Programming Languages:
Theory and Practice

(Working Draft of September 12, 2001)

Robert Harper
Carnegie Mellon University

Spring Semester, 2001
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 and 2000 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 Fundamentals 1

1 Inductive Definitions 3
  1.1 Informal Overview ........................................... 3
    1.1.1 Judgements and Rules ................................. 3
    1.1.2 Rule Induction ........................................ 4
    1.1.3 Defining Functions by Rule Induction ............. 6
  1.2 Formal Development .................................... 6
    1.2.1 Universes ........................................... 7
    1.2.2 Inference Rules ..................................... 7
    1.2.3 Rule Induction ..................................... 8
    1.2.4 Simultaneous Inductive Definitions ............. 10

2 Concrete Syntax 13
  2.1 Context-Free Grammars ................................. 13
  2.2 Ambiguity ............................................. 15

3 First-Order Abstract Syntax 19
  3.1 Abstract Syntax ....................................... 19
  3.2 Structural Induction .................................. 20
  3.3 Parsing ................................................. 21
  3.4 Taking Liberties When Specifying Syntax .......... 22

4 Higher-Order Abstract Syntax 23
  4.1 Binding and Scope ..................................... 23
  4.2 Renaming Bound Variables ............................. 25
  4.3 Substitution and Capture ............................. 27

WORKING DRAFT SEPTEMBER 12, 2001
## CONTENTS

4.4 de Bruijn Indices ........................................... 28

II Effect-free Languages ........................................ 31

5 MinML: A MiniMaL Language ............................... 33
5.1 Abstract Syntax ............................................. 33

6 Static Semantics ................................................ 35
6.1 Type System .................................................. 35
6.2 Properties of Typing ........................................ 37

7 Dynamic Semantics ........................................... 41
7.1 The Dynamic Semantics of MinML ....................... 42
7.2 Properties of the Dynamic Semantics ................... 44

8 Type Safety .................................................... 47
8.1 Defining Type Safety ....................................... 47
8.2 Type Safety of MinML ....................................... 48

III Evaluation Semantics ........................................ 53

9 Evaluation Semantics .......................................... 55
9.1 Evaluation Semantics ....................................... 55
9.2 Correspondences ............................................ 56
9.3 Type Safety For Evaluation ............................... 57

10 Cost Semantics ............................................... 61
10.1 A Cost Semantics for MinML ............................ 61
10.2 Relating Cost Semantics to Transition Semantics .... 62

IV Data Structures and Dynamic Typing ................. 65

11 Aggregate Data Structures ................................. 67
11.1 Products .................................................... 67
11.2 Sums ......................................................... 69
11.3 Recursive Types ............................................ 70
## CONTENTS

### 12 Dynamic Typing
- 12.1 Dynamic Typing ........................................ 75
- 12.2 Implementing Dynamic Typing ......................... 76
- 12.3 Dynamic Typing as Static Typing ....................... 78

### V Control Flow

### 13 Abstract Machines
- 13.1 Control Flow ........................................... 84
- 13.2 Environments ........................................... 90

### 14 Continuations
- 14.1 Informal Overview of Continuations .................. 99
- 14.2 Semantics of Continuations ............................ 102
- 14.3 Coroutines .............................................. 104

### 15 Exceptions

### VI Effect-ful Languages

### 16 TinyC, A Tiny Fragment of C
- 16.1 Syntax .................................................. 120
- 16.2 Static Semantics ....................................... 122
- 16.3 Dynamic Semantics .................................... 125
- 16.4 Block Structure ......................................... 129
- 16.5 Type Safety ............................................. 129

### 17 Mutable Storage
- 17.1 References .............................................. 131

### 18 Lazy Evaluation
- 18.1 Lazy Types ............................................. 139
  - 18.1.1 Lazy Lists in an Eager Language ................. 141
  - 18.1.2 Delayed Evaluation and Lazy Data Structures ... 147
- 18.2 Lazy Languages ......................................... 149
  - 18.2.1 Call-by-Name and Call-by-Need .................. 151
  - 18.2.2 Strict Types in a Lazy Language ................ 153
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VII Type Checking</td>
<td>157</td>
</tr>
<tr>
<td>19 Type Checking</td>
<td>159</td>
</tr>
<tr>
<td>20 Type Reconstruction</td>
<td>161</td>
</tr>
<tr>
<td>20.1 Informal Overview</td>
<td>161</td>
</tr>
<tr>
<td>20.2 Type Reconstruction</td>
<td>164</td>
</tr>
<tr>
<td>20.3 Constraint Generation</td>
<td>169</td>
</tr>
<tr>
<td>20.4 Solving Constraints</td>
<td>174</td>
</tr>
<tr>
<td>VIII Polymorphism and Data Abstraction</td>
<td>181</td>
</tr>
<tr>
<td>21 Expression Equivalence</td>
<td>183</td>
</tr>
<tr>
<td>21.1 Expression Equivalence</td>
<td>184</td>
</tr>
<tr>
<td>21.2 Some Laws of Equivalence</td>
<td>186</td>
</tr>
<tr>
<td>21.2.1 General Laws</td>
<td>186</td>
</tr>
<tr>
<td>21.2.2 Symbolic Evaluation</td>
<td>187</td>
</tr>
<tr>
<td>21.2.3 Extensionality</td>
<td>188</td>
</tr>
<tr>
<td>21.2.4 Strictness Properties</td>
<td>188</td>
</tr>
<tr>
<td>21.2.5 Arithmetic Laws</td>
<td>189</td>
</tr>
<tr>
<td>21.2.6 Products</td>
<td>189</td>
</tr>
<tr>
<td>21.2.7 Lists</td>
<td>190</td>
</tr>
<tr>
<td>22 Polymorphism</td>
<td>193</td>
</tr>
<tr>
<td>22.1 Polymorphic MinML</td>
<td>194</td>
</tr>
<tr>
<td>22.2 ML-style Type Inference</td>
<td>200</td>
</tr>
<tr>
<td>22.3 Parametricity</td>
<td>201</td>
</tr>
<tr>
<td>22.3.1 Informal Discussion</td>
<td>202</td>
</tr>
<tr>
<td>22.3.2 Relational Parametricity</td>
<td>205</td>
</tr>
<tr>
<td>23 Data Abstraction</td>
<td>211</td>
</tr>
<tr>
<td>23.1 Existential Types</td>
<td>212</td>
</tr>
<tr>
<td>23.1.1 Abstract Syntax</td>
<td>212</td>
</tr>
<tr>
<td>23.1.2 Correspondence With ML</td>
<td>212</td>
</tr>
<tr>
<td>23.1.3 Static Semantics</td>
<td>214</td>
</tr>
<tr>
<td>23.1.4 Dynamic Semantics</td>
<td>215</td>
</tr>
<tr>
<td>23.1.5 Safety</td>
<td>216</td>
</tr>
<tr>
<td>23.2 Representation Independence</td>
<td>216</td>
</tr>
</tbody>
</table>
## IX Subtyping and Inheritance

### 24 Subtyping

24.1 MinML With Subtyping .......................... 223
24.2 Varieties of Subtyping .......................... 225
  24.2.1 Arithmetic Subtyping ......................... 225
  24.2.2 Function Subtyping .......................... 226
  24.2.3 Product and Record Subtyping ................. 228
  24.2.4 Reference Subtyping ........................ 229
24.3 Type Checking With Subtyping ..................... 230
24.4 Implementation of Subtyping ....................... 232
  24.4.1 Coercions ................................ 232

### 25 Inheritance and Subtyping in Java

25.1 Inheritance Mechanisms in Java ..................... 237
  25.1.1 Classes and Instances ......................... 237
  25.1.2 Subclasses ................................ 239
  25.1.3 Abstract Classes and Interfaces ............... 240
25.2 Subtyping in Java ................................ 242
  25.2.1 Subtyping ................................ 242
  25.2.2 Subsumption ................................ 243
  25.2.3 Dynamic Dispatch ............................ 244
  25.2.4 Casting ................................ 245
25.3 Methodology .................................. 247

### 26 Featherweight Java

26.1 Abstract Syntax ................................ 249
26.2 Static Semantics ................................. 252
26.3 Dynamic Semantics ............................... 254
26.4 Type Safety .................................. 256
26.5 Acknowledgement ................................ 257

## X Parallelism and Concurrency

### 27 Parallelism

27.1 Bounded Parallelism .............................. 261
27.2 Work and Depth ................................ 263
27.3 Unbounded Parallelism ............................ 265
27.4 Provable Implementations (Summary) ............... 269
Part I

Fundamentals
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. In subsequent chapters we will make frequent use of inductive definitions.

1.1 Informal Overview

1.1.1 Judgements and Rules

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

The rules determine the conditions under which a judgement is derivable. In general a rule is an implication stating that a judgement is derivable if some other judgements (perhaps none) are derivable. Rules are usually 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 is to be understood as an implication stating that if $j_1, \ldots, j_n$ are derivable, then so is $j$. Axioms state that a judgement is unconditionally derivable; all other rules are conditional on the derivability
of their premises. To determine whether a judgement is derivable, simply read the rules as implications and perform ordinary logical deduction.

For example, consider the following set, $\mathcal{R}_N$, of rules for deriving judgements of the form $x$ nat:

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

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

\[\text{succ(succ(zero)) nat}\]

is derivable using these rules.

To take another example, consider the following set, $\mathcal{R}_T$, of rules for deriving judgements of the form $t$ tree:

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

In words, empty is a binary tree, and if $x$ and $y$ are binary trees, so is $\text{node(x, y)}$. It is easy to see that the judgement

\[\text{node(empty, node(empty, empty)) tree}\]

is derivable according to these rules.

Here is a set, $\mathcal{R}_{TF}$, of rules for deriving judgements of the form $t$ tree and $f$ forest.

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

### 1.1.2 Rule Induction

The rules constituting an inductive definition are exhaustive in the sense that the only way that a judgement can be derived is in accordance with these rules. This means that we can “run the rules backward” when reasoning about the derivable judgements. For example, suppose that $x$ nat is derivable from the rules $\mathcal{R}_N$ given above. Since the rules $\mathcal{R}_N$ are exhaustive, the judgement $x$ nat must be derived either by the first rule, in which case $x = \text{zero}$, or by the second rule, in which case $x = \text{succ(y)}$ where $y$ nat is also derivable. Similarly, if $t$ tree is derivable according to the rules $\mathcal{R}_T$
given above, then either \( t = \text{empty} \) or \( t = \text{node}(t_1, t_2) \), where \( t_1 \) is a tree and \( t_2 \) is a tree are both derivable.

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

\[
\frac{j_1 \quad \ldots \quad j_n}{j},
\]

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 derivable judgement.

Note that when showing that \( P \) holds of the conclusion of a rule, we may assume that \( P \) holds for each of its premises. This is may be justified informally by thinking of the premises as being derived “earlier” than the conclusion, so that the property \( P \) “already” holds of the premises by the time we consider the conclusion. For an axiom there are no premises, so we must show that \( P \) holds of the conclusion outright. This corresponds to the axioms holding at the “beginning of time”.

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 \( P(x) \), rather than \( P_l(x) \) or \( P(x l) \).

Returning to the example rule sets given above, to show that if \( x \text{ nat} \) then \( P_{\text{nat}}(x) \), it is enough to show

1. \( P_{\text{nat}}(\text{zero}) \).
2. if \( P_{\text{nat}}(x) \), then \( P_{\text{nat}}(\text{succ}(x)) \).

This is just the ordinary principle of mathematical induction.

Similarly, to show that if \( t \text{ tree} \), then \( P_{\text{tree}}(t) \), it is enough to show

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

This is sometimes called tree induction.

For a simultaneous definition of several judgements, we may reason by simultaneous induction. For example, to show that if \( t \text{ tree} \), then \( P_{\text{tree}}(t) \) and if \( f \text{ forest} \), then \( P_{\text{forest}}(f) \), 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 $P_{\text{forest}}(\text{cons}(t, f))$.

Notice that the assumptions and conclusions are intertwined in a pattern closely following the rules themselves.

### 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, the following equations determine the height of a binary tree:

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

We prove by rule induction that if $t$ tree then there exists a unique $n \geq 0$ such that $\text{hgt}_{\text{tree}}(t) = n$. We consider each rule in $\mathcal{R}_T$ in turn. The first rule, stating that $\text{empty}$ tree, 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 these 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 $\text{hgt}_{\text{forest}}$.

### 1.2 Formal Development

In this section we give a more rigorous treatment of inductive definitions as subsets of a given universe.
1.2 Formal Development

1.2.1 Universes

We will work with inductively defined subsets of a given universe of discourse. That is, we fix a set of all possible objects under consideration, and define subsets of this set. The choice of universe is essentially arbitrary — it need only be an infinite set. However, in practice we will be concerned with universes derived from two basic sets: the universe of strings over a given alphabet of symbols, and the universe of terms over a given set of operators.

Let $\Sigma$ be a countable set of symbols, or characters. For example, $\Sigma$ might be the set of ASCII or UniCode characters. The set $\Sigma^*$ is defined to be the set of strings, or finite sequences, of characters 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.

Let $O$ be a countable set of operators assigned arities by a function $\alpha : O \to N$. An operator $o \in O$ 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 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 terms. $^1$

We often work with combinations of these basic universes. For example, we may consider inductive subsets of $T \times T$, the set of ordered pairs of terms, and so forth.

1.2.2 Inference Rules

An inductive definition of a subset of a universe $U$ consists of a collection of inference rules defining the conditions for membership in that set (in a sense to be made precise shortly). An inference rule over $U$ is a configuration of the form

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

where $x \in U$ and, for each $1 \leq i \leq n$, $x_i \in U$. 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, all over the same universe $U$.

A subset $A \subseteq U$ is closed under rule set $R$, or $A$ is $R$-closed, iff whenever

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

$^1$This is itself an example of an inductive definition! However, we have to start somewhere, so we assume that the definition of $T$ is understood. In set theory we take the set $\mathbb{N}$ as given ("the axiom of infinity"), and work from there.
is a rule in $R$ with $x_i \in A$ for every $1 \leq i \leq n$, then $x \in A$. A rule set $R$ over a universe $U$ determines a set $I(R) \subseteq U$ defined by the equation

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

This is called the set inductively defined by $R$.

For example, here is a set, $R_P$, of rules for deriving strings that, as we shall prove later, are palindromes (equal to themselves when reversed):

$$\varepsilon \quad a \quad \text{s a s a}$$

The set of rules $R_P$ 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 infinitely 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 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 $s \in \Sigma^*$.

### 1.2.3 Rule Induction

The set inductively defined by a rule set $R$ is the least set closed under $R$.

**Theorem 1.1**

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

1. $I$ is $R$-closed.

2. If $J$ is $R$-closed, then $I \subseteq J$.

**Proof:**

1. Suppose that

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

is a rule in $R$, and that

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

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

Working Draft September 12, 2001
2. If \( J \) is \( \mathcal{R} \)-closed, then it is among the sets in the intersection defining \( I \). So \( I \subseteq J \).

\[ \begin{array}{c}
\end{array} \]

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.\(^2\)

When proving that a set is \( \mathcal{R} \)-closed, we consider each rule in turn, assuming that the premises are in the set, and showing that the conclusion is in the set. If there are no premises, this reduces to showing directly that the conclusion is in the set. These are the base cases of the proof. Otherwise we assume that the premises are all in the set, and deduce that the conclusion must also be. These assumptions are the inductive hypotheses of the proof; the inductive step is to show that these hypotheses suffice for the conclusion to be in the desired set.

\(^2\)You might also like to prove that every palindrome is a member of \( P \). This can be achieved by induction over the length of palindrome \( s \). Notice that two base cases are required!
A common use of rule induction is to justify the definition of a function whose domain is an inductively-defined set. 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 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 \( R_P \). 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 \( a \, s \, a \).

### 1.2.4 Simultaneous Inductive Definitions

It is often useful to define several subsets of the universe simultaneously by giving for each set a collection of rules defining the conditions for membership in that set. The definition is “simultaneous” in the sense that the rule sets may “cross reference” one another in a mutually recursive manner. Such a simultaneous inductive definition can be reduced to a single, ordinary inductive definition by a simple trick.

Let \( L \) be a finite set of labels. A simultaneous inductive definition of a family of subsets \( \{ I_l \}_{l \in L} \) of a universe \( U \) is simply an ordinary inductive definition over the set

\[
U \times L = \{ (x, l) \mid x \in U, \, l \in L \}.
\]

\[\text{It suffices to consider finite sets of integers, but in practice we choose more informative labels.}\]
The elements of $U \times L$ are pairs of the form $(x, l)$, where $x \in U$ and $l \in L$. We often write these pairs in the form $x l$, as we did in the informal development.

We may think of $U \times L$ as the disjoint union of $L$ copies of $U$:

$$U \times L = \bigcup_{l \in L} U \times \{l\}$$

Given a subset $X \subseteq U \times L$ and a label $l \in L$, the section $X_l \subseteq U$ of $X$ is defined by the equation

$$X_l = \{ x \mid (x, l) \in X \}.$$

That is, $X_l$ consists of those elements of $X$ that are labelled by $l \in L$. We may recover $X$ as the disjoint union of its sections:

$$X = \bigcup_{l \in L} X_l \times \{l\}$$

When working over the universe $U \times L$, all rules have the form

$$\frac{(x_1, l_1) \ldots (x_k, l_k)}{(x, l)}$$

for some $k \geq 0$. Such a rule specifies a condition for membership in the set with label $l$ in terms of the members of the sets with labels $l_1, \ldots, l_k$, respectively.

Let $R$ be a rule set over $U \times L$, and let $I = I(R)$ be the set inductively defined by $R$. This determines a family of sets $\{ I_l \}_{l \in L}$ consisting of the sections of $I$ for each of the labels $L$. This family of sets is simultaneously closed under $R$ in the sense that if

$$\frac{(x_1, l_1) \ldots (x_k, l_k)}{(x, l)}$$

is a rule in $R$, and $x_i \in I_{l_i}$ for each $1 \leq i \leq k$, then $x \in I_l$. Moreover, it is the smallest such family in the sense that if $\{ X_l \}_{l \in L}$ is simultaneously closed under $R$, then $I_l \subseteq X_l$ for each $l \in L$.

This observation licenses the principle of simultaneous rule induction:

To show that $I_l \subseteq X_l$ for each $l \in L$, it is enough to show that the family $\{ X_l \}_{l \in L}$ is simultaneously closed under $R$.

An example will make this all clear. Let $I = I(R_{TF})$ be the set inductively defined by the rule set $R_{TF}$ given in Section Section 1.1. The sets of
trees and forests are simply the sections of \( I \) by the labels \textit{tree} and \textit{forest}, respectively.

\[ I_{\text{tree}} = \{ t \in T \mid t \text{ tree} \in I \} \]

and

\[ I_{\text{forest}} = \{ f \in T \mid f \text{ forest} \in I \}. \]

We may prove by simultaneous induction that every tree has a height that is one more than the height of its forest of children, where the height of a forest is the maximum height of its constituent trees. We first define

\[ hgt_{\text{tree}}(\text{node}(f)) = 1 + hgt_{\text{forest}}(f) \]

and

\[ hgt_{\text{forest}}(\text{nil}) = 0 \]

\[ hgt_{\text{forest}}(\text{cons}(t, f)) = \max(hgt_{\text{tree}}(t), hgt_{\text{forest}}(f)). \]

To show that each tree and forest has a unique height, we show that the sets

\[ H_{\text{tree}} = \{ t \in T \mid \exists! n \ hgt_{\text{tree}}(t) = n \} \]

and

\[ H_{\text{forest}} = \{ f \in T \mid \exists! n \ hgt_{\text{forest}}(f) = n \} \]

are simultaneously closed under the rules defining variadic trees and forests.
Chapter 2

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.

2.1 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 $N$ of non-terminals that stand for the syntactic categories.

3. A set $P$ 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:

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

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

\[ t_1 A_1 \cdots t_n A_n \]

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

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.

\begin{align*}
\text{Digits} & \quad d ::= 0 | 1 | \cdots | 9 \\
\text{Numbers} & \quad n ::= d | n d \\
\text{Expressions} & \quad e ::= n | e + e | e * e
\end{align*}

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} & \quad \cdots \quad 9 \text{ digit} \\
\hline
\text{d digit} & \quad n \text{ number} \quad \text{d digit} \\
\text{d number} & \quad n d \text{ number}
\end{align*}
2.2 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 $I_{expr}$, and so forth.

### 2.2 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 $1+2*3$ 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 $(1+2)*3$; the latter corresponds to the expression $1+(2*3)$.

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 $eval$ 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*}
eval_{dig}(0) &= 0 \\
&\vdots \\
eval_{dig}(9) &= 9 \\

eval_{num}(d) &= \eval_{dig}(d) \\
\eval_{num}(n\,d) &= 10 \times \eval_{num}(n) + \eval_{dig}(d) \\
\eval_{exp}(n) &= \eval_{num}(n) \\
\eval_{exp}(e_1 + e_2) &= \eval_{exp}(e_1) + \eval_{exp}(e_2) \\
\eval_{exp}(e_1 \times e_2) &= \eval_{exp}(e_1) \times \eval_{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 multiplication).

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.

```
Digits   d  ::=  0 | 1 | ··· | 9
Numbers  n  ::=  d | nd
Expressions  e  ::=  t | t+e
Terms    t  ::=  f | f*t
Factors  f  ::=  n | (e)
```

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
2.2 Ambiguity

the new syntactic categories.

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

A straightforward proof by rule induction shows that these functions are well-defined.
Chapter 3

First-Order Abstract Syntax

The concrete syntax of a language is an inductively-defined set of strings over a given alphabet. Its abstract syntax is an inductively-defined set of terms whose outermost constructor uniquely determines the rule used to construct it. Since constructors are one-to-one functions, there is no question of ambiguity — a term has a unique outermost constructor that can be directly “read off” of the term itself. The representation of abstract syntax as terms is called first-order abstract syntax. In Chapter 4 we will discuss a refinement, called higher-order abstract syntax.

3.1 Abstract Syntax

The first-order abstract syntax of a language is specified by an inductive definition of a set of terms. For example, the abstract syntax of arithmetic expressions is given by the following rules:

\[
\text{number}(n) \Rightarrow \text{expr} \tag{3.1}
\]

\[
\frac{e_1 \Rightarrow \text{expr} \quad e_2 \Rightarrow \text{expr}}{\text{plus}(e_1, e_2) \Rightarrow \text{expr}} \tag{3.2}
\]

\[
\frac{e_1 \Rightarrow \text{expr} \quad e_2 \Rightarrow \text{expr}}{\text{times}(e_1, e_2) \Rightarrow \text{expr}} \tag{3.3}
\]

Notice that the conclusion of each rule is a term whose outermost constructor uniquely identifies the rule used to construct it.
As a convenience we often use a notation similar to context-free grammars to specify the abstract syntax. The difference compared to 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 terms.

For example, the abstract syntax of the language of arithmetic expressions introduced in Chapter 2 may be defined by the following grammar:

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

This has the same meaning as the earlier inductive definition.

### 3.2 Structural Induction

When applied to the rules defining the abstract syntax of a language, the principle of rule induction is called *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 \( \text{number}(n) \) 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

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

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 \( \text{eval}(e) = n \).
3.3 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 terms), then a parser is a function $\text{parse} : C \rightarrow A$ mapping strings to terms. 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 2. Since we wish to define a function on the concrete syntax, it should be clear from the discussion in Section 2.2 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:

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

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.

### 3.4 Taking Liberties When Specifying Syntax

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.

For example, we might define the (concrete and abstract) syntax of the language of arithmetic expressions by the following grammar:

\[
\text{Expressions } e ::= n \mid e_1 + e_2 \mid e_1 \ast e_2
\]

This grammar can be read as an ambiguous specification of the concrete syntax that is implicitly disambiguated using standard methods, and as an unambiguous specification of the abstract syntax in the form given above. 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.
Chapter 4

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. This makes it possible to work with an expression without fear of ambiguity.

Higher-order abstract syntax takes this process one step further by exposing the “binding structure” of an expression in the sense of making clear both the point at which a variable is \textit{bound} in an expression and also its \textit{scope}, or range of significance.

4.1 Binding and Scope

As an illustration, let us extend the language of arithmetic expressions with a means of defining variables. Here is the complete grammar:

\[
\text{Expressions} \quad e \ ::= \quad x \mid n \mid e_1 + e_2 \mid e_1 * e_2 \mid \text{let } x \text{ be } e_1 \text{ in } e_2
\]

Here \(x\) ranges over some countable set of variables, and \(\text{let } x \text{ be } e_1 \text{ in } e_2\) is a new expression form for defining variables. In keeping with the conventions discussed earlier, this grammar may be read either as a specification of the concrete syntax (in which case some additional disambiguation is needed) or as a specification of the abstract syntax (in which case the constructors are implicit).

In the expression \(\text{let } x \text{ be } e_1 \text{ in } e_2\) the variable \(x\) is \textit{bound} by the \textit{let} expression, with the \textit{scope} of \(x\) being its body, \(e_2\). The bound variable \(x\) may
be used within $e_2$ to refer to its binding, $e_1$. The actual name of that bound variable does not matter; the variable is just a reference to its binding site. Thus the expression

$$\text{let } x \text{ be } 2 \text{ in } x + x$$

is entirely indistinguishable from the expression

$$\text{let } y \text{ be } 2 \text{ in } y + y,$$

in which we’ve used a different name for the bound variable of the expression. This convention is called renaming of bound variables, or, for historical reasons, $\alpha$-conversion.

Variables that are not bound in an expression are free. For example, in the expression

$$\text{let } x \text{ be } 10 \text{ in } x + y$$

the variable $x$ is bound in the expression $x + y$, but the variable $y$ is free in that same expression. Every variable must be bound somewhere in a program, but when examining a sub-expression in isolation we may encounter variables that are not bound.

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

$$\text{let } x = 10 \text{ in } (\text{let } x = 11 \text{ in } x + x) + x$$

Both the outer and the inner let expressions bind the variable $x$. The inner let binds all occurrences of $x$ within its body; the outer let also binds all occurrences of $x$ within its body, except those that occur within another binding of $x$.

The general rule is this: a non-binding occurrence of a variable is bound at the nearest enclosing binder for that variable determined by moving “upwards” in the piece of abstract syntax in which it occurs. If no binder for that variable is encountered, the variable is free within that phrase; otherwise, we determine its binder and consider it bound. Since a binder for a variable can be enclosed within the scope of another binder for the same variable, the inner occurrence is said to shadow the outer variable, or to puncture the scope of the outer binder. Shadowing can always be avoided by renaming bound variables. For example, we can rewrite

$$\text{let } x = 10 \text{ in } (\text{let } x = 11 \text{ in } x + x) + x$$

as
4.2 Renaming Bound Variables

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

by choosing a different name for the inner binding let.

The set of free variables occurring in an expression $e$ is defined by structural induction on $e$ as follows:

- $FV(var(x)) = \{x\}$
- $FV(number(n)) = \emptyset$
- $FV(plus(e_1, e_2)) = FV(e_1) \cup FV(e_2)$
- $FV(times(e_1, e_2)) = FV(e_1) \cup FV(e_2)$
- $FV(let(x, e_1, e_2)) = FV(e_1) \cup (FV(e_2) \setminus \{x\})$

Observe that the free variables of a let expression do not include its bound variable!

4.2 Renaming Bound Variables

While it may be handy to assume that distinct bound variables have distinct names, it is an undue burden on the programmer to insist on this rule. After all, suppose that you are linking with a million-line library: how would you know what variable names to avoid, and why on earth should you care? However, since the names of bound variables are not semantically significant, we can accrue the benefits of unique names by adopting a simple convention — we identify all phrases that differ only in the names of bound variables. This is convention is called identification up to $\alpha$-conversion. Under this convention the expressions

let x be 10 in (let x be 11 in x+x)+x

and

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

are regarded as being identical.

