Programming Languages: Theory and Practice

(WORKING DRAFT OF DECEMBER 17, 2004)

Robert Harper
Carnegie Mellon University

Spring Semester, 2002

Preface

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

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

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

Enjoy!

Contents

Preface ii

I Preliminaries 1

1 Inductive Definitions 2
1.1 Relations and Judgements . . . 2
1.2 Rules and Derivations . . . 2
1.3 Examples of Inductive Definitions . . . 4
1.4 Rule Induction . . . 5
1.5 Iterated and Simultaneous Inductive Definitions . . . 6
1.6 Examples of Rule Induction . . . 7
1.7 Admissible and Derivable Rules . . . 7
1.8 Defining Functions by Rules . . . 9
1.9 Foundations . . . 10

2 Transition Systems 12
2.1 Transition Systems . . . 12
2.2 Exercises . . . 13

II Defining a Language 14

3 Concrete Syntax 15
3.1 Strings . . . 15
3.2 Context-Free Grammars . . . 16
3.3 Ambiguity . . . 18
3.4 Exercises . . . 20

4 Abstract Syntax Trees **21**
4.1 Abstract Syntax Trees 21
4.2 Structural Induction 22
4.3 Parsing 23
4.4 Exercises 25

5 Abstract Binding Trees **26**
5.1 Names 26
5.2 Abstract Syntax With Names 27
5.3 Abstract Binding Trees 27
5.4 Renaming 29
5.5 Structural Induction 31

6 Static Semantics **33**
6.1 Static Semantics of Arithmetic Expressions 33
6.2 Exercises 34

7 Dynamic Semantics **35**
7.1 Structured Operational Semantics 35
7.2 Evaluation Semantics 38
7.3 Relating Transition and Evaluation Semantics 39
7.4 Exercises 40

8 Relating Static and Dynamic Semantics **41**
8.1 Preservation for Arithmetic Expressions 41
8.2 Progress for Arithmetic Expressions 42
8.3 Exercises 42

III A Functional Language **43**

9 MinML, A Minimal Functional Language **44**
9.1 Syntax 44
9.1.1 Concrete Syntax 44
9.1.2 Abstract Syntax 45
9.2 Static Semantics 46
9.3 Properties of Typing 47
9.4 Dynamic Semantics 50
9.5 Properties of the Dynamic Semantics 52

9.6 Exercises 53

10 Type Safety for MinML **54**
10.1 Defining Type Safety 54
10.2 Type Safety of MinML 55
10.3 Run-Time Errors and Safety 58

IV Control and Data Flow **61**

11 Abstract Machines **62**
11.1 Control Flow 63
11.2 Environments 70

12 Continuations **77**
12.1 Informal Overview of Continuations 78
12.2 Semantics of Continuations 82
12.3 Coroutines 85
12.4 Exercises 90

13 Exceptions **91**
13.1 Exercises 97

V Imperative Functional Programming **98**

14 Mutable Storage **99**
14.1 References 99

15 Monads **104**
15.1 Monadic MinML 105
15.2 Reifying Effects 107
15.3 Exercises 108

VI Cost Semantics and Parallelism **109**

16 Cost Semantics **110**
16.1 Evaluation Semantics 110

16.2 Relating Evaluation Semantics to Transition Semantics . . . 111
16.3 Cost Semantics . . . 112
16.4 Relating Cost Semantics to Transition Semantics . . . 113
16.5 Exercises . . . 114

17 Implicit Parallelism **115**
17.1 Tuple Parallelism . . . 115
17.2 Work and Depth . . . 118
17.3 Vector Parallelism . . . 120

18 A Parallel Abstract Machine **124**
18.1 A Simple Parallel Language . . . 124
18.2 A Parallel Abstract Machine . . . 126
18.3 Cost Semantics, Revisited . . . 128
18.4 Provable Implementations (Summary) . . . 129

VII Data Structures and Abstraction **132**

19 Aggregate Data Structures **133**
19.1 Products . . . 134
19.2 Sums . . . 136
19.3 Recursive Types . . . 137

20 Polymorphism **140**
20.1 Polymorphic MinML . . . 141
20.2 ML-style Type Inference . . . 147
20.3 Parametricity . . . 149
20.3.1 Informal Discussion . . . 150
20.3.2 Relational Parametricity . . . 153

21 Data Abstraction **157**
21.1 Existential Types . . . 158
21.1.1 Abstract Syntax . . . 158
21.1.2 Correspondence With ML . . . 159
21.1.3 Static Semantics . . . 160
21.1.4 Dynamic Semantics . . . 161
21.1.5 Safety . . . 162
21.2 Representation Independence . . . 162

VIII Lazy Evaluation 167

22 Lazy Types 168
22.1 Lazy Types in MinML . . . 170
22.1.1 Lazy Lists in an Eager Language . . . 172
22.1.2 Delayed Evaluation and Lazy Data Structures . . . 179

23 Lazy Languages 182
23.0.3 Call-by-Name and Call-by-Need . . . 184
23.0.4 Strict Types in a Lazy Language . . . 186

IX Dynamic Typing 189

24 Dynamic Typing 190
24.1 Dynamic Typing . . . 192
24.2 Implementing Dynamic Typing . . . 193
24.3 Dynamic Typing as Static Typing . . . 195

25 Featherweight Java 199
25.1 Abstract Syntax . . . 199
25.2 Static Semantics . . . 202
25.3 Dynamic Semantics . . . 204
25.4 Type Safety . . . 206
25.5 Acknowledgement . . . 207

X Subtyping and Inheritance 208

26 Subtyping 209
26.1 MinML With Subtyping . . . 209
26.2 Varieties of Subtyping . . . 211
26.2.1 Arithmetic Subtyping . . . 211
26.2.2 Function Subtyping . . . 212
26.2.3 Product and Record Subtyping . . . 214
26.2.4 Reference Subtyping . . . 216
26.3 Type Checking With Subtyping . . . 217
26.4 Implementation of Subtyping . . . 219
26.4.1 Coercions . . . 219

27 Inheritance and Subtyping in Java **223**
27.1 Inheritance Mechanisms in Java 223
27.1.1 Classes and Instances 223
27.1.2 Subclasses 225
27.1.3 Abstract Classes and Interfaces 227
27.2 Subtyping in Java 228
27.2.1 Subtyping 229
27.2.2 Subsumption 230
27.2.3 Dynamic Dispatch 231
27.2.4 Casting 232
27.3 Methodology 233

XI Concurrency **235**

28 Concurrent ML **236**

XII Storage Management **237**

29 Storage Management **238**
29.1 The A Machine 238
29.2 Garbage Collection 242

Part I

Preliminaries

Chapter 1

Inductive Definitions

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

1.1 Relations and Judgements

We take as given the notion of an n-place, or n-ary, relation, R, among $n \geq 1$ objects. A unary (*i.e.*, 1-ary) relation is called a *predicate*, or a *class*.

If R is an n-ary relation and $\vec{x}$ is an n-tuple of objects, the assertion that the objects $\vec{x}$ stand in the relation R, written $R\,(x_1, \ldots, x_n)$ or $(x_1, \ldots, x_n)\,R$, is called a *judgement*. The relation R itself is called a *judgement form*, and the tuple $\vec{x}$ is an *instance* of that judgement form.

If X and Y are classes, then a *function* from X to Y is a binary relation, f, such that for every x such that $X\,x$ there exists a unique y such that $Y\,y$ and $f\,x, y$. A *partial function* from X to Y is a relation such that if $X\,x$, then there is at most one y such that $y\,Y$ and $f\,x, y$.

