22)

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 \( \tau \).

\[
\Gamma^* \vdash u_1 \leadsto e_1^* : \tau_1^* \phi_1 \quad \ldots \quad \Gamma^* \vdash u_n \leadsto e_n^* : \tau_n^* \phi_n
\]

\[
\Gamma^* \vdash o(u_1, \ldots, u_n) \leadsto o(e_1^*, \ldots, e_n^*) : \tau[\phi_1 \land \cdots \land \phi_n \land \tau_1^* = \tau_1 \land \cdots \land \tau_n^* = \tau_n]
\] (30.23)

Conditionals generate constraints for the sub-expressions, together with the requirement that the type of the test expression be \( \text{bool} \) 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.

\[
\Gamma^* \vdash u \leadsto e^* : \tau^* \phi \quad \Gamma^* \vdash u_1 \leadsto e_1^* : \tau_1^* \phi_1 \quad \Gamma^* \vdash u_2 \leadsto e_2^* : \tau_2^* \phi_2
\]

\[
\Gamma^* \vdash \text{if } u \text{ then } u_1 \text{ else } u_2 \text{ fi} \leadsto \text{if } e^* \text{ then } e_1^* \text{ else } e_2^* \text{ fi} : \alpha[\phi \land \phi_1 \land \phi_2 \land \tau^* = \text{bool} \land \alpha = \tau_1^* \land \alpha = \tau_2^*]
\] (30.24)

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.

\[ \Gamma^* \vdash e^* : \tau^*[\phi] \]

\[ \Gamma^* \vdash \text{fun } f(x) \text{ end } \leadsto \text{fun } f : \alpha_1 \rightarrow \alpha_2 \text{ is } e^* \text{ end } : \alpha_1 \rightarrow \alpha_2 \left[ \phi \land \tau^* = \alpha_2 \right] \]

(30.25)

Applications are handled by introducing a fresh type variable to stand for the result type of the application.

\[ \Gamma^* \vdash e_1^* : \tau_1^*[\phi_1] \quad \Gamma^* \vdash e_2^* : \tau_2^*[\phi_2] \]

\[ \Gamma^* \vdash \text{apply}(u_1, u_2) \leadsto \text{apply}(e_1^*, e_2^*) : \alpha \left[ \phi_1 \land \phi_2 \land \tau_1^* = \tau_2^* \rightarrow \alpha \right] \]

(30.26)

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 \text{int} makes the equation \( \alpha \rightarrow \beta = \text{int} \rightarrow \text{int} \) 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 \( \text{dom}(S) \) of the substitution \( S \) is defined by \( \text{dom}(S) = \{ \alpha \mid S(\alpha) \neq \alpha \} \). 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, \( S(\alpha) = S(S(\alpha)) \) 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 \( S(\alpha) = \beta \rightarrow \text{int} \) and \( S(\beta) = \text{bool} \) is not idempotent, since \( S(S(\alpha)) = \text{bool} \rightarrow \text{int} \neq \beta \rightarrow \text{int} = S(\alpha) \).

The application of a substitution, \( S \), to a type scheme, \( \tau^* \), written \( S(\tau^*) \), is defined by induction on the structure of \( \tau^* \) as follows:

\[
\begin{align*}
\hat{S}(\alpha) & = S(\alpha) \\
\hat{S}(\text{int}) & = \text{int} \\
\hat{S}(\text{bool}) & = \text{bool} \\
\hat{S}(\tau_1 \rightarrow \tau_2) & = \hat{S}(\tau_1) \rightarrow \hat{S}(\tau_2)
\end{align*}
\]

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 \( S(\Gamma^*) \) for the result of applying \( S \) to every type scheme in \( \Gamma^* \). By a similar abuse, we also write \( S(e) \) 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,

\[
S \models \top \quad \text{iff} \quad \text{always}
\]

\[
S \models \tau_1^* = \tau_2^* \quad \text{iff} \quad \hat{S}(\tau_1^*) = \hat{S}(\tau_2^*)
\]

\[
S \models \phi_1 \land \phi_2 \quad \text{iff} \quad S \models \phi_1 \text{ and } S \models \phi_2
\]

We may now state the key properties of the constraint generation relation.