Higher-order abstract syntax is, by definition, first-order abstract syntax “up to $\alpha$-conversion”. Technically, this means that a piece of higher-order abstract syntax is an equivalence class of first-order abstract syntax, where any two terms that differ only in the names of their bound variables are regarded as equivalent. This relation is called $\alpha$-equivalence, and is an example of the more general concept of a structural equivalence on expressions.
The relation of \( \alpha \)-equivalence between expressions, written \( e \equiv e' \) is inductively defined by the following rules.

\[
\frac{(x' \notin \text{FV}(e_2))}{\text{let } x \text{ be } e_1 \text{ in } e_2 \equiv \text{let } x' \text{ be } e_1 \text{ in } [x'/x]e_2} \quad (4.1)
\]

\[
e \equiv e \quad (4.2)
\]

\[
e \equiv e' \\
e' \equiv e \quad (4.3)
\]

\[
e \equiv e' \quad e' \equiv e'' \\
e \equiv e'' \quad (4.4)
\]

\[
e_1 \equiv e'_1 \quad e_2 \equiv e'_2 \\
e_1 + e_2 \equiv e'_1 + e'_2 \quad (4.5)
\]

\[
e_1 \equiv e'_1 \quad e_2 \equiv e'_2 \\
e_1 \ast e_2 \equiv e'_1 \ast e'_2 \quad (4.6)
\]

\[
\frac{e_1 \equiv e'_1 \quad e_2 \equiv e'_2}{\text{let } x \text{ be } e_1 \text{ in } e_2 \equiv \text{let } x \text{ be } e'_1 \text{ in } e'_2} \quad (4.7)
\]

The first rule is the axiom of variable renaming, stating that the names of bound variables do not matter. It is stated in terms of an operation to rename free variables, written \([x'/x]e_2\). The next three rules state that \( \alpha \)-equivalence is an equivalence relation (i.e., is reflexive, symmetric, and transitive). The remaining rules state that \( \alpha \)-equivalence is a congruence, meaning that it is compatible with all of the term-forming operators. Altogether these rules may be summed up by saying that \( \alpha \)-equivalence is the least congruence containing the variable-renaming axiom.

As usual, an equivalence class is designated by choosing a representative — that is, by making a specific choice of bound variable names. This choice can always be made so as to satisfy any finite number of variable avoidance conditions. For example, if we wish to avoid the variable \( x \), we may choose the expression

\[
\text{let } y \text{ be } 10 \text{ in } y + y
\]
as the representative of the equivalence class that also contains

\[
\text{let } x \text{ be } 10 \text{ in } x + x.
\]

The key to making higher-order abstract syntax work is this. Whenever we introduce a new form of expression, we must explicitly indicate which variables are bound in which sub-expressions, if any are bound at all. Once we have done so, we have automatically determined the renaming convention in effect, and can avoid further problems with choice of variable names.

### 4.3 Substitution and Capture

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

\[
\text{let } y=10 \text{ in } x+y
\]

and \(e'\) is the expression

\[
2*y+1
\]

The result of substituting \(e'\) for free occurrences of \(x\) in \(e\) is not

\[
\text{let } y=10 \text{ in } (2*y+1)+y
\]

but rather

\[
\text{let } z=10 \text{ in } (2*y+1)+z
\]

Proper substitution requires that we rename \(y\) in \(e\) before substituting to avoid capturing free occurrences of \(y\) in \(e'\). The idea is that the free occurrence of \(y\) in \(e'\) is a different variable from the bound occurrence introduced by the \texttt{let} expression. By renaming the bound variable before substituting, we may avoid confusing these distinct variables.

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

\[
\begin{align*}
    [e'/x]\text{var}(x) &= e' \\
    [e'/x]\text{var}(y) &= y \\
    [e'/x]\text{number}(n) &= \text{number}(n) \\
    [e'/x]\text{plus}(e_1, e_2) &= \text{plus}([e'/x]e_1, [e'/x]e_2) \\
    [e'/x]\text{times}(e_1, e_2) &= \text{times}([e'/x]e_1, [e'/x]e_2) \\
    [e'/x]\text{let}(x, e_1, e_2) &= \text{let}(x, [e'/x]e_1, e_2) \\
    [e'/x]\text{let}(y, e_1, e_2) &= \text{let}(y, [e'/x]e_1, [e'/x]e_2) \quad (y \notin \text{FV}(e'))
\end{align*}
\]

Notice that in the last clause, we may tacitly assume that \( y \) does not occur free in \( e' \) — if it does, we may simply choose another variable before commencing substitution.

### 4.4 de Bruijn Indices

The terms of first-order abstract syntax may be regarded as labelled trees. Each node is labelled with an operator \( o \), and has \( n \) children, where \( n \) is the arity of \( o \). The terms of higher-order abstract syntax may be regarded as directed graphs, in which occurrences of bound variables are replaced by edges pointing to the binding occurrence of that variable. Having done so, there is no need for equivalence classes, which simplifies the treatment of higher-order abstract syntax considerably.

However, this graphical representation is not very user friendly. To write down a piece of higher-order abstract syntax means to draw the graph in some manner, perhaps using nodes and arrows. An alternative, invented by the Dutch mathematicians N. G. de Bruijn, is to use an indexing scheme, called de Bruijn indices, to refer to the binding occurrence of a variable. A bound variable is represented by a number representing the number of enclosing binders to the binding occurrence of that variable.

An example will clarify the idea. The expression

\[
\text{let } x=10 \text{ in let } y=11 \text{ in } x+y
\]

is represented in de Bruijn notation by

\[
\text{let } 10 \text{ in let } 11 \text{ in } [2]+[1]
\]

Here \([i]\) is a de Bruijn index referring to the variable bound by \( i \)th enclosing \text{let} expression, starting from 1.
A peculiarity of de Bruijn indices is that the same variable may be referred to by several different indices, depending on the nesting depth of the occurrence. For example, the expression

\[
\text{let } x=10 \text{ in } (\text{let } y=11 \text{ in } y+x)+x
\]

is represented in de Bruijn notation by the expression

\[
\text{let } 10 \text{ in } (\text{let } 11 \text{ in } 1+2)+1.
\]

The two uses variable that we called \(x\) in the named form are represented by the occurrence of the de Bruijn index \([2]\) and the second occurrence of the de Bruijn index \([1]\). The first occurrence of \([1]\) refers not to \(x\), but to \(y\). These observations are simply a consequence of the “counting outwards” indexing scheme used in the de Bruijn representation of higher-order abstract syntax.
Part II

Effect-free Languages
Chapter 5

MinML: A MiniMaL 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 two base types, \texttt{int} and \texttt{bool}, and (partial) function types $\tau_1 \rightarrow \tau_2$. It is a call-by-value variant of Plotkin’s PCF language, a stripped-down version of ML.

5.1 Abstract Syntax

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

These categories are all defined by the following grammar:

\[
\begin{align*}
\text{Variables} & \quad x ::= \ldots \\
\text{Numerals} & \quad n ::= \ldots \\
\text{Operators} & \quad o ::= + | * | - | = | < \\
\text{Types} & \quad \tau ::= \text{int} \mid \text{bool} \mid \tau_1 \rightarrow \tau_2 \\
\text{Expressions} & \quad e ::= x \mid n \mid o(e_1, \ldots, e_n) \mid \text{true} \mid \text{false} \\
& \quad \quad \quad \quad \quad \quad \quad \text{if} \, e \text{then} \, e_1 \text{else} \, e_2 \text{fi} \\
& \quad \quad \quad \quad \quad \quad \quad \text{fun} \, f \, x : \tau_1 : \tau_2 \text{ is } e \text{ end} \\
& \quad \quad \quad \quad \quad \quad \quad \text{apply} \, (e_1, e_2) \\
\text{Programs} & \quad p ::= e
\end{align*}
\]

We do not specify precisely the sets of numbers or variables. We generally write $x$, $y$, \textit{etc.} for variables, and we write numbers in ordinary decimal notation.

\textbf{Working Draft} \hfill September 12, 2001
In keeping with the conventions discussed in Chapter 3, this grammar may be understood as specifying the abstract syntax and as providing a rough guide to the concrete syntax of the language. We will not be very specific about the concrete syntax, relying on the reader to fill in the missing details. Here are a few conventions that we will employ without further comment:

1. We will freely use parentheses to disambiguate the parsing of phrases.

2. The function space type constructor is right-associative, so that the type \( \text{int} \to \text{int} \to \text{int} \) parses as \( \text{int} \to (\text{int} \to \text{int}) \), rather than as the type \((\text{int} \to \text{int}) \to \text{int}\).

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

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

To specify the higher-order abstract syntax of MinML we need only define specify which expression-forming operators bind variables and, for those that do, specify the range of significance, or scope, of each bound variable. There is only one variable binding operator in MinML, the function expression \( \text{fun} f \ (x: \tau_1) : \tau_2 \ is \ e \ end \). In such an expression the variables \( f \) and \( x \) are both bound within the body of the function, \( e \).
Chapter 6

Static Semantics

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

6.1 Type System

Not all expressions in MinML are well-formed. For example, the expression $\text{if } \text{int } 3 \text{ then } 1 \text{ else } 0 \text{ fi}$ 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 $x \notin \text{dom}(\Gamma)$, 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:

$$\Gamma \vdash x : \Gamma(x) \quad (6.1)$$

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

$$\Gamma \vdash n : \text{int} \quad (6.2)$$

$$\Gamma \vdash \text{true} : \text{bool} \quad (6.3)$$

$$\Gamma \vdash \text{false} : \text{bool} \quad (6.4)$$

$$\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \quad \Gamma \vdash + (e_1, e_2) : \text{int} \quad (6.5)$$

$$\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \quad \Gamma \vdash * (e_1, e_2) : \text{int} \quad (6.6)$$

$$\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \quad \Gamma \vdash - (e_1, e_2) : \text{int} \quad (6.7)$$

$$\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \quad \Gamma \vdash = (e_1, e_2) : \text{bool} \quad (6.8)$$

$$\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \quad \Gamma \vdash < (e_1, e_2) : \text{bool} \quad (6.9)$$

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

$$\Gamma \vdash e : \text{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash \text{if} \; \tau \; e \; \text{then} \; e_1 \; \text{else} \; e_2 \; \text{fi} : \tau \quad (6.10)$$

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

$$\Gamma[f : \tau_1 \rightarrow \tau_2][x : \tau_1] \vdash e : \tau_2 \quad \Gamma \vdash \text{fun} \; f \; (x : \tau_1) : \tau_2 \; \text{is} \; \text{end} : \tau_1 \rightarrow \tau_2 \quad (6.11)$$
Here we require that the variables $f$ and $x$ be chosen (by suitable renaming of the function expression) so that $\{ f, x \} \cap \text{dom} (\Gamma) = \emptyset$.

$$\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_2$$

$$\Gamma \vdash \text{apply} (e_1, e_2) : \tau$$

(6.12)

This completes the typing rules for MinML.

### 6.2 Properties of Typing

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

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

**Exercise 6.1**

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

A simple — but important — consequence of syntax-directedness are the following **inversion principles** for typing. The typing rules define **sufficient** conditions for typing. For example, to show that

$$\Gamma \vdash \text{if}_{\tau} e \text{ then } e_1 \text{ else } e_2 \text{ fi} : \tau,$$

it suffices to show that $\Gamma \vdash e : \text{bool}$, $\Gamma \vdash e_1 : \tau$, and $\Gamma \vdash e_2 : \tau$, because of rule Rule 6.10. Since there is exactly one typing rule for each expression, the typing rules also express **necessary** conditions for typing. For example, if $\Gamma \vdash \text{if}_{\tau} e \text{ then } e_1 \text{ else } e_2 \text{ fi} : \tau$, 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 6.2 (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 $\Gamma \vdash \text{if } t \text{ then } e_1 \text{ else } e_2 \text{ fi} : \tau$, then $\Gamma \vdash e : \text{bool}$, $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$.
5. If $\Gamma \vdash \text{fun } f (x : \tau_1) : \tau_2 \text{ is } e \text{ end} : \tau$, 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.

Exercise 6.3
Can you think of a type system for a variant of MinML in which inversion fails? What form would such a type system have to take? Hint: think about overloading arithmetic operations.

Lemma 6.4
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 Rule 6.1) 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 Rule 6.11) We have that $e' = \text{fun } f \ (y : \tau_1) : \tau_2 \text{ is } e_2 \text{ end}$ and $\tau' = \tau_1 \to \tau_2$. 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 Rule 6.11,
\[
\Gamma[x:\tau][f:\tau_1 \to \tau_2][y:\tau_1] \vdash e_2 : \tau_2,
\]
it follows that
\[
\Gamma[f:\tau_1 \to \tau_2][y:\tau_1] \vdash [e/x]e_2 : \tau_2.
\]
Hence
\[
\Gamma \vdash \text{fun } f \ (y : \tau_1) : \tau_2 \text{ is } [e/x]e_2 \text{ end} : \tau_1 \to \tau_2,
\]
as required.

\[\blacksquare\]
Chapter 7

Dynamic Semantics

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

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

In contrast, a language-based model is one for which the semantics of a program is given entirely in terms of the constructs of the language itself. This means that no mysterious implementation-specific problems can arise, greatly enhancing portability. Moreover, language-based models support reasoning about program behavior entirely at the level of the program it-
self, avoiding the need to understand how the language is compiled on a given platform. An analogy may prove helpful in understanding the distinction. In high school you learned to calculate with polynomials. Given a polynomial, you learn to “plug in” (substitute) a value for one or more variables, then carry out some arithmetic calculations to determine the value. A polynomial may be thought of as a kind of program, for which the “plugging” and “calculating” steps constitute a dynamic semantics. You never have to leave the realm of polynomials to carry out the evaluation. In particular, there is no need to specify how a polynomial is “compiled” in order to determine its meaning as a function of its indeterminates.

Languages that admit language-based models are called abstract, or high-level, languages because they abstract away from the low-level details of the implementation. MinML is an abstract language in this sense, and we shall give it a language-based semantics. Languages that do not admit language-based models are called concrete, low-level, languages because they are closely tied to the low-level details of their implementation on actual computers. Nearly all assembly languages, and early dialects of the C language\textsuperscript{1}, are concrete languages, and are usually given a machine-based semantics.

7.1 The Dynamic Semantics of MinML

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.\textsuperscript{2}

First the instructions governing the primitive operations. We assume

\textsuperscript{1}Subsequent standardization efforts have attempted to give a more abstract semantics to C.

\textsuperscript{2}Some languages are, by contrast, non-deterministic, notably those involving concurrent interaction. We'll come back to those later.
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:

\[
+(m, n) \rightarrow m + n
\]  

(7.1)

The other primitive operations are defined similarly.

The primitive instructions for conditional expressions are as follows:

\[
\text{if true then } e_1 \text{ else } e_2 \text{ fi} \rightarrow e_1
\]  

(7.2)

\[
\text{if false then } e_1 \text{ else } e_2 \text{ fi} \rightarrow e_2
\]  

(7.3)

The primitive instruction for application is as follows:

\[
(v = \text{fun } f (x: \tau_1): \tau_2 \text{ is } e \text{ end}) \\
\text{apply}(v, v_1) \rightarrow [v, v_1/f, x]e
\]  

(7.4)

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 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:

\[
e_1 \rightarrow e'_1 \\
+(e_1, e_2) \rightarrow +(e'_1, e_2)
\]  

(7.5)

\[
e_2 \rightarrow e'_2 \\
+(v_1, e_2) \rightarrow +(v_1, e'_2)
\]  

(7.6)

The other primitive operations are handled similarly.

For the conditional, we evaluate the test expression.

\[
e \rightarrow e' \\
\text{if } e \text{ then } e_1 \text{ else } e_2 \text{ fi} \rightarrow \text{if } e' \text{ then } e_1 \text{ else } e_2 \text{ fi}
\]  

(7.7)
For applications, we first evaluate the function position; once that is complete, we evaluate the argument position.

\[
\begin{align*}
e_1 &\mapsto e'_1 \\
\text{apply}(e_1, e_2) &\mapsto \text{apply}(e'_1, e_2)
\end{align*}
\]  
(7.8)

\[
\begin{align*}
e_2 &\mapsto e'_2 \\
\text{apply}(v_1, e_2) &\mapsto \text{apply}(v_1, e'_2)
\end{align*}
\]  
(7.9)

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

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

\[
\begin{align*}
e &\rightarrow^* e
\end{align*}
\]  
(7.10)

\[
\begin{align*}
e &\rightarrow e' \\
 e' &\rightarrow^* e'' \\
 e &\rightarrow^* e''
\end{align*}
\]  
(7.11)

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 \textit{Kleene closure}, or \textit{reflexive-transitive closure}, of the relation \( \rightarrow \).

### 7.2 Properties of the Dynamic Semantics

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

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

\textbf{Lemma 7.1}

\textit{For every closed expression \( e \), there exists at most one \( e' \) such that \( e \mapsto e' \). In other words, the relation \( \mapsto \) is a partial function.}
7.2 Properties of the Dynamic Semantics

**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 7.2**

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.
Chapter 8

Type Safety

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.

8.1 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 \texttt{apply (3, 4)} — it is easy to check that no transition rule applies. Fortunately, this expression is also ill-typed! Progress tells us that this will always be the case.

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

8.2 Type Safety of MinML

Theorem 8.1 (Preservation)

\( e : \tau \) 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 Rule 7.1) Here \( e = +(m, n), \tau = \texttt{int} \), and \( e' = m + n \). Clearly \( e' : \texttt{int} \), as required. The other primitive operations are handled similarly.

(Rule Rule 7.2) Here \( e = \texttt{if \_ true then} e_1 \texttt{else} e_2 \texttt{fi} \) and \( e' = e_1 \). Since \( e : \tau \), by inversion \( e_1 : \tau \), as required.

(Rule Rule 7.3) Here \( e = \texttt{if \_ false then} e_1 \texttt{else} e_2 \texttt{fi} \) and \( e' = e_2 \). Since \( e : \tau \), by inversion \( e_2 : \tau \), as required.
8.2 Type Safety of MinML

(Rule Rule 7.4) Here \(e = \text{apply}(v_1, v_2)\), where \(v_1 = \text{fun} f \ (x: \tau_2): \tau \) is \(e_2\) end, and \(e' = [v_1, v_2/f, x]e_2\). By inversion applied to \(e\), we have \(v_1 : \tau_2 \rightarrow \tau\) and \(v_2 : \tau_2\). By inversion applied to \(v_1\), we have \([f: \tau_2 \rightarrow \tau][x: \tau_2] \vdash e_2 : \tau\). Therefore, by substitution we have \([v_1, v_2/f, x]e_2 : \tau\), as required.

(Rule Rule 7.5) 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 Rule 7.6) 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 Rule 7.7) Here \(e = \text{if} \tau e_1 \text{then} e_2 \text{else} e_3 \text{fi}\) and \(e' = \text{if} \tau e'_1 \text{then} e_2 \text{else} e_3 \text{fi}\). 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 Rule 7.8) 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 Rule 7.9) 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 8.2 (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 \(v = \text{fun} f \ (x: \tau_1) : \tau_2\) is \(e\) end for some \(f, x\), and \(e\).

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

WORKING DRAFT  SEPTEMBER 12, 2001
Exercise 8.3
Give a proof of the canonical forms lemma.

Theorem 8.4 (Progress)
If $e : \tau$, then either $e$ is a value, or there exists $e'$ such that $e \mapsto e'$.

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

(Rule Rule 6.1) Cannot occur, since $e$ is closed.

(Rules Rule 6.2, Rule 6.3, Rule 6.4, Rule 6.11) In each case $e$ is a value, which completes the proof.

(Rule Rule 6.5) 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 Rule 6.10) Here $e = \text{if}_\tau e_1 \text{ then } e_2 \text{ else } e_3 \text{ fi}$, 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 $e' = \text{if}_\tau e'_1 \text{ then } e_2 \text{ else } e_3 \text{ fi}$, by rule Rule 7.7.

(Rule Rule 6.12) Here $e = \text{apply} (e_1, e_2)$, with $e_1 : \tau_2 \to \tau$ 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 Rule 7.8, 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 Rule 7.4.

■
Theorem 8.5 (Safety)
If $e$ is well-typed, then evaluation of $e$ can only terminate with a value of the same type. In particular, evaluation cannot "abort" due to an ill-defined state.
Part III

Evaluation Semantics
Chapter 9

Evaluation Semantics

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

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

9.1 Evaluation Semantics

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

\[
\frac{}{n \Downarrow n} \quad (9.1)
\]

\[
\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{+(e_1, e_2) \Downarrow n_1 + n_2} \quad (9.2)
\]
Evaluation Semantics

(and similarly for the other primitive operations).

\[
\begin{align*}
\text{true} & \Downarrow \text{true} & \text{false} & \Downarrow \text{false} \\
& \Downarrow v_1 & & \Downarrow v_2 \\
\text{if } e \text{ then } e_1 \text{ else } e_2 \text{ fi} & \Downarrow v \\
\end{align*}
\]

\[
\begin{align*}
& \Downarrow v_1 & & \Downarrow v_2 \\
\text{if } e \text{ then } e_1 \text{ else } e_2 \text{ fi} & \Downarrow v \\
\end{align*}
\]

\[
\begin{align*}
\text{fun } f \colon \tau_1 : \tau_2 \text{ is } e \text{ end} & \Downarrow \text{fun } f \colon \tau_1 : \tau_2 \text{ is } e \text{ end} \\
& \Downarrow v_1 & & \Downarrow v_2 \\
\text{apply} (e_1, e_2) & \Downarrow v \\
\end{align*}
\]

\[
\begin{align*}
& \Downarrow v_1 & & \Downarrow v_2 \\
\text{apply} (e_1, e_2) & \Downarrow v \\
\end{align*}
\]

(9.3)

(9.4)

(9.5)

(9.6)

(9.7)

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

### 9.2 Correspondences

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

**Theorem 9.1**

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 \( v_1 = \text{fun } f \colon \tau_1 : \tau_2 \text{ is } e \text{ end} \), \( e_2 \Downarrow v_2 \), 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 \( v_1 = \text{fun} f (x: \tau_1) : \tau_2 \text{ is } e \text{ end} \), 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 \( e_1 \rightarrow e_1' \). 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 \( e \rightarrow^* v \), then \( e \Downarrow v \). The base case, \( v \rightarrow^* v \), follows from the fact that \( v \Downarrow v \). Now suppose that \( e \rightarrow e' \rightarrow^* v \). By induction \( e' \Downarrow v \), and hence \( e \Downarrow v \) by what we have just proved.

9.3 Type Safety For Evaluation

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

Exercise 9.2
Prove type preservation for evaluation semantics by induction on the rules defining evaluation.

A standard technique for expressing progress in the framework of evaluation semantics is to reduce progress to preservation for an extended evaluation relation. Specifically, we extend the evaluation relation with rules governing “stuck” expressions stating that the value of such an expression is a special token wrong that is, by definition, ill-typed. Since preservation ensures that the value of a well-typed expression is well-typed, the value cannot be wrong. This leads to the slogan “well-typed programs cannot go wrong.”
More precisely, we introduce a special syntactic category of *answers* defined by the following grammar:

\[
\text{Answers} \quad a \quad ::= \quad \text{ok}(v) \mid \text{wrong}
\]

Here \(v\) is a syntactic value, as defined earlier.

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

\[
\frac{\vdash v : \tau}{\vdash \text{ok}(v) : \tau}
\]  \hspace{1cm} (9.8)

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

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

\[
\frac{}{n \Downarrow \text{ok}(n)}
\]  \hspace{1cm} (9.9)

\[
\frac{e_1 \Downarrow \text{ok}(n_1) \quad e_2 \Downarrow \text{ok}(n_2)}{(e_1, e_2) \Downarrow \text{ok}(n_1 + n_2)}
\]  \hspace{1cm} (9.10)

\[
\frac{e_1 \Downarrow \text{wrong}}{(e_1, e_2) \Downarrow \text{wrong}}
\]  \hspace{1cm} (9.11)

\[
\frac{e_1 \Downarrow \text{ok}(v_1) \quad e_2 \Downarrow \text{wrong}}{(e_1, e_2) \Downarrow \text{wrong}}
\]  \hspace{1cm} (9.12)

\[
\frac{e_1 \Downarrow \text{ok}(v_1) \quad v_1 \neq n_1}{(e_1, e_2) \Downarrow \text{wrong}}
\]  \hspace{1cm} (9.13)

\[
\frac{e_1 \Downarrow \text{ok}(n_1) \quad e_2 \Downarrow \text{ok}(v_2) \quad v_2 \neq n_2}{(e_1, e_2) \Downarrow \text{wrong}}
\]  \hspace{1cm} (9.14)

(The other primitive operations are handled similarly.)

\[
\frac{}{\text{true} \Downarrow \text{ok}(\text{true})}
\]  \hspace{1cm} (9.15)

\[
\frac{}{\text{false} \Downarrow \text{ok}(\text{false})}
\]  \hspace{1cm} (9.16)
The changes we have made are these:

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

2. Introduce rules that propagate \( \text{wrong} \) from sub-expressions. Once evaluation goes \( \text{wrong} \) it stays \( \text{wrong} \)!

Why does this work? We have the following correspondence:
Theorem 9.3

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

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

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

Exercise 9.4

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

It is important that you understand how the proof of progress is embedded (contrapositively) in the proof of preservation for the extended semantics.
Chapter 10

Cost Semantics

A structured operational semantics provides a natural notion of time complexity for programs, namely the number of steps required to reach a final state. For example, if $f$ is the (obvious) MinML program to compute the factorial function, then $f(n)$ evaluates to $n!$ in $O(n)$ steps, so we may say that the time complexity of $f$ is $O(n)$.

An evaluation semantics, on the other hand, does not provide such a direct notion of complexity. Since the individual steps required to complete an evaluation are suppressed, we cannot “read off” the number of steps required to evaluate to a value. Instead we must augment the evaluation relation with a cost measure, resulting in a cost semantics.

The notion of a cost semantics is rather general. In Chapter 27 we will consider a cost semantics corresponding to execution on an idealized parallel machine. One may also consider the space required by a computation, in either a sequential or parallel setting. In this chapter we restrict attention to the time complexity of execution.

10.1 A Cost Semantics for MinML

We will introduce a cost semantics for MinML that reflects the number of steps required to complete evaluation according to the structured operational semantics given in Chapter 7.

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

\[ n \Downarrow^0 n \quad (10.1) \]
\[
\begin{align*}
\frac{e_1 \downarrow^{k_1} v_1 \quad e_2 \downarrow^{k_2} v_2}{(e_1, e_2) \downarrow^{k_1+k_2+1} \ n_1 + n_2}
\end{align*}
\]  
(10.2)

(and similarly for the other primitive operations).

\[
\begin{align*}
\text{true} \downarrow^{0} \text{true} & \quad \text{false} \downarrow^{0} \text{false} \\
\end{align*}
\]  
(10.3)

\[
\begin{align*}
e \downarrow^{k} \text{true} & \quad e_1 \downarrow^{k_1} v \\
\text{if} \ e \text{then} e_1 \text{else} e_2 \text{fi} \downarrow^{k+k_1+1} v \\
\end{align*}
\]  
(10.4)

\[
\begin{align*}
e \downarrow^{k} \text{false} & \quad e_2 \downarrow^{k_2} v \\
\text{if} \ e \text{then} e_1 \text{else} e_2 \text{fi} \downarrow^{k+k_2+1} v \\
\end{align*}
\]  
(10.5)

\[
\begin{align*}
\text{fun} \ f \ (x: \tau_1) : \tau_2 \ is \ e \ end \ \downarrow^{0} \ \text{fun} \ f \ (x: \tau_1) : \tau_2 \ is \ e \ end
\end{align*}
\]  
(10.6)

\[
\begin{align*}
e_1 \downarrow^{k_1} v_1 & \quad e_2 \downarrow^{k_2} v_2 \quad [v_1, v_2/f, x] e \downarrow^{k} v \\
\text{apply} \ (e_1, e_2) \downarrow^{k_1+k_2+k+1} v
\end{align*}
\]  
(10.7)

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

This completes the definition of the cost semantics for MinML.

### 10.2 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 \( e \mapsto^{k} e' \) by induction on \( k \) as follows. If \( k = 0 \), \( e \mapsto^{0} e' \) iff \( e = e' \). For \( k = k' + 1 \), \( e \mapsto^{k} e' \) holds iff \( e \mapsto^{e''} \mapsto^{k'} e' \).

**Theorem 10.1**

*For any closed expression \( e \) and closed value \( v \) of the same type, \( e \downarrow^{k} v \) iff \( e \mapsto^{k} v \).*

**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
10.2 Relating Cost Semantics to Transition Semantics

1. \( e_1 \Downarrow v_1 \),
2. \( e_2 \Downarrow v_2 \),
3. \( v_1 = \text{fun } f (x : \tau_1) : \tau_2 \text{ is } e \text{ end} \),
4. \( [v_1, v_2/f, x]e \Downarrow v \).

By induction we have

1. \( e_1 \mapsto^{k_1} v_1 \),
2. \( e_2 \mapsto^{k_2} v_2 \),
3. \( [v_1, v_2/f, x]e \mapsto^{k} v \),

and hence

\[
\begin{align*}
\begin{array}{l}
e_1 (e_2) \mapsto^{k_1} v_1 (e_2) \\
\mapsto^{k_2} v_1 (v_2) \\
\mapsto [v_1, v_2/f, x]e \\
\mapsto^{k} v
\end{array}
\end{align*}
\]

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 rules we may check that \( v \Downarrow^0 v \) for every value \( v \). For \( k = k' + 1 \), we must show that if \( e \mapsto e' \) 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 \( e = e_1 (e_2) \mapsto e'_1 (e_2) = e' \), with \( e_1 \mapsto e'_1 \). 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. \( v_1 = \text{fun } f (x : \tau_1) : \tau_2 \text{ is } e \text{ end} \),
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. ■
Part IV

Data Structures and Dynamic Typing
Chapter 11

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 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.