1.2 Rules and Derivations

An *inductive definition* of an n-ary relation R consists of a collection of *inference rules* of the form

$$\frac{\vec{x}_1\,R \quad \cdots \quad \vec{x}_k\,R}{\vec{x}\,R}.$$

Here $\vec{x}$ and each $\vec{x}_1 \ldots, \vec{x}_k$ are n-tuples of objects, and R is the relation being defined. The judgements above the horizontal line are called the *premises* of the rule, and the judgement below is called the *conclusion* of the rule. If a rule has no premises (*i.e.*, $n = 0$), the rule is called an *axiom*; otherwise it is a *proper rule*.

A relation P is *closed* under a rule

$$\frac{\vec{x}_1\, R \quad \cdots \quad \vec{x}_k\, R}{\vec{x}\, R}$$

iff $\vec{x}\, P$ whenever $\vec{x}_1\, P, \ldots, \vec{x}_k\, P$. The relation P is closed under a set of such rules iff it is closed under each rule in the set. If $\mathcal{S}$ is a set of rules of the above form, then the relation, R, *inductively defined* by the rule set, $\mathcal{S}$, is the strongest (most restrictive) relation closed under $\mathcal{S}$. This means that R is closed under $\mathcal{S}$, and that if P is also closed under $\mathcal{S}$, then $\vec{x}\, R$ implies $\vec{x}\, P$.

If R is inductively defined by a rule set $\mathcal{S}_R$, then $\vec{x}\, R$ holds if and only if it has a *derivation* consisting of a composition of rules in $\mathcal{S}_R$, starting with axioms and ending with $\vec{x}\, R$. A derivation may be depicted as a "stack" of rules of the form

$$\frac{\begin{array}{c}\vdots\\ \mathcal{D}_1\end{array}}{\vec{x}_1\, R} \quad \cdots \quad \frac{\begin{array}{c}\vdots\\ \mathcal{D}_k\end{array}}{\vec{x}_k\, R}$$
$$\vec{x}\, R$$

where

$$\frac{\vec{x}_1\, R \quad \cdots \quad \vec{x}_k\, R}{\vec{x}\, R}$$

is an inference rule, and each $\mathcal{D}_i$ is a derivation of $\vec{x}_i\, R$.

To show that a judgement is derivable we need only find a derivation for it. There are two main methods for finding a derivation, called *forward chaining* and *backward chaining*. Forward chaining starts with the axioms and works forward towards the desired judgement, whereas backward chaining starts with the desired judgement and works backwards towards the axioms.

More precisely, forward chaining search maintains a set of derivable judgements, and continually extends this set by adding to it the conclusion of any rule all of whose premises are in that set. Initially, the set is empty; the process terminates when the desired judgement occurs in the

set. Assuming that all rules are considered at every stage, forward chaining will eventually find a derivation of any derivable judgement, but it is impossible (in general) to decide algorithmically when to stop extending the set and conclude that the desired judgement is not derivable. We may go on and on adding more judgements to the derivable set without ever achieving the intended goal. It is a matter of understanding the global properties of the rules to determine that a given judgement is not derivable.

Forward chaining is undirected in the sense that it does not take account of the end goal when deciding how to proceed at each step. In contrast, backward chaining is goal-directed. Backward chaining search maintains a set of current goals, judgements whose derivations are to be sought. Initially, this set consists solely of the judgement we wish to derive. At each stage, we remove a judgement from the goal set, and consider all rules whose conclusion is that judgement. For each such rule, we add to the goal set the premises of that rule. The process terminates when the goal set is empty, all goals having been achieved. As with forward chaining, backward chaining will eventually find a derivation of any derivable judgement, but there is no algorithmic method for determining in general whether the current goal is derivable. Thus we may futilely add more and more judgements to the goal set, never reaching a point at which all goals have been satisfied.

1.3 Examples of Inductive Definitions

Let us now consider some examples of inductive definitions. The following set of rules, $\mathcal{S}_N$, constitute an inductive definition of the judgement form nat:

$$\frac{}{\mathtt{zero}\ \mathsf{nat}} \qquad \frac{x\ \mathsf{nat}}{\mathtt{succ}(x)\ \mathsf{nat}}$$

The first rule states that $\mathtt{zero}$ is a natural number. The second states that if x is a natural number, so is $\mathtt{succ}(x)$. Quite obviously, the judgement $x\ \mathsf{nat}$ is derivable from rules $\mathcal{S}_N$ iff x is a natural number. For example, here is a derivation of the judgement $\mathtt{succ}(\mathtt{succ}(\mathtt{zero}))\ \mathsf{nat}$:

$$\dfrac{\dfrac{\dfrac{}{\mathtt{zero}\ \mathsf{nat}}}{\mathtt{succ}(\mathtt{zero})\ \mathsf{nat}}}{\mathtt{succ}(\mathtt{succ}(\mathtt{zero}))\ \mathsf{nat}}$$

The following set of rules, $\mathcal{S}_T$, form an inductive definition of the judgement form tree:

$$\frac{}{\mathtt{empty}\ \mathsf{tree}} \qquad \frac{x\ \mathsf{tree} \quad y\ \mathsf{tree}}{\mathtt{node}(x,y)\ \mathsf{tree}}$$

The first states that an empty tree is a binary tree. The second states that a node with two binary trees as children is also a binary tree.

Using the rules $\mathcal{S}_T$, we may construct a derivation of the judgement

$$\mathtt{node}(\mathtt{empty},\mathtt{node}(\mathtt{empty},\mathtt{empty}))\ \mathsf{tree}$$

as follows:

$$\dfrac{\dfrac{}{\mathtt{empty}\ \mathsf{tree}} \quad \dfrac{\dfrac{}{\mathtt{empty}\ \mathsf{tree}} \quad \dfrac{}{\mathtt{empty}\ \mathsf{tree}}}{\mathtt{node}(\mathtt{empty},\mathtt{empty})\ \mathsf{tree}}}{\mathtt{node}(\mathtt{empty},\mathtt{node}(\mathtt{empty},\mathtt{empty}))\ \mathsf{tree}}$$

1.4 Rule Induction

Suppose that the relation R is inductively defined by the rule set $\mathcal{S}_R$. The principle of *rule induction* is used to show $\vec{x}\,P$, whenever $\vec{x}\,R$. Since R is the strongest relation closed under $\mathcal{S}_R$, it is enough to show that P is closed under $\mathcal{S}_R$. Specifically, for every rule

$$\frac{\vec{x}_1\,R \quad \cdots \quad \vec{x}_k\,R}{\vec{x}\,R}$$

in $\mathcal{S}_R$, we must show $\vec{x}\,P$ under the assumptions $\vec{x}_1\,P$, $\ldots$, $\vec{x}_k\,P$. The assumptions $\vec{x}_1\,P$, $\ldots$, $\vec{x}_k\,P$ are the *inductive hypotheses*, and the conclusion is called the *inductive step*, corresponding to that rule.

Rule induction is also called *induction on derivations*, for if $\vec{x}\,R$ holds, then there must be some derivation of it from the rules in $\mathcal{S}_R$. Consider the final rule in the derivation, whose conclusion is $\vec{x}\,R$ and whose premises are $\vec{x}_1\,R$, $\ldots$, $\vec{x}_k\,R$. By induction we have $\vec{x}_1\,P$, $\ldots$, $\vec{x}_k\,P$, and hence to show $\vec{x}\,P$, it suffices to show that $\vec{x}_1\,P$, $\ldots$, $\vec{x}_k\,P$ imply $\vec{x}\,P$.

1.5 Iterated and Simultaneous Inductive Definitions

Inductive definitions are often *iterated*, meaning that one inductive definition builds on top of another. For example, the following set of rules, $\mathcal{S}_L$, defines the predicate list, which expresses that an object is a list of natural numbers:

$$\frac{}{\mathsf{nil}\ \mathsf{list}} \qquad \frac{x\ \mathsf{nat} \quad y\ \mathsf{list}}{\mathsf{cons}(x,y)\ \mathsf{list}}$$

Notice that the second rule makes reference to the judgement nat defined earlier.