**Theorem 30.4**

1. **Constraint generation is a total function:** For every \(\Gamma^*\) and \(u\) there exists a unique \(e^*, \tau^*\), and \(\phi\) such that \(\Gamma^* \vdash u \leadsto e^* : \tau^* [\phi]\).

2. **Constraint generation is sound:** If \(\Gamma^* \vdash u \leadsto e^* : \tau^* [\phi]\) and \(S \models \phi\), then \(\hat{S}\Gamma^* \vdash u \leadsto \hat{S}e^* : \hat{S}\tau^*\).

3. **Constraint generation is complete:** Suppose that \(\Gamma^* \vdash u \leadsto e^* : \tau^* [\phi]\). Let \(S\) be a substitution such that \(\hat{S}\Gamma^*\) contains no type variables. If \(\hat{S}\Gamma^* \vdash u \leadsto e : \tau\), then there exists substitutions \(S_1\) and \(S_2\) such that \(S_2 = S_1 \circ S\), \(\hat{S}V(\phi) \subseteq \text{dom}(S_2)\), \(S_2 \models \phi\), and \(e = \hat{S}_2e^*\) 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.

### 30.4 Solving Constraints

Constraint solving (also known as unification) is performed by a process similar to Gaussian 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 $\text{dom}(S) \cap \text{TV}(\phi) = \emptyset$. The restriction to idempotent substitutions means that we can regard $S$ as a set of ordered pairs $(\alpha, \tau)$ where $\text{TV}(\tau) \cap \text{dom}(S) = \emptyset$. 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:

\[
\begin{align*}
\top \land \phi & \equiv \phi \\
\phi \land \top & \equiv \phi \\
\phi_1 \land \phi_2 & \equiv \phi_2 \land \phi_1 \\
\phi_1 \land (\phi_2 \land \phi_3) & \equiv (\phi_1 \land \phi_2) \land \phi_3 \\
\tau_1^* = \tau_2^* & \equiv \tau_2^* = \tau_1^*
\end{align*}
\]

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 vari-
ables may be dropped.

\[(S, \text{int} = \text{int} \land \phi) \mapsto (S, \phi)\]
\[(S, \text{bool} = \text{bool} \land \phi) \mapsto (S, \phi)\]
\[(S, \alpha = \alpha \land \phi) \mapsto (S, \phi)\]

Second, an equation between function types reduces to two equations between their respective domains and ranges.

\[(S, \tau_1^* \rightarrow \tau_2^* = \sigma_1^* \rightarrow \sigma_2^* \land \phi) \mapsto (S, \tau_1^* = \sigma_1^* \land \tau_2^* = \sigma_2^* \land \phi)\] (30.27)

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.)

\[(S, \alpha = \tau^* \land \phi) \mapsto ([\alpha = \tau^*] \circ S, \{\tau^*/\alpha\} \phi)\] (30.28)

provided that \(\alpha \notin \text{TV}(\tau^*)\). 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 \((S, \text{int} = \text{bool})\) 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. \(\text{int} = \text{bool}\)
2. \(\tau_1^* \rightarrow \tau_2^* = \tau^*\), where \(\tau^*\) is neither a function type nor a type variable.
3. \(\alpha = \tau^*\), where \(\alpha \in \text{TV}(\tau^*)\).

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 \((S_1, \phi_1) \rightarrow (S_2, \phi_2) \rightarrow \cdots\). 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 \(s < s')\), 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 30.5 (Termination)**

There is no infinite sequence of states \((S_1, \phi_1) \rightarrow (S_2, \phi_2) \rightarrow \cdots\).