### 11.1 Products

The first-order abstract syntax associated with nullary and binary product types is given by the following grammar:

- **Types**
  \[ \tau :: = \text{unit} | \tau_1 \times \tau_2 \]

- **Expressions**
  \[ e :: = () | \text{check } e_1 \text{ is } () \text{ in } e_2 \text{ end } | (e_1, e_2) | \text{split } e_1 \text{ as } (x, y) \text{ in } e_2 \text{ end} \]

- **Values**
  \[ v :: = () | (v_1, v_2) \]
The higher-order abstract syntax is given by stipulating that in the expression `split e_1 as (x, y) in e_2 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:

\[
\frac{}{Γ ⊢ (): \text{unit}} \quad (11.1)
\]

\[
\frac{Γ ⊢ e_1: \text{unit} \quad Γ ⊢ e_2: τ_2}{Γ ⊢ \text{check } e_1 \text{ is } () \text{ in } e_2 \text{ end}: τ_2} \quad (11.2)
\]

\[
\frac{Γ ⊢ e_1: τ_1 \quad Γ ⊢ e_2: τ_2}{Γ ⊢ (e_1, e_2): τ_1 * τ_2} \quad (11.3)
\]

\[
\frac{Γ ⊢ e_1: τ_1 * τ_2 \quad Γ, x: τ_1, y: τ_2 ⊢ e_2: τ}{Γ ⊢ \text{split } e_1 \text{ as } (x, y) \text{ in } e_2 \text{ end}: τ} \quad (11.4)
\]

The dynamic semantics is given by these instructions and search rules:

\[
\text{check } (\text{)} \text{ is } () \text{ in } e \text{ end} \mapsto e \quad (11.5)
\]

\[
\frac{e_1 \mapsto e'_1}{\text{check } e_1 \text{ is } () \text{ in } e_2 \text{ end} \mapsto \text{check } e'_1 \text{ is } () \text{ in } e_2 \text{ end}} \quad (11.6)
\]

\[
\frac{e_1 \mapsto e'_1}{(e_1, e_2) \mapsto (e'_1, e_2)} \quad (11.7)
\]

\[
\frac{e_2 \mapsto e'_2}{(v_1, e_2) \mapsto (v_1, e'_2)} \quad (11.8)
\]

\[
\frac{\text{split } (v_1, v_2) \text{ as } (x, y) \text{ in } e \text{ end} \mapsto [v_1, v_2/x \, y]e}{\quad (11.9)}
\]

\[
\frac{e_1 \mapsto e'_1}{\text{split } e_1 \text{ as } (x, y) \text{ in } e_2 \text{ end} \mapsto \text{split } e'_1 \text{ as } (x, y) \text{ in } e_2 \text{ end}} \quad (11.10)
\]

**Exercise 11.1**

*State and prove the soundness of this extension to MinML.*
Exercise 11.2
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 11.3
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 11.4
Another variation is to considered labelled products in which the components are accessed directly by referring to their labels (in a manner similar to \texttt{C struct}'s). Formalize this notion.

11.2 Sums

The first-order abstract syntax of nullary and binary sums is given by the following grammar:

\[
\begin{align*}
\text{Types} & \quad \tau ::= \text{void} | \tau_1 + \tau_2 \\
\text{Expressions} & \quad e ::= \text{inl} \tau_1 + \tau_2 (e_1) | \text{inr} \tau_1 + \tau_2 (e_2) | \\
& \quad \text{case}_\tau e_0 \text{ of inl} (x : \tau_1) => e_1 | \text{inr} (y : \tau_2) => e_2 \text{ end} \\
\text{Values} & \quad v ::= \text{inl} \tau_1 + \tau_2 (v_1) | \text{inr} \tau_1 + \tau_2 (v_2)
\end{align*}
\]

The higher-order abstract syntax is given by noting that in the expression \(\text{case}_\tau e_0 \text{ of inl} (x : \tau_1) => e_1 | \text{inr} (y : \tau_2) => e_2 \text{ end}\), 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:

\[
\begin{align*}
\Gamma \vdash e_1 : \tau_1 & \quad \Gamma \vdash \text{inl} \tau_1 + \tau_2 (e_1) : \tau_1 + \tau_2 \tag{11.11} \\
\Gamma \vdash e_2 : \tau_2 & \quad \Gamma \vdash \text{inl} \tau_1 + \tau_2 (e_2) : \tau_1 + \tau_2 \tag{11.12} \\
\Gamma \vdash e_0 : \tau_1 + \tau_2 & \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau \\
\Gamma \vdash \text{case}_\tau e_0 \text{ of inl} (x_1 : \tau_1) => e_1 | \text{inr} (x_2 : \tau_2) => e_2 \text{ end} : \tau \tag{11.13}
\end{align*}
\]
The evaluation rules are as follows:

\[ e \mapsto e' \]
\[ \text{inl}_{\tau_1 + \tau_2} (e) \mapsto \text{inl}_{\tau_1 + \tau_2} (e') \quad (11.14) \]
\[ e \mapsto e' \]
\[ \text{inr}_{\tau_1 + \tau_2} (e) \mapsto \text{inr}_{\tau_1 + \tau_2} (e') \quad (11.15) \]

\[ \text{case} \tau \text{ inl}_{\tau_1 + \tau_2} (v) \text{ of inl} (x_1 : \tau_1) =\mapsto e_1 | \text{inr} (x_2 : \tau_2) =\mapsto e_2 \text{ end} \mapsto [v/x_1]e_1 \quad (11.16) \]

\[ \text{case} \tau \text{ inr}_{\tau_1 + \tau_2} (v) \text{ of inl} (x_1 : \tau_1) =\mapsto e_1 | \text{inr} (x_2 : \tau_2) =\mapsto e_2 \text{ end} \mapsto [v/x_2]e_2 \quad (11.17) \]

**Exercise 11.5**

*State and prove the soundness of this extension.*

**Exercise 11.6**

*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.*

### 11.3 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 *value constructors* for the type \( \text{ilist} \) of integer lists. We may program with lists using a form of case analysis, written

\[ \text{listcase} \ e \text{ of nil} =\mapsto e_1 | \text{cons} (x, y) =\mapsto e_2 \text{ end}, \]

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 11.7
Give a formal definition of the type 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 ilist and unit+(int*ilist) are isomorphic, meaning that there is a one-to-one correspondence between values of type 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 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 11.8
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:

\[
\begin{align*}
\text{Types} & \quad \tau ::= t | \text{rectis}\tau \\
\text{Expressions} & \quad e ::= \text{roll}(e) | \text{unroll}(e) \\
\text{Values} & \quad v ::= \text{roll}(v)
\end{align*}
\]

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 \( \text{rectisunit+}(\text{int}^*t) \) is the recursive type of lists of integers. It is isomorphic to its unrolling, the type

\[
\text{unit+}(\text{int}^*\text{rectisunit+}(\text{int}^t)).
\]
If we write \texttt{ilist} for \texttt{rectisunit+(int*)}, then we obtain the isomorphism described informally above.

The abstract "pointers" witnessing the isomorphism are written \texttt{roll(e)}, which "allocates" a pointer to (the value of) \( e \), and \texttt{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:

\[
\Gamma \vdash e : \texttt{[rectis\tau/t]\tau} \\
\Gamma \vdash \texttt{roll}(e) : \texttt{rectis}\tau 
\]  
(11.18)

\[
\Gamma \vdash e : \texttt{rectis}\tau \\
\Gamma \vdash \texttt{unroll}(e) : \texttt{[rectis\tau/t]\tau} 
\]  
(11.19)

These primitive operations move back and forth between a recursive type and its unrolling.

The dynamic semantics is given by the following rules:

\[
\texttt{unroll(roll(v))} \mapsto v 
\]  
(11.20)

\[
\begin{align*}
\Gamma \vdash e & \mapsto e' \\
\texttt{unroll}(e) & \mapsto \texttt{unroll}(e')
\end{align*}
\]  
(11.21)

\[
\begin{align*}
\Gamma \vdash e & \mapsto e' \\
\texttt{roll}(e) & \mapsto \texttt{roll}(e')
\end{align*}
\]  
(11.22)

\textbf{Exercise 11.9}

State and prove the soundness of this extension of \textit{MinML}.

\textbf{Exercise 11.10}

Consider the definition of the type \texttt{ilist} as a recursive type given above. Give definitions of \texttt{nil}, \texttt{cons}, and \texttt{listcase} in terms of the operations on recursive types, sums, and products.
Chapter 12

Dynamic Typing

The formalization of type safety given in Chapter 8 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.

The first issue we addressed was the possibility of states such as $3 \div 0$, which are well-typed, yet stuck, in apparent violation of the progress theorem. We discussed two possible moves to handle such a situation. One is to enrich the type system to ensure that such an expression is ill-typed. We argued, however, that this takes us considerably beyond the capabilities of current type systems for practical programming languages. This left only the possibility to ensure that such states are not stuck, but rather make a transition to a designated “error” state that is propagated throughout the program. To do so we introduced the notion of a checked error, one which is explicitly detected and signalled during execution, as opposed to an unchecked error, such as a type error, which is ruled out by the static type discipline.

A second issue is the observation that, in the presence of a progress theorem, while all stuck states are ill-typed, not all ill-typed states are stuck! For example, the expression

$$(\text{if } \text{true then } 7 \text{ else } "7") + 1$$

is ill-typed, because the branches of the conditional do not have the same type, yet executes successfully without getting stuck or incurring a checked
error. While this example is rather contrived — there is no need to perform a conditional test on a boolean constant — such states arise during the course of execution. For example, the initial state might have been the expression

\[(\text{if } e \text{ then } 7 \text{ else } "7") + 1\]

for some complex expression \(e\) whose outcome cannot easily be predicted prior to execution. Such an expression is therefore ruled out in the MinML type system (and many like it), even though it will not “get stuck” so long as \(e\) evaluates to \text{true}.

A closely related issue is the treatment of heterogeneous data structures, those whose “elements” may be of many different types. For example, in ML all of the elements of a list must have the same type, yet we might wish to form the list

\[[\text{true}, 1, 3.4, \text{fn } x\Rightarrow x]\]

and would be precluded from doing so. Yet, it is argued, there is nothing inherently wrong with such a list, particularly since the list constructors are insensitive to the type of elements of a list — they simply allocate a new node in the heap, and initialize it appropriately.

These examples suggest that our formalization of safety in terms of static typing — the assignment of a type to an expression without executing it — rules out programs of practical interest. The alternative, called dynamic typing, eschews static type checking in favor of run-time checks that catch errors as late, rather than as early, as possible. Thus, a conditional such as the one given above might well execute without error in all situations of interest, and a program might manipulate the values of a heterogeneous list in a manner entirely consistent with their types (e.g., by adhering to the invariant that the odd-numbered elements are integers, and the even-number elements are strings, and acting accordingly).

In this chapter we will investigate a dynamically typed variant of MinML to show how and why type safety may be ensured in the absence of a static semantics. While this may seem, at first, to undermine our framework for defining and analyzing programming languages, we will demonstrate that in fact it merely reinforces the basic principles we have developed so far. Static and dynamic typing are often presented as if they are in opposition to one another, two design choices yielding safe languages in different ways. However, our analysis will reveal that, far from being opposed, dynamic

---

1This is debatable, but bear with me.
typing is a special case of static typing. 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.

### 12.1 Dynamic Typing

The fundamental idea in a dynamically typed language is to render type errors as checked, rather than unchecked, errors. That is, the dynamic semantics is instrumented with rules that explicitly check for stuck states that correspond to ill-typed programs. 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 is possible from such a state, namely by a transition to an error state. Thus type errors are on a par with all other checked errors, such as division by zero or arithmetic overflow.

The idea is easily illustrated by example. Consider the rules for function application in MinML given in Chapter 6, which we repeat here for convenience:

\[
\begin{align*}
& v \text{ value } v_1 \text{ value } (v = \text{fun } f (x: \tau_1): \tau_2 \text{ is } e \text{ end}) \\
& \quad \text{apply}(v, v_1) \mapsto [v, v_1/f, x]e \\
& \quad e_1 \mapsto e'_1 \\
& \quad \text{apply}(e_1, e_2) \mapsto \text{apply}(e'_1, e_2)
\end{align*}
\]

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

In addition to these rules, which govern the well-typed case, we add the following rules governing the ill-typed case:

\[
\begin{align*}
& v \text{ value } v_1 \text{ value } (v \neq \text{fun } f (x: \tau_1): \tau_2 \text{ is } e \text{ end}) \\
& \quad \text{apply}(v, v_1) \mapsto \text{error} \\
& \quad \text{apply}(\text{error}, e_2) \mapsto \text{error} \\
& \quad v_1 \text{ value } \mapsto \text{error} \\
& \quad \text{apply}(v_1, \text{error}) \mapsto \text{error}
\end{align*}
\]
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 \textit{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.\footnote{We may then simplify the language by dropping type declarations on variables and functions, since these are no longer of any use.} Having done 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 \texttt{error}, ensuring that there are no “stuck” states. Thus, the dynamically typed variant of \textit{MinML} is type safe in same sense as for the statically typed variant. The meaning of safety does not change, rather only the means by which it is achieved.

\subsection*{12.2 Implementing Dynamic Typing}

Since both the statically typed and the dynamically typed variants of \textit{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? These are significant methodological questions that we will not tackle here.\footnote{In Chapter 23 we will see that static type systems support modular programming far better than do dynamic type systems.} Instead, we will show that the apparent dichotomy between static and dynamic typing is illusory by showing that dynamic typing is a \textit{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 \textit{given} situation), rather than a doctrinal debate about the design of a programming language (which to use in \textit{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 \textit{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 problem is that the “natural” representations
of data values 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. Given an arbitrary word, it is not possible, in general, to determine whether it represents a number or a code pointer.

To support run-time type checking, we must adulterate our data representations to ensure that the required checks are possible. 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. We must also be able to mark a “raw” data value with the appropriate information to identify what sort of value it represents. There are many schemes for achieving this, but they all amount to attaching a tag to a “raw” value that uniquely identifies whether the value is an integer, boolean, or function (or, in richer languages, other form of data). Run-time type checking then amounts to checking and stripping tags from data during computation.\footnote{The terminology “run-time type checking” is therefore misleading, because we are not checking types at run-time, but rather tags. As we will see shortly, tags indicate only the outermost form of a value, and therefore carry less information than types.}

To fix ideas, let us make explicit the role of tags in a dynamically typed language. We will illustrate the main ideas by example.

First, we extend the language with a new class of tagged values, given by the following grammar:

\[
v :: \text{Integer} (n) | \text{Boolean} (\text{true}) | \text{Boolean} (\text{false}) | \text{Function} (\text{fun} f (x : \tau_1) : \tau_2 \text{ is } \text{end})
\]

We do not include expressions of the form \text{Integer} (e), \text{Boolean} (e), or \text{Function} (e) for a general expression e. Rather, tagged values are to be thought of as “molecules” consisting of two “atoms”, a tag and a raw value.

Second, we introduce tag checking rules that determine whether or not a value has a given tag, and, if so, extracts the underlying value. We give rules for deriving judgements of the form \text{v is fun } f, which checks that \text{v} has the form \text{Function} (f), and extracts \text{f} from it if so, and for judgements of the form \text{v isnt fun}, that checks that \text{v} does not have the form \text{Function} (f).

\[
\begin{align*}
\text{Function (v) is fun } v \\
\text{Integer (x) isnt fun} & \quad \text{Boolean (x) isnt fun}
\end{align*}
\]
There are similar judgements and rules for identifying integers and booleans.

Using these, we may re-formulate the rules for performing a function call as follows:

\[
\frac{v_1 \text{ value } v \text{ is fun } f (x : \tau_1) : \tau_2 \text{ is end}}{\text{apply} (v, v_1) \mapsto [v, v_1/f, x]e}
\]

\[
\frac{v \text{ value } v \text{ isnt fun}}{\text{apply} (v, v_1) \mapsto \text{error}}
\]

These rules rely on the tag-checking judgements to determine whether a run-time type fault has occurred.

### 12.3 Dynamic Typing as Static Typing

The commitment to dynamic typing has an unfortunate consequence. The cost of maintaining and checking tags is imposed, even if the program is (statically) well-typed. Since statically well-typed programs satisfy preservation and progress, we can guarantee a priori that tags are not necessary. In this sense dynamically typed languages violate the pay-as-you-go principle of language design: you pay the cost of dynamic typing, even if you don’t take advantage of it in your program!

Ideally, we would like to pay the overhead of dynamic typing only to the extent that we actually use it. For example, if we only use homogeneous lists of values, then no type tags are required to determine the type of its elements. However, if we do wish to use heterogeneous lists, then tagging is necessary to ensure safety, and we would expect to have to pay additional overhead for the privilege. The choice should be in the hands of the programmer, not the language designer.

The key to achieving this is a simple, but powerful, observation: dynamic typing is but a mode of use of a (sufficiently rich) static type system. All that is necessary is to introduce a type of tagged data into a statically-typed language. Dynamic typing corresponds to manipulation of values of the type of tagged values. Whenever dynamic typing is required, simply use the type of tagged data to ensure safety. Not only does this provide the means to perform dynamic typing within a statically typed language, it does so in a manner that is 100% faithful to its “direct” implementation in a dynamically typed language. All we are doing is to expose the underlying infrastructure of dynamic type checking.

Here is how this can be done within ML. First, we introduce the type of tagged data values:
12.3 Dynamic Typing as Static Typing

(* The type of tagged values. *)
datatype tagged =
    Integer of int |
    Boolean of bool |
    Function 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.

exception TypeError

fun checked_add (m:tagged, n:tagged):tagged =
    case (m, n) of
        (Integer a, Integer b) => Integer (a+b)
    | (_, _) => raise TypeError

fun checked_apply (f:tagged, a:tagged):tagged =
    case f of
        Function 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 type tagged list.

val het_list : tagged list =
    [Integer 1, Boolean true, Function (fn x => x)]
val f : tagged = hd(tl(tl het_list))
val x : tagged = checked_apply (f, Integer 5)

The tags on the elements serve to identify what sort of element it is: an integer, a boolean, or a function.

It is also possible to add the type tagged to MinML as a new primitive type. However, in Chapter 11 we will see how to encode ML-style data types, including the type tagged defined above. Therefore we will not explicitly formalize tagging as an extension to MinML.
Part V

Control Flow
Chapter 13

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 \textit{start state}, executing instructions one-by-one until no more instructions remains; the result of the computation is the \textit{final state}. Thus an abstract machine is essentially a \textit{transition system} between states of that machine.

According to this definition the dynamic semantics of MinML is an abstract machine, the \textit{M} machine. The states of the \textit{M} machine are closed MinML expressions \(e\), and the transitions are given by the one-step evaluation relation \(e \rightarrow_{M} e'\) 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 \textit{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.

\section{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 \textit{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 \textit{C machine}, to remind us that it is defined to capture the idea of control flow in a computation.

The states of the \textit{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:

\begin{align}
\text{stack} & \\
\text{frame } & \downarrow k \text{ stack} \\
& \frac{f \text{ frame } k \text{ stack}}{f \uparrow k \text{ stack}}
\end{align}

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.
13.1 Control Flow

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

\[\begin{align*}
\text{frame} & \quad \text{val} \quad \text{frame} \\
+ & (\square, e_2) \quad v_1 \quad + & (v_1, \square)
\end{align*}\] (13.3, 13.4)

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

\[\begin{align*}
\text{frame} & \quad \text{val} \quad \text{frame} \\
\text{if} \ & \square \text{then} \ & e_1 \ & \text{else} \ & e_2 \ & \text{fi} \\
\text{apply} & (\square, e_2) \quad v_1 \quad \text{apply} & (v_1, \square)
\end{align*}\] (13.5, 13.6, 13.7)

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.

\[ (k, +(e_1, e_2)) \mapsto_C (+((\square, e_2) \triangleright k, e_1) \quad (13.8) \]

\[ (+((\square, e_2) \triangleright k, v_1) \mapsto_C (+((v_1, \square) \triangleright k, e_2) \quad (13.9) \]

\[ (+((n_1, \square) \triangleright k, n_2) \mapsto_C (k, n_1 + n_2) \quad (13.10) \]

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

Next, we consider the rules for the conditional.

\[ (k, \text{if } e \text{ then } e_1 \text{ else } e_2 \text{ fi}) \mapsto_C (\text{if } \square \text{ then } e_1 \text{ else } e_2 \text{ fi} \triangleright k, e) \quad (13.11) \]

\[ (\text{if } \square \text{ then } e_1 \text{ else } e_2 \text{ fi} \triangleright k, \text{true}) \mapsto_C (k, e_1) \quad (13.12) \]
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.

\[(k, \text{apply}(e_1, e_2)) \rightarrow_C (\text{apply}(\Box, e_2) \triangleright k, e_1)\]  \hspace{1cm} (13.14)

\[(\text{apply}(\Box, e_2) \triangleright k, v_1) \rightarrow_C (\text{apply}(v_1, \Box) \triangleright k, e_2)\]  \hspace{1cm} (13.15)

\[(\text{apply}(v_1, \Box) \triangleright k, v_2) \rightarrow_C (k, [v_1, v_2/f, x]e)\]  \hspace{1cm} (13.16)

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 \((\bullet, v)\) consisting of the empty stack (no further work to do) and a value \(v\).

The rules defining the \(C\) machine have no premises — they are all simple transitions, without any hypotheses. We’ve made explicit the management of the “subgoals” required for evaluating expressions using the \(M\) 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 \text{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 \text{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:

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

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

**Theorem 13.1**

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

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

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.

---

3Half 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

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

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

(b) Consider the transition

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

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

(c) Consider the transition

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

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

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

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 \(k' @ \text{apply}(v_1, v_2)\) 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) \(e = \text{apply}(v_1, v_2) \mapsto_M [v_1, v_2/f, x]e_2 = e'\), where the value \(v_1 = \text{fun} f (x: \tau_2) : \tau \text{ is e end}\). Suppose that \((k, e') \mapsto_C^* (\bullet, v)\). By the definition of the \(C\) machine transition relation,

\[
(k, e) \mapsto_C (\text{apply}(\Box, v_2) \triangleright k, v_1) \mapsto_C (\text{apply}(v_1, \Box) \triangleright k, v_2) \mapsto_C (k', e')
\]

From this, the result follows immediately.

(b) \(e = \text{apply}(e_1, e_2) \mapsto_M \text{apply}(e'_1, e_2) = e'\), where \(e_1 \mapsto_M e'_1\). Suppose that \((k, e') \mapsto_C^* (\bullet, v)\). Since \(e' = \text{apply}(e'_1, e_2)\), and
since the C machine is deterministic, this transition sequence must have the form

\[(k, e') = (k, apply(e_1', e_2)) \rightarrow_C (apply(\Box, e_2) \triangleright k, e_1') \rightarrow^*_C (\bullet, v)\]

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

\[(apply(\Box, e_2) \triangleright k, e_1) \rightarrow^*_C (\bullet, v).\]

Now since

\[(k, e) = (k, apply(e_1, e_2)) \rightarrow_C (apply(\Box, e_2) \triangleright k, e_1)\]

the result follows immediately.

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

\[(k, e') = (k, apply(v_1, e_2')) \rightarrow_C (apply(v_1, \Box) \triangleright k, e_2') \rightarrow^*_C (\bullet, v)\]

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

\[(apply(v_1, \Box) \triangleright k, e_2) \rightarrow^*_C (\bullet, v).\]

Now since

\[(k, e) = (k, apply(v_1, e_2)) \rightarrow_C (apply(v_1, \Box) \triangleright k, e_1)\]

the result follows immediately.

\[\square\]

Exercise 13.2

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

Corollary 13.3

1. If \((k, e) \rightarrow^*_C (\bullet, v),\) then \(k \mathbin{@} e \rightarrow^* M v.\) 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 \((\bullet, e) \rightarrow^*_C (\bullet, v).\)
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 \( k @ e = v \rightarrow^* v \), as required.

   (b) Reverse execution. Here we have \((k', e') \rightarrow^* (k, e) \rightarrow^*_C (\bullet, v)\). By induction \( k @ e \rightarrow^*_M v \), and by Theorem 13.1 \( k' @ e' \rightarrow^*_M k @ e \), so \( k' @ e' \rightarrow^*_M v \).

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 \( e \rightarrow^*_M e'' \rightarrow^*_M e' \) and \((k, e') \rightarrow^*_C (\bullet, v)\).

By induction \((k, e'') \rightarrow^*_M (\bullet, v)\) and by Theorem 13.1 we have \((k, e) \rightarrow^*_C (\bullet, v)\), as required.

13.2 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.\footnote{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.} 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 \textit{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 \textit{machine stack}, \(E\) is an \textit{environment}, a finite function mapping variables to \textit{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 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 \textit{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 \(\text{int} \to \text{int} \to \text{int}\). When applied to an integer \(n\), the result \(\text{apply}(f, n)\) yields a function of type \(\text{int} \to \text{int}\). For example, \(f\) might be the following function:

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

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

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

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\(^6\)

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

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

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

$$\begin{align*}
\text{n mvalue} & \quad \text{true mvalue} & (13.17) \\
\text{true mvalue} & \quad \text{false mvalue} & (13.18) \\
\text{fun } f (x:\tau_1) : \tau_2 \text{ is } e \text{ end}[E] & \quad \text{mvalue} & (13.20)
\end{align*}$$

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

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

$$\begin{align*}
\bullet & \quad \text{mstack} & (13.21)
\end{align*}$$

\(^5\)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.
13.2 Environments

\[
\frac{F \text{ mframe} \quad K \text{ mstack}}{F \triangleright K \text{ mstack}}. \tag{13.22}
\]

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

\[
\frac{+(\square, e_2)[E] \text{ mframe}}{V_1 \text{ mvalue}} \tag{13.23}
\]

\[
\frac{+(V_1, \square) \text{ mframe}}{\text{if} \; \square \; \text{then} \; e_1 \; \text{else} \; e_2 \; \text{fi}[E] \text{ mframe}} \tag{13.24}
\]

\[
\frac{\text{apply} \; (\square, e_2) \; [E] \text{ mframe}}{V_1 \text{ mvalue}} \tag{13.25}
\]

\[
\frac{\text{apply} \; (V_1, \square) \; \text{ mframe}}{\text{apply} \; (V_1, \square) \; \text{ mframe}} \tag{13.26}
\]

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 \((K, V)\), 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:

\[
(K, E, e) \xrightarrow{E} (K', E', e') \quad \text{process expression}
\]

\[
(K, E, v) \xrightarrow{E} (K', V') \quad \text{pass value from expression to stack}
\]

\[
(K, V) \xrightarrow{E} (K', V') \quad \text{pass value to stack}
\]

\[
(K, V) \xrightarrow{E} (K', E', e') \quad \text{process pending expression}
\]

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.

$$(K, E, x) \mapsto_E (K, E(x))$$

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

$$(K, E, n) \mapsto_E (K, n)$$

$$(K, E, \text{true}) \mapsto_E (K, \text{true})$$

$$(K, E, \text{false}) \mapsto_E (K, \text{false})$$

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

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

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

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

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

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

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

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.

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

\[
(+([\square, e_2])E[K, V_1]) \mapsto_E (+([V_1, \square])K, E, e_2)
\] (13.36)

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

\[
(+([n_1, \square])K, n_2) \mapsto_E (K, n_1 + n_2)
\] (13.37)

Pending conditional branches are handled in the obvious manner.

\[
(if \square then e_1 else e_2 \text{[if]} E[K, \text{true}]) \mapsto_E (K, E, e_1)
\] (13.38)

\[
(if \square then e_1 else e_2 \text{[if]} E[K, \text{false}]) \mapsto_E (K, E, e_2)
\] (13.39)

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

Pending function applications are handled as follows.

\[
(\text{apply}([\square, e_2])E[K, V]) \mapsto_E (\text{apply}(V, \square)E[K, e_2])
\] (13.40)

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.

\[
(\text{apply}(V, \square)E[K, V_2]) \mapsto_E (K, E[f \mapsto V][x \mapsto V_2], e)
\] (13.41)

where

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

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 \((\bullet, V)\), 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 MinML program of base type (\(\text{int}\) or \(\text{bool}\)), then \((\bullet, e) \mapsto^*_E (\bullet, v)\) iff \((\bullet, \emptyset, e) \mapsto^*_E (\bullet, v)\). (The restriction to base type is necessary if we are to claim that both machines return the same value.)
Chapter 14

Continuations

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

The idea of reification of control stacks can be taken a step further, by allowing them to be passed as values within a program and to be restored at a later point, even if control has long since returned past the point of reification. Reified control stacks of this kind are called first-class continuations, where the qualification “first class” stresses that they are ordinary values with an indefinite lifetime that can be passed and returned at will in a computation. In contrast to set points in C first-class continuations never “expire”, and it is always sensible to reinstate a continuation without compromising safety. Thus first-class continuations support unlimited “time travel” — we can go back to a previous point in the computation and then return to some point in its future, at will.
How is this achieved? The key to implementing first-class continuations is to arrange that control stacks are persistent data structures, just like any other data structure in ML that does not involve mutable references. By a persistent data structure we mean one for which operations on it yield a “new” version of the data structure without disturbing the old version. For example, lists in ML are persistent in the sense that if we cons an element to the front of a list we do not thereby destroy the original list, but rather yield a new list with an additional element at the front, retaining the possibility of using the old list for other purposes. In this sense persistent data structures allow time travel — we can easily switch between several versions of a data structure without regard to the temporal order in which they were created. This is in sharp contrast to more familiar ephemeral data structures for which operations such as insertion of an element irrevocably mutate the data structure, preventing any form of time travel.