It is also common to give a *simultaneous* inductive definition of several relations, $R_1, \ldots, R_k$, by a single set of rules, $\mathcal{S}_{R_1,\ldots,R_k}$. Each rule in the set has the form

$$\frac{\vec{x}_1\ R_{i_1} \quad \cdots \quad \vec{x}_m\ R_{i_m}}{\vec{x}\ R_i}$$

where $1 \leq i_j \leq k$ for each $1 \leq j \leq k$.

The principle of rule induction for such a simultaneous inductive definition gives a sufficient condition for a family $P_1, \ldots, P_k$ of relations such that $\vec{x}\ P_i$ whenever $\vec{x}\ R_i$, for each $1 \leq i \leq k$. To show this, it is sufficient to show for each rule

$$\frac{\vec{x}_1\ R_{i_1} \quad \cdots \quad \vec{x}_m\ R_{i_m}}{\vec{x}\ R_i}$$

if $\vec{x}_1\ P_{i_1}, \ldots, \vec{x}_m\ R_{i_m}$, then $\vec{x}\ R_i$.

For example, consider the rule set, $\mathcal{S}_{eo}$, which forms a simultaneous inductive definition of the judgement forms $x\ \mathsf{even}$, stating that x is an even natural number, and $x\ \mathsf{odd}$, stating that x is an odd natural number.

$$\frac{}{\mathtt{zero}\ \mathsf{even}} \qquad \frac{x\ \mathsf{odd}}{\mathtt{succ}(x)\ \mathsf{even}}$$

$$\frac{x\ \mathsf{even}}{\mathtt{succ}(x)\ \mathsf{odd}}$$

These rules must be interpreted as a simultaneous inductive definition because the definition of each judgement form refers to the other.

1.6 Examples of Rule Induction

Consider the rule set $\mathcal{S}_N$ defined earlier. The principle of rule induction for $\mathcal{S}_N$ states that to show $x\ P$ whenever x nat, it is enough to show

1. zero P;
2. if $y\ P$, then succ$(y)\ P$.

This is just the familiar principal of *mathematical induction*.

The principle of rule induction for $\mathcal{S}_T$ states that if we are to show that $x\ P$ whenever x tree, it is enough to show

1. empty P;
2. if $x_1\ P$ and $x_2\ P$, then node$(x_1, x_2)\ P$.

This is sometimes called the principle of *tree induction*.

The principle of rule induction naturally extends to simultaneous inductive definitions as well. For example, the rule induction principle corresponding to the rule set $\mathcal{S}_{eo}$ states that if we are to show $x\ P$ whenever x even, and $x\ Q$ whenever x odd, it is enough to show

1. zero P;
2. if $x\ P$, then succ$(x)\ Q$;
3. if $x\ Q$, then succ$(x)\ P$.

These proof obligations are derived in the evident manner from the rules in $\mathcal{S}_{eo}$.

1.7 Admissible and Derivable Rules

Let $\mathcal{S}_R$ be an inductive definition of the relation R. There are two senses in which a rule

$$\frac{\vec{x}_1\ R \quad \cdots \quad \vec{x}_k\ R}{\vec{x}\ R}$$

may be thought of as being "valid" for $\mathcal{S}_R$: it can be either *derivable* or *admissible*.

A rule is said to be *derivable* iff there is a derivation of its conclusion from its premises. This means that there is a composition of rules starting with the premises and ending with the conclusion. For example, the following rule is derivable in $\mathcal{S}_N$:

$$\frac{x\ \mathsf{nat}}{\mathtt{succ}(\mathtt{succ}(\mathtt{succ}(x)))\ \mathsf{nat}.}$$

Its derivation is as follows:

$$\dfrac{\dfrac{\dfrac{x\ \mathsf{nat}}{\mathtt{succ}(x)\ \mathsf{nat}}}{\mathtt{succ}(\mathtt{succ}(x))\ \mathsf{nat}}}{\mathtt{succ}(\mathtt{succ}(\mathtt{succ}(x)))\ \mathsf{nat}.}$$

A rule is said to be *admissible* iff its conclusion is derivable from no premises whenever its premises are derivable from no premises. For example, the following rule is *admissible* in $\mathcal{S}_N$:

$$\frac{\mathtt{succ}(x)\ \mathsf{nat}}{x\ \mathsf{nat}\quad}.$$

First, note that this rule is *not* derivable for any choice of x. For if x is $\mathtt{zero}$, then the only rule that applies has no premises, and if x is $\mathtt{succ}(y)$ for some y, then the only rule that applies has as premise $y\ \mathsf{nat}$, rather than $x\ \mathsf{nat}$. However, this rule *is* admissible! We may prove this by induction on the derivation of the premise of the rule. For if $\mathtt{succ}(x)\ \mathsf{nat}$ is derivable from no premises, it can only be by second rule, which means that $x\ \mathsf{nat}$ is also derivable, as required. (This example shows that not every admissible rule is derivable.)

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

$$\frac{}{\mathtt{succ}(\mathtt{junk})\ \mathsf{nat}.}$$

Now it is no longer the case that the rule

$$\frac{\mathtt{succ}(x)\ \mathsf{nat}}{x\ \mathsf{nat}}.$$

is admissible, for if the premise were derived using the additional rule, there is no derivation of $\mathtt{junk}\,\mathsf{nat}$, as would be required for this rule to be admissible.

Since admissibility is sensitive to which rules are *absent*, as well as to which are *present*, a proof of admissibility of a non-derivable rule must, at bottom, involve a use of rule induction. A proof by rule induction contains a case for each rule in the given set, and so it is immediately obvious that the argument is not stable under an expansion of this set with an additional rule. The proof must be reconsidered, taking account of the additional rule, and there is no guarantee that the proof can be extended to cover the new case (as the preceding example illustrates).

1.8 Defining Functions by Rules

A common use of inductive definitions is to give an inductive definition of its graph, a relation, which we then prove is a function. For example, the following rules constitute a definition of the addition function on the natural numbers:

$$\frac{m\ \mathsf{nat}}{A\,(m, \mathtt{zero}, m)} \qquad \frac{A\,(m, n, p)}{A\,(m, \mathtt{succ}(n), \mathtt{succ}(p))}$$

These rules constitute an inductive definition of a three-place relation. But is it a function?

We prove that if $m\ \mathsf{nat}$ and $n\ \mathsf{nat}$, then there exists a unique p such that $A\,(m, n, p)$ by rule induction. There are two cases:

1. From $m\ \mathsf{nat}$ and $\mathtt{zero}\ \mathsf{nat}$, show that there exists a unique p such that $A\,(m, n, p)$. Taking p to be m, it is easy to see that $A\,(m, n, p)$.

2. From $m\ \mathsf{nat}$ and $\mathtt{succ}(n)\ \mathsf{nat}$ and the assumption that there exists a unique p such that $A\,(m, n, p)$, we are to show that there exists a unique q such that $A\,(m, \mathtt{succ}(n), q)$. Taking $q = \mathtt{succ}(p)$ does the job.

Given this, we are entitled to write $A\,(m, n, p)$ as the equation $m + n = p$.

Often the rule format for defining functions illustrated above can be a bit unwieldy. In practice we often present such rules in the more convenient form of *recursion equations*. For example, the addition function may be defined by the following recursion equations:

$$\begin{aligned} m + \mathtt{zero} &=_{df} m \\ m + \mathtt{succ}(n) &=_{df} \mathtt{succ}(m + n) \end{aligned}$$

These equations clearly define a relation, namely the three-place relation given above, but we must prove that they constitute an implicit definition of a function.

1.9 Foundations

The foregoing account of inductive definitions leaves unsaid just what sorts of objects may occur in judgements and rules. For example, the inductive definition of binary trees makes use of `empty` and $\mathtt{node}(-,-)$ without saying precisely what these things really are. More importantly, one may naturally wonder just what sorts of objects are permissible in an inductive definition. This is a matter of foundations that we will only touch on briefly here.