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 30.6**

\[ S \models \{\tau^*/\alpha\} \phi \iff S \circ [\alpha = \tau^*] \models \phi. \]

\(^1\)This is the “dictionary ordering” for two-letter words, where the “letters” are natural numbers.
30.4 Solving Constraints

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 30.7
Suppose that $(S, \phi) \to (S', \phi')$. 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 $\phi = \alpha = \tau^* \land \psi$, $\phi' = \{\tau^*/\alpha\} \psi$, 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

$$
T = U' \circ S' \quad \text{by assumption on } T
$$
$$
= U' \circ [\alpha = \tau^*] \circ S \quad \text{by assumption on } S'
$$
$$
= U \circ S \quad \text{by definition of } U
$$

Now we must show that $T \models \alpha = \tau^* \land \psi$. First, observe that by the invariants on the state, $\hat{S}(\tau^*) = \tau^*$ and $S(\alpha) = \alpha$. Next, note that

$$
T(\alpha) = \hat{U}(S(\alpha)) \quad \text{since } T = U \circ S
$$
$$
= \hat{U}(\alpha) \quad \text{by state invariant}
$$
$$
= \hat{U}'(\tau^*) \quad \text{by definition of } U
$$

and that

$$
\check{T}(\tau^*) = \hat{U}(\hat{S}(\tau^*)) \quad \text{since } T = U \circ S
$$
$$
= \hat{U}(\tau^*) \quad \text{by state invariants}
$$
$$
= \hat{U}'(\{\tau^*/\alpha\} \tau^*) \quad \text{by definition of } U
$$
$$
= \hat{U}'(\tau^*) \quad \text{by the occurs check, } \alpha \notin TV(\tau^*)
$$

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 substitution $U'$ such that $T = U' \circ S'$ and $T \models \phi'$. Since $T \models \phi$,
Type Reconstruction

$T(\alpha) = \tilde{T}(\tau^*)$ 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

$$U(\alpha) = \tilde{U}(S(\alpha)) \quad \text{by the state invariants}$$
$$= T(\alpha) \quad \text{since } T = U \circ S$$
$$= \tilde{T}(\tau^*) \quad \text{since } T \models \alpha=\tau^*$$
$$= \tilde{U}(S(\tau^*)) \quad \text{since } T = U \circ S$$
$$= \tilde{U}(\tau^*) \quad \text{by the state invariants}$$

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

$$(U' \circ [\alpha=\tau^*])(\beta) = U'(\beta) \quad \text{since } \beta \neq \alpha$$
$$= U(\beta) \quad \text{by definition of } U'$$

and

$$(U' \circ [\alpha=\tau^*])(\alpha) = \tilde{U}'(\tau^*) \quad \text{by composition of substitutions}$$
$$= \tilde{U}(\tau^*) \quad \text{since } \alpha \notin TV(\tau^*)$$
$$= U(\alpha) \quad \text{since } U(\alpha) = \tilde{U}(\tau^*)$$

Therefore, $U' \circ S' = U \circ [\alpha=\tau^*] \circ S = U \circ S = T$.

An important corollary of Theorems 30.7 and 30.5 states that the transition system computes principal solutions of constraints.

**Corollary 30.8**

Suppose that $(I, \phi) \rightarrow^* (S, \top)$. 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 30.7 by induction on the length of the transition sequence from the initial 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 30.9 (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 \((I, \phi) \rightarrow^* (S, \phi')\). 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 \(\Gamma \vdash u \leadsto e^* : \tau^*[\phi]\).
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 \(S_0(e^*)\) and \(S_0(\tau^*)\).
Part XIV

Storage Management
Chapter 31

Storage Management

The dynamic semantics for MinML given in Chapter 8, and even the C-machine given in Chapter 11, ignore questions of storage management. In particular, all values, be they integers, booleans, functions, or tuples, are treated the same way. But this is unrealistic. Physical machines are capable of handling only rather “small” values, namely those that can fit into a word. Thus, while it is reasonable to treat, say, integers and booleans as values directly, it is unreasonable to do the same with “large” objects such as tuples or functions.

In this chapter we consider an extension of the C-machine to account for storage management. We proceed in two steps. First, we give an abstract machine, called the A-machine, that includes a heap for allocating “large” objects. This introduces the problem of garbage, storage that is allocated for values that are no longer needed by the program. This leads to a discussion of automatic storage management, or garbage collection, which allows us to reclaim unused storage in the heap.