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

Why are first-class continuations useful? Fundamentally, they are representations of the control state of a computation at a given point in time. Using first-class continuations we can “checkpoint” the control state of a program, save it in a data structure, and return to it later. In fact this is precisely what is necessary to implement threads (concurrently executing programs) — the thread scheduler must be able to checkpoint a program and save it for later execution, perhaps after a pending event occurs or another thread yields the processor. In Chapter 2 we will show how to build a threads package for concurrent programming using continuations.
14.1 Informal Overview of Continuations

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

Informally, evaluation of letcc $x$ in $e$ binds the current continuation\(^2\) to $x$ and evaluates $e$. The current continuation as, as we've discussed, a reification of the current control stack, which represents the current point in the evaluation of the program. The type of $x$ is $\tau_{\text{cont}}$, where $\tau$ is the type of $e$. The intuition is that the current continuation is the point to which $e$ returns when it completes evaluation. Consequently, the control stack expects a value of type $\tau$, which then determines how execution proceeds. Thus $x$ is bound to a stack expecting a value of type $\tau$, that is, a value of type $\tau_{\text{cont}}$. 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 ref expression.)

We may “jump” to a saved control point by throwing a value to a continuation, written throw $e_1$ to $e_2$. The expression $e_2$ must evaluate to a $\tau_1_{\text{cont}}$, 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 =  
  callcc  
    (fn ret =>  
      let fun mult (nil) = 1  
      | mult (0::l) = throw ret 0  
      | mult (n::l) = n * mult (l)  
      in mult (l) end)
```

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

\(^2\)Hence the name “letcc”.

WORKING DRAFT  SEPTEMBER 12, 2001
Ignoring the callcc for the moment, the body of mult_list is a let expression that defines a recursive procedure mult, and applies it to the argument of mult_list. The job of mult is to return the product of the elements of the list times the value of the accumulator; by calling mult with 1 and 1, we obtain the product of the elements of l. Ignoring the second line of mult, it should be clear why and how this code works.

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

Here’s a slicker formulation of the same function:

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

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

Exercise 14.1

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

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

```
fun compose (f:τ’→τ,k:τ cont):τ’ cont = ...
```

The function compose will have type
Exercise 14.2

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

The first problem is to obtain the continuation we wish to return. The second problem is how to return it. The continuation we seek is the one in effect at the point of the ellipsis in the expression \(\text{throw } f(...)\) 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 \text{letcc}, writing

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

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, in SML/NJ notation:

\[
\text{fun compose } (f, k) =
\begin{align*}
\text{callcc} & \quad (* \text{return point} *)
\text{throw } k (f (\text{callcc} (\text{fn } k' => \text{throw return } k')))
\end{align*}
\]

The type of \text{return} 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 \text{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:

\[
\text{fun cps_mult nil k = k 1}
\mid \text{cps_mult } (n::l) k = \text{cps_mult } l \ (\text{fn } r => k \ (n \ast r))
\text{fun mult } l = \text{cps_mult } l \ (\text{fn } r => r)
\]
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 k0 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

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

and that the type of the second version becomes

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

These transformations are representative of the general case.

### 14.2 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 suprisingly 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:

\[
e \ ::= \ldots \mid \text{letcc} x \in e \mid \text{throw}_{e_1} e_2 \mid K
\]

In the expression `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 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:

\[
\begin{align*}
K & \text{ stack} \\
\bar{K} & \text{ value}
\end{align*}
\]
14.2 Semantics of Continuations

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

\[ e_2 \text{ expr} \]
\[ \text{throw} \square \text{to} e_2 \text{ frame} \]  \( (14.2) \)

\[ v_1 \text{ value} \]
\[ \text{throw} v_1 \text{ to} \square \text{ frame} \]  \( (14.3) \)

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

\[ \Gamma[x: \tau \text{ cont}] \vdash e : \tau \]
\[ \Gamma \vdash \text{letcc} \: x \: \text{in} \: e : \tau \]
\[ (14.4) \]

\[ \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_1 \text{ cont} \]
\[ \Gamma \vdash \text{throw} e_1 \text{ to} e_2 : \tau' \]
\[ (14.5) \]

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:

\[ \vdash K : \tau \text{ cont} \]
\[ \Gamma \vdash K : \tau \text{ cont} \]
\[ (14.6) \]

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

Finally, the dynamic semantics. We use the 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:

\[ (K, \text{letcc} \: x \: \text{in} \: e) \mapsto (K, [K/x]e) \]  \( (14.7) \)

\[ (\text{throw} v \text{ to} \square \text{ to} K, K') \mapsto (K', v) \]  \( (14.8) \)

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

\[ (K, \text{throw} e_1 \text{ to} e_2) \mapsto (\text{throw} \square \text{ to} e_2 \: K, e_1) \]  \( (14.9) \)

\[ (\text{throw} \square \text{ to} e_2 \: K, e_1) \mapsto (\text{throw} v_1 \text{ to} \square \text{ to} K, e_2) \]  \( (14.10) \)

Notice that evaluation of letcc duplicates the control stack, and that evaluation of throw eliminates the current control stack.
Exercise 14.3
Simulate the evaluation of compose \((f, k)\) on the empty stack. Observe that the control stack substituted for \(x\) is

\[
\text{apply}(f, \Box) \triangleright \text{throw} \Box \triangleright k \triangleleft \bullet
\]  

(14.11)

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

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

The preservation theorem is stated as follows:

**Theorem 14.4 (Preservation)**
If \((K, e)\) ok and \((K, e) \rightarrow (K', e')\), 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 14.5 (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 14.6 (Progress)**
If \((K, e)\) ok then either \(K = \bullet\) and \(e\) value, or there exists \(K'\) and \(e'\) such that \((K, e) \rightarrow (K', e')\).

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

### 14.3 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 handoff from producer to consumer. Here is an outline of a simple producer-consumer relationship:

```plaintext
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:

```plaintext
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))
```

WORKING DRAFT SEPTEMBER 12, 2001
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:

```haskell
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:

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

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

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

Here is a simple implementation of threads:

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

Using the above thread interface we may implement the simple producer-consumer example as follows:
structure Client = struct
  open Threads
  val buffer : int ref = ref (˜1)
  fun producer (n) = 
    (buffer := n ; yield () ; producer (n+1))
  fun consumer () = 
    (print (Int.toString (!buffer)); yield (); consumer())
  fun run () = 
    (fork (consumer); producer 0)
end

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

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

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

There is much more to be said about threads! We will return to this later in the course. For now, the main idea is to give a flavor of how first-class continuations can be used to implement a user-level threads package with very little difficulty. A more complete implementation is, of course, somewhat more complex, but not much more. We can easily provide all that is necessary for sophisticated thread programming in a few hundred lines of ML code.
Chapter 15

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:

\[ e ::= \ldots | \text{fail} | \text{try} e_1 \text{ow} e_2 \]

The expression \text{fail} raises an exception. The expression \text{try} e_1 \text{ow} e_2 evaluates \( c_1 \). If it terminates normally, we return its value; otherwise, if it fails, we continue by evaluating \( e_2 \).

The static semantics of exceptions is quite straightforward:

\[
\Gamma \vdash \text{fail} : \tau
\] (15.1)
\[
\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{try} \ e_1 \ \text{ow} \ e_2 : \tau} \quad (15.2)
\]

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:

\[
\frac{e_2 \ \text{expr}}{\text{try} \ \Box \ \text{ow} \ e_2 \ \text{frame}} \quad (15.3)
\]

The evaluation rules are extended as follows:

\[
\frac{}{(K, \text{try} \ e_1 \ \text{ow} \ e_2) \mapsto (\text{try} \ \Box \ \text{ow} \ e_2 \triangleright K, e_1)} \quad (15.4)
\]

\[
\frac{}{(\text{try} \ \Box \ \text{ow} \ e_2 \triangleright K, v) \mapsto (K, v)} \quad (15.5)
\]

\[
\frac{}{(\text{try} \ \Box \ \text{ow} \ e_2 \triangleright K, \text{fail}) \mapsto (K, e_2)} \quad (15.6)
\]

\[
\frac{(F \neq \text{try} \ \Box \ \text{ow} \ e_2)}{(F \triangleright K, \text{fail}) \mapsto (K, \text{fail})} \quad (15.7)
\]

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 \textit{unwinding} the control stack. Finally, we regard the state \((\bullet, \text{fail})\) as a final state of computation, corresponding to an uncaught exception.

**Exercise 15.1**

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

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

\[
\frac{}{\vdash K : \tau \ \text{stack} \quad \vdash e : \tau \quad (K, e) \ \text{ok}} \quad (15.8)
\]
That is, a state \((K, e)\) is well-formed iff \(e\) is an expression of type \(\tau\) and \(K\) is a \(\tau\)-accepting control stack. The latter is defined by the following rules:

\[
\begin{align*}
\vdash \cdot &: \tau \text{stack} \\
\vdash F : (\tau, \tau') \text{ frame} &\vdash K : \tau' \text{ stack} \\
\vdash F \triangleright K &: \tau \text{ stack}
\end{align*}
\] (15.9)

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

\[
\begin{align*}
\vdash e_2 &: \text{int} \\
\vdash (+, e_2) &: (\text{int}, \text{int}) \text{ frame} \\
\vdash v_1 &: \text{value} \\
\vdash v_1, e &: \text{frame} \\
\vdash e_1 &: \tau \\
\vdash e_2 &: \tau \\
\vdash \text{if} \square \text{then } e_1 \text{ else } e_2 \text{ fi} &: (\text{bool}, \tau) \text{ frame} \\
\vdash e_2 &: \tau_2 \\
\vdash \text{apply}(\square, e_2) &: (\tau_2 \to \tau, \tau) \text{ frame} \\
\vdash v_1 &: \text{value} \\
\vdash v_1, e &: \text{frame} \\
\vdash e_2 &: \tau \\
\vdash \text{try} \square \text{ ow } e_2 &: (\tau, \tau') \text{ frame}
\end{align*}
\] (15.10)

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 “\(\square\)” in the frame; the result is the type of the frame once its hole has been filled with an expression of the given type.

With this in place we can state and prove safety.

**Theorem 15.2 (Preservation)**

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

**Proof:** By induction on evaluation.  

\[\square\]

**Working Draft**  

**September 12, 2001**
Exercise 15.3
Prove Theorem 15.2.

Theorem 15.4 (Progress)
If \((K, e)\) ok then either

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

Proof: By induction on typing.

Exercise 15.5
Prove Theorem 15.4.

Exercise 15.6
Combine the treatment of references and exceptions to form a language with both of these features. You will face a choice of how to define the interaction between mutation and exceptions:

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

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

The dynamic semantics of exceptions is somewhat unsatisfactory because of the explicit unwinding of the control stack to find the nearest enclosing handler. While this does effect a non-local transfer of control, it does so by rather crude means, rather than by a direct “jump” to the handler. In practice exceptions are implemented as jumps, using the following ideas. A dedicated register is set aside to contain the “current” exception handler. When an exception is raised, the current handler is retrieved from the exception register, and control is passed to it. Before doing so, however, we must reset the exception register to contain the nearest handler enclosing the new handler. This ensures that if the handler raises an exception the correct handler is invoked. How do we recover this handler? We maintain a stack of pending handlers that is pushed whenever a handler is installed,
and popped whenever a handler is invoked. The exception register is the
top element of this stack. Note that we must restore the control stack to the
point at which the handler was installed before invoking the handler!

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

- \(H\) is a handler stack;
- \(K\) is a control stack;
- \(e\) is a closed expression

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

\[
\text{hstack} \quad (H, K, e) \quad (15.17)
\]

\[
\begin{array}{c}
K \quad \text{stack} \\
\text{e expr} \\
H \quad \text{hstack}
\end{array} \quad \Rightarrow \quad (K, e) \quad (15.18)
\]

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:

\[
((K', e') \dashv H, K, \text{fail}) \Rightarrow (H, K', e') \quad (15.19)
\]

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 ma-
chine:

\[
(\cdot, K, \text{fail}) \Rightarrow (\cdot, \cdot, \text{fail}) \quad (15.20)
\]

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

\[
(H, K, \text{try} e_1 \text{ow} e_2) \Rightarrow ((K, e_2) \dashv H, \text{try} \Box \text{ow} e_2 \dashv K, e_1) \quad (15.21)
\]

We “freeze dry” the control stack, associate it with the unevaluated han-
dler, 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 \(e_1\):

\[
((K, e_2) \dashv H, \text{try} \Box \text{ow} e_2 \dashv K, v_1) \Rightarrow (H, K, v_1) \quad (15.22)
\]
Exercise 15.7
State and prove the safety of this formulation of exceptions.

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

Exercise 15.8
Prove that the prefix property is preserved by every step of evaluation.

Finally, let us consider value-passing exceptions such as are found in Standard ML. The main idea is to replace the failure expression, \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:

\[
\Gamma \vdash e : \tau_{\text{exn}} \\
\Gamma \vdash \text{raise}(e) : \tau
\]

\[(15.23)\]

\[
\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau_{\text{exn}} \rightarrow \tau \\
\Gamma \vdash \text{try } e_1 \text{ ow } e_2 : \tau
\]

\[(15.24)\]

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, \texttt{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

\[
\text{fun div } (m, 0) = \text{raise } "$\text{Division by zero attempted.}$"
\]

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

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

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

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

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

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

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

```
exception Div
exception Overflow
```

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

Effect-ful Languages
Chapter 16

TinyC, A Tiny Fragment of C

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 application of the techniques developed so far for an imperative language, one whose execution model is based on assignment to variables.
16.1 Syntax

The syntax of TinyC is given by the following grammar.¹

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

This grammar certainly makes clear the “stripped down” nature of C syntax!

A statement is either an expression, executed for its effect and value, a conditional statement, a while statement, a return 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 to declare variables. All variables must be initialized to a given constant value.

¹The notation “\(\circ\)” stands for the empty sequence.
²The 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.
The higher-order abstract syntax of TinyC is somewhat tricky to specify precisely, because it provides for “bulk” declarations of several variables with sequential scope. The following rules define which variables and functions are declared in which phrases, and what is their range of significance:

1. In a program $g \ s$, the functions and variables declared in $g$ are bound within $g$ (as specified below) and in $s$.
2. In a block \{ $l \ ss \} \$, the variables declared in $l$ are bound in $ss$.
3. The variable $x$ is declared in the variable declaration $\text{int } x = v \;;$.
4. The function $f$ is declared in the function declaration $\text{int } f(\text{as}) \ s$.
5. In an empty global declaration sequence, $\circ$, no variables are declared.
6. In a compound declaration sequence $vd \ g$, the variable declared in $vd$ is bound in $g$.
7. In a compound declaration sequence $fd \ g$, the function declared in $fd$ is bound in $g$.
8. The variables and functions declared in $vd$ and $g$ are declared in the sequence $vd \ g$; those declared in $fd$ and $g$ are declared in the sequence $fd \ g$.
9. In an empty local declaration sequence, $\circ$, no variables are declared.
10. The variables declared in $vd$ and $l$ are declared in the sequence $vd \ l$.
11. In a function declaration of the from $\text{int } f(\text{as}) \ s$, the variables declared in $as$ are bound in $s$.
12. The variable declared in $a$ is bound in the length-1 sequence $a$.
13. In $a$, as the variable declared in $a$ is bound in $as$. The variables declared in $a$ and in $as$ are declared in $a$, $as$.
14. The variable $x$ is declared in the argument $\text{int } x$.

With these definitions in mind we can easily give a precise definition of the set of free variables in a phrase and of equivalence up to renaming of bound variables.
16.2 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 integers 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 \ ok & \quad \text{\( p \) is well-formed} \\
F, V \vdash s \ ok & \quad \text{\( s \) is well-formed wrt \( F \) and \( V \)} \\
F, V \vdash ss \ ok & \quad \text{\( ss \) is well-formed wrt \( F \) and \( V \)} \\
F, V \vdash e \ ok & \quad \text{\( e \) is well-formed wrt \( F \) and \( V \)} \\
F, V \vdash es \ ok n & \quad \text{\( es \) is well-formed wrt \( F \) and \( V \)} \\
F, V \vdash \text{decs} \, F', V' & \quad \text{\( g \) declares functions \( F' \) and variables \( V' \)} \\
F, V \vdash \text{decs} \, V' & \quad \text{\( l \) declares variables \( V' \)} \\
F, V \vdash \text{decs} \, x & \quad \text{\( v \) declares variable \( x \)} \\
F, V \vdash \text{decs} \, f : n & \quad \text{\( f \) declares function \( f \) with arity \( n \)} \\
F, V \vdash \text{decs} \, x & \quad \text{\( a \) declares variable \( x \)} \\
F, V \vdash \text{decs} \, V' & \quad \text{\( a s \) 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.

\[
\begin{align*}
\emptyset, \emptyset \vdash \text{decs} \, F, V & \quad F, V \vdash s \ ok \\
\hline \\
\vdash \text{decs} \, F, V \\
\end{align*}
\]

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

\[
\begin{align*}
F, V \vdash e \ ok & \quad F, V \vdash e ; \ ok \\
\hline \\
F, V \vdash \text{if} \ (e) \ s_1 \ \text{else} \ s_2 \ ok & \quad F, V \vdash \text{if} \ (e) \ s_1 \ \text{else} \ s_2 \ ok \\
\end{align*}
\]
16.2 Static Semantics

\[
\frac{F, V \vdash e \text{ ok} \quad F, V \vdash s \text{ ok}}{F, V \vdash \text{while (e) s ok}} \quad (16.4)
\]

\[
\frac{F, V \vdash e \text{ ok}}{F, V \vdash \text{return e; ok}} \quad (16.5)
\]

\[
\frac{F, V \vdash l \text{ decs } V' \quad F, V \cup V' \vdash ss \text{ ok} \quad (V \cap V' = \emptyset)}{F, V \vdash \{ l \text{ ss } \} \text{ ok}} \quad (16.6)
\]

A statement sequence is well-formed if its constituent statements are:

\[
\frac{F, V \vdash o \text{ ok}}{} \quad (16.7)
\]

\[
\frac{F, V \vdash s \text{ ok} \quad F, V \vdash ss \text{ ok}}{F, V \vdash s \text{ ss ok}} \quad (16.8)
\]

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.

\[
\frac{F, V \vdash v \text{ ok}}{} \quad (16.9)
\]

\[
\frac{(x \in V)}{F, V \vdash x \text{ ok}} \quad (16.10)
\]

\[
\frac{F, V \vdash e_1 \text{ ok} \quad F, V \vdash e_2 \text{ ok}}{F, V \vdash e_1+e_2 \text{ ok}} \quad (16.11)
\]

\[
\frac{F, V \vdash e_1 \text{ ok} \quad F, V \vdash e_2 \text{ ok}}{F, V \vdash e_1-e_2 \text{ ok}} \quad (16.12)
\]

\[
\frac{F, V \vdash e \text{ ok} \quad (x \in V)}{F, V \vdash x=e \text{ ok}} \quad (16.13)
\]

\[
\frac{F, V \vdash es \text{ ok} \quad (F(f) = n)}{F, V \vdash f(es) \text{ ok}} \quad (16.14)
\]
Sequences of expressions are considered from left to right.

$$\begin{align*}
F, V \vdash & \ e \ \text{ok} \\
F, V & \vdash \ e \ \text{ok} \ 1
\end{align*}$$

(16.15)

$$\begin{align*}
F, V \vdash & \ e \ \text{ok} \quad F, V \vdash \ es \ \text{ok} \ n \\
F, V & \vdash \ e, \ es \ \text{ok} \ n + 1
\end{align*}$$

(16.16)

A sequence of global declarations may not re-declare a variable or function.

$$F, V \vdash \ o \ \text{decs} \ \emptyset$$

(16.17)

$$\begin{align*}
F, V \vdash & \ vd \ \text{decs} \ x \\
F, V \cup & \ \{ x \} \vdash \ g \ \text{decs} \ F', V' \quad (x \notin V')
\end{align*}$$

$$F, V \vdash \ vd \ g \ \text{decs} \ F', \{ x \} \cup V'$$

(16.18)

$$\begin{align*}
F, V \vdash & \ fd \ \text{decs} \ f : n \\
F \cup & \ \{ f : n \}, V \vdash \ g \ \text{decs} \ F', V' \quad (f \notin F')
\end{align*}$$

$$F, V \vdash \ fd \ g \ \text{decs} \ \{ f : n \} \cup F', V'$$

(16.19)

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

$$F, V \vdash \ o \ \text{decs} \ \emptyset$$

(16.20)

$$\begin{align*}
F, V \vdash & \ vd \ \text{decs} \ x \\
F, V \cup & \ \{ x \} \vdash \ l \ \text{decs} \ V' \quad (x \notin V')
\end{align*}$$

$$F, V \vdash \ vd \ l \ \text{decs} \ \{ x \} \cup V'$$

(16.21)

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

$$\begin{align*}
(x \notin V) \\
F, V \vdash & \ \text{int} \ x = v \ ; \ \text{decs} \ x
\end{align*}$$

(16.22)

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!).

$$\begin{align*}
F \cup & \ \{ f : n \}, V \cup V' \vdash \ ok \\
F, V \vdash & \ as \ \text{decs} \ V' \quad (|V'| = n, \ f \notin F)
\end{align*}$$

$$F, V \vdash \ \text{int} \ f (as) \ s \ \text{decs} \ f : n$$

(16.23)
An argument must not already be a declared variable. A sequence of arguments may not re-declare a variable.

\[
(x \notin V) \\
F, V \vdash \text{int } x \text{ decs } x
\]  
(16.24)

\[
F, V \vdash a \text{ decs } x \\
F, V \vdash a \text{ decs } \{x\}
\]  
(16.25)

\[
F, V \vdash a \text{ decs } x \quad F, V \vdash as \text{ decs } V' \quad (x \notin V') \\
F, V \vdash a, as \text{ decs } \{x\} \cup V'
\]  
(16.26)

### 16.3 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:

- **Program execution**: \( p \rightarrow p' \)
- **Statement execution**: \( (g, s) \rightarrow (g', s') \)
- **Expression execution**: \( (g, e) \rightarrow (g', e') \)
- **Expression list execution**: \( (g, es) \rightarrow (g', es') \)

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.

\[
(g, s) \rightarrow (g', s') \\
g s \rightarrow g's'
\]  
(16.27)
An expression is executed for its effect on the environment.

\[
(g, e) \mapsto (g', e') \\
(g, e; ) \mapsto (g', e'; )
\]  
(16.28)

A conditional is executed by evaluating the test expression, then branching on whether the result is non-zero.

\[
(g, e) \mapsto (g', e') \\
(g, \text{if} (e) s_1 \text{else} s_2) \mapsto (g', \text{if} (e') s_1 \text{else} s_2)
\]  
(16.29)

\[
(g, \text{if} (0) s_1 \text{else} s_2) \mapsto (g, s_2)
\]  
(16.30)

\[
(v \neq 0) \\
(g, \text{if} (v) s_1 \text{else} s_2) \mapsto (g, s_1)
\]  
(16.31)

A while statement is executed by conditionally unrolling the body.

\[
(g, \text{while} (e) s) \mapsto (g, \text{if} (e) \{ s \text{ while} (e) s \} \text{else} \{ 0; \})
\]  
(16.32)

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

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

\[
(g, e) \mapsto (g', e') \\
(g, \text{return} e; ) \mapsto (g', \text{return} e'; )
\]  
(16.33)

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.\(^3\)

\[
(g \cdot l, ss) \mapsto^* (g' \cdot l', v; ) \\
(g, \{ l ss \}) \mapsto (g', v; )
\]  
(16.34)

\(^3\)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.
(16.3 Dynamic Semantics 127)

\[
\frac{(g \cdot l, ss) \rightarrow^* (g' \cdot l', \text{return } v)}{(g, \{ \ell ss \}) \rightarrow (g', \text{return } v)}
\]

(16.35)

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.

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.

\[
\frac{(g, \circ) \rightarrow (g, 0 ;)}{}
\]

(16.36)

\[
\frac{(g, s) \rightarrow (g', s')}{(g, s ss) \rightarrow (g', s' ss)}
\]

(16.37)

\[
\frac{(g, v ; ss) \rightarrow (g, ss)}{}
\]

(16.38)

\[
\frac{(g, \text{return } v; ss) \rightarrow (g, \text{return } v;)}{}
\]

(16.39)

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

\[
\frac{g \mathbin{@} x = v}{(g, x) \rightarrow (g, v)}
\]

(16.40)

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

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

\[
\frac{(g, e_1) \rightarrow (g', e'_1)}{(g, e_1 + e_2) \rightarrow (g', e'_1 + e_2)}
\]

(16.41)

\[
\frac{(g, e_2) \rightarrow (g', e'_2)}{(g, n_1 + e_2) \rightarrow (g', n_1 + e'_2)}
\]

(16.42)

\[
\frac{(n = n_1 + n_2)}{(g, n_1 + n_2) \rightarrow (g, n)}
\]

(16.43)
The rules for subtraction — and other arithmetic operations — are similar.
Assignments are evaluated by evaluating the right-hand side, then updating the environment.

\[(g, e) \mapsto (g', e')\]
\[(g, x = e) \mapsto (g', x = e')\]  (16.44)

\[g \@ x \leftarrow v = g'\]
\[(g, x = v) \mapsto (g', v)\]  (16.45)

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

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.

\[(g, es) \mapsto (g', es')\]
\[(g, f(es)) \mapsto (g', f(es'))\]  (16.46)

\[g \@ f = \text{int } f(\text{int } x_1, \ldots \text{int } x_k, \circ ) \ s\]
\[(g, \{ \text{int } x_1 = v_1; \cdots \text{int } x_k = v_k; \circ s \}) \mapsto (g', \text{return } v; )\]
\[(g, f(v_1, \ldots, v_k)) \mapsto (g', v)\]  (16.47)

\[g \@ f = \text{int } f(\text{int } x_1, \ldots \text{int } x_k, \circ ) \ s\]
\[(g, \{ \text{int } x_1 = v_1; \cdots \text{int } x_k = v_k; \circ s \}) \mapsto (g', v; )\]
\[(g, f(v_1, \ldots, v_k)) \mapsto (g', v)\]  (16.48)

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

\[(g, e) \mapsto (g', e')\]
\[(g, e) \mapsto (g', e')\]  (16.49)

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

\[(g, e) \mapsto (g', e')\]
\[(g, e, es) \mapsto (g', e', es)\]  (16.50)

\[(g, es) \mapsto (g', es')\]
\[(g, v, es) \mapsto (g', v, es')\]  (16.51)
16.4 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 the rules Rule 16.47 and Rule 16.48. 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.

16.5 Type Safety

The safety of TinyC is stated as follows.
Theorem 16.1

If $\vdash p \ ok$, 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.
Chapter 17

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 will be studied in Chapter 15 and in Chapter 14. Store effects are dynamic modifications to mutable storage. This chapter is concerned with store effects.

17.1 References

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

$$e ::= l | ref(e) | !e | e_1 := e_2$$

As in Standard ML, $ref(e)$ 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:

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

(17.1)

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

(17.2)

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

(17.3)

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

(17.4)

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:

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

(17.5)

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

(17.6)

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

(17.7)
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 \(\dvdash M : \Lambda\), is inductively defined by the following rule:

\[
\text{dom}(M) = \text{dom}(\Lambda) \quad \forall l \in \text{dom}(\Lambda) \quad \Lambda; \dvdash M(l) : \Lambda(l) \\
\dvdash M : \Lambda
\]

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

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

(* loop forever when called *)
fun diverge (x:int):int = diverge x

(* allocate a reference cell *)
val fc : (int->int) ref = ref (diverge)

(* define a function that ``recurs'' through fc *)
fun f 0 = 1 | f n = n * (!fc)(n-1)

(* tie the knot *)
val _ = fc := f

(* now call f *)
val n = f 5

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

**Exercise 17.1**

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

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

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

\[
\frac{\vdash M : \Lambda \quad \Lambda; \bullet \vdash e : \tau}{(M, e) \text{ ok}} \quad (17.13)
\]

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

**Theorem 17.2 (Preservation)**

If $(M, e)$ ok and $(M, e) \mapsto (M', e')$, then $(M', e')$ ok.

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

**Exercise 17.3**

Prove Theorem Theorem 17.2. 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 17.4 (Progress)**

If $(M, e) \text{ ok}$ 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 17.5**

*Prove Theorem Theorem 17.4 by induction on typing of machine states.*
Chapter 18

Lazy Evaluation

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.

### 18.1 Lazy Types

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

\[ \text{nats} = \text{iterate} \ \text{successor} \ 0 \]

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

\[ \text{nats} = \text{lcons} \ (0, \text{map} \ \text{successor} \ \text{nats}) , \]

where successor and map 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 map successor to nats, then inspect the head element of the result. This is successor(0), or 1; its 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.
18.1 Lazy Types

18.1.1 Lazy Lists in an Eager Language

The additional constructs required to add lazy lists to MinML are given by the following grammar:

\[
\begin{align*}
\text{Types} \quad \tau & ::= \text{llist} \\
\text{Expressions} \quad e & ::= \text{lnil} | \text{lcons}(e_1, e_2) | \text{lazy} \, x \, \text{is} \, e \\
& \quad | \text{lcase} \, e \, \text{of} \, \text{lnil} => e_0 | \text{lcons}(x, y) => e_1 \text{ end}
\end{align*}
\]

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 \( \text{lnil} \) 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:

\[
\text{lazy} \, x \, \text{is} \, \text{lcons} \,(1, \, \text{lmap} \, \text{successor} \, x).
\]

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:

\[
\begin{align*}
\text{fun} \, \text{map}(f: \text{int} \to \text{int}) : \text{llist} \to \text{llist} \, \text{is} \\
\text{fun} \, \text{lmapf}(l: \text{llist}) \, \text{is} \\
\quad \text{lcase} \, l \\
\quad \text{of} \, \text{lnil} => \, \text{lnil} \\
\quad \mid \, \text{lcons}(x, y) => \, \text{lcons} \,(f \, x, \, \text{lmapf} \, y).
\end{align*}
\]

Unfortunately this definition doesn’t work as expected! Suppose that \( f \) is a function of type \( \text{int} \to \text{int} \) 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 \( \text{lmapf} \), 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
    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 18.1
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:

\[ \Gamma \vdash \text{x:llist} \vdash e : \text{llist} \]  \hspace{1cm} (18.1)

\[ \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{llist} \quad \Gamma \vdash \text{lcons}(e_1, e_2) : \text{llist} \]  \hspace{1cm} (18.2)

\[ \Gamma, x: \text{llist} \vdash e : \text{llist} \quad \Gamma \vdash \text{lazy}x\text{is}: e : \text{llist} \]  \hspace{1cm} (18.3)

\[ \Gamma \vdash e : \text{llist} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x: \text{int}, y: \text{llist} \vdash e_1 : \tau \quad \Gamma \vdash \text{lcase} e \text{of} \text{lnil} => e_0 \mid \text{lcons}(x, y) => e_1 \end{equation} \]  \hspace{1cm} (18.4)
In Rule 18.2 the body, $e$, of the lazy list expression $\text{l lazy list is } e$ 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 $\text{lnil, lcons } (e_1, e_2)$, and $\text{l lazy x i s } e$ to be values, independently of whether their constituent expressions are values. Second, we evaluate case analyses according to the following transition rules:

\begin{align*}
\text{lcase } \text{lnil of } \text{lnil} := e_0 | \text{lcons } (x, y) := e_1 \end{align*}

(18.5)

\begin{align*}
\text{lcase } \text{lcons } (e_h, e_t) \text{ of } \text{lnil} := e_0 | \text{lcons } (x, y) := e_1 \end{align*}

(18.6)

\begin{align*}
\text{lcase } \text{lcons } (\text{l lazy z is } e) \text{ of } \text{lnil} := e_0 | \text{lcons } (x, y) := e_1 \\
\text{lcase } [\text{l lazy z is } e/z] e \text{ of } \text{lnil} := e_0 | \text{lcons } (x, y) := e_1 \\
\text{let } x : \text{int be } e_h \text{ in let } y : \text{llist be } e_t \text{ in } e_1 \text{ end end}
\end{align*}

(18.7)

\begin{align*}
\text{e := e'} \\
\text{lcase } e \text{ of } \text{lnil} := e_0 | \text{lcons } (x, y) := e_1 \\
\text{lcase } e' \text{ of } \text{lnil} := e_0 | \text{lcons } (x, y) := e_1
\end{align*}

(18.8)

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 18.2**

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**  
**September 12, 2001**
Exercise 18.3

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 $\text{l list}$. 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

$$\text{lazy } x \text{ is } (\text{l map succ } x)$$

that, when forced, maps the successor function over itself.

What is it that makes the recursive list

$$\text{lazy } \text{nats is } \text{l cons } (0, \text{l map succ } \text{nats})$$

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 $\text{l map}$, because $\text{l map}$ is defined to be a lazy function, one yielding a delayed computation of a list.

Exercise 18.4

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 18.5

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
evaluations; 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 fol-
lows:

\[
\begin{align*}
(x \notin \text{dom}(M)) & \quad \Rightarrow (M, \text{lazy } z \text{ is } e) \\
(M, \text{lnil}) & \quad \Rightarrow (M, \text{lnil}) \\
(M, \text{lcons}(e_1, e_2)) & \quad \Rightarrow (M, \text{lcons}(e_1, e_2)) \\
(M(z) = \text{lnil}) & \quad \Rightarrow (M, \text{lnil}) \\
(M(z) = \text{lcons}(v_h, v_t)) & \quad \Rightarrow (M, \text{lcons}(v_h, v_t)) \\
(M(z) = \text{lnil} => e_0 | \text{lcons}(x, y) => e_1 \text{ end}) & \quad \Rightarrow (M, e_0) \\
(M(z) = \text{lcons}(e_h, e_t)) & \quad \Rightarrow (M', e_h) \\
(M(z) = \text{lnil} => e_0 | \text{lcons}(x, y) => e_1 \text{ end}) & \quad \Rightarrow (M', e_t) \\
(M(z) = \text{lazy } z \text{ is } e) & \quad \Rightarrow (M', e) \\
\end{align*}
\]
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, •, 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 18.6
Convince yourself that the replacement of \( z \) by • in the second premise of Rule Rule 18.14 is redundant — the location \( z \) is already guaranteed to be bound to •.

Exercise 18.7
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 18.8
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 18.9
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 18.10
Consider again the definition of the natural numbers as the lazy list

\[
\text{lazy nats is (lcons (0, lmap succ nats))}.
\]

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).

18.1.2 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 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 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:

\[
\begin{align*}
\text{Types} & \quad \tau ::= \tau\ \text{computation} \\
\text{Expressions} & \quad e ::= \text{delay}\ x\ is\ e | \text{eval}\ e_1\ as\ x\ in\ e_2\ end
\end{align*}
\]

In the expression \(\text{delay}\ x\ is\ e\) the variable \(x\) is bound within \(e\), and in the expression \(\text{eval}\ e_1\ as\ x\ in\ e_2\ end\) the variable \(x\) is bound within \(e_2\). The expression \(\text{delay}\ x\ is\ e\) both delays evaluation of \(e\) and gives it a name that can be used within \(e\) to stand for the computation itself. The expression \(\text{eval}\ e_1\ as\ x\ in\ e_2\ end\) 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:

\[
\Gamma \vdash e : \tau \\
\Gamma \vdash \text{delay}\ x\ is\ e : \tau\ \text{computation}
\]

\[(18.17)\]
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:

\[
\begin{align*}
\Gamma, x : \tau_1 & \vdash e_1 : \tau_1 \\
\Gamma & \vdash \text{eval } e_1 \text{ as } x \text{ in } e_2 \text{ end} : \tau_2 
\end{align*}
\] (18.18)

Exercise 18.11

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 \texttt{lcell} of lazy lists may be defined to be the type where \texttt{lcell} has the following constructors and destructors:

\[
\begin{align*}
\Gamma & \vdash \text{cnil} : \text{lcell} \\
\Gamma & \vdash e_h : \text{int} \quad \Gamma & \vdash e_l : \text{llist} \\
\Gamma & \vdash \text{ccons} (e_h, e_l) : \text{lcell} \\
\Gamma, x : \tau_1, y : \text{llist} & \vdash e_c : \tau \\
\Gamma & \vdash \text{ccase } e \text{ of } \text{cnil} \Rightarrow e_h \mid \text{ccons} (x, y) \Rightarrow e_c : \tau 
\end{align*}
\] (18.25)

Observe that the “tail” of a \texttt{ccons} is of type \texttt{llist}, not \texttt{lcell}. Using these primitives we may define the lazy list constructors as follows:
18.2 Lazy Languages

\[
\begin{align*}
\text{lnil} &= \text{lazy} \cdot \text{is cnil} \\
\text{lcons}(e_h, e_t) &= \text{lazy} \cdot \text{is ccons}(e_h, e_t) \\
\text{lcase } e \text{ of nil }&\Rightarrow e_n \mid \text{cons}(x, y) \Rightarrow e_c = \\
&\quad \text{force } z = e \\
&\quad \text{case } z \text{ of cnil }&\Rightarrow e_n \mid \text{ccons}(x, y) \Rightarrow e_c
\end{align*}
\]

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 \text{llist} with \text{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 \text{llist} of lazy lists (delayed computations of cells) and the type \text{lcell} of lazy cells (which specify emptiness and define the first element of non-empty lists).

18.2 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-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

\[ \text{val } x : \text{int} = 1 + x \]

or

\[ \text{val } x : \text{int list} = \text{cons} (1, x). \]

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

\[ \text{val } x : \text{int delayed} = \text{delay} (1 + x) \]

and

\[ \text{val } x : \text{int list delayed} = \text{delay} (\text{lcons} (1, x)) \]

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

\[ \text{val rec } x : \text{int} = 1 + x. \]

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.

### 18.2.1 Call-by-Name and Call-by-Need

To model lazy languages we simply extend MinML with an additional construct for recursively-defined computations, written \( \text{rec} \ x: \tau \ is \ e \). The variable \( x \) is bound in \( e \), and may be renamed at will. Recursive computations are governed by the following typing rule:

\[
\Gamma, x: \tau \vdash e : \tau \\
\Gamma \vdash \text{rec} x: \tau \ is \ e : \tau
\]  

(18.26)

In addition we replace the recursive function expression \( \text{fun} \ f \ (x: \tau_1) : \tau_2 \ is \ e \ end \) with the non-recursive form \( \text{fn} \ \tau: x \ in \ e \ end \), since the former may be defined by the expression

\[
\text{rec} f: \tau_1 \rightarrow \tau_2 \ is \ e \ \text{fn} \ \tau_1: x \ in \ e \ end.
\]
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:

\[
\text{fn } \tau : x \text{ in } e \text{ end}(e') \mapsto [\text{fn } \tau : x \text{ in } e \text{ end}, e'/x]e
\] (18.27)

This is known as the call-by-name rule, according to which arguments are 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:

\[
\text{rec } x : \tau \text{ is } e \mapsto [\text{rec } x : \tau \text{ is } e/x]e
\] (18.28)

To evaluate a recursive computation, simply unroll the recursion by one step and continue from there.

Exercise 18.12
Show that the behavior of the recursive function expression \(\text{fun } f (x : \tau_1) : \tau_2 \text{ is } e \text{ end}\) is correctly defined by

\[
\text{rec } f : \tau_1 \rightarrow \tau_2 \text{ is } \text{fn } \tau_1 : x \text{ in } e \text{ end}
\]

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 un-evaluated, 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.

\(^2\)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.
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:

\[(M, e_1) \mapsto (M', e'_1) \]
\[(M, + (e_1, e_2)) \mapsto (M', + (e'_1, e'_2)) \] (18.29)

\[(M, e_2) \mapsto (M', e'_2) \]
\[(M, + (v_1, e_2)) \mapsto (M', + (v'_1, e'_2)) \] (18.30)

\[(M, +(n_1, n_2)) \mapsto (M, n_1 + n_2) \] (18.31)

The main differences are in the rule for function application and the need for additional rules for variables and recursive computations.

\[(x \notin \text{dom}(M)) \]
\[(M, \text{fn } \tau : x \text{ in } e \text{ end } (e')) \mapsto (M[x = e'], e) \] (18.32)

\[(M(x) = v) \]
\[(M, x) \mapsto (M, v) \] (18.33)

\[(M(x) = e) \quad (M[x = \bullet], e) \Rightarrow^* (M', v) \]
\[(M, x) \mapsto (M'[x = v], v) \] (18.34)

\[(x \notin \text{dom}(M)) \]
\[(M, \text{rec } x : \tau \text{ is } e) \mapsto (M[x = e], e) \] (18.35)

Observe that we employ the “black holing” technique to catch ill-defined recursive definitions.

### 18.2.2 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.\footnote{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.}

One consequence, alluded to above, is that recursive type equations have overly rich solutions. For example, in this setting the recursive type equation
\[
data llist = lnil | lcons of int * list
\]
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 computation of another list, so we can easily use recursion equations such as
\[
rec ones is lcons (1, ones)
\]
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 \texttt{cons} evaluate its tail whenever it is applied. This is called a \textit{strictness} annotation. If \texttt{cons} is strict in its second argument, then the equation
\[
rec ones is cons (1, ones)
\]
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 \texttt{lnil} as a value, and regard \texttt{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:
\[
\begin{align*}
\text{lcase} lnil of lnil &\Rightarrow e_n & \text{lcons}(x, y) &\Rightarrow e_c \text{ end} &\Rightarrow e_n
\end{align*}
\]
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 VII

Type Checking
Chapter 19

Type Checking
Chapter 20

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.

20.1 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:

```
fun map (f, l) is
  if null(l) then
    nil
  else
    cons (f (hd l), map (f, (tl l)))
  fi
end
```
We may think of type reconstruction as being performed in several stages. First, we introduce type variables (written \(\alpha, \beta, \gamma\), etc) in place of the omitted type information on functions. These are intended as placeholders for type information that we will appear in the reconstructed typed term. We obtain the following term scheme, a pattern, or scheme, for a typed term:

\[
\begin{align*}
\text{fun} \ \text{map} \ (f:\alpha, \ l:\beta):\gamma \ &\text{is} \\
\quad \text{if} \ \text{null} (l) \ \text{then} \\
\quad \quad \text{nil} \\
\quad \text{else} \\
\quad \quad \text{cons} \ (f \ (\text{hd}(l)), \ \text{map} \ (f, \ \text{tl}(l))) \\
\quad \text{fi} \\
\text{end}
\end{align*}
\]

The type variables \(\alpha\) and \(\beta\) are placeholders for the argument types of \(\text{map}\), and the type variable \(\gamma\) is a placeholder for its result type.

In the next phase we examine the code of \(\text{map}\) 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 \(\text{map}\) to determine the constraints that govern its reconstruction.

1. The expression \(\text{null} (l)\) 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 \(\text{null} (l)\) is \(\text{bool}\).

2. The expression \(\text{nil}\) 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 \(\text{hd} (l)\) and \(\text{tl} (l)\) induce the requirements that \(\beta = \beta''\) list and \(\beta = \beta'''\) list, where \(\beta''\) and \(\beta'''\) are fresh type variables. The type of \(\text{hd} (l)\) is therefore \(\beta''\) and the type of \(\text{tl} (l)\) is \(\beta'''\) list.
4. The expression \( \text{map} \ (f, \ tl(l)) \) 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 \( \text{cons} (f \ (\text{hd}(l)), \text{map}(f, \ tl(l))) \) 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 \text{then} and \text{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:

\[
\begin{align*}
\beta &= \beta' \text{list} \\
\beta &= \beta'' \text{list} \\
\beta &= \beta''' \text{list} \\
\alpha &= \alpha \\
\beta &= \beta''' \text{list} \\
\alpha &= \beta'' \rightarrow \alpha' \\
\gamma &= \gamma' \text{list} \\
\alpha' &= \gamma' \\
\delta \text{list} &= \gamma' \text{list} \\
\gamma &= \delta \text{list}
\end{align*}
\]

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

\[
\begin{align*}
\alpha &= \beta' \rightarrow \gamma' \\
\beta &= \beta' \text{list} \\
\gamma &= \gamma' \text{list}
\end{align*}
\]

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):

\[
\text{fun map } (f: \alpha \rightarrow \beta, x: \alpha \text{list}): \beta \text{list} \text{ is}
\begin{align*}
\text{if null(l) then} \\
\text{nil} \\
\text{else} \\
\text{cons } (f (\text{hd}(l)), \text{map } (f, \text{tl}(l))) \\
\text{fi}
\end{align*}
\text{end}
\]

### 20.2 Type Reconstruction

The syntax of \textit{MinML} is \textit{explicitly typed} (or, more briefly, \textit{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 \textit{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

\[
\text{fun } f(x) \text{ is if } x=0 \text{ then } 1 \text{ else } x \ast f(x-1) \text{ fi end}
\]

rather than in the fully explicit form

\[
\text{fun } f(x: \text{int}): \text{int} \text{ is if } x=0 \text{ then } 1 \text{ else } x \ast f(x-1) \text{ fi end}
\]

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

\[
\text{fun } f(x) \text{ is fun } g(y) \text{ is } x+y \text{ end end}
\]
In explicit form we would have to write
\[
\text{fun } f(x:\text{int}):\text{int} \to \text{int} \text{ is fun } g(y:\text{int}):\text{int} \text{ is } x+y \text{ end end}
\]
As programs grow larger it becomes more and more burdensome to supply the required type information.

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

\[
\begin{align*}
\text{x} & \quad \text{\( \pi \)}} \\
\text{true} & \quad \text{false} \\
\text{u}_1 \ldots \text{u}_n & \quad \text{o}(u_1, \ldots, u_n) \\
\text{u} \quad \text{u}_1 \quad \text{u}_2 & \quad \text{if } u \text{ then } u_1 \text{ else } u_2 \text{ fi} \\
\text{u} & \quad \text{fun } f(x) \text{ is } u \text{ end} \\
\text{u}_1 \quad \text{u}_2 & \quad \text{apply}(u_1, u_2)
\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 \textit{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:

\[
\begin{align*}
x^- & = x \\
n^- & = n \\
\text{true}^- & = \text{true} \\
\text{false}^- & = \text{false} \\
o(e_1, \ldots, e_n)^- & = o(e_1^-, \ldots, e_n^-) \\
\text{if}^- \text{ } e \text{ then } e_1 \text{ else } e_2 \text{ fi}^- & = \text{if } e^- \text{ then } e_1^- \text{ else } e_2^- \text{ fi} \\
\text{fun } f(x:\tau_1) : \tau_2 \text{ is } e \text{ end}^- & = \text{fun } f(x) \text{ is } e^- \text{ end}
\end{align*}
\]

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 \texttt{fun f(x) is x end} can be reconstructed to any typed expression of the form \texttt{fun f(x: \tau): \tau is x end}, where $\tau$ is a type.

How are the typed and untyped syntaxes of \texttt{MinML} related? We will define a four-place elaboration relation $\Gamma \vdash u \leadsto e : \tau$, 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.

\[
\Gamma(x) = \tau \\
\Gamma \vdash x \leadsto x : \tau
\]

(20.7)

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

\[
\Gamma \vdash n \leadsto n : \text{int}
\]

(20.8)

\[
\Gamma \vdash \text{true} \leadsto \text{true} : \text{bool}
\]

(20.9)

\[
\Gamma \vdash \text{false} \leadsto \text{false} : \text{bool}
\]

(20.10)

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

\[
\Gamma \vdash u_1 \leadsto e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash u_n \leadsto e_n : \tau_n \\
\Gamma \vdash o(u_1, \ldots, u_n) \leadsto o(e_1, \ldots, e_n) : \tau
\]

(20.11)

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 \texttt{bool}.

\[
\Gamma \vdash u \leadsto e : \text{bool} \quad \Gamma \vdash u_1 \leadsto e_1 : \tau \quad \Gamma \vdash u_2 \leadsto e_2 : \tau \\
\Gamma \vdash \text{if } u \text{ then } u_1 \text{ else } u_2 \text{ fi} \leadsto \text{if } \tau \text{ then } e_1 \text{ else } e_2 \text{ fi} : \tau
\]

(20.12)
Functions are reconstructed by “guessing” domain and range types, which are then assigned to the function name and parameter while reconstructing the body.

\[
\Gamma[f : \tau_1 \rightarrow \tau_2][x : \tau_1] \vdash u \rightsquigarrow e : \tau_2
\]
\[
\Gamma \vdash \text{fun } f(x) \text{ is } u \text{ end} \rightsquigarrow \text{fun } (x : \tau_1) : \tau_2 \text{ is } e \text{ end} : \tau_1 \rightarrow \tau_2 \quad (20.13)
\]

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.

\[
\Gamma \vdash u_1 \rightsquigarrow e_1 : \tau_2 \rightarrow \tau \quad \Gamma \vdash u_2 \rightsquigarrow e_2 : \tau_2
\]
\[
\Gamma \vdash \text{apply} \ (u_1, u_2) \rightsquigarrow \text{apply} \ (e_1, e_2) : \tau \quad (20.14)
\]

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 20.1
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:

\[
(f n \ f \ i n \ (f n \ x \ i n \ f \ (f \ x) \ e n d) \ e n d) \ (f n \ x \ i n \ x \ e n d)
\]

(Here \(f n x i n e n d\) stands for \(\text{fun } f(x) \text{ is } e \text{ end}\), where \(f \notin \text{FV}(e)\).) 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 20.2**

1. The elaboration relation is **sound**: if $\Gamma \vdash u \leadsto e : \tau$, then $\Gamma \vdash e : \tau$ and $e^- = u$.

2. The elaboration relation is **complete**: if $\Gamma \vdash e : \tau$ and $e^- = u$, then $\Gamma \vdash u \leadsto e : \tau$.

Soundness ensures that given $\Gamma$ and $u$ we can derive *only* well-typed reconstructions of $u$ relative to $\Gamma$. Completeness ensures that we can derive *all* well-typed reconstructions of $u$, relative to $\Gamma$.

**Proof:**

1. By induction on the elaboration relation. We proceed by considering each rule in turn, applying the inductive hypothesis to the premises of the rule. We will illustrate one case here, leaving the rest as an exercise for the reader.

Suppose that

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

where

$$\Gamma[f:\tau_1 \rightarrow \tau_2][x:\tau_1] \vdash u \leadsto e : \tau_2.$$

By induction we know that

$$\Gamma[f:\tau_1 \rightarrow \tau_2][x:\tau_1] \vdash e : \tau_2.$$

Therefore

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

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 \( \Gamma \vdash u \leadsto e : \tau \). 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

\[
\Gamma \left[ f : \tau \right] \left[ x : \tau_1 \right] \vdash u_2 \leadsto e_2 : \tau_2,
\]

for then it follows immediately that

\[
\Gamma \vdash u \leadsto e : \tau,
\]

as required. But this follows immediately by the inductive hypothesis, since \( \Gamma \left[ f : \tau \right] \left[ x : \tau_1 \right] \vdash e_2 : \tau_2 \) and \( u_2 = e_2 \).

\[\square\]

Exercise 20.3

Complete the proof of soundness and completeness of the elaboration relation.

20.3 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 \( \text{int list} \) 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:

\[
\begin{align*}
\alpha & \quad \text{int} \quad \text{bool} \\
\tau_1^* & \tau_2^* \\
\frac{}{\tau_1^* \rightarrow \tau_2^*}
\end{align*}
\]

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:

\[
\begin{align*}
\top & \\
\tau_1^* = \tau_2^* \\
\frac{}{\phi_1 \land \phi_2}
\end{align*}
\]

The constraint generation relation is written $\Gamma^* \vdash u \rightsquigarrow e^* : \tau^* [\phi]$. The intended meaning is that in context $\Gamma^*$ the untyped term $u$ reconstructs to term scheme $e^*$ with type scheme $\tau^*$, subject to the constraints $\phi$. (The exact meaning will be given by Theorem 20.4 below.) This relation will be total function of $\Gamma^*$ and $u$, provided that $FV(u) \subseteq \text{dom}(\Gamma)$ — that is, for every $\Gamma^*$
and $u$ there will be a unique $e^*$ and $\tau^*$ such that $\Gamma^* \vdash u \rightsquigarrow e^* : \tau^* [\phi]$, even if $u$ is ill-formed.

The constraint generation relation is inductively-defined by the following rules.

Variables and constants are straightforward; no constraints are generated.

\[
\Gamma^* \vdash x \rightsquigarrow x : \Gamma^*(x) [\top] \tag{20.19}
\]

\[
\Gamma^* \vdash n \rightsquigarrow n : \text{int} [\top] \tag{20.20}
\]

\[
\Gamma^* \vdash \text{true} \rightsquigarrow n : \text{bool} [\top] \tag{20.21}
\]

\[
\Gamma^* \vdash \text{false} \rightsquigarrow n : \text{bool} [\top] \tag{20.22}
\]

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 $\tau$.

\[
\begin{align*}
\Gamma^* \vdash u_1 \rightsquigarrow e^*_1 : \tau^*_1 [\phi_1] & \quad \cdots & \quad \Gamma^* \vdash u_n \rightsquigarrow e^*_n : \tau^*_n [\phi_n] \\
\Gamma^* \vdash o(u_1, \ldots, u_n) \rightsquigarrow o(e^*_1, \ldots, e^*_n) : \tau & [\phi_1 \land \cdots \land \phi_n \land \tau^*_1 = \tau_1 \land \cdots \land \tau^*_n = \tau_n] 
\end{align*}
\tag{20.23}
\]

Conditionals generate constraints for the sub-expressions, together with the requirement that the type of the test expression be bool and that the "then" and "else" clauses have the same type. The type variable $\alpha$ is uniquely associated with this occurrence of the conditional; it may not otherwise appear in a constraint.

\[
\begin{align*}
\Gamma^* \vdash u \rightsquigarrow e^* : \tau^* [\phi] & \quad \Gamma^* \vdash u_1 \rightsquigarrow e^*_1 : \tau^*_1 [\phi_1] & \quad \Gamma^* \vdash u_2 \rightsquigarrow e^*_2 : \tau^*_2 [\phi_2] \\
\Gamma^* \vdash \text{if } u \text{ then } u_1 \text{ else } u_2 \text{ fi} \rightsquigarrow \text{if } \alpha \text{ e}^* \text{ then } e^*_1 \text{ else } e^*_2 \text{ fi} : \alpha & [\phi \land \phi_1 \land \phi_2 \land \tau^* = \text{bool} \land \alpha = \tau^*_1 \land \alpha = \tau^*_2] 
\end{align*}
\tag{20.24}
\]

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.

\[ \Gamma^*[f : \alpha_1 \rightarrow \alpha_2][x : \alpha_1] \vdash u \leadsto e^* : \tau^*[\phi] \]

\[ \Gamma^* \vdash \text{fun } f(x) \text{ end} \leadsto \text{fun } f(x : \alpha_1) : \alpha_2 \text{ is } u \text{ end} : \alpha_1 \rightarrow \alpha_2 [\phi \land \tau^* = \alpha_2] \]  

(20.25)

Applications are handled by introducing a fresh type variable to stand for the result type of the application.

\[ \Gamma^* \vdash u_1 \leadsto e_1^* : \tau_1^*[\phi_1] \quad \Gamma^* \vdash u_2 \leadsto e_2^* : \tau_2^*[\phi_2] \]

\[ \Gamma^* \vdash \text{apply } (u_1, u_2) \leadsto \text{apply } (e_1^*, e_2^*) : \alpha [\phi_1 \land \phi_2 \land \tau_1^* = \tau_2^* \rightarrow \alpha] \]  

(20.26)

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, \( S(\alpha) = \hat{S}(S(\alpha)) \) for all type variables \( \alpha \). This means that the type variables occurring in the range of \( S \) are not also in its domain. The substitution \( S \) such that \( S(\alpha) = \beta \rightarrow \text{int} \) and \( S(\beta) = \text{bool} \) is not idempotent, since \( \hat{S}(S(\alpha)) = \text{bool} \rightarrow \text{int} \neq \beta \rightarrow \text{int} = S(\alpha) \).

The application of a substitution, \( S \), to a type scheme, \( \tau^* \), written \( \hat{S}(\tau^*) \), is defined by induction on the structure of \( \tau^* \) as follows:

\[ \hat{S}(\alpha) = S(\alpha) \]
\[ \hat{S}(\text{int}) = \text{int} \]
\[ \hat{S}(\text{bool}) = \text{bool} \]
\[ \hat{S}(\tau_1^* \rightarrow \tau_2^*) = \hat{S}(\tau_1^*) \rightarrow \hat{S}(\tau_2^*) \]

In other words, we replace all of the type variables in \( \tau^* \) by their bindings in \( S \). The type scheme \( \hat{S}(\tau^*) \) is called an instance of the type scheme \( \tau^* \). Abusing notation, we write \( \hat{S}(\Gamma^*) \) for the result of applying \( S \) to every type scheme in \( \Gamma^* \). By a similar abuse, we also write \( \hat{S}(e) \) for the application of a substitution \( S \) to a term scheme \( e \), replacing all type variables by their bindings in \( S \). We leave the precise definition to the reader.
20.3 Constraint Generation

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 20.4**

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 \(S\Gamma^*\) contains no type variables. If \(\hat{S}\Gamma^* \vdash u \leadsto e: \tau\), then there exists substitutions \(S_1\) and \(S_2\) such that \(S_2 = S_1 \circ S\), TV(\(\phi\)) \(\subseteq \text{dom}(S_2)\), \(S_2 \models \phi\), 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.

### 20.4 Solving Constraints

solve

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 var-
20.4 Solving Constraints 175

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.

\[(S, \tau_1^* \to \tau_2^* = \sigma_1^* \to \sigma_2^* \land \phi) \mapsto (S, \tau_1^* = \sigma_1^* \land \tau_2^* = \sigma_2^* \land \phi)\] (20.27)

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.)

\[(S, \alpha=\tau^* \land \phi) \mapsto ([\alpha=\tau^*] \circ S, [\tau^*/\alpha] \phi)\] (20.28)

provided that \(\alpha \notin \text{TV}(\tau^*)\). The reason for this restriction is that an equation
such as \(\alpha=\alpha \to \alpha\) 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 irre-
ducible. 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 equa-
tions of one of the following forms:

1. \(\text{int}=\text{bool}\)
2. \(\tau_1^* \to \tau_2^* = \tau^*\), 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 \((S_1, \phi_1) \mapsto (S_2, \phi_2) \mapsto \cdots\). To prove this, we define a “size measure” for states that is decreased by each transition. Examining the transitions in turn, we can make the following observations:

1. The first set of transitions reduces the size of the constraint (measured as the sum of the sizes of the type schemes occurring in it).

2. The second transition also reduces the size of the constraint, since it eliminates two occurrences of the \(\to\) 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 20.5 (Termination)**
There is no infinite sequence of states \((S_1, \phi_1) \mapsto (S_2, \phi_2) \mapsto \cdots\).

The next observation concerns the relationship between successive states. A complete solution for a state \((S, \phi)\) consists of a substitution \(T\) such that \(T = U \circ S\) for some \(U\) and \(T \models \phi\). In other words, \(T\) is an extension of the partial solution \(S\) to a solution for \(\phi\). It is easy to check that stuck states have no complete solution, and that for a final state the partial solution contained within it is a complete solution.

The following technical lemma will prove useful.

**Lemma 20.6**
\[S \models [\tau^*/\alpha]\phi \iff S \circ [\alpha=\tau^*] \models \phi.\]

\(^1\)This is the “dictionary ordering” for two-letter words, where the “letters” are natural numbers.
20.4 Solving Constraints

**Proof:** By induction on the structure of $\phi$. $\blacksquare$

The key property of the transition system is that it *preserves* and *reflects* complete solutions of states.

**Theorem 20.7**

*Suppose that* $(S, \phi) \mapsto (S', \phi')$. *Then* $T$ *is a complete solution for* $(S, \phi)$ *iff* $T$ *is a complete solution for* $(S', \phi')$.

**Proof:** We consider each of the transitions in turn. The case of transitions that drop identical base types or identical type variables are trivial — the presence or absence of such constraints does not change the complete solution. The transition for an equation between function types is similarly straightforward; a complete solution to the left-hand state is clearly a complete solution to the right-hand state, and vice versa.

Finally, let us consider the third transition, for which $\phi = \alpha = \tau^* \land \psi$, $\phi' = ([\tau^*/\alpha]\psi$, 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

$$
\hat{T}(\tau^*) = \hat{U}(\hat{S}(\tau^*)) \quad \text{since } T = U \circ S
= \hat{U}(\tau^*) \quad \text{by state invariants}
= \hat{U}'([\tau^*/\alpha]\tau^*) \quad \text{by definition of } U
= \hat{U}'(\tau^*) \quad \text{by the occurs check, } \alpha \notin \text{TV}(\tau^*)
$$

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 sustitution $U'$ 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

\[
U(\alpha) = \tilde{U}(S(\alpha)) \quad \text{by the state invariants}
= T(\alpha) \quad \text{since } T = U \circ S
= \tilde{T}(\tau^*) \quad \text{since } T \models [\alpha=\tau^*]
= \tilde{U}(S(\tau^*)) \quad \text{since } T = U \circ S
= \hat{U}(\tau^*) \quad \text{by the state invariants}
\]

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

\[
(U' \circ [\alpha=\tau^*])(\alpha) = \hat{U}'(\tau^*) \quad \text{by composition of substitutions}
= \hat{U}(\tau^*) \quad \text{since } \alpha \notin \text{TV}(\tau^*)
= U(\alpha) \quad \text{since } U(\alpha) = \hat{U}(\tau^*)
\]

Therefore, \( U' \circ S' = U \circ [\alpha=\tau^*] \circ S = U \circ S = T \). 

An important corollary of Theorems 20.7 and 20.5 states that the transition system computes principal solutions of constraints.

**Corollary 20.8**

Suppose that \( (I, \phi) \mapsto^* (S, \top) \). The substitution \( T \) solves \( \phi \) iff \( T = U \circ S \) for some substitution \( U \).

The substitution \( S \) is called the principal solution (or most general unifier) for \( \phi \).

**Proof:** It follows from Theorem 20.7 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 20.9 (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.
20.4 Solving Constraints

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 VIII

Polymorphism and Data Abstraction
Chapter 21

Expression Equivalence

One of the beauties of functional programming is the ease with which we may reason about equivalence of expressions. Informally, we say that two expressions $e_1$ and $e_2$ of the same type are equivalent iff replacing $e_1$ by $e_2$ in a complete program doesn’t change its final result. By a “complete program” we mean a closed expression of type `int` or `bool`; the final result of a complete program is therefore a number or a Boolean constant. What is important here is that the final outcome be finitely observable — we can see immediately that the answer is `false` or `17`. Since functions are essentially “infinite” objects (in the sense that the graph of a function on the integers is infinite), we would not regard functions as observable outcomes of a complete program.

We can think of a usage of an expression in a complete program as an “experiment” or “observation” performed on that expression. The idea is that the program “uses” the expression to compute an observable quantity that we can regard as a kind of test performed on that expression. For this reason the notion of equivalence just described is sometimes called observational equivalence — two expressions are observationally equivalent iff any experiment performed on one yields the same observable outcome as the same experiment performed on the other. This relation is also called contextual equivalence to emphasize that equivalence is determined by considering all contexts in which the two expressions might be used to form a complete program. In philosophical logic this relation is known as Leibniz’s Principle of Identity of Indiscernibles — two things are equal iff we cannot tell them apart.

Observational equivalence is very difficult to handle. To determine whether or not $e_1$ and $e_2$ are observationally equivalent requires us to con-
sider all possible programs that use them to compute an integer! This quickly gets out of hand. What we need are alternative criteria for establishing observational equivalences that avoid the need to explicitly consider all possible usages of the expressions in question. Unfortunately, a rigorous development of such alternatives would take us far beyond the scope of the course. We will content ourselves with stating, without proof, a collection of laws of equivalence that are useful for deriving equations between expressions such as these:

1. $x + (y + z)$ is equivalent to $(x + y) + z$.

2. $\text{rev}(\text{rev}(x))$ is equivalent to $x$ (where $\text{rev}$ is the list reversal function).

3. If $v = \text{fun} f (x: \tau_1): \tau_2 \text{ is } e \text{ end}$ and $v, \tau_1$ is a value, then $v(v_1)$ is equivalent to $[v, v_1/f, x]e$.