One approach is to simply take as given that the constructions we have mentioned so far are intuitively acceptable, and require no further justification or explanation. More generally, one might permit any form of "finitary" construction, whereby finite entities are built up from other such finite entities by finitely executable processes in some intuitive sense. This is the attitude that we shall adopt in the sequel. We will simply assume without further comment that the constructions we describe are self-evidently meaningful, and do not require further justification.

Another approach is to build on a widely accepted foundation, such as set theory. While this leads to a mathematically satisfactory account, it ignores the very real question of whether and how our constructions can be justified on computational grounds. After all, the study of programming languages is all about things we can implement on a machine! Conventional set theory makes it difficult to discuss such matters. A standard

halfway point is to insist that all work take place in the universe of *hereditarily finite sets*, which are finite sets of finite sets of ... of finite sets. Any construction that can be carried out in this universe is taken as meaningful, and tacitly assumed to be effectively executable on a machine.

An alternative is to work over a universe of *well-founded, finitely branching trees* (which are therefore finite, by König's Lemma). These can be seen as "pictures" of hereditarily finite sets, so there is no essential difference between the two approaches. More radically, one may dispense with the well-foundedness requirement, and permit finitely branching trees of potentially infinite height. Such structures are pictures of *hereditarily finite, non-well-founded sets*, which may be used to model self-referential structures.

Chapter 2

Transition Systems

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

2.1 Transition Systems

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

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

A state $s \in S$ for which there is no $s' \in S$ such that $s \mapsto s'$ is said to be *stuck*. Not all stuck states are final! Non-final stuck states correspond to run-time errors, states for which there is no well-defined next state.

A transition system is *deterministic* iff for every $s \in S$ there exists at most one $s' \in S$ such that $s \mapsto s'$. Most of the transition systems we will consider in this book are deterministic, the notable exceptions being those used to model concurrency.

The *reflexive, transitive closure*, $\stackrel{*}{\mapsto}$, of the transition relation $\mapsto$ is inductively defined by the following rules:

$$\frac{}{s \stackrel{*}{\mapsto} s} \qquad \frac{s \mapsto s' \quad s' \stackrel{*}{\mapsto} s''}{s \stackrel{*}{\mapsto} s''}$$

It is easy to prove by rule induction that $\stackrel{*}{\mapsto}$ is indeed reflexive and transitive.

The *complete* transition relation, $\stackrel{!}{\mapsto}$ is the restriction to $\stackrel{*}{\mapsto}$ to $S \times F$. That is, $s \stackrel{!}{\mapsto} s'$ iff $s \stackrel{*}{\mapsto} s'$ and $s' \in F$.

The *multistep* transition relation, $\stackrel{n}{\longmapsto}$, is defined by induction on $n \geq 0$ as follows:

$$\frac{}{s \stackrel{0}{\longmapsto} s} \qquad \frac{s \mapsto s' \quad s' \stackrel{n}{\longmapsto} s''}{s \stackrel{n+1}{\longmapsto} s''}$$

It is easy to show that $s \stackrel{*}{\mapsto} s'$ iff $s \stackrel{n}{\longmapsto} s'$ for some $n \geq 0$.

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

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

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

2.2 Exercises

1. Prove that $s \stackrel{*}{\mapsto} s'$ iff there exists $n \geq 0$ such that $s \stackrel{n}{\longmapsto} s'$.

Part II

Defining a Language

Chapter 3

Concrete Syntax

The *concrete syntax* of a language is a means of representing expressions as strings, linear sequences of characters (or symbols) that may be written on a page or entered using a keyboard. The concrete syntax usually is designed to enhance readability and to eliminate ambiguity. While there are good methods (grounded in the theory of formal languages) for eliminating ambiguity, improving readability is, of course, a matter of taste about which reasonable people may disagree. Techniques for eliminating ambiguity include precedence conventions for binary operators and various forms of parentheses for grouping sub-expressions. Techniques for enhancing readability include the use of suggestive key words and phrases, and establishment of punctuation and layout conventions.

3.1 Strings

To begin with we must define what we mean by characters and strings. An *alphabet*, Σ, is a set of *characters*, or *symbols*. Often Σ is taken implicitly to be the set of ASCII or UniCode characters, but we shall need to make use of other character sets as well. The judgement form char is inductively defined by the following rules (one per choice of $c \in \Sigma$):

$$\frac{(c \in \Sigma)}{c\ \mathsf{char}}$$

The judgment form string_Σ states that s is a string of characters from Σ.

It is inductively defined by the following rules:

$$\frac{}{\varepsilon\ \mathsf{string}_\Sigma} \qquad \frac{c\ \mathsf{char} \quad s\ \mathsf{string}_\Sigma}{c \cdot s\ \mathsf{string}_\Sigma}$$

In most cases we omit explicit mention of the alphabet, Σ, and just write $s\ \mathsf{string}$ to indicate that s is a string over an implied choice of alphabet.

In practice strings are written in the usual manner, *abcd* instead of the more proper $a \cdot (b \cdot (c \cdot (d \cdot \varepsilon)))$. The function $s_1\hat{}s_2$ stands for string concatenation; it may be defined by induction on s_1. We usually just juxtapose two strings to indicate their concatentation, writing $s_1\ s_2$, rather than $s_1\hat{}s_2$.

3.2 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 Σ of *terminals*.
2. A finite set $\mathcal{N}$ of *non-terminals* that stand for the syntactic categories.
3. A set $\mathcal{P}$ of *productions* of the form $A ::= \alpha$, where A is a non-terminal and α is a string of terminals and non-terminals.

Whenever there is a set of productions

$$\begin{array}{rcl} A & ::= & \alpha_1 \\ & \vdots & \\ A & ::= & \alpha_n. \end{array}$$

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

$$A ::= \alpha_1 \mid \cdots \mid \alpha_n.$$

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

$$A ::= s_1\ A_1\ s_2\ \ldots\ s_{n-1}\ A_n\ s_n$$

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

$$\frac{t_1\ A_1 \quad \ldots \quad t_n\ A_n}{s_1\, t_1\, s_2\, \ldots\, s_{n-1}\, t_n\, s_n\ A}$$

to the rule set R. For each non-terminal A, we say that *s is a string of syntactic category A* iff $s\ A$ is derivable according to the rule set R so obtained.

An example will make these ideas clear. Let us give a grammar defining the syntax of a simple language of arithmetic expressions.

$$\begin{array}{lllll} \textit{Digits} & d & ::= & 0 \mid 1 \mid \cdots \mid 9 \\ \textit{Numbers} & n & ::= & d \mid n\,d \\ \textit{Expressions} & e & ::= & n \mid e{+}e \mid e{*}e \end{array}$$

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:

$$\overline{0\ \mathsf{digit}} \quad \cdots \quad \overline{9\ \mathsf{digit}} \tag{3.1}$$

$$\frac{d\ \mathsf{digit}}{d\ \mathsf{number}} \qquad \frac{n\ \mathsf{number} \quad d\ \mathsf{digit}}{n\,d\ \mathsf{number}} \tag{3.2}$$

$$\frac{n\ \mathsf{number}}{n\ \mathsf{expr}} \tag{3.3}$$

$$\frac{e_1\ \mathsf{expr} \qquad e_2\ \mathsf{expr}}{e_1{+}e_2\ \mathsf{expr}} \tag{3.4}$$

$$\frac{e_1\ \mathsf{expr} \qquad e_2\ \mathsf{expr}}{e_1{*}e_2\ \mathsf{expr}} \tag{3.5}$$

Each syntactic category of the grammar determines a judgement form. For example, the category of expressions corresponds to the judgement form expr, and so forth.