31.1 The A Machine

The A-machine is defined for an extension of MinML in which we add an additional form of expression, a location, \( l \), which will serve as a “reference” or “pointer” into the heap.

Values are classified into two categories, small and large, by the following rules:

\[
\frac{(l \in \text{Loc})}{l \text{ svalue}} \quad (31.1)
\]
A state of the A-machine has the form \((H, k, e)\), where \(H\) is a heap, a finite function mapping locations to large values, \(k\) is a control stack, and \(e\) is an expression. A heap \(H\) is said to be self-contained iff \(FL(H) \subseteq \text{dom}(H)\), where \(FL(H)\) is the set of locations occurring free in any location in \(H\), and \(\text{dom} H\) is the domain of \(H\).

Stack frames are similar to those of the C-machine, but refined to account for the distinction between small and large values.

\[
\begin{align*}
& e_2 \text{ expr} \\
& + (\Box, e_2) \text{ frame}
\end{align*}
\]

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

\[
\begin{align*}
& e_1 \text{ expr} \quad e_2 \text{ expr} \\
& \text{if} \Box \text{ then } e_1 \text{ else } e_2 \text{ if } \text{ frame}
\end{align*}
\]

\[
\begin{align*}
& e_2 \text{ expr} \\
& \text{apply} (\Box, e_2) \text{ frame}
\end{align*}
\]

\[
\begin{align*}
& v_1 \text{ svalue} \\
& \text{apply} (v_1, \Box) \text{ frame}
\end{align*}
\]

Notice that \(v_1\) is required to be a small value; a function is represented by a location in the heap, which is small.

As with the C-machine, a stack is a sequence of frames:

\[
\begin{align*}
& \bullet \text{ stack}
\end{align*}
\]
The dynamic semantics of the A-machine is given by a set of rules defining the transition relation \((H, k, e) \rightarrow_A (H', k', e')\). The rules are similar to those for the C-machine, except for the treatment of functions.

Arithmetic expressions are handled as in the C-machine:

\[
(H, k, +(e_1, e_2)) \rightarrow_A (H, +([\square], e_2) \triangleright k, e_1)
\]

\[(H, +(\square, e_2) \triangleright k, v_1) \rightarrow_A (H, +(v_1, \square) \triangleright k, e_2)\]

\[(H, +(n_1, \square) \triangleright k, n_2) \rightarrow_A (H, k, n_1 + n_2)\]

Note that the heap is simply “along for the ride” in these rules.

Booleans are also handled similarly to the C-machine:

\[(H, k, \text{if } e \text{ then } e_1 \text{ else } e_2 \text{ fi}) \rightarrow_A (H, \text{if } \square \text{ then } e_1 \text{ else } e_2 \text{ fi} \triangleright k, e)\]

\[(H, \text{if } \square \text{ then } e_1 \text{ else } e_2 \text{ fi} \triangleright k, \text{true}) \rightarrow_A (H, k, e_1)\]

\[(H, \text{if } \square \text{ then } e_1 \text{ else } e_2 \text{ fi} \triangleright k, \text{false}) \rightarrow_A (H, k, e_2)\]

Here again the heap plays no essential role.

The real difference between the C-machine and the A-machine is in the treatment of functions. A function expression is no longer a (small) value, but rather requires an execution step to allocate it on the heap.

\[
(H, k, \text{fun } x \ (y: \tau_1) : \tau_2 \text{ is } e \text{ end}) \rightarrow_A (H[l \mapsto \text{fun } x \ (y: \tau_1) : \tau_2 \text{ is } e \text{ end}], k, l)
\]

where \(l\) is chosen so that \(l \notin \text{dom } H\).