In the next section we make precise the definition of observational equivalence, and state an alternative characterization of it that is often easier to handle. In the subsequent section we enumerate a collection of valid principles of equivalence for variants of MinML.

### 21.1 Expression Equivalence

We begin with the notion of Kleene equality between complete programs. Kleene equivalence captures what we mean by “same outcome” for complete programs. Two complete programs $p_1$ and $p_2$ are Kleene equivalent, written $p_1 \simeq p_2$, exactly when $p_1 \downarrow v$ iff $p_2 \downarrow v$. That is, either both $p_1$ and $p_2$ diverge (fail to halt), or both converge to the same number or Boolean constant (perhaps in very different ways, using very different amounts of time and space).

A context $C$ is a complete program with a single “hole” into which we may insert an expression. That is, $C$ has the form $\ldots \bullet \ldots$, where the $\bullet$ indicates the “hole” in the program. We write $C\{e\}$ for the result of filling the hole in $C$ with the expression $e$ to obtain $\ldots e \ldots$. The expression $e$ might have free variables that are captured when inserted into the hole. For example, if $C$ is the program context $(\text{fun} f (x: \text{int}): \text{int is } \bullet \text{ end})(3)$ and $e$ is the expression $x + 5$, then $C\{x + 5\}$ is the program

$$(\text{fun} f (x: \text{int}): \text{int is } x + 5 \text{ end})(3).$$
21.1 Expression Equivalence

Suppose that \( \Gamma \vdash e_1 : \tau \) and \( \Gamma \vdash e_2 : \tau \). We define the observational equivalence relation \( \Gamma \vdash e_1 \equiv_{obs} e_2 : \tau \) to hold iff for every program context \( C \) such that \( C\{e_1\} \) and \( C\{e_2\} \) are programs, \( C\{e_1\} \simeq C\{e_2\} \). That is, every use of \( e_1 \) has the same observable outcome as the corresponding use of \( e_2 \), and vice-versa.

As we remarked earlier, it is rather difficult to establish that two expressions are observationally equivalent. A direct application of the definition leaves us no recourse but to consider all possible program contexts \( C \), which quickly gets out of hand.

A more usable characterization of observational equivalence, called applicative equivalence, is defined as follows. For closed expressions \( e_1 \) and \( e_2 \) of type \( \tau \), we define \( e_1 \equiv_{app} e_2 : \tau \) by induction on the structure of \( \tau \) as follows:

- If \( \tau = \text{int}, \) or \( \tau = \text{bool}, \) then \( e_1 \equiv_{app} e_2 : \tau \) iff \( e_1 \simeq e_2. \)

- If \( \tau = \tau_1 \rightarrow \tau_2, \) then \( e_1 \equiv_{app} e_2 : \tau \) iff for every \( v : \tau_1, e_1(v) \equiv_{app} e_2(v) : \tau_2. \)

This relation is extended to open expressions by substitution of closed values of appropriate type for the free variables. Let \( \Gamma \) be the context \( x_1: \tau_1, \ldots, x_n: \tau_n. \)

We define \( \Gamma \vdash e_1 \equiv_{app} e_2 : \tau \) to hold iff

\[
[v_1, \ldots, v_n/x_1, \ldots, x_n]e_1 \equiv_{app} [v_1, \ldots, v_n/x_1, \ldots, x_n]e_2
\]

for every substitution of closed values \( v_1, \ldots, v_n \) of type \( \tau_1, \ldots, \tau_n \) for \( x_1, \ldots, x_n. \)

An important result of Milner's states that applicative and observational equivalence coincide. The proof is non-trivial, and is omitted from this brief exposition.

**Theorem 21.1 (Milner’s Context Lemma)**

\( \Gamma \vdash e_1 \equiv_{obs} e_2 : \tau \) iff \( \Gamma \vdash e_1 \equiv_{app} e_2 : \tau. \)

The point of the context lemma is that we may interpret the relation \( \Gamma \vdash e_1 \equiv_{obs} e_2 : \tau \) as expressing the universally-quantified formula

\[
\forall v_1 : \tau_1 \ldots \forall v_n : \tau_n [v_1, \ldots, v_n/x_1, \ldots, x_n]e_1 \equiv_{app} [v_1, \ldots, v_n/x_1, \ldots, x_n]e_2 : \tau
\]

where the quantifiers range over closed values of the appropriate type. For example,

\[
x : \text{int}, y : \text{int} \vdash x + y \equiv_{obs} y + x : \text{int}
\]

means that for every \( m \) and \( n, m + n \simeq n + m. \)
21.2 Some Laws of Equivalence

In this section we summarize some useful principles of equivalence. These are all valid observational equivalences, but we will not prove this. What is important is to get a feeling for what are some valid principles of equivalence, and how to use them in practice. Since observational and applicative equivalence coincide, we will write \( \Gamma \vdash e_1 \cong e_2 : \tau \) for equality of expressions of type \( \tau \) relative to a context \( \Gamma \), where we tacitly assume that \( \Gamma \vdash e_i : \tau \) for \( i = 1, 2 \).

In the presentation of the rules, we use \( v \) to stand for an open value, either a variable, a constant, or an function expression (perhaps with free variables occurring within it). We admit variables as values because, in a call-by-value language, variables are only ever bound to values, and hence may be taken as standing for a fixed, but unknown, value.

It will be convenient to make use of a designated non-terminating expression of each type \( \tau \), written \( \Omega_\tau \), which is defined to be the expression

\[
(fun\ f(x:int):\tau\ is\ f(x)\ end\)(0).
\]

It is easy to check that \( \Omega_\tau \) diverges (loops forever) when evaluated.

21.2.1 General Laws

First, equivalence is indeed an equivalence relation — it is reflexive, symmetric, and transitive.

\[
\Gamma \vdash e \cong e : \tau \tag{21.1}
\]

\[
\begin{align*}
\Gamma & \vdash e_2 \cong e_1 : \tau \\
\Gamma & \vdash e_1 \cong e_2 : \tau
\end{align*}
\]

\[
\Gamma \vdash e_1 \cong e_2 : \tau \\
\Gamma \vdash e_2 \cong e_3 : \tau
\]

\[
\frac{\Gamma \vdash e_1 \cong e_2 : \tau \quad \Gamma \vdash e_2 \cong e_3 : \tau}{\Gamma \vdash e_1 \cong e_3 : \tau} \tag{21.3}
\]

Second, equivalence is a congruence — we may replace a sub-expression of any expression by an equivalent one to obtain an equivalent expression. This is most easily stated by a collection of rules that ensure that we may replace equivalent sub-expressions to obtain equivalent expressions. We will give just a few of these here; the rest follow a similar pattern.

\[
\begin{align*}
\Gamma & \vdash e_1 \cong e_1' : \tau_1 \\
\vdots
\end{align*}
\]

\[
\frac{\Gamma \vdash e_n \cong e_n' : \tau_n}{\Gamma \vdash o(e_1, \ldots, e_n) \cong o(e_1', \ldots, e_n') : \tau} \tag{21.4}
\]
21.2 Some Laws of Equivalence

\[ \Gamma \vdash e_1 \cong e_1' : \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 \cong e_2' : \tau_2 \]
\[ \Gamma \vdash e_1(e_2) \cong e_1'(e_2') : \tau \]  
(21.5)

\[ \Gamma[f : \tau_1 \rightarrow \tau_2][x : \tau_1] \vdash e \cong e' : \tau_2 \]
\[ \Gamma \vdash \text{fun } f \ (x : \tau_1) \ : \tau_2 \text{ is e end} \cong \text{fun } f \ (x : \tau_1) \ : \tau_2 \text{ is e' end} : \tau_1 \rightarrow \tau_2 \]  
(21.6)

Finally, equivalence is stable under substitution of values for free variables.
\[ \Gamma[x : \tau] \vdash e \cong e' : \tau' \]
\[ \Gamma \vdash [v/x]e \cong [v/x]e' : \tau' \]  
(21.7)

The restriction to values is essential; this rule is not true for general expression substitution! A counterexample is given in the next subsection.

21.2.2 Symbolic Evaluation

Evaluation of an expression in accordance with the rules of the operational semantics results in an equivalent expression. This is called “symbolic evaluation” because the transformations may involve expressions with free variables, which are regarded as values for the purposes of these rules.

An application of a primitive operation may be simplified if we know the values of its arguments:
\[ \Gamma \vdash o(v_1, \ldots, v_n) \cong v : \tau, \]  
(21.8)

where \( v \) is the result of applying \( o \) to \( v_1, \ldots, v_n \).

Similarly, if we know the result of the boolean test, then a conditional may be simplified:
\[ \Gamma \vdash \text{if } \text{true then } e_1 \text{ else } e_2 \text{ fi} \cong e_1 : \tau \]  
(21.9)

\[ \Gamma \vdash \text{if } \text{false then } e_1 \text{ else } e_2 \text{ fi} \cong e_2 : \tau \]  
(21.10)

An application may be simplified if we know the function and the argument is a value. Note that either the function or argument may be open expressions (containing free variables)!
\[ \Gamma \vdash v(v_1) \cong [v, v_1/f, x]e : \tau_2 \]  
(21.11)

where \( v = \text{fun } f \ (x : \tau_1) : \tau_2 \text{ is e end} \).
Exercise 21.2
Using these rules, check that \((fn \times \text{in } 3 \text{ end}) (z)\) is equivalent to 3. Show that \((fn \times \text{in } 3 \text{ end}) (\Omega_{\text{int}})\) is not equivalent to 3. Conclude that substitution of non-values for free variables does not preserve equivalence.

21.2.3 Extensionality
Two functions are equivalent if they are equivalent on all arguments.

\[
\frac{\Gamma[x:\tau_1] \vdash e (x) \cong e' (x) : \tau_2}{\Gamma \vdash e \cong e' : \tau_1 \rightarrow \tau_2} \tag{21.12}
\]

In other words, if two functions are equal for all closed argument values, then they are equal.

21.2.4 Strictness Properties
The evaluation rules of MinML impose a call-by-value evaluation order on function applications and primitive operations. This can be captured equationally by a set of strictness equations that are defined in terms of the divergent expressions \(\Omega_\tau\). We may state that an expression \(e\) of type \(\tau\) diverges as an equation by stating that \(e \cong \Omega_\tau\). The following rules give some conditions under which expressions are divergent.

If any argument of a primitive operation is divergent, so is the whole expression:

\[
\frac{\Gamma \vdash o (e_1, \ldots, e_{i-1}, \Omega_{\tau_i}, e_{i+1}, \ldots, e_n) \cong \Omega_\tau : \tau}{\tag{21.13}}
\]

If the test expression of a conditiona is divergent, so is the conditional.

\[
\frac{\Gamma \vdash \text{if } \Omega_{\text{bool}} \text{ then } e_1 \text{ else } e_2 \text{ fi} \cong \Omega_\tau : \tau}{\tag{21.14}}
\]

If the function or argument of an application is divergent, so is the entire expression:

\[
\frac{\Gamma \vdash \Omega_{\tau_2 \rightarrow \tau} (e_2) \cong \Omega_\tau : \tau}{\tag{21.15}}
\]

\[
\frac{\Gamma \vdash e_1 (\Omega_{\tau_2}) \cong \Omega_\tau : \tau}{\tag{21.16}}
\]
21.2 Some Laws of Equivalence

21.2.5 Arithmetic Laws

Arithmetic and comparison operations behave as expected. For example, addition is associative and commutative, and equality test on integers is an equivalence relation. In general appropriate laws governing the primitive operations on integers hold, provided that they hold mathematically. The same could not be said for floating point (for which addition is not even associative!). Observe that these laws fail, in general, in the presence of effects such as writing to the screen or destructively updating a reference cell! In that case we must restrict attention to values, not general expressions.

\[ \Gamma \vdash e_1 + e_2 \cong e_2 + e_1 : \text{int} \quad (21.17) \]

\[ \Gamma \vdash e_1 + (e_2 + e_3) \cong (e_1 + e_2) + e_3 : \text{int} \quad (21.18) \]

\[ \Gamma \vdash e_1 = e_2 \cong e_2 = e_1 : \text{bool} \quad (21.19) \]

21.2.6 Products

For the extension of MinML to product types,\(^1\) we have the following symbolic evaluation rule:

\[ \Gamma \vdash \text{bind} (x_1 : \tau_1, x_2 : \tau_2) \to (v_1, v_2) \text{ in end} \cong [v_1, v_2/x_1, x_2]e \quad (21.20) \]

We may, in general, replace equals by equals:

\[ \Gamma \vdash e_1 \cong e'_1 : \tau_1 \quad \Gamma \vdash e_2 \cong e'_2 : \tau_2 \]

\[ \Gamma \vdash (e_1, e_2) \cong (e'_1, e'_2) : \tau_1 \times \tau_2 \quad (21.21) \]

\[ \Gamma \vdash e_1 \cong e'_1 : \tau_1 \times \tau_2 \quad \Gamma[x_1 : \tau_1][x_2 : \tau_2] \vdash e_2 \cong e'_2 : \tau' \]

\[ \Gamma \vdash \text{bind} (x_1 : \tau_1, x_2 : \tau_2) \to e_1 \text{ in } e_2 \text{ end} \cong \text{bind} (x_1 : \tau_1, x_2 : \tau_2) \to e'_1 \text{ in } e'_2 \text{ end} \quad (21.22) \]

\(^1\)We do not consider nested or wildcard patterns, for the sake of simplicity. It is a simple matter to extend these rules to the more general case.
Finally, the expected strictness properties hold:

\[ \Gamma \vdash (\Omega_{\tau_1}, e_2) \cong \Omega_{(\tau_1, \tau_2)} \]  
(21.23)

\[ \Gamma \vdash (e_1, \Omega_{\tau_2}) \cong \Omega_{(\tau_1, \tau_2)} \]  
(21.24)

\[ \Gamma \vdash \text{bind}(x_1:\tau_1, x_2:\tau_2) \text{ to } \Omega_{\tau_1 * \tau_2} \text{ in } e \text{ end } \cong \Omega_{\tau'} \]  
(21.25)

### 21.2.7 Lists

We may extend MinML with list types \( \tau \text{list} \) by adding the expressions \( \text{nil}, \text{cons}(e_1, e_2), \text{and listcase } e \text{ of } \text{nil} \Rightarrow e' | \text{cons}(x, y) \Rightarrow e'' \text{ end} \), with the following typing rules:

\[ \Gamma \vdash \text{nil} : \tau \text{list} \]  
(21.26)

\[ \Gamma \vdash \text{cons}(e_1, e_2) : \tau \text{list} \]  
(21.27)

\[ \begin{align*}  
\Gamma \vdash e : \tau \text{list} & \quad \Gamma \vdash e' : \tau' \\
\Gamma[x:\tau][y:\tau \text{list}] \vdash e'' : \tau' 
\end{align*} \]  
(21.28)

The following symbolic evaluation and strictness rules express the evaluation of these constructs:

\[ \Gamma \vdash \text{listcase } e \text{ of } \text{nil} \Rightarrow e' | \text{cons}(x, y) \Rightarrow e'' \text{ end } \cong e' : \tau' \]  
(21.29)

\[ \begin{align*}  
\Gamma \vdash \text{listcase } \text{cons}(v_1, v_2) \text{ of } \text{nil} \Rightarrow e' | \text{cons}(x_1, x_2) \Rightarrow e'' \text{ end } \cong [v_1, v_2/x_1, x_2]e'' : \tau' 
\end{align*} \]  
(21.30)

\[ \Gamma \vdash \text{cons}(\Omega_{\tau}, e) \cong \Omega_{\tau \text{list}} : \tau \text{list} \]  
(21.31)

\[ \Gamma \vdash \text{cons}(e, \Omega_{\tau \text{list}}) \cong \Omega_{\tau \text{list}} : \tau \text{list} \]  
(21.32)
21.2 Some Laws of Equivalence 191

\[ \Gamma \vdash \text{listcase} \Omega_{\tau \text{list}} \begin{cases} \text{of nil} & \Rightarrow e' \\ \text{cons}(x_1, x_2) & \Rightarrow e'' \end{cases} \equiv \Omega_{\tau'} : \tau' \]

(21.33)

Most importantly, we may prove equivalences by induction on the structure of a list. Suppose that \( \Gamma[x:\tau \text{list}] \vdash e_i : \tau' (i = 1, 2) \). To prove that

\[ \Gamma[x:\tau \text{list}] \vdash e_1 \equiv e_2 : \tau' \]

it suffices to show the following two facts:

1. \( \Gamma \vdash [\text{nil}/x]e_1 \equiv [\text{nil}/x]e_2 : \tau' \)

2. For every \( \Gamma \vdash v_h : \tau \) and \( \Gamma \vdash v_t : \tau \text{list} \), if \( \Gamma \vdash [v_t/x]e_1 \equiv [v_t/x]e_2 : \tau' \), then \( \Gamma \vdash [\text{cons}(v_h, v_t)/x]e_1 \equiv [\text{cons}(v_h, v_t)/x]e_2 : \tau' \).

Exercise 21.3
The list append and reversal functions are defined as follows:

\[
\begin{align*}
\text{fun app(l:}\tau \text{list, m:}\tau \text{list):}\tau \text{list is} \\
\text{listcase l of nil} & \Rightarrow m \\
& | h::t \Rightarrow h :: \text{app (t, m)} \end{align*}
\]

\[
\begin{align*}
\text{fun rev (l:}\tau \text{list) is} \\
\text{listcase l of nil} & \Rightarrow \text{nil} \\
& | h::t \Rightarrow \text{app (rev(t), [h])}
\end{align*}
\]

Use list induction and the laws of expression equivalence to prove the following two facts:

1. \( x:\tau \text{list}, y:\tau \text{list} \vdash \text{rev(app(x, y))} \equiv \text{app(rev(y)) rev(x)} : \tau \text{list} \)

2. \( x:\tau \text{list} \vdash \text{rev(rev(x))} \equiv x : \tau \text{list} \)

---

2Officially, the two-argument append function is written using bind as follows:

\[
\begin{align*}
\text{fun app(lm:}\tau \text{list * } \tau \text{list):}\tau \text{list is bind (l:}\tau \text{list, m:}\tau \text{list) to lm in ...end end.}
\end{align*}
\]

We use infix notation for the append function for the sake of clarity.
Chapter 22

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

\[
\text{fun } \ (f:\text{string}\rightarrow\text{int}) : (\text{char}\rightarrow\text{string}) \rightarrow (\text{string}\rightarrow\text{int}) \text{ is}
\]

\[
\text{fun } \ (g:\text{char}\rightarrow\text{string}) : \text{string}\rightarrow\text{int} \text{ is}
\]

\[
\text{fun } \ (x:\text{string}) : \text{int} \text{ is apply}(f, \text{apply}(g, x)),
\]

and here is another

\[
\text{fun } \ (f:\text{float}\rightarrow\text{double}) : (\text{int}\rightarrow\text{float}) \rightarrow (\text{int}\rightarrow\text{double}) \text{ is}
\]

\[
\text{fun } \ (g:\text{int}\rightarrow\text{float}) : \text{int}\rightarrow\text{double} \text{ is}
\]

\[
\text{fun } \ (x:\text{int}) : \text{double} \text{ is apply}(f, \text{apply}(g, x)).
\]

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.

22.1 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:

\[
\begin{align*}
\text{Polytypes} & \quad \sigma ::= \tau \mid \forall t(\sigma) \\
\text{Monotypes} & \quad \tau ::= \ldots \mid t \\
\text{Expressions} & \quad e ::= \ldots \mid \text{Fun } t \text{ in } e \text{ end} \mid \text{inst } (e, \tau) \\
\text{Values} & \quad v ::= \ldots \mid \text{Fun } t \text{ in } e \text{ end}
\end{align*}
\]

The variable \( t \) ranges over a set of type variables, which are written ML-style \( \texttt{'a}, \texttt{'b}, \) 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 \( \text{Fun } t \text{ in } e \text{ end} \) 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:

\[
\begin{align*}
\Delta \vdash \tau \text{ ok} & \quad \tau \text{ is a well-formed type in } \Delta \\
\Gamma \vdash \Delta e : \sigma & \quad e \text{ is a well-formed expression of type } \sigma \text{ in } \Gamma \text{ and } \Delta
\end{align*}
\]
The rules for validity of types are as follows:

\[
\frac{t \in \Delta}{\Delta \vdash t \text{ ok}} 
\]

\[\text{(22.1)}\]

\[
\Delta \vdash \text{int ok} 
\]

\[\text{(22.2)}\]

\[
\Delta \vdash \text{bool ok} 
\]

\[\text{(22.3)}\]

\[
\frac{\Delta \vdash \tau_1 \text{ ok} \quad \Delta \vdash \tau_2 \text{ ok}}{\Delta \vdash \tau_1 \rightarrow \tau_2 \text{ ok}} 
\]

\[\text{(22.4)}\]

\[
\frac{\Delta \cup \{t\} \vdash \sigma \text{ ok} \quad t \notin \Delta}{\Delta \vdash \forall t(\sigma) \text{ ok}} 
\]

\[\text{(22.5)}\]

The auxiliary judgement \(\Delta \vdash \Gamma\) is defined by the following rule:

\[
\frac{\Delta \vdash \Gamma(x) \text{ ok} \quad (\forall x \in \text{dom}(\Gamma))}{\Delta \vdash \Gamma \text{ ok}} 
\]

\[\text{(22.6)}\]

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 \textsc{PolyMinML}; the remaining rules are those of \textsc{MinML}, augmented with a set \(\Delta\) of type variables.

\[
\frac{\Gamma \vdash_{\Delta \cup \{t\}} e : \sigma \quad t \notin \Delta}{\Gamma \vdash_{\Delta} \text{Fun in e end} : \forall t(\sigma)} 
\]

\[\text{(22.7)}\]

\[
\frac{\Gamma \vdash_{\Delta} e : \forall t(\sigma) \quad \Delta \vdash \tau \text{ ok}}{\Gamma \vdash_{\Delta} \text{inst}(e, \tau) : [\tau/t] \sigma} 
\]

\[\text{(22.8)}\]

For example, here is the polymorphic composition function in \textsc{PolyMinML}:

\[
\text{Fun t in} \\
\text{Fun u in} \\
\text{Fun v in} \\
\quad \text{fun \_\,(f:u->v):(t->u)->(t->v) is} \\
\quad \text{fun \_\,(g:t->u):t->v is} \\
\quad \text{fun \_\,(x:t):v is apply(f, apply(g, x))} 
\]
It is easy to check that it has type

$$\forall t (\forall u (\forall v ((u \to v) \to (t \to u) \to (t \to v))))$$.

We will need the following technical lemma stating that typing is preserved under instantiation:

**Lemma 22.1 (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 22.2 (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:

\[
\begin{align*}
\text{inst} (\text{Fun } t \text{ in } e \text{ end}, \tau) & \mapsto [\tau/t]e \\
\text{inst} (e, \tau) & \mapsto \text{inst} (e', \tau)
\end{align*}
\] (22.9)

(22.10)

It is then a simple matter to prove safety for this language.

**Theorem 22.3 (Preservation)**

If \( e : \sigma \) and \( e \mapsto e' \), then \( e' : \sigma \).

The proof is by induction on evaluation.

**Theorem 22.4 (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 evaluation.
First-Class Polymorphism

The syntax given above describes an ML-like treatment of polymorphism, \textit{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 $f \left( \text{Fun}\, t \rightarrow \text{fun}\, \left( x : t \rightarrow t \right) : t \rightarrow t \right)$ is well-formed. However, the application $f \left( \text{fun}\, \left( x : \text{int} \rightarrow \text{int} \right) : \text{int} \rightarrow \text{int} \right)$ 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 $\text{inst} \left( g, \text{int} \right) \left( \text{succ} \right)$ 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:

\begin{verbatim}
let
  val r : ('a -> 'a) ref = ref (fn x:'a => x)
in
  r := (fn x:int => x+1) ; (!r)(true)
end
\end{verbatim}
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 \( \forall t (t \to t \text{ref}) \) of polymorphic functions yielding reference cells containing a function from a type to itself.

2. The type \( \forall t (t \to t \text{ref}) \) 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:

\[
\text{let}
\]
\[
\text{val } r : \text{All } 'a (('a \to 'a) \text{ref}) =
\]
\[
\text{Fun } 'a \text{ in ref (fn } x : 'a \Rightarrow x) \text{ end}
\]
\[
\text{in}
\]
\[
\text{r[\text{int}]} := (\text{fn } x : \text{int} \Rightarrow x+1) ; (\text{!}(r[\text{bool}]))(\text{true})
\]
\[
\text{end}
\]

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 two reference cells, not 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”!).
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

\[
\text{let} \\
\quad \text{val } r : (\text{All } 'a ('a -> 'a)) \text{ ref } = \\
\quad \text{ref (Fun } 'a \text{ in fn } x:'a => x \text{ end)} \\
\quad \text{in} \\
\quad r := (\text{fn } x:\text{int } => x+1) \ ; \ (!r)[\text{bool}](true) \\
\text{end}
\]

then the expression is well-typed, and behaves sanely, precisely because we have assigned to \( r \) a sufficiently polymorphic function.

### 22.2 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

\[
\text{val } I : 'a -> 'a = \text{fn } x => x  \\
\text{val } n : \text{int} = I(I)(3)
\]

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.
22.3 Parametricity