3.3 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:

$$\begin{aligned}
eval_{\mathsf{dig}}(0) &=_{df} 0 \\
&\vdots \\
eval_{\mathsf{dig}}(9) &=_{df} 9 \\
\\
eval_{\mathsf{num}}(d) &=_{df} eval_{\mathsf{dig}}(d) \\
eval_{\mathsf{num}}(n\,d) &=_{df} 10 \times eval_{\mathsf{num}}(n) + eval_{\mathsf{dig}}(d) \\
\\
eval_{\mathsf{exp}}(n) &=_{df} eval_{\mathsf{num}}(n) \\
eval_{\mathsf{exp}}(e_1{+}e_2) &=_{df} eval_{\mathsf{exp}}(e_1) + eval_{\mathsf{exp}}(e_2) \\
eval_{\mathsf{exp}}(e_1{*}e_2) &=_{df} eval_{\mathsf{exp}}(e_1) \times eval_{\mathsf{exp}}(e_2)
\end{aligned}$$

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 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 intepretation.

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

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

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

$$\begin{array}{lllll} \textit{Digits} & d & ::= & 0 \mid 1 \mid \cdots \mid 9 \\ \textit{Numbers} & n & ::= & d \mid n\,d \\ \textit{Expressions} & e & ::= & t \mid t{+}e \\ \textit{Terms} & t & ::= & f \mid f{*}t \\ \textit{Factors} & f & ::= & n \mid (e) \end{array}$$

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

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

the new syntactic categories.

$$\begin{array}{rcl}
\mathit{eval}_{\mathsf{dig}}(0) & =_{\mathit{df}} & 0 \\
& \vdots & \\
\mathit{eval}_{\mathsf{dig}}(9) & =_{\mathit{df}} & 9
\end{array}$$

$$\begin{array}{rcl}
\mathit{eval}_{\mathsf{num}}(d) & =_{\mathit{df}} & \mathit{eval}_{\mathsf{dig}}(d) \\
\mathit{eval}_{\mathsf{num}}(n\,d) & =_{\mathit{df}} & 10 \times \mathit{eval}_{\mathsf{num}}(n) + \mathit{eval}_{\mathsf{dig}}(d)
\end{array}$$

$$\begin{array}{rcl}
\mathit{eval}_{\mathsf{exp}}(t) & =_{\mathit{df}} & \mathit{eval}_{\mathsf{trm}}(t) \\
\mathit{eval}_{\mathsf{exp}}(t{+}e) & =_{\mathit{df}} & \mathit{eval}_{\mathsf{trm}}(t) + \mathit{eval}_{\mathsf{exp}}(e)
\end{array}$$

$$\begin{array}{rcl}
\mathit{eval}_{\mathsf{trm}}(f) & =_{\mathit{df}} & \mathit{eval}_{\mathsf{trm}}(f) \\
\mathit{eval}_{\mathsf{trm}}(f{*}t) & =_{\mathit{df}} & \mathit{eval}_{\mathsf{trm}}(f) \times \mathit{eval}_{\mathsf{trm}}(t)
\end{array}$$

$$\begin{array}{rcl}
\mathit{eval}_{\mathsf{trm}}(n) & =_{\mathit{df}} & \mathit{eval}_{\mathsf{num}}(n) \\
\mathit{eval}_{\mathsf{trm}}(\mathtt{(}e\mathtt{)}) & =_{\mathit{df}} & \mathit{eval}_{\mathsf{exp}}(e)
\end{array}$$

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

3.4 Exercises

Chapter 4

Abstract Syntax Trees

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 *abstract syntax trees*, or *ast's*, over a set of operators. Abstract syntax avoids the ambiguities of concrete syntax by employing operators that determine the outermost form of any given expression, rather than relying on parsing conventions to disambiguate strings.

4.1 Abstract Syntax Trees

Abstract syntax trees are constructed from other abstract syntax trees by combining them with an *constructor*, or *operator*, of a specified *arity*. The arity of an operator, o, is the number of arguments, or sub-trees, required by o to form an ast. A *signature* is a mapping assigning to each $o \in \mathrm{dom}(\Omega)$ its arity $\Omega(o)$. The judgement form term_Ω is inductively defined by the following rules:

$$\frac{t_1\ \mathsf{term}_\Omega \quad \cdots \quad t_n\ \mathsf{term}_\Omega \quad (\Omega(o) = n)}{o(t_1, \ldots, t_n)\ \mathsf{term}_\Omega}$$

Note that we need only one rule, since the arity of o might well be zero, in which case the above rule has no premises.

For example, the following signature, Ω_{expr}, specifies an abstract syn-

tax for the language of arithmetic expressions:

Operator	*Arity*
$\texttt{num}[n]$	0
`plus`	2
`times`	2

Here n ranges over the natural numbers; the operator $\texttt{num}[n]$ is the nth numeral, which takes no arguments. The operators `plus` and `times` take two arguments each, as might be expected. The abstract syntax of our language consists of those t such that $t\,\mathsf{term}_{\Omega_{expr}}$.

Specializing the rules for abstract syntax trees to the signature Ω_{expr} (and suppressing explicit mention of it), we obtain the following inductive definition:

$$\frac{(n \in \mathbb{N})}{\texttt{num}[n]\,\mathsf{term}} \quad \frac{t_1\,\mathsf{term} \quad t_2\,\mathsf{term}}{\texttt{plus}(t_1,t_2)\,\mathsf{term}} \quad \frac{t_1\,\mathsf{term} \quad t_2\,\mathsf{term}}{\texttt{times}(t_1,t_2)\,\mathsf{term}}$$

It is common to abuse notation by presenting these rules in grammatical form, as follows:

$$\textit{Terms} \quad t \;::=\; \texttt{num}[n] \mid \texttt{plus}(t_1,t_2) \mid \texttt{times}(t_1,t_2)$$

Although it has the form of a grammar, this description is to be understood as defining the *abstract*, not the *concrete*, syntax of the language.

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

4.2 Structural Induction

The principal of rule induction for abstract syntax 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.

In the case of arithmetic expressions the principal of structural induction is as follows. To show that $t\ J$ is evident whenever $t\,\mathsf{term}$, it is enough to show:

1. $\mathtt{num}[n]\ J$ for every $n \in \mathbb{N}$;
2. if $t_1\ J$ and $t_2\ J$, then $\mathtt{plus}(t_1, t_2)\ J$;
3. if $t_1\ J$ and $t_2\ J$, then $\mathtt{times}(t_1, t_2)\ J$;

For example, we may prove that the equations

$$\begin{array}{rcl} \mathit{eval}(\mathtt{num}[n]) & =_{\mathit{df}} & n \\ \mathit{eval}(\mathtt{plus}(t_1, t_2)) & =_{\mathit{df}} & \mathit{eval}(t_1) + \mathit{eval}(t_2) \\ \mathit{eval}(\mathtt{times}(t_1, t_2)) & =_{\mathit{df}} & \mathit{eval}(t_1) \times \mathit{eval}(t_2) \end{array}$$

determine a function *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 $\mathit{eval}(t) = n$.

4.3 Parsing

The process of translation from concrete to abstract syntax is called *parsing*. Typically the concrete syntax is specified by an inductive definition defining the grammatical strings of the language, and the abstract syntax is given by an inductive definition of the abstract syntax trees that constitute the language. In this case it is natural to formulate parsing as an inductively defined function mapping concrete the abstract syntax. Since parsing is to be a function, there is exactly one abstract syntax tree corresponding to a well-formed (grammatical) piece of concrete syntax. Strings that are not derivable according to the rules of the concrete syntax are not grammatical, and can be rejected as ill-formed.