Evaluation of the function and argument position of an application is handled similarly to the C-machine.

\[
(H, k, \text{apply}(e_1, e_2)) \rightarrow_A (H, \text{apply}(\square, e_2) \triangleright k, e_1)
\]
Execution of a function call differs from the corresponding C-machine instruction in that the function must be retrieved from the heap in order to determine the appropriate instance of its body. Notice that the location of the function, and not the function itself, is substituted for the function variable:

\[
\begin{align*}
\text{loc } H(v_1) &= \text{fun } f(x:t_1) : t_2 \text{ is } e \text{ end} \\
(H, \text{apply}(v_1, \Box) \triangleright k, v_2) &\rightarrow_{A} (H, k, \{v_1, v_2/f, x\} e)
\end{align*}
\] (31.22)

The A-machine preserves self-containment of the heap. This follows from observing that whenever a location is allocated, it is immediately given a binding in the heap, and that the bindings of heap locations are simply those functions that are encountered during evaluation.

**Lemma 31.1**

If \( H \) is self-contained and \( (H, k, e) \rightarrow_{A} (H', k', e') \), then \( H' \) is also self-contained. Moreover, if \( \text{FL}(k) \cup \text{FL}(e) \subseteq \text{dom } H \), then \( \text{FL}(k') \cup \text{FL}(e') \subseteq \text{dom } H' \).

It is not too difficult to see that the A-machine and the C-machine have the same “observable behavior” in the sense that both machines determine the same value for closed expressions of integer type. However, it is somewhat technically involved to develop a precise correspondence. The main idea is to define the heap expansion of an A-machine state to be the C-machine state obtained by replacing all locations in the stack and expression by their values in the heap. (It is important to take care that the locations occurring in a value stored are themselves replaced by their values in the heap!) We then prove that an A-machine state reaches a final state in accordance with the transition rules of the A-machines iff its expansion does in accordance with the rules of the C-machine. Finally, we observe that the value of a final state of integer type is the same for both machines.

Formally, let \( \widehat{H}(e) \) stand for the substitution

\[
\{H(l_1), \ldots, H(l_n)/l_1, \ldots, l_n\} e,
\]

where \( \text{dom } H = \{l_1, \ldots, l_n\} \). Similarly, let \( \widehat{H}(k) \) denote the result of performing this substitution on every expression occurring in the stack \( k \).
Theorem 31.2
If \((H, k, e) \rightarrow_A (H', k', e')\), then \((\bar{H}(k), \bar{H}(e)) \rightarrow^{0,1}_C (\bar{H}'(k'), \bar{H}'(e'))\).

Notice that the allocation of a function in the A-machine corresponds to zero steps of execution on the C-machine, because in the latter case functions are values.

31.2 Garbage Collection

The purpose of the A-machine is to model the memory allocation that would be required in an implementation of MinML. This raises the question of garbage, storage that is no longer necessary for a computation to complete. The purpose of a garbage collector is to reclaim such storage for further use. Of course, in a purely abstract model there is no reason to perform garbage collection, but in practice we must contend with the limitations of finite, physical computers. For this reason we give a formal treatment of garbage collection for the A-machine.

The crucial issue for any garbage collector is to determine which locations are unnecessary for computation to complete. These are deemed garbage, and are reclaimed so as to conserve memory. But when is a location unnecessary for a computation to complete? Consider the A-machine state \((H, k, e)\). A location \(l \in \text{dom}(H)\) is unnecessary, or irrelevant, for this machine state iff execution can be completed without referring to the contents of \(l\). That is, \(l \in \text{dom} H\) is unnecessary iff \((H, k, e) \rightarrow_A^* (H', \bullet, v)\) iff \((H_l, k, e) \rightarrow_A^* (H''_l, \bullet, v)\), where \(H_l\) is \(H\) with the binding for \(l\) removed, and \(H''\) is some heap.

Unfortunately, a machine cannot decide whether a location is unnecessary!

Theorem 31.3
It is mechanically undecidable whether or not a location \(l\) is unnecessary for a given state of the A-machine.

Intuitively, we cannot decide whether \(l\) is necessary without actually running the program. It is not hard to formulate a reduction from the halting problem to prove this theorem: simply arrange that \(l\) is used to complete a computation iff some given Turing machine diverges on blank input.