```
val I : ∀(t→t) = Fun in fun (x:t) : t is x end end
val n : int = inst(I, int→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→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 ∀((t→t) 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.

22.3 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 underlie 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 polymorphic type, then the strictures of polymorphism vastly cut down the set of well-typed programs with that type. Since the intended program is one of 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 complexity. 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 languages with mutable references!

22.3.1 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 \( \forall t (t \to t) \). 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 \( \tau \to \tau \) 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 $\forall t (t \rightarrow t)$. 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 $\forall t (t \to (t \to t) \to t)$. What are the interesting functions of the type $N$? Given any type $\tau$, and values $z : \tau$ and $s : \tau \to \tau$, 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

```haskell
Fun t in
  fn z:t in
    fn s:t->t in
      s(s(\ldots s(z)\ldots))
    end
  end
end
```

where there are $n$ occurrences of $s$ in the expression. Observe that if we instantiate $\pi$ at the built-in type int and apply the result to 0 and 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:

```plaintext
fn m:N in
  fn n:N in
    Fun t in
      fn z:t in
        fn s:t->t in
          ...
          end
        end
    end
  end
end
```

To see that it works, instantiate the result at \( \tau \), apply it to \( z \) and \( s \), and observe the result.

### 22.3.2 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 \( \bar{n} \). 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:

\[
e \equiv_{\text{exp}} e' : \tau \iff e \mapsto^* v \iff e' \mapsto^* v.
\]

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:

\[
\begin{align*}
v &\equiv_{\text{val}} v' : \text{bool} & \iff v = v' = \text{true or } v = v' = \text{false} \\
v &\equiv_{\text{val}} v' : \text{int} & \iff v = v' = n \text{ for some } n \geq 0 \\
v &\equiv_{\text{val}} v' : \tau_1 \rightarrow \tau_2 & \iff v_1 \equiv_{\text{val}} v'_1 : \tau_1 \text{ implies } v(v_1) \equiv_{\text{exp}} v'(v'_1) : \tau_2
\end{align*}
\]

The following lemma states two important properties of this notion of equality.

**Lemma 22.5**

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 \textit{a priori} what we mean by equality at that type, precisely because it is “unknown”. Therefore we \textit{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:

$$v \cong_{\text{val}} v' : \forall t(\sigma) \iff \text{for all } \tau \text{ and } \tau', \text{ and all } R : \tau \leftrightarrow \tau', \text{ } v \{ \tau \} \cong_{\text{exp}} v' \{ \tau' \} : \sigma$$

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 \textit{proviso}: when quantifying over relations, we must restrict attention to what are called \textit{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:

\textbf{Theorem 22.6 (Parametricity)}

\textit{If } $e : \sigma$ \textit{is a closed expression, then } $e \cong_{\text{exp}} e : \sigma$.

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.

\textbf{Theorem 22.7}

\textit{If } $f : \forall t(t \rightarrow t)$ \textit{is an interesting value, then } $f \cong_{\text{val}} \text{id} : \forall t(t \rightarrow t)$, \textit{where id is the polymorphic identity function}.

\textbf{Proof:} Suppose that $\tau$ and $\tau'$ are monotypes, and that $R : \tau \leftrightarrow \tau'$. We wish to show that

$$f \{ \tau \} \cong_{\text{exp}} \text{id} \{ \tau' \} : t \rightarrow t,$$
where equality at type \( t \) is taken to be the relation \( R \).

Since \( f \) (and \( \text{id} \)) are interesting, there exists values \( f_\tau \) and \( \text{id}_{\tau'} \) such that

\[
f [\tau] \mapsto^* f_\tau
\]

and

\[
\text{id} [\tau'] \mapsto^* \text{id}_{\tau'}.
\]

We wish to show that

\[
f_\tau \cong_{\text{val}} \text{id}_{\tau'} : t \rightarrow t.
\]

Suppose that \( v_1 \cong_{\text{val}} v'_1 : t \), 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

\[
f_\tau (v_1) \cong_{\text{exp}} \text{id}_{\tau'} (v'_1) : t
\]

By the assumption that \( f \) is interesting (and the fact that \( \text{id} \) is interesting), there exists values \( v_2 \) and \( v'_2 \) such that

\[
f_\tau (v_1) \mapsto^* v_2
\]

and

\[
\text{id}_{\tau'} (v'_1) \mapsto^* v'_2.
\]

By the definition of \( \text{id} \), 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 \( R' : \tau \leftrightarrow \tau \) to be the set \( \{ (v, v) \mid v R v'_1 \} \). Since \( f : \forall t (t \rightarrow t) \), we have by the Parametricity Theorem that \( f \cong_{\text{val}} f : \forall t (t \rightarrow t) \), 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

\[
f_\tau (v_1) \cong_{\text{exp}} \text{id}_{\tau'} (v_1) : t.
\]

Hence \( v_2 R v'_1 \), as required.

You might reasonably wonder, at this point, what the relationship \( f \cong_{\text{val}} \text{id} : \forall t (t \rightarrow t) \) 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 \( f : \forall t (t \rightarrow t) \), we can show that for every \( \tau \) and every \( v : \tau \), \( f [\tau] (v) \) evaluates to \( v \). By the preceding theorem \( f \cong_{\text{val}} \text{id} : \forall t (t \rightarrow t) \), Suppose that \( \tau \) is some monotype and \( v : \tau \) is some closed value. Define the relation \( R : \tau \leftrightarrow \tau \) by

\[
v_1 R v_2 \text{ iff } v_1 = v_2 = v.
\]
Then we have by the definition of equality for polymorphic values

\[ f[\tau](v) \cong_{\exp id[\tau]} (v) : t, \]

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 23

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.

The purpose of this note 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.
23.1 Existential Types

23.1.1 Abstract Syntax

The syntax of PolyMinML is extended with the following constructs:

\[
\begin{align*}
\text{Polytypes} & \quad \sigma ::= \ldots \\
\text{Expressions} & \quad e ::= \ldots \\
\text{Values} & \quad v ::= \ldots
\end{align*}
\]

- Polytypes: \(\sigma\) ::= \ldots \mid \exists t(\sigma)
- Expressions: \(e ::= \ldots\) \mid \text{pack } \tau \text{ with } e \text{ as } \sigma \text{ end}
- Values: \(v ::= \ldots\) \mid \text{pack } \tau \text{ with } v \text{ as } \exists t(\sigma) \text{ end}

The polytype \(\exists t(\sigma)\) is called an existential type. An existential type is the interface of an abstract type. An implementation of the existential type \(\exists t(\sigma)\) is a package value of the form \(\text{pack } \tau \text{ with } v \text{ as } \exists t(\sigma) \text{ end}\) 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 \(\text{open } e_i \text{ as } t \text{ with } x : \sigma \text{ in } e_c \text{ end}\), where \(e_i\) is an implementation of the interface \(\exists t(\sigma)\), and \(e_c\) 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 \(\exists t(\sigma)\) the type variable \(t\) is bound in \(\sigma\), and may be renamed at will to satisfy uniqueness requirements. In an expression of the form \(\text{open } e_i \text{ as } t \text{ with } x : \sigma \text{ in } e_c \text{ end}\) the type variable \(t\) and the ordinary variable \(x\) are bound in \(e_c\) 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.

23.1.2 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:
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
\[ e_q := \text{pack int list with } \nu_q \text{ as } \sigma_q \text{ end,} \]
where
\[ \nu_q := (\text{nil}, (\nu_i, \nu_r)) \]
where \( \nu_i \) and \( \nu_r \) 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:

```ml
local
  open QL
in
  ...
end
```

This corresponds to writing

```ml
open QL as q with <n,i,r> : \tau_q \text{ in } ... \text{ end}
```

in the existential type formalism, renaming variables for convenience.

### 23.1.3 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:

\[
\frac{\Delta \cup \{ t \} \vdash \sigma \text{ ok } t \notin \Delta}{\Delta \vdash \exists \tau(\sigma) \text{ ok}} \quad (23.1)
\]

The requirement \( t \notin \Delta \) may always be met by renaming the bound variable.

The typing rule for packages is as follows:

\[
\frac{\Delta \vdash \tau \text{ ok } \Delta \vdash \exists \tau(\sigma) \text{ ok } \Gamma \vdash \exists t : \tau}{\Gamma \vdash \delta \text{ as } \tau \text{ with } e \text{ as } \exists t(\sigma) \text{ end}} \quad (23.2)
\]

The implementation, \( e \), of the operations “knows” the representation type, \( \tau \), of the ADT.

The typing rule for opening a package is as follows:

\[
\frac{\Delta \vdash \tau' \text{ ok } \Gamma, x: \sigma \vdash \Delta \cup \{ t \} \ e_c : \tau_c \Gamma \vdash \exists \tau(\sigma) \ t \notin \Delta}{\Gamma \vdash \delta \text{ as } \tau \text{ with } x : \sigma \text{ in } e_c \text{ end : } \tau_c} \quad (23.3)
\]

This is a complex rule, so study it carefully! Two things to note:
23.1 Existential Types

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 23.2 below.

23.1.4 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:

\[
\frac{e \mapsto e'}{\text{pack } \tau \text{ with } e \text{ as } \sigma \text{ end } \mapsto \text{pack } \tau \text{ with } e' \text{ as } \sigma \text{ end}} \tag{23.4}
\]

Opening a package begins by evaluating the package expressions:

\[
\frac{e_i \mapsto e_i'}{\text{open } e_i \text{ as } t \text{ with } x : \sigma \text{ in } e_c \text{ end } \mapsto \text{open } e_i' \text{ as } t \text{ with } x : \sigma \text{ in } e_c \text{ end}} \tag{23.5}
\]

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:

\[
\frac{\text{open pack } \tau \text{ with } v \text{ as } \sigma \text{ end as } t \text{ with } x : \sigma \text{ in } e_c \text{ end } \mapsto [\tau, v/t, x]_e}{\text{end}} \tag{23.6}
\]

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.

23.1.5 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 23.1 (Preservation)
If $e : \tau$ and $e \rightarrow e'$, then $e' : \tau$.

Lemma 23.2 (Canonical Forms)
If $v : \exists t(\sigma)$ is a value, then $v = \text{pack } \tau \text{ with } v' \text{ as } \exists t(\sigma) \text{ end}$ for some mono-
type $\tau$ and some value $v' : [\tau/t]\sigma$.

Theorem 23.3 (Progress)
If $e : \tau$ then either $e$ value or there exists $e'$ such that $e \rightarrow e'$.

23.2 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.
23.2 Representation Independence

Since the client $e_c$ of a data abstraction of type $\exists t(\sigma)$ is essentially a polymorphic function of type $\forall t(\sigma \rightarrow \tau_c)$, where $t \notin \text{FTV}(\tau_c)$, it follows from the Parametricity Theorem that

$$[\tau_1, v_1/t, x]e_c \equiv_{\exp} [\tau_2, v_2/t, x]e_c : \tau_c$$

whenever $R$ is such a bisimulation. Consequently,

$$\text{open } e_1 \text{ as } t \text{ with } x: \sigma \text{ in } e_c \text{ end} \equiv_{\exp} \text{open } e_2 \text{ as } t \text{ with } x: \sigma \text{ in } e_c \text{ end} : \tau_c.$$ 

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 $R : \text{int list} \leftrightarrow \text{int list} \ast \text{int list}$ as follows:

$$R = \{ (l, (b,f)) | \exists_{val} b@rev(f) \}$$
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

$$QL.\text{remove}(l) \cong_{\text{exp}} QFB.\text{remove}((b, f)) : \text{int*}t,$$

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

$$\text{nil} \circ \text{rev}(\text{nil}) \equiv_{\text{exp}} \text{nil} : \text{int list},$$

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

$$m : l \equiv_{\text{exp}} (n : b) \circ \text{rev}(f) : \text{int list}.$$
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 \(m \equiv_{\val} n : \text{int}\), 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

\[
\text{QL.remove}(l) \equiv_{\exp} \text{QFB.remove}((b, f)) : \text{int}^*t,
\]
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 \(\text{QFB.remove}\),

\[
\text{QFB.remove}((b, f)) \equiv_{\exp} (n, (b, f')) : \text{int}^*t,
\]
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

Subtyping and Inheritance
Chapter 24

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.

24.1 MinML With Subtyping

We will consider two extensions of MinML with subtyping. The first, MinML with implicit subtyping, is obtained by adding the following rule of implicit subsumption to the typing rules of MinML:

\[
\Gamma \vdash e : \sigma \quad \sigma <: \tau \\
\Gamma \vdash e : \tau
\]

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 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.
Subtyping

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:

\[
\tau \prec \tau \\
\rho \prec \sigma \quad \sigma \prec \tau \quad \Rightarrow \\
\rho \prec \tau
\]

Note that pure MinML is obtained as an instance of MinML^i<; by giving no subtyping rules beyond these two, so that \(\sigma \prec \tau\) 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 \(\sigma \prec \tau\), 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 24.1 (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<;.
Theorem 24.2 (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 \( e \mapsto e' \).

**Proof:** By induction on typing, appealing to the subtyping condition on primitive operations in the case of primitive instruction steps.

24.2 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 \( (\tau) v \mapsto v \), unless otherwise specified.

24.2.1 Arithmetic Subtyping

In informal mathematics we tacitly treat integers as real numbers, even though \( \mathbb{Z} \nsubseteq \mathbb{R} \). 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 \texttt{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 24.4 below.

### 24.2.2 Function Subtyping

Suppose that \texttt{int} <: \texttt{float}. What subtyping relationships, if any, should hold among the following four types?

1. \texttt{int} -> \texttt{int}
2. \texttt{int} -> \texttt{float}
3. \texttt{float} -> \texttt{int}
4. \texttt{float} -> \texttt{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 \( f : \texttt{int} \rightarrow \texttt{int} \). If we apply \( f \) to \( x : \texttt{int} \), the result has type \texttt{int}, and hence, by the arithmetic subtyping axiom, has type \texttt{float}. This suggests that

\[
\texttt{int} \rightarrow \texttt{int} <: \texttt{int} \rightarrow \texttt{float}
\]

is a valid subtype relationship. By similar reasoning, we may derive that

\[
\texttt{float} \rightarrow \texttt{int} <: \texttt{float} \rightarrow \texttt{float}
\]

$^3$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 \( f : \text{float} \to \text{int} \). 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

\[
\text{float} \to \text{int} <: \text{int} \to \text{int}
\]

is a valid subtype relationship. Since \( \text{int} \to \text{int} <: \text{int} \to \text{float} \), it follows that

\[
\text{float} \to \text{int} <: \text{int} \to \text{float}
\]

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:

\[
\frac{\tau_1 <: \sigma_1 \quad \sigma_2 <: \tau_2}{\sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2}
\]

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:

\[
(\tau_1 \to \tau_2) \to \text{fn } x : \tau_1 \text{ in } (\tau_2) \to \text{fn } x : (\sigma_1) \to \text{fn } x : \text{end}
\]

Here \( v \) has type \( \sigma_1 \to \sigma_2 \), \( \tau_1 <: \sigma_1 \), 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.
24.2.3 Product and Record Subtyping

In Chapter 11 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 $\tau_1 \times \cdots \times \tau_n$ 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.i. Similarly, record types have as elements records of the form $\{l_1:e_1, \ldots, l_n:e_n\}$ whose $l$th component is accessed by field selection, e.l.

Using the subsumption principle as a guide, it is natural to consider a tuple type to be a subtype of any of its prefixes:

$$\frac{m > n}{\tau_1 \times \cdots \times \tau_m \prec \prec \tau_1 \times \cdots \times \tau_n}$$

Given a value of type $\tau_1 \times \cdots \times \tau_m$, 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 $\tau_1 \times \cdots \times \tau_m$, 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:

$$\frac{m > n}{\{l_1:\tau_1, \ldots, l_m:\tau_m\} \prec \prec \{l_1:\tau_1, \ldots, l_n:\tau_n\}}$$

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,

$$\forall 1 \leq i \leq n \sigma_i \prec \prec \tau_i \frac{\sigma_1 \times \cdots \times \sigma_n \prec \prec \tau_1 \times \cdots \times \tau_n}{\sigma_1 \times \cdots \times \sigma_n}$$

and

$$\forall 1 \leq i \leq n \sigma_i \prec \prec \tau_i \frac{\forall 1 \leq i \leq n \sigma_i \prec \prec \tau_i}{\{l_1:\sigma_1, \ldots, l_n:\sigma_n\} \prec \prec \{l_1:\tau_1, \ldots, l_n:\tau_n\}}.$$
These are called *depth subtyping* rules for tuples and records, respectively.

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:

\[
m \geq n \\
(\tau_1 \times \cdots \times \tau_n) <v_1, \ldots, v_m> \mapsto <(\tau_1) v_1, \ldots, (\tau_n) v_n>.
\]

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:

\[
<v_1, \ldots, v_i, \ldots> \mapsto v_i.
\]

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:

\[
\{l; v, \ldots\} \mapsto v.
\]

The ellision expresses the independence of field selection from any “extra” fields.

### 24.2.4 Reference Subtyping

Finally, let us consider the reference types of Chapter Chapter 17. What should be the variance rule for reference types? Suppose that \(r\) has type \(\sigma ref\). 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:

\[
\sigma <: \tau \quad \sigma ref <: \tau ref.
\]
On the other hand, if \( \tau <: \sigma \), then we may store a value of type \( \tau \) into \( r \). This suggests that references are contravariant:

\[
\frac{\tau <: \sigma}{\sigma \text{ref} <: \tau \text{ref}}.
\]

Given that we may perform either operation on a reference cell, we must insist that reference types are invariant:

\[
\frac{\sigma <: \tau \quad \tau <: \sigma}{\sigma \text{ref} <: \tau \text{ref}}.
\]

The premise of the rule is often strengthened to the requirement that \( \sigma \) and \( \tau \) be equal:

\[
\frac{\sigma = \tau}{\sigma \text{ref} <: \tau \text{ref}}
\]

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.

### 24.3 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 \( \Gamma \vdash e : \tau \), 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 \( e_1(\varepsilon_2) \). 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 MinML<\text{e} by using the following rule to synthesize a type for a cast:

\[
\Gamma \vdash e \Rightarrow \sigma \quad \sigma \prec \tau \\
\Gamma \vdash (\tau) e \Rightarrow \tau
\]

Extending this method to MinML<\text{j} 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 MinML<\text{j} are as follows:

\[
(\Gamma(x) = \tau) \\
\Gamma \vdash x \Rightarrow \tau \\
\Gamma \vdash n \Rightarrow \text{int} \\
\Gamma \vdash \text{true} \Rightarrow \text{bool} \\
\Gamma \vdash \text{false} \Rightarrow \text{bool} \\
\Gamma \vdash e_1 \Rightarrow \sigma_1 \quad \sigma_1 \prec \tau_1 \\
\vdots \\
\Gamma \vdash e_n \Rightarrow \sigma_n \quad \sigma_n \prec \tau_n \\
\Gamma \vdash o(e_1, \ldots, e_n) \Rightarrow \tau
\]

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.

\[
\Gamma \vdash e \Rightarrow \sigma \\
\Gamma \vdash e_1 \Rightarrow \tau_1 \\
\Gamma \vdash e_2 \Rightarrow \tau_2 \\
\Gamma \vdash \text{if}_{\tau} e \text{then } e_1 \text{ else } e_2 \text{ fi} \Rightarrow \tau
\]

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.\(^4\)

\[
\Gamma[f: \tau_1 \rightarrow \tau_2][x: \tau_1] \vdash e \Rightarrow \tau_2 \\
\Gamma \vdash \text{fun} f(x: \tau_1) : \tau_2 \text{ is } e \text{ end} \Rightarrow \tau_1 \rightarrow \tau_2
\]

\(^4\)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 24.3**
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.

### 24.4 Implementation of Subtyping
#### 24.4.1 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...
the types of \(3\) and \(4\) from \(\text{int}\) to \(\text{float}\). 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:

\[
\begin{align*}
\tau <: \tau \Rightarrow \text{id}_\tau & \quad \rho <: \sigma \Rightarrow v \quad \sigma <: \tau \Rightarrow v' \\
\int <: \text{float} \Rightarrow \text{to_float} & \quad \tau_1 <: \sigma_1 \Rightarrow v_1 \quad \sigma_2 <: \tau_2 \Rightarrow v_2 \\
\sigma_1 \rightarrow \sigma_2 <: \tau_1 \rightarrow \tau_2 \Rightarrow v_1 \rightarrow v_2
\end{align*}
\]

These rules make use of the following auxiliary functions:

1. Primitive conversion: \(\text{to_float}\).
2. Identity: \(\text{id}_\tau = \text{fn}\ x:\tau\ \text{in}\ x\ \text{end}\).
3. Composition: \(v; v' = \text{fn}\ x:\tau\ \text{in}\ v'(v(x))\ \text{end}\).
4. Functions: \(v_1 \rightarrow v_2 = \text{fn}\ f:\sigma_1 \rightarrow \sigma_2\ \text{in}\ \text{fn}\ x:\tau_1\ \text{in}\ v_2(f(v_1(x)))\ \text{end end}\).

The coercion interpretation is type correct. Moreover, there is at most one coercion between any two types:

**Theorem 24.4**

1. If \(\sigma <: \tau \Rightarrow v\), then \(v : \sigma \rightarrow \tau\).

2. If \(\sigma <: \tau \Rightarrow v_1\) and \(\sigma <: \tau \Rightarrow v_2\), then \(v_1 \cong v_2 : \sigma \rightarrow \tau\).

**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) $\text{id} \rightarrow \text{id} \cong \text{id}$.

(c) $(v_1 \rightarrow v_2); (v_1' \rightarrow v_2') \cong (v_1' ; v_1) \rightarrow (v_2; v_2')$.

The type checking relation is augmented with a translation from $\text{MinML}_{<;}$ to pure $\text{MinML}$ that eliminates uses of subsumption by introducing coercions:

$$\Gamma \vdash e : \sigma \bowtie e' \quad \sigma < ; \tau \bowtie v$$

$$\Gamma \vdash e : \tau \bowtie v(e')$$

The remaining rules simply commute with the translation. For example, the rule for function application becomes

$$\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \bowtie e_1' \quad \Gamma \vdash e_2 : \tau_2 \bowtie e_2'$$

$$\Gamma \vdash e_1(e_2) : \tau \bowtie e_1'(e_2')$$

**Theorem 24.5**

1. If $\Gamma \vdash e : \tau \bowtie e'$, then $\Gamma \vdash e' : \tau$ in pure $\text{MinML}$.

2. If $\Gamma \vdash e : \tau \bowtie e_1$ and $\Gamma \vdash e : \tau \bowtie e_2$, then $\Gamma \vdash e_1 \cong e_2 : \tau$ in pure $\text{MinML}$.

3. If $e : \text{int} \bowtie e'$ 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 $\{\}$ 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.
Here, then, are the augmented rules for width and depth subtyping for records:

\[
\frac{m > n}{\{l_1 : \tau_1, \ldots, l_m : \tau_m\} <: \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \rightsquigarrow \text{drop}_{m,n,l,\tau}
\]

\[
\sigma_1 <: \tau_1 \rightsquigarrow v_1 \ldots \sigma_n <: \tau_n \rightsquigarrow v_n
\]

\[
\{l_1 : \sigma_1, \ldots, l_n : \sigma_n\} <: \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \rightsquigarrow \text{copy}_{n,l,\sigma,v}
\]

These rules make use of the following coercion functions:

\[
\text{drop}_{m,n,l,\sigma} = \text{fn} \; x : \{l_1 : \sigma_1, \ldots, l_m : \sigma_m\} \; \text{in} \; \{l_1 : x, l_1, \ldots, l_n : x, l_n\} \; \text{end}
\]

\[
\text{copy}_{n,l,\sigma,v} = \text{fn} \; x : \{l_1 : \sigma_1, \ldots, l_n : \sigma_n\} \; \text{in} \; \{l_1 : v_1(x, l_1), \ldots, l_n : v_n(x, l_n)\} \; \text{end}
\]

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 \(l \cdot e\) becomes a two-stage process:

\[
\text{snd}(e) \ [\text{fst}(e) \ . \ l]
\]

Finally, coerccions copy the view, without modifying the contents. If \(\sigma = \{l_1 : \sigma_1, \ldots, l_n : \sigma_n\}\) and \(\tau = \{l_1 : \text{int}, \ldots, l_n : \text{int}\}\), 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 25

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.