For example, consider the language of arithmetic expressions discussed in Chapter 3. Since we wish to define a function on the concrete syntax, it should be clear from the discussion in Section 3.3 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:[1]

$$\begin{array}{rcl}
\mathit{parse}_{\mathsf{dig}}(0) & = & 0 \\
& \vdots & \\
\mathit{parse}_{\mathsf{dig}}(9) & = & 9
\end{array}$$

$$\begin{array}{rcl}
\mathit{parse}_{\mathsf{num}}(d) & = & \mathtt{num}[\mathit{parse}_{\mathsf{dig}}(d)] \\
\mathit{parse}_{\mathsf{num}}(n\,d) & = & \mathtt{num}[10 \times k + \mathit{parse}_{\mathsf{dig}}\,d],\ \mathit{where}\ \mathit{parse}_{\mathsf{num}}\,n = \mathtt{num}[k]
\end{array}$$

$$\begin{array}{rcl}
\mathit{parse}_{\mathsf{exp}}(t) & = & \mathit{parse}_{\mathsf{trm}}(t) \\
\mathit{parse}_{\mathsf{exp}}(t\mathtt{+}e) & = & \mathtt{plus}(\mathit{parse}_{\mathsf{trm}}(t), \mathit{parse}_{\mathsf{exp}}(e))
\end{array}$$

$$\begin{array}{rcl}
\mathit{parse}_{\mathsf{trm}}(f) & = & \mathit{parse}_{\mathsf{fct}}(f) \\
\mathit{parse}_{\mathsf{trm}}(f\mathtt{*}t) & = & \mathtt{times}(\mathit{parse}_{\mathsf{fct}}(f), \mathit{parse}_{\mathsf{trm}}(t))
\end{array}$$

$$\begin{array}{rcl}
\mathit{parse}_{\mathsf{fct}}(n) & = & \mathit{parse}_{\mathsf{num}}(n) \\
\mathit{parse}_{\mathsf{fct}}(\mathtt{(}e\mathtt{)}) & = & \mathit{parse}_{\mathsf{exp}}(e)
\end{array}$$

It is a simple matter to prove by rule induction that these rules define a function from grammatical strings to abstract syntax.

There is one remaining issue about this specification of the parsing function that requires further remedy. Look closely at the definition of the function $\mathit{parse}_{\mathsf{num}}$. 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:

$$\mathit{Numbers} \quad n \quad ::= \quad d \mid d\,n$$

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*

[1] These are, of course, definitional equalities, but here (and elsewhere) we omit the subscript "*df*" for perspicuity.

descent parser. This is the method of choice for hand-coded parsers. Parser generators, which automatically create parsers from grammars, make use of a different technique that is more efficient, but much harder to implement by hand.

4.4 Exercises

1. Give a concrete and (first-order) abstract syntax for a language.
2. Write a parser for that language.

Chapter 5

Abstract Binding Trees

Abstract syntax trees make explicit the hierarchical relationships among the components of a phrase by abstracting out from irrelevant surface details such as parenthesization. *Abstract binding trees*, or *abt*'s, go one step further and make explicit the binding and scope of identifiers in a phrase, abstracting from the "spelling" of bound names so as to focus attention on their fundamental role as abstract names.

5.1 Names

Names are widely used in programming languages: names of variables, names of fields in structures, names of communication channels, names of locations in the heap, and so forth. Names have no structure beyond their identity; in particular, the "spelling" of a name is of no significance. Consequently, we shall treat names as atoms, and abstract away any internal structure such as the "spelling" of the name. We assume given a judgement form name such that $x\ \mathsf{name}$ for infinitely many x.

We will often make use of n-tuples of names $\vec{x} = x_1, \ldots, x_n$, where $n \geq 0$. The constituent names of $\vec{x}$ are written x_i, where $1 \leq i \leq n$, and we tacitly assume that if $i \neq j$, then x_i and x_j are distinct names. In any such tuple the names x_i are tacitly assumed to be pairwise distinct. If $\vec{x}$ is an n-tuple of names, we define its *length*, $|\vec{x}|$, to be n.

5.2 Abstract Syntax With Names

Suppose that we enrich the language of arithmetic expressions given in Chapter 4 with a means of binding the value of an arithmetic expression to an identifier for use within another arithmetic expression. To support this we extend the abstract syntax with two additional constructs:[1]

$$\frac{x\ \mathsf{name}}{\mathtt{id}(x)\ \mathsf{term}_\Omega} \qquad \frac{x\ \mathsf{name} \quad t_1\ \mathsf{term}_\Omega \quad t_2\ \mathsf{term}_\Omega}{\mathtt{let}(x,t_1,t_2)\ \mathsf{term}_\Omega}$$

The ast $\mathtt{id}(x)$ represents a use of a name, x, as a variable, and the ast $\mathtt{let}(x,t_1,t_2)$ introduces a name, x, that is to be bound to (the value of) t_1 for use within t_2.

The difficulty with abstract syntax trees is that they make no provision for specifying the binding and scope of names. For example, in the ast $\mathtt{let}(x,t_1,t_2)$, the name x is available for use within t_2, but not within t_1. That is, the name x is bound by the `let` construct for use within its scope, the sub-tree t_2. But there is nothing intrinsic to the ast that makes this clear. Rather, it is a condition imposed on the ast "from the outside", rather than an intrinsic property of the abstract syntax. Worse, the informal specification is vague in certain respects. For example, what does it mean if we nest bindings for the same identifier, as in the following example?

$$\mathtt{let}(x,t_1,\mathtt{let}(x,id(x),id(x)))$$

Which occurrences of x refer to which bindings, and why?

5.3 Abstract Binding Trees

Abstract binding trees are a generalization of abstract syntax trees that provide intrinsic support for binding and scope of names. Abt's are formed from names and abstractors by *operators*. Operators are assigned (generalized) *arities*, which are finite sequences of *valences*, which are natural numbers. Thus an arity has the form $(m_1,\ldots,m_k)$, specifying an operator with k arguments, each of which is an abstractor of the specified valence. Abstractors are formed by associating zero or more names with an abt; the

[1] One may also devise a concrete syntax, for example writing `let` x `be` e_1 `in` e_2 for the binding construct, and a parser to translate from the concrete to the abstract syntax.

valence is the number of names attached to the abstractor. The present notion of arity generalizes that given in Chapter 4 by observing that the arity n from Chapter 4 becomes the arity $(0, \ldots, 0)$, with n copies of 0.

This informal description can be made precise by a simultaneous inductive definition of two judgement forms, abt_Ω and abs_Ω, which are parameterized by a signature assigning a (generalized) arity to each of a finite set of operators. The judgement $t\ \mathsf{abt}_\Omega$ asserts that t is an abt over signature Ω, and the judgement $\vec{x}.t\ \mathsf{abs}_\Omega^n$ states that $\vec{x}.t$ is an abstractor of valence n.

$$\frac{x\ \mathsf{name}}{x\ \mathsf{abt}_\Omega} \qquad \frac{\beta_1\ \mathsf{abs}_\Omega^{m_1} \quad \cdots \quad \beta_k\ \mathsf{abs}_\Omega^{m_k}}{o(\beta_1, \ldots, \beta_k)\ \mathsf{abt}_\Omega}\ (\Omega(o) = (m_1, \ldots, m_k))$$

$$\frac{t\ \mathsf{abt}_\Omega}{t\ \mathsf{abs}_\Omega^0} \qquad \frac{x\ \mathsf{name} \quad \beta\ \mathsf{abs}_\Omega^n}{x.\beta\ \mathsf{abs}_\Omega^{n+1}}$$

An abstractor of valence n has the form $x_1.x_2.\ldots x_n.t$, which is sometimes abbreviated $\vec{x}.t$, where $\vec{x} = x_1, \ldots, x_n$. We tacitly assume that no name is repeated in such a sequence, since doing so serves no useful purpose. Finally, we make no distinction between an abstractor of valence zero and an abt.

The language of arithmetic expressions may be represented as abstract binding trees built from the following signature.