Given this fundamental limitation, practical garbage collectors must employ a conservative approximation to determine which locations are unnecessary in a given machine state. The most popular criterion is based
on reachability. A location \( l_n \) is unreachable, or inaccessible, iff there is no sequence of locations \( l_1, \ldots, l_n \) such that \( l_1 \) occurs in either the current expression or on the control stack, and \( l_i \) occurs in \( l_{i+1} \) for each \( 1 \leq i < n \).

**Theorem 31.4**

*If a location \( l \) is unreachable in a state \((H, k, e)\), then it is also unnecessary for that state.*

Each transition depends only on the locations occurring on the control stack or in the current expression. Some steps move values from the heap onto the stack or current expression. Therefore in a multi-step sequence, execution can depend only on reachable locations in the sense of the definition above.

The set of unreachable locations in a state may be determined by tracing. This is easily achieved by an iterative process that maintains a finite set of locations, called the roots, containing the locations that have been found to be reachable up to that point in the trace. The root set is initialized to the locations occurring in the expression and control stack. The tracing process completes when no more locations can be added. Having found the reachable locations for a given state, we then deem all other heap locations to be unreachable, and hence unnecessary for computation to proceed. For this reason the reachable locations are said to be live, and the unreachable are said to be dead.

Essentially all garbage collectors used in practice work by tracing. But since reachability is only a conservative approximation of necessity, all practical collectors are conservative! So-called conservative collectors are, in fact, incorrect collectors that may deem as garbage storage that is actually necessary for the computation to proceed. Calling such a collector “conservative” is misleading (actually, wrong), but it is nevertheless common practice in the literature.

The job of a garbage collector is to dispose of the unreachable locations in the heap, freeing up memory for later use. In an abstract setting where we allow for heaps of unbounded size, it is never necessary to collect garbage, but of course in practical situations we cannot afford to waste unlimited amounts of storage. We will present an abstract model of a particular form of garbage collection, called copying collection, that is widely used in practice. The goal is to present the main ideas of copying collection, and to prove that garbage collection is semantically “invisible” in the sense that it does not change the outcome of execution.
The main idea of copying collection is to simultaneously determine which locations are reachable, and to arrange that the contents of all reachable locations are preserved. The rest are deemed garbage, and are reclaimed. In a copying collector this is achieved by partitioning storage into two parts, called *semi-spaces*. During normal execution allocation occurs in one of the two semi-spaces until it is completely filled, at which point the collector is invoked. The collector proceeds by copying all reachable storage from the current, filled semi-space, called the *from space*, to the other semi-space, called the *to space*. Once this is accomplished, execution continues using the “to space” as the new heap, and the old “from space” is reclaimed in bulk. This exchange of roles is called a *flip*.

By copying all and only the reachable locations the collector ensures that unreachable locations are reclaimed, and that no reachable locations are lost. Since reachability is a conservative criterion, the collector may preserve more storage than is strictly necessary, but, in view of the fundamental undecidability of necessity, this is the price we pay for mechanical collection. Another important property of copying collectors is that their execution time is proportion to the size of the live data; no work is expended manipulating reclaimable storage. This is the fundamental motivation for using semi-spaces: once the reachable locations have been copied, the unreachable ones are eliminated by the simple measure of “flipping” the roles of the spaces. Since the amount of work performed is proportional to the live data, we can amortize the cost of collection across the allocation of the live storage, so that garbage collection is (asymptotically) “free”. However, this benefit comes at the cost of using only half of available memory at any time, thereby doubling the overall storage required.

Copying garbage collection may be formalized as an abstract machine with states of the form \((H_f, S, H_t)\), where \(H_f\) is the “from” space, \(H_t\) is the “to” space, and \(S\) is the scan set, the set of reachable locations. The initial state of the collector is \((H, S, \emptyset)\), where \(H\) is the “current” heap and \(\emptyset \neq S \subseteq \text{dom}(H_f)\) is the set of locations occurring in the program or control stack. The final state of the collector is \((H_f, \emptyset, H_t)\), with an empty scan set.