25.1 Inheritance Mechanisms in Java

25.1.1 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.

\(^1\)Fields 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 \texttt{new } \texttt{C(e_1,\ldots,e_n)}, where \texttt{C} is a class and \texttt{e_1,\ldots,e_n} are arguments to the constructor. 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 \texttt{C} are accessed using “dot notation”. If \texttt{f} is a static field of \texttt{C}, a client may refer to it by writing \texttt{C.f}. Similarly, if \texttt{m} is a static method of \texttt{C}, a client may invoke it by writing \texttt{C.m(e_1,\ldots,e_n)}, where \texttt{e_1,\ldots,e_n} 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 \texttt{c} of a class \texttt{C} are similarly accessed using “dot notation”, albeit from the instance, rather than the class. That is, if \texttt{f} is a public dynamic field of \texttt{C}, then \texttt{c.f} refers to the \texttt{f} field of the instance \texttt{c}. Since distinct instances have distinct fields, there is no essential connection between \texttt{c.f} and \texttt{c’.f} when \texttt{c} and \texttt{c’} are

\footnote{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 `this`, which is bound to the instance itself. The methods of an instance may call one another (or themselves) by sending a message to `this`. Although Java defines conventions whereby explicit reference to `this` may be omitted, it is useful to eschew these conveniences and always use `this` to refer to the components of an instance from within code for that instance. We may think of `this` as an implicit argument to all methods that allows the method to access to object itself.

### 25.1.2 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 `this`. If a method $m$ is overridden in a subclass $D$ of a class $C$, then all methods of $D$ that invoke $m$ via `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 `super.m`. The keyword `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 super\((e_1, \ldots, e_n)\), 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.

### 25.1.3 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.
25.1 Inheritance Mechanisms in Java

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 the implements declarations. As we mentioned above, a class may be declared to extend a single class to inherit from it.\(^4\) 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.

\(^4\)Classes that do not specify a superclass implicitly extend the class Object of all objects.
25.2 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.

25.2.1 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, $C \blacktriangleleft I$, is defined to hold exactly when a class $C$ is declared to implement an interface that inherits from $I$. 
The Java subtype relation is inductively defined by the following rules. Subtyping is reflexive and transitive:

\[
\tau <: \tau
\]  
(25.1)

\[
\frac{\tau <: \tau' \quad \tau' <: \tau''}{\tau <: \tau''}
\]  
(25.2)

Arrays are covariant type constructors, in the sense of this rule:

\[
\tau [\ ] <: \tau' [\ ]
\]  
(25.3)

Inheritance implies subtyping:

\[
C <: C'
\]

(25.4)

Implementation implies subtyping:

\[
C \triangleleft I
\]

(25.5)

Every class is a subclass of the distinguished “root” class Object:

\[
\tau <: \text{Object}
\]  
(25.6)

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).

25.2.2 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 \texttt{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 \textit{ad hoc} requirement that either the type of \( e_1 \) be a subtype of the type of \( e_2 \), or \textit{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.

### 25.2.3 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 subsumption, 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 *dynamic dispatch*, or, less perspicuously, *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.\(^5\)

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 *every single array assignment incurs a run-time check* whenever the array contains objects.\(^6\)

### 25.2.4 Casting

A *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 $\textit{Object}$, a general container is essentially a container whose elements are of type $\textit{Object}$. This allows the container operations to be defined once for all

---

\(^5\)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.

\(^6\)Arrays of integers and floats do not incur this overhead, because numbers are not objects.
element types. However, when retrieving an element from a container its static type is \texttt{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 \textit{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 \texttt{Object}, but its true type will usually be some class further down the type hierarchy. Therefore a cast applied to an expression of type \texttt{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. \textit{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. \textit{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. \textit{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.
25.3 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.
Chapter 26

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.

26.1 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
\[ k :::= c(c_x) \{ \text{super } (x); \text{this } f=x; \} \]

Methods
\[ d :::= c m(c_x) \{ \text{return } e; \} \]

Types
\[ \tau :::= c \]

Expressions
\[ e :::= x | e.f | e.m(c_x) | \text{new } c(c_x) | (c) e \]

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 “overbarring” 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 $c f$ 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

$$\text{class } c \text{ extends } c' \{ c f; k d \}$$

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

$$c (c' x', c x) \{ \text{super } (x'); \text{this.f=x}; \}$$

declares the constructor for class $c$ with arguments $c' x', c x$, 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 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 integer and 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 $e$. The expression new $c (e)$ creates a new instance of class $c$, passing arguments $e$ 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 this, standing for the instance itself. We may think of this as a bound variable of the instance, but we will arrange things so that renaming of this is never necessary to avoid conflicts.
26.1 Abstract Syntax

```java
class Pt extends Object {
    int x;
    int y;
    Pt (int x, int y) {
        super(); this.x = x; this.y = y;
    }
    int getx () { return this.x; }
    int gety () { return this.y; }
}
class CPt extends Pt {
    color c;
    CPt (int x, int y, color c) {
        super(x,y);
        this.c = c;
    }
    color getc () { return this.c; }
}

Figure 26.1: A Sample FJ Program
```

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 26.1. 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.
26.2 Static Semantics

The static semantics of FJ is defined by a collection of judgments of the following forms:

- $\tau \prec \tau'$: subtyping
- $\Gamma \vdash e : \tau$: expression typing
- $d \text{ ok in } c$: well-formed method
- $C \text{ ok}$: well-formed class
- $T \text{ ok}$: well-formed class table
- $\text{fields}(c) = c.f$: field lookup
- $\text{type}(m, c) = c \rightarrow c$: method type

The rules defining the static semantics follow.

Every variable must be declared:

$$
\begin{align*}
\Gamma & \vdash x : \tau \\
\Gamma & \vdash \Gamma(x) = \tau
\end{align*}
$$

(26.1)

The types of fields are defined in the class table.

$$
\begin{align*}
\Gamma & \vdash e_0 : c_0 \\
\Gamma & \vdash \text{fields}(c_0) = c.f \\
\Gamma & \vdash e_0.f_i : c_i
\end{align*}
$$

(26.2)

The argument and result types of methods are defined in the class table.

$$
\begin{align*}
\Gamma & \vdash e_0 : c_0 \\
\Gamma & \vdash e : c \\
\Gamma & \vdash \text{type}(m, c_0) = c' \rightarrow c \\
\Gamma & \vdash e_0.m(e) : c
\end{align*}
$$

(26.3)

Instantiation must provide values for all instance variables as arguments to the constructor.

$$
\begin{align*}
\Gamma & \vdash e : c \\
\Gamma & \vdash e < : c' \\
\Gamma & \vdash \text{fields}(c) = c'.f \\
\Gamma & \vdash \text{new } c(e) : c
\end{align*}
$$

(26.4)

All casts are statically valid, but must be checked at run-time.

$$
\begin{align*}
\Gamma & \vdash e_0 : d \\
\Gamma & \vdash (c) e_0 : c
\end{align*}
$$

(26.5)
The subtyping relation is read directly from the class table. Subtyping is the smallest reflexive, transitive relation containing the subclass relation:

\[
\tau <: \tau
\]  

\[
\tau <: \tau' \quad \tau' <: \tau'' \\
\tau <: \tau''
\]  

\[
T(c) = \text{class } c \text{ extends } c'\{\ldots; \ldots\} \\
c <: c'
\]  

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.

\[
k = c(c', x', c) \{\text{super}(x') ; \\text{this}.f = x;\} \\
\text{fields}(c') = c'f' c'' \text{ ok in } c \\
\text{class } c \text{ extends } c'\{c f ; k c''\} \text{ ok}
\]  

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.

\[
T(c) = \text{class } c \text{ extends } c'\{\ldots; \ldots\} \\
\text{type}(m, c') = c \rightarrow c_0 \quad x; c ; \text{this} : c; \vdash e_0 : c_0 \\
c_0 \text{ m(c x) } \{\text{return } e_0 ;\} \text{ ok in } c
\]  

A method table is well-formed iff all of its classes are well-formed:

\[
\forall c \in \text{dom}(T) \quad T(c) \text{ ok} \\
T \text{ ok}
\]  

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:

\[
T \text{ ok} \quad \emptyset \vdash e : \tau \\
(\langle T, e \rangle \text{ ok})
\]  

}\text{ WORKING DRAFT}\}  

\text{ SEPTEMBER 12, 2001}
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:

\[
\text{fields(} \text{Object} \text{)} = \bullet
\]

(26.13)

\[
T(c) = \text{class } c \text{ extends } c' \{c \; f; \; \ldots\} \quad \text{fields}(c') = c' \; f'
\]

\[
\text{fields}(c) = c' \; f', \; c \; f
\]

(26.14)

The type of a method is determined by the following rules:

\[
T(c) = \text{class } c \text{ extends } c' \{\ldots; \ldots \; c''\}
\]

\[
\text{\textcolor{black}{\textcolor{black}{c''}}}_i = c_i \; m \; (c_i \; x) \{\text{return } e;\}
\]

\[
\text{type}(m, c) = c_i \rightarrow c_i
\]

(26.15)

\[
T(c) = \text{class } c \text{ extends } c' \{\ldots; \ldots \; c''\}
\]

\[
m \notin c'' \quad \text{type}(m, c') = c_i \rightarrow c_i
\]

\[
\text{type}(m, c) = c_i \rightarrow c_i
\]

(26.16)

### 26.3 Dynamic Semantics

The dynamic semantics of Featherweight Java (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 \( \text{new } c \; (\epsilon) \), where each \( e_i \) is a value.

\[
\epsilon \; \text{value} \quad \text{new } c \; (\epsilon) \; \text{value}
\]

(26.17)

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.

\[
\text{fields}(c) = c' f' c f \quad \text{value} \quad \text{value} \\
\text{new } c (e', e) . f' i \mapsto e'_i 
\]

(26.18)

Message send replaces \texttt{this} by the instance itself, and replaces the method parameters by their values.

\[
\text{body}(m, c) = x \to e_0 \quad \text{value} \quad \text{value} \\
\text{new } c (e) . m (e') \mapsto [e'/x][\text{new } c (e) / \text{this}]e_0 
\]

(26.20)

Casting checks that the instance is of a sub-class of the target class, and yields the instance.

\[
c <: c' \quad \text{value} \\
(c') \text{ new } c (e) \mapsto \text{ new } c (e) 
\]

(26.21)

These rules determine the order of evaluation:

\[
e_0 \mapsto e'_0 \\
e_0 . f \mapsto e'_0 . f 
\]

(26.22)

\[
e_0 \mapsto e'_0 \\
e_0 . m (e) \mapsto e'_0 . m (e) 
\]

(26.23)

\[
e_0 \text{ value } \quad \text{value} \mapsto e'_0 \\
e_0 . m (e) \mapsto e_0 . m (e') 
\]

(26.24)

\[
e \mapsto e'_0 \\
\text{new } c (e) \mapsto \text{ new } c (e') 
\]

(26.25)

\[
e_0 \mapsto e'_0 \\
(c) e_0 \mapsto (c) e'_0 
\]

(26.26)
Dynamic dispatch makes use of the following auxiliary relation to find the correct method body.

\[
T(c) = \text{class } c \text{ extends } c' \{ \ldots; \ldots d \}
\]
\[
d_i = c_i \cdot m_i(c_i \cdot x) \{ \text{return } e; \}
\]
\[
\text{body}(m_i, c) = x \rightarrow e
\] (26.27)

\[
T(c) = \text{class } c \text{ extends } c' \{ \ldots; \ldots d \}
\]
\[
m \notin d \quad \text{type}(m, c') = x \rightarrow e
\]
\[
\text{body}(m, c) = x \rightarrow e
\] (26.28)

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).

\[
e_1 \text{ value} \ldots e_{i-1} \text{ value} \quad e_i \mapsto e'_i
\]
\[
e_1, \ldots, e_{i-1}, e_i, e_{i+1}, \ldots, e_n \mapsto e_1, \ldots, e_{i-1}, e'_i, e_{i+1}, \ldots, e_n
\] (26.29)

\[
e_1 \text{ value} \ldots e_n \text{ value}
\]
\[
e \text{ value}
\] (26.30)

This completes the dynamic semantics of FJ.

### 26.4 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 26.1 (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 \( \tau' \prec \tau \).

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.
26.5 Acknowledgement

Theorem 26.2 (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 $(c) \text{new } d(e_0)$ with $e_0$ value and $d \not< : c$, 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 26.3 (Canonical Forms)
If $e : c$ and $e$ value, then $e$ has the form $\text{new } d(e_0)$ with $e_0$ value and $d < : c$.

26.5 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 X

Parallelism and Concurrency
Chapter 27

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.

27.1 Bounded 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, and since the nesting depth of tuples is similarly limited (in the absence of recursive types), the degree of parallelism is strictly bounded by the maximum of the product of the tuple width and nesting
depth in the program. In Section 27.3 we will consider mechanisms to support unbounded parallelism determined as a function of the input data.

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 11. However, we now write \( e \mapsto \text{seq} e' \) for the transition relation to stress that this is the sequential semantics. The sequential evaluation rules for pairs are as follows:

\[
\frac{e_1 \mapsto \text{seq} e'_1}{(e_1, e_2) \mapsto \text{seq} (e'_1, e_2)} \tag{27.1}
\]

\[
\begin{align*}
&\text{value} & e_2 & \mapsto \text{seq} e'_2 \\
&\frac{v_1 & \mapsto \text{seq} (v_1, e'_2)}{(v_1, e_2) & \mapsto \text{seq} (v_1, e'_2)} \tag{27.2}
\end{align*}
\]

\[
\begin{align*}
&\text{value} & v_2 & \mapsto \text{seq} e'_2 \\
&\frac{v_1 & \mapsto \text{seq} (v_1, e'_2)}{\text{split } (v_1, v_2) & \text{ as } (x, y) \text{ in } e \text{ end} \mapsto \text{seq} [v_1, v_2/x, y] e} \tag{27.3}
\end{align*}
\]

\[
\frac{e_1 \mapsto \text{seq} e'_1}{\text{split } e_1 \text{ as } (x, y) \text{ in } e \text{ end} \mapsto \text{seq} \text{ split } e'_1 \text{ as } (x, y) \text{ in } e \text{ end}} \tag{27.4}
\]

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:\footnote{It might be preferable to admit progress on either \( e_1 \) or \( e_2 \) alone, without requiring the other to be a value.}

\[
\begin{align*}
&\text{par} & e_1 & \mapsto \text{par} e'_1 \\
&\text{par} & e_2 & \mapsto \text{par} e'_2 \\
&\frac{e_1 & \mapsto \text{par} e'_1 & e_2 & \mapsto \text{par} e'_2}{(e_1, e_2) & \mapsto \text{par} (e'_1, e'_2)} \tag{27.5}
\end{align*}
\]

\[
\begin{align*}
&\text{par} & e_1 & \mapsto \text{par} e'_1 \\
&\text{value} & v_2 & \mapsto \text{par} e'_2 \\
&\frac{v_1 & \mapsto \text{par} (v_1, e'_2)}{(v_1, e_2) & \mapsto \text{par} (v_1, e'_2)} \tag{27.6}
\end{align*}
\]

\[
\begin{align*}
&\text{value} & v_2 & \mapsto \text{par} e'_2 \\
&\frac{v_1 & \mapsto \text{par} (v_1, e'_2)}{(v_1, e_2) & \mapsto \text{par} (v_1, e'_2)} \tag{27.7}
\end{align*}
\]

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”.

**Theorem 27.1**
For every closed, well-typed expression \( e \), \( e \rightarrow^*_{\text{seq}} v \) iff \( e \rightarrow^*_{\text{par}} v \).

**Proof:** For the implication from left to right, it suffices to show that if \( e \rightarrow^*_{\text{seq}} e' \rightarrow^*_{\text{par}} v \), then \( e \rightarrow^*_{\text{par}} v \). This is proved by induction on the sequential evaluation relation. For example, suppose that

\[
(e_1, e_2) \rightarrow_{\text{seq}} (e'_1, e'_2) \rightarrow^*_{\text{par}} (v_1, v_2),
\]

where \( e_1 \rightarrow_{\text{seq}} e'_1 \). By inversion of the parallel evaluation sequence, we have \( e'_1 \rightarrow^*_{\text{par}} v_1 \) and \( e_2 \rightarrow^*_{\text{par}} v_2 \). Hence, by induction, \( e_1 \rightarrow^*_{\text{par}} v_1 \), from which it follows immediately that \( (e_1, e_2) \rightarrow^*_{\text{par}} (v_1, v_2) \). 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 \( e \rightarrow_{\text{par}} e' \rightarrow^*_{\text{seq}} v \), then \( e \rightarrow^*_{\text{seq}} v \). We proceed by induction on the definition of the parallel evaluation relation. For example, suppose that we have

\[
(e_1, e_2) \rightarrow_{\text{par}} (e'_1, e'_2) \rightarrow^*_{\text{seq}} (v_1, v_2)
\]

with \( e_1 \rightarrow_{\text{par}} e'_1 \) and \( e_2 \rightarrow_{\text{par}} e'_2 \). We are to show that \( (e_1, e_2) \rightarrow^*_{\text{seq}} (v_1, v_2) \).

Since \( (e'_1, e'_2) \rightarrow^*_{\text{seq}} (v_1, v_2) \), it follows that \( e'_1 \rightarrow^*_{\text{seq}} v_1 \) and \( e'_2 \rightarrow^*_{\text{seq}} v_2 \). By induction \( e_1 \rightarrow^*_{\text{seq}} v_1 \) and \( e_2 \rightarrow^*_{\text{seq}} v_2 \), 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 affects the efficiency of programs.

### 27.2 Work and Depth

An operational semantics for a language determines the time complexity of programs written in that language. The **sequential complexity** of an expression \( e \) is the number of steps of the sequential semantics required to complete evaluation of \( e \). The **parallel complexity** of \( e \) is the number of steps
in the parallel semantics required to finish execution of \( e \).\(^2\) The sequential complexity of an expression is called the work of that expression, for reasons that we shall see shortly. Its parallel complexity is called the depth because it represents the length of the longest sequential dependency arising during its evaluation. (A sequential dependency between expressions arises whenever the semantics specifies that one expression is to be evaluated before another.)

Let us define the work of a (sequential or parallel) evaluation sequence of an expression to be the total number of primitive instruction steps in that sequence. The work of a sequence of steps is the sum of the work of the individual steps. The work of an axiom step is one; the work of a non-axiom step is the sum of the work of its premise(s). For the sequential semantics, the work of an evaluation sequence is precisely its length, because each rule has at most one premise. For the parallel semantics, the work is, in general, greater than the length of the sequence, because the rule for evaluation of pairs allows us to perform many instructions in a single step.

The work required to evaluate an expression is the same, regardless of whether we use the sequential or the parallel semantics.

**Theorem 27.2**
\[
e \rightarrow_{seq}^* v \text{ with work } w \text{ iff } e \rightarrow_{par}^* v \text{ with work } w.
\]

**Proof:** The proof is very similar to the equivalence theorem of the preceding section, modified to account for the work involved.\(^\blacksquare\) This justifies referring to the sequential complexity of an expression as the work of that expression. Its depth will continue to refer to its parallel complexity, which has no analogue in the sequential semantics.

The notion of a cost semantics introduced in Chapter 10 may be extended to account for parallelism by recording both the work and the depth cost 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:

\[
\frac{e_1 \Downarrow_{w_1,d_1}^v_1 \quad e_2 \Downarrow_{w_2,d_2}^v_2}{(e_1, e_2) \Downarrow_{w_1+w_2, \max(d_1,d_2)}^{v_1,v_2}} \quad (27.8)
\]

The remaining rules are easily derived from the sequential cost semantics, with both work and depth being additively combined at each step.\(^3\)

\(^2\)If we admit unequal progress of parallel computations, then we must revise this definition to consider the least number of steps required to reach a value.

\(^3\)If we choose, we might evaluate arguments of primop’s in parallel, in which case the
The correctness of the cost semantics states that the work and depth costs are consistent with the sequential and parallel complexity, respectively, of the expression.

**Theorem 27.3**
For any closed, well-typed expression $e$, $e \downarrow^{w,d} v$ iff $e \downarrow_{seq}^{w} v$ and $e \downarrow_{par}^{d} v$.

**Proof:** From left to right, we proceed by induction on the cost semantics. For example, we must show that if $e_1 \downarrow_{par}^{d_1} v_1$ and $e_2 \downarrow_{par}^{d_2} v_2$, then $(e_1, e_2) \downarrow_{par}^{d} (v_1, v_2)$, 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 $e_1 \downarrow_{par}^{d_1} v_1$ and $e_2 \downarrow_{par}^{d_2} e'_2 \downarrow_{par}^{d'} v_2$. It follows that

$$(e_1, e_2) \downarrow_{par}^{d_1} (v_1, e'_2) \downarrow_{par}^{d'} (v_1, v_2).$$

For the converse, we proceed by considering work and depth costs separately. For work, we proceed as in Chapter 10. For depth, it suffices to show that if $e \downarrow_{par} e'$ and $e' \downarrow^{d} v$, then $e \downarrow^{d} v$.

For example, suppose that $(e_1, e_2) \downarrow_{par} (e'_1, e'_2)$, with $e_1 \downarrow_{par} e'_1$ and $e_2 \downarrow_{par} e'_2$. 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 $e_1 \downarrow^{d_1} v_1$ and $e_2 \downarrow^{d_2} v_2$ and hence $(e_1, e_2) \downarrow^{d} (v_1, v_2)$, as desired. ■

### 27.3 Unbounded Parallelism

To support *unbounded 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 multi-processor*, or SMP, in constant depth for an arbitrary vector. This cost model corresponds to an idealized parallel computer that can sustain unbounded parallelism. (In Section 27.4 we will briefly discuss the cost of realizing this assumption on a $p$-processor SMP.)

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.

---

*The work component of the cost is suppressed here for the sake of clarity.*
The following primitives are added to MinML to support vectors:

**Types**  \( \tau ::= \tau \text{vector} \)

**Expr's**  
\( e ::= [e_0, \ldots, e_{n-1}] | \text{elt}(e_1, e_2) | \text{size}(e) | \text{index}(e) | \text{map}(e_1, e_2) | \text{update}(e_1, e_2) \)

**Values**  
\( v ::= [v_0, \ldots, v_{n-1}] \)

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:

1. \(\Gamma \vdash e_1 : \tau \quad \cdots \quad \Gamma \vdash e_n : \tau\)
   \(\Gamma \vdash [e_0, \ldots, e_{n-1}] : \tau \text{vector} \) \hspace{1cm} (27.9)

2. \(\Gamma \vdash e_1 : \tau \text{vector} \quad \Gamma \vdash e_2 : \text{int}\)
   \(\Gamma \vdash \text{elt}(e_1, e_2) : \tau\) \hspace{1cm} (27.10)

3. \(\Gamma \vdash e : \tau \text{vector}\)
   \(\Gamma \vdash \text{size}(e) : \text{int}\) \hspace{1cm} (27.11)

4. \(\Gamma \vdash e : \text{int}\)
   \(\Gamma \vdash \text{index}(e) : \tau \text{vector}\) \hspace{1cm} (27.12)

5. \(\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau \text{vector}\)
   \(\Gamma \vdash \text{map}(e_1, e_2) : \tau' \text{vector}\) \hspace{1cm} (27.13)

6. \(\Gamma \vdash e_1 : \tau \text{vector} \quad \Gamma \vdash e_2 : (\text{bool} \times \text{int} \times \tau) \text{vector}\)
   \(\Gamma \vdash \text{update}(e_1, e_2) : \tau \text{vector}\) \hspace{1cm} (27.14)
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 unbounded parallelism:

\[\forall i \in I \; (e_i \mapsto_{\text{par}} e'_i) \quad \forall i \notin I \; (e'_i = e_i \& e_i \text{ value})\]

\[\left[ e_0, \ldots, e_{n-1} \right] \mapsto_{\text{par}} \left[ e'_0, \ldots, e'_{n-1} \right]\]  
(27.15)

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:

\[\text{elt} \left( \left[ v_0, \ldots, v_{n-1} \right], i \right) \mapsto_{\text{par}} v_i\]  
(27.16)

\[\text{size} \left( \left[ v_0, \ldots, v_{n-1} \right] \right) \mapsto_{\text{par}} n\]  
(27.17)

\[\text{index} \left( n \right) \mapsto_{\text{par}} \left[ 0, \ldots, n-1 \right]\]  
(27.18)

\[\text{map} \left( v, \left[ v_0, \ldots, v_{n-1} \right] \right) \mapsto_{\text{par}} \left[ \text{apply} \left( v, v_0 \right), \ldots, \text{apply} \left( v, v_{n-1} \right) \right]\]  
(27.19)

\[\text{update} \left( \left[ v_0, \ldots, v_{n-1} \right], \left[ (b_0, i_0, x_0), \ldots, (b_{k-1}, i_{k-1}, x_{k-1}) \right] \right) \mapsto_{\text{par}} \left[ v'_0, \ldots, v'_{n-1} \right]\]  
(27.20)

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:

\[e_i \mapsto_{\text{seq}} e'_i\]

\[\left[ v_0, \ldots, v_{i-1}, e_i, e_{i+1}, \ldots, e_{n-1} \right] \mapsto \left[ v_0, \ldots, v_{i-1}, e'_i, e_{i+1}, \ldots, e_{n-1} \right]\]  
(27.21)

We write \(e \mapsto_{\text{seq}} e'\) 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.

\[
\forall 0 \leq i < n \quad (e_i \Downarrow^{w_i,d_i} v_i) \quad [e_0, \ldots, e_{n-1}] \Downarrow^{w,d} [v_0, \ldots, v_{n-1}] \tag{27.22}
\]

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.

\[
e_1 \Downarrow^{w_1,d_1} [v_0, \ldots, v_{n-1}] \quad e_2 \Downarrow^{w_2,d_2} i \quad (0 \leq i < n) \quad \text{elt}(e_1, e_2) \Downarrow^{w_1+w_2+1,d_1+d_2+1} v_i \tag{27.23}
\]

Retrieving the size of a vector takes constant work and depth.

\[
e \Downarrow^{w,d} [v_0, \ldots, v_{n-1}] \quad \text{size}(e) \Downarrow^{w+1,d+1} n \tag{27.24}
\]

Creating an index vector takes linear work and constant depth.

\[
e \Downarrow^{w,d} n \quad \text{index}(e) \Downarrow^{w+n,d+1} [0, \ldots, n-1] \tag{27.25}
\]

Mapping a function across a vector takes constant work and depth beyond the cost of the function applications.

\[
e_1 \Downarrow^{w_1,d_1} v \quad e_2 \Downarrow^{w_2,d_2} [v_0, \ldots, v_{n-1}] \quad [\text{apply}(v, v_0), \ldots, \text{apply}(v, v_{n-1})] \Downarrow^{w,d} [v'_0, \ldots, v'_{n-1}] \quad \text{map}(e_1, e_2) \Downarrow^{w_1+w_2+w+1,d_1+d_2+d+1} [v'_0, \ldots, v'_{n-1}] \tag{27.26}
\]

Updating a vector takes linear work and constant depth.

\[
e_1 \Downarrow^{w_1,d_1} [v_0, \ldots, v_{n-1}] \quad e_2 \Downarrow^{w_2,d_2} [(b_1, i_1, x_1), \ldots, (b_k, i_k, x_k)] \quad \text{update}(e_1, e_2) \Downarrow^{w_1+w_2+k+n,d_1+d_2+1} [v'_0, \ldots, v'_{n-1}] \tag{27.27}
\]
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 27.4**

For the extension of MinML with vectors, $e \downarrow^{w,d} v$ iff $e \rightarrow^{\text{par}}_d v$ and $e \rightarrow^{\text{seq}}_w v$.

### 27.4 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 27.5 (Blelloch and Greiner)**

If $e \downarrow^{w,d} v$, then $e$ can be evaluated on an SMP with $p$-processors in time $O\left(\frac{w}{p} + d \log p\right)$.

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, $\log p$, 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 \textit{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 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 simple consequence of Brent’s Theorem, which states that a total workload \( w \) divided into \( d \) parallel steps may be implemented on \( p \) processors in \( O(n/p + d \lg p) \) time. The argument relies on certain assumptions about the SMP, including the ability to perform a parallel fetch-and-add operation in constant time.
Chapter 28

Concurrency

28.1 Introduction
28.2 Shared Variables
28.3 Message Passing
Part XI

Storage Management
Chapter 29

Automatic Storage Management