Operator	*Arity*
`num`$[n]$	$()$
`plus`	$(0,0)$
`times`	$(0,0)$
`let`	$(0,1)$

The arity of the "let" operator makes clear that no name is bound in the first position, but that one name is bound in the second.

5.4 Renaming

The *free names*, $FN(t)$, of an abt, t, is inductively defined by the following recursion equations:

$$\begin{array}{rcl} FN(x) & =_{df} & \{\, x \,\} \\ FN(o(\beta_1, \ldots, \beta_k)) & =_{df} & FN(\beta_1) \cup \cdots \cup FN(\beta_k) \end{array}$$

$$FN(\vec{x}.t) =_{df} FN(t) \setminus \vec{x}$$

Thus, the set of *free names* in t are those names occurring in t that do not lie within the scope of an abstractor.

We say that the abt t_1 *lies apart* from the abt t_2, written $t_1 \# t_2$, whenever $FN(t_1) \cap FN(t_2) = \emptyset$. In particular, $x \# t$ whenever $x \notin FN(t)$, and $x \# y$ whenever x and y are distinct names. We write $\vec{t} \# \vec{u}$ to mean that $t_i \# u_j$ for each $1 \leq i \leq |\vec{t}|$ and each $1 \leq j \leq |\vec{u}|$.

The operation of *swapping* one name, x, for another, y, within an abt, t, written $[x \leftrightarrow y]t$, is inductively defined by the following recursion equations:

$$\begin{array}{rcl} [x \leftrightarrow y]x & =_{df} & y \\ [x \leftrightarrow y]y & =_{df} & x \\ [x \leftrightarrow y]z & =_{df} & z \quad (\mathit{if}\ z \# x, z \# y) \\ [x \leftrightarrow y]o(\beta_1, \ldots, \beta_k) & =_{df} & o([x \leftrightarrow y]\beta_1, \ldots, [x \leftrightarrow y]\beta_k) \end{array}$$

$$[x \leftrightarrow y](\vec{x}.t) =_{df} [x \leftrightarrow y]\vec{x}.[x \leftrightarrow y]t$$

In the above equations, and elsewhere, if $\vec{t}$ is an n-tuple of abt's, then $[x \leftrightarrow y]\vec{t}$ stands for the n-tuple $[x \leftrightarrow y]t_1, \ldots, [x \leftrightarrow y]t_n$.

A chief characteristic of a binding operator is that the choice of bound names does not matter. This is captured by treating as equivalent and two abt's that differ only in the choice of bound names, but are otherwise identical. This relation is called, for historical reasons, α-equivalence. It is inductively defined by the following rules:

$$\frac{}{x =_\alpha x\ \mathsf{abt}_\Omega} \qquad \frac{\beta_1 =_\alpha \gamma_1\ \mathsf{abs}_\Omega^{m_1} \quad \cdots \quad \beta_k =_\alpha \gamma_k\ \mathsf{abs}_\Omega^{m_k} \quad (\Omega(o) = (m_1, \ldots, m_k))}{o(\beta_1, \ldots, \beta_k) =_\alpha o(\gamma_1, \ldots, \gamma_k)\ \mathsf{abt}_\Omega}$$

$$\frac{t =_\alpha u\ \mathsf{abt}_\Omega}{t =_\alpha u\ \mathsf{abs}^0_\Omega} \qquad \frac{\beta =_\alpha \gamma\ \mathsf{abs}^n_\Omega}{x.\beta =_\alpha x.\gamma\ \mathsf{abs}^{n+1}_\Omega} \qquad \frac{x \# y \qquad y \# \beta \quad [x{\leftrightarrow}y]\beta =_\alpha \gamma\ \mathsf{abs}^n_\Omega}{x.\beta =_\alpha y.\gamma\ \mathsf{abs}^{n+1}_\Omega}$$

In practice we abbreviate these relations to $t =_\alpha u$ and $\beta =_\alpha \gamma$, respectively.

As an exercise, check the following α-equivalences and inequivalences using the preceding definitions specialized to the signature given earlier.

$$\begin{array}{rcl}
\mathtt{let}(x, x.x) & =_\alpha & \mathtt{let}(x, y.y) \\
\mathtt{let}(y, x.x) & =_\alpha & \mathtt{let}(y, y.y) \\
\mathtt{let}(x, x.x) & \neq_\alpha & \mathtt{let}(y, y.y) \\
\mathtt{let}(x, x.\mathtt{plus}(x, y)) & =_\alpha & \mathtt{let}(x, z.\mathtt{plus}(z, y)) \\
\mathtt{let}(x, x.\mathtt{plus}(x, y)) & \neq_\alpha & \mathtt{let}(x, y.\mathtt{plus}(y, y))
\end{array}$$

It may be shown by rule induction that α-equivalence is, in fact, an equivalence relation (*i.e.*, it is reflexive, symmetric, and transitive). For transitivity we must show simultaneously that (i) $t =_\alpha u$ and $u =_\alpha v$ imply $t =_\alpha v$, and (ii) $\beta =_\alpha \gamma$ and $\gamma =_\alpha \delta$ imply $\beta =_\alpha \delta$. Let us consider the case that $\beta = x.\beta'$, $\gamma = y.\gamma'$, and $\delta = z.\delta'$. Suppose that $\beta =_\alpha \gamma$ and $\gamma =_\alpha \delta$. We are to show that $\beta =_\alpha \delta$, for which it suffices to show either

1. $x = z$ and $\beta' =_\alpha \delta'$, or
2. $x \# z$ and $z \# \beta'$ and $[x{\leftrightarrow}z]\beta' =_\alpha \delta'$.

There are four cases to consider, depending on the derivation of $\beta =_\alpha \gamma$ and $\gamma =_\alpha \delta$. Consider the case in which $x \# y$, $y \# \beta'$, and $[x{\leftrightarrow}y]\beta' =_\alpha \gamma'$ from the first assumption, and $y \# z$, $z \# \gamma'$, and $[y{\leftrightarrow}z]\gamma' =_\alpha \delta'$ from the second. We proceed by cases on whether $x = z$ or $x \# z$.

1. Suppose that $x = z$. Since $[x{\leftrightarrow}y]\beta' =_\alpha \gamma'$ and $[y{\leftrightarrow}z]\gamma' =_\alpha \delta'$, it follows that $[y{\leftrightarrow}z][x{\leftrightarrow}y]\beta' =_\alpha \delta'$. But since $x = z$, we have $[y{\leftrightarrow}z][x{\leftrightarrow}y]\beta' = [y{\leftrightarrow}x][x{\leftrightarrow}y]\beta' = \beta'$, so we have $\beta' =_\alpha \delta'$, as desired.

2. Suppose that $x \# z$. Note that $z \# \gamma'$, so $z \# [x{\leftrightarrow}y]\beta'$, and hence $z \# \beta'$ since $x \# z$ and $y \# z$. Finally, $[x{\leftrightarrow}y]\beta' =_\alpha \gamma'$, so $[y{\leftrightarrow}z][x{\leftrightarrow}y]\beta' =_\alpha \delta'$, and hence $[x{\leftrightarrow}z]\beta' =_\alpha \delta'$, as required.

This completes the proof of this case; the other cases are handled similarly.

From this point onwards we identify any two abt's t and u such that $t =_\alpha u$. This means that an abt implicitly stands for its α-equivalence class,

and that we tacitly assert that all operations and relations on abt's respects α-equivalence. Put the other way around, any operation or relation on abt's that fails to respect α-equivalence is illegitimate, and therefore ruled out of consideration. In this way we ensure that the choice of bound names does not matter.

One consequence of this policy on abt's is that whenever we encounter an abstract $x.\beta$, we may assume that x is fresh in the sense that it may be implicitly chosen to not occur in any specified finite set of names. For if X is such a finite set and $x \in X$, then we may choose another representative of the α-equivalence class of $x.\beta$, say $x'.\beta'$, such that $x' \notin X$, meeting the implicit assumption of freshness.