The collector is invoked by adding the following instruction to the A-machine:

\[
(H, \text{FL}(k) \cup \text{FL}(e), \emptyset) \xrightarrow{\text{G}} (H'', \emptyset, H')
\]

\[
(H, k, e) \xrightarrow{\text{A}} (H', k, e)
\] (31.23)

The scan set is initialized to the set of free locations occurring in either the current stack or the current expression. These are the locations that are immediately reachable in that state; the collector will determine those that
are transitively reachable, and preserve their bindings. Once the collector has finished, the "to" space is installed as the new heap.

Note that a garbage collection can be performed at any time! This correctly models the unpredictability of collection in an implementation, but avoids specifying the exact criteria under which the collector is invoked. As mentioned earlier, this is typically because the current heap is exhausted, but in an abstract setting we impose no fixed limit on heap sizes, preferring instead to simply allow collection to be performed spontaneously according to unspecified criteria.

The collection machine is defined by the following two rules:

\[(H_f[l = v], S \cup \{l\}, H_t) \rightarrow_G (H_f, S \cup \text{FL}(v), H_t[l = v])\]  \hspace{1cm} (31.24)

\[(H_f, S \cup \{l\}, H_t[l = v]) \rightarrow_G (H_f, S, H_t[l = v])\]  \hspace{1cm} (31.25)

The first rule copies a reachable binding in the "from" space to the "to" space, and extends the scan set to include those locations occurring in the copied value. This ensures that we will correctly preserve those locations that occur in a reachable location. The second rule throws away any location in the scan set that has already been copied. This rule is necessary because when the scan set is updated by the free locations of a heap value, we may add locations that have already been copied, and we do not want to copy them twice!

The collector is governed by a number of important invariants.

1. The scan set contains only "valid" locations: \(S \subseteq \text{dom } H_f \cup \text{dom } H_t\);

2. The "from" and "to" space are disjoint: \(\text{dom } H_f \cap \text{dom } H_t = \emptyset\);

3. Every location in "to" space is either in "to" space, or in the scan set: \(\text{FL}(H_t) \subseteq S \cup \text{dom } H_t\);

4. Every location in "from" space is either in "from" or "to" space: \(\text{FL}(H_f) \subseteq \text{dom } H_f \cup \text{dom } H_t\).

The first two invariants are minimal “sanity” conditions; the second two are crucial to the operation of the collector. The third states that the "to" space contains only locations that are either already copied into "to" space, or will eventually be copied, because they are in the scan set, and hence in "from" space (by disjointness). The fourth states that locations in "from" space contain only locations that either have already been copied or are yet to be copied.
These invariants are easily seen to hold of the initial state of the collector, since the “to” space is empty, and the “from” space is assumed to be self-contained. Moreover, if these invariants hold of a final state, then $\text{FL}(H_t) \subseteq \text{dom} H_t$, since $S = \emptyset$ in that case. Thus the heap remains self-contained after collection.

**Theorem 31.5 (Preservation of Invariants)**

If the collector invariants hold of $(H_f, S, H_t)$ and $(H_f, S, H_t) \rightarrow_G (H'_f, S', H'_t)$, then the same invariants hold of $(H'_f, S', H'_t)$.

The correctness of the collector follows from the following lemma.

**Lemma 31.6**

If $(H_f, S, H_t) \rightarrow_G (H'_f, S', H'_t)$, then $H_f \cup H_t = H'_f \cup H'_t$ and $S \cup \text{dom} H_t \subseteq S' \cup \text{dom} H'_t$.

The first property states that the union of the semi-spaces never changes; bindings are only copied from one to the other. The second property states that the domain of the “to” space together with the scan set does not change.

From this lemma we obtain the following crucial facts about the collector. Let $S = \text{FL}(k) \cup \text{FL}(e)$, and suppose that

$$(H, S, \emptyset) \rightarrow^*_G (H'', \emptyset, H').$$

Then we have the following properties:

1. The reachable locations are bound in $H'$: $\text{FL}(k) \cup \text{FL}(e) \subseteq \text{dom} H'$.
   
   This follows from the lemma, since the initial “to” space and the final scan set are empty.

2. The reachable data is correctly copied: $H' \subseteq H$. This follows from the lemma, which yields $H = H'' \cup H'$.
Bibliography