5.5 Structural Induction

The principle of structural induction for ast's generalizes to abt's, subject to freshness conditions that ensure bound names are not confused. To show simultaneously that

1. for all t such that $t\ \mathsf{abt}_\Omega$, the judgement $t\ J$ holds, and
2. for every β and n such that $\beta\ \mathsf{abs}^n_\Omega$, the judgement $\beta\ K^n$ holds,

then it is enough to show the following:

1. For any name x, the judgement $x\ J$ holds.
2. For each operator, o, of arity $(m_1, \ldots, m_k)$, if $\beta_1\ K^{m_1}$ and $\ldots$ and $\beta_k\ K^{m_k}$, then $o(\beta_1, \ldots, \beta_k)\ J$.
3. If $t\ J$, then $t\ K^0$.
4. For some/any "fresh" name x, if $\beta\ K^n$, then $x.\beta\ K^{n+1}$.

In the last clause the choice of x is immaterial: *some* choice of fresh names is sufficient iff *all* choices of fresh names are sufficient. The precise meaning of "fresh" is that the name x must not occur free in the judgement K.

Another example of a proof by structural induction is provided by the definition of *substitution*. The operation $[\vec{x} \leftarrow \vec{u}]t$ performs the *simultaneous*, *capture-avoiding* substitution of u_i for free occurrences of x_i in t for each

$1 \leq i \leq |\vec{x}| = |\vec{u}|$. It is inductively defined by the following recursion equations:

$$\begin{array}{rcl} [\vec{x}\leftarrow\vec{u}]x_i & =_{df} & t_i \\ [\vec{x}\leftarrow\vec{u}]y & =_{df} & y \quad (\mathit{if}\ y \# \vec{x}) \\ [\vec{x}\leftarrow\vec{u}]o(\beta_1,\ldots,\beta_k) & =_{df} & o([\vec{x}\leftarrow\vec{u}]\beta_1,\ldots,[\vec{x}\leftarrow\vec{u}]\beta_k) \end{array}$$

$$[\vec{x}\leftarrow\vec{u}](\vec{y}.t) \quad =_{df} \quad \vec{y}.[\vec{x}\leftarrow\vec{u}]t \quad (\mathit{if}\ \vec{y} \# \vec{u})$$

The condition on the last clause can always be met, by the freshness assumption. More precisely, we may prove by structural induction that substitution is *total* in the sense that for any $\vec{u}$, $\vec{x}$, and t, there exists a unique t' such that $[\vec{x}\leftarrow\vec{u}]t = t'$. The crucial point is that the principal of structural induction for abstract binding trees permits us to choose bound names to lie apart from $\vec{x}$ when applying the inductive hypothesis.

Chapter 6

Static Semantics

The *static semantics* of a language determines which pieces of abstract syntax (represented as ast's or abt's) are *well-formed* according to some context-sensitive criteria. A typical example of a well-formedness constraint is *scope resolution*, the requirement that every name be declared before it is used.

6.1 Static Semantics of Arithmetic Expressions

We will give an inductive definition of a static semantics for the language of arithmetic expressions that performs scope resolution. A *well-formedness judgement* has the form $\Gamma \vdash e\ \mathsf{ok}$, where Γ is a finite set of variables and e is the abt representation of an arithmetic expression. The meaning of this judgement is that e is an arithmetic expression all of whose free variables are in the set Γ. Thus, if $\emptyset \vdash e\ \mathsf{ok}$, then e has no unbound variables, and is therefore suitable for evaluation.

$$\frac{(x \in \Gamma)}{\Gamma \vdash x\ \mathsf{ok}} \qquad \frac{(n \geq 0)}{\Gamma \vdash \mathtt{num}[n]\ \mathsf{ok}}$$

$$\frac{\Gamma \vdash e_1\ \mathsf{ok} \quad \Gamma \vdash e_2\ \mathsf{ok}}{\Gamma \vdash \mathtt{plus}(e_1, e_2)\ \mathsf{ok}} \qquad \frac{\Gamma \vdash e_1\ \mathsf{ok} \quad \Gamma \vdash e_2\ \mathsf{ok}}{\Gamma \vdash \mathtt{times}(e_1, e_2)\ \mathsf{ok}}$$

$$\frac{\Gamma \vdash e_1\ \mathsf{ok} \quad \Gamma \cup \{ x \} \vdash e_2\ \mathsf{ok} \quad (x \notin \Gamma)}{\Gamma \vdash \mathtt{let}(e_1, x.e_2)\ \mathsf{ok}}$$

There are a few things to notice about these rules. First, a variable is well-formed iff it is in Γ. This is consistent with the informal reading of the judgement. Second, a `let` expression adds a *new* variable to Γ for use within e_2. The "newness" of the variable is captured by the requirement that $x \notin \Gamma$. Since we identify abt's up to choice of bound names, this requirement can always be met by a suitable renaming prior to application of the rule. Third, the rules are *syntax-directed* in the sense that there is one rule for each form of expression; as we will see later, this is not always the case for a static semantics.

6.2 Exercises

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

Chapter 7

Dynamic Semantics

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

7.1 Structured Operational Semantics

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

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

$$S = \{\, e \mid \exists \Gamma\ \Gamma \vdash e\ \mathsf{ok} \,\}.$$

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

$$I = \{\, e \mid \emptyset \vdash e\ \mathsf{ok} \,\}.$$

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

$$F = \{\, \mathtt{num}[n] \mid n \geq 0 \,\}.$$

The transition relation $\mapsto \subseteq S \times S$ is inductively defined by the following rules:

$$\frac{(p = m + n)}{\texttt{plus}(\texttt{num}[m], \texttt{num}[n]) \mapsto \texttt{num}[p]} \qquad \frac{(p = m \times n)}{\texttt{times}(\texttt{num}[m], \texttt{num}[n]) \mapsto \texttt{num}[p]}$$

$$\frac{}{\texttt{let}(\texttt{num}[n], x.e) \mapsto \{\texttt{num}[n]/x\}e}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{plus}(e_1, e_2) \mapsto \texttt{plus}(e_1', e_2)} \qquad \frac{e_2 \mapsto e_2'}{\texttt{plus}(\texttt{num}[n_1], e_2) \mapsto \texttt{plus}(\texttt{num}[n_1], e_2')}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{times}(e_1, e_2) \mapsto \texttt{times}(e_1', e_2)} \qquad \frac{e_2 \mapsto e_2'}{\texttt{times}(\texttt{num}[n_1], e_2) \mapsto \texttt{times}(\texttt{num}[n_1], e_2')}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{let}(e_1, x.e_2) \mapsto \texttt{let}(e_1', x.e_2)}$$

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

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

$$\frac{(p = m + n)}{m\texttt{+}n \mapsto p} \qquad \frac{(p = m \times n)}{m\texttt{*}n \mapsto p}$$

$$\frac{}{\texttt{let}\ x\ \texttt{be}\ n\ \texttt{in}\ e \mapsto \{n/x\}e}$$

$$\frac{e_1 \mapsto e_1'}{e_1\texttt{+}e_2 \mapsto e_1'\texttt{+}e_2} \qquad \frac{e_2 \mapsto e_2'}{n_1\texttt{+}e_2 \mapsto n_1\texttt{+}e_2'}$$

$$\frac{e_1 \mapsto e_1'}{e_1\texttt{*}e_2 \mapsto e_1'\texttt{*}e_2} \qquad \frac{e_2 \mapsto e_2'}{n_1\texttt{*}e_2 \mapsto n_1\texttt{*}e_2'}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{let}\ x\ \texttt{be}\ e_1\ \texttt{in}\ e_2 \mapsto \texttt{let}\ x\ \texttt{be}\ e_1'\ \texttt{in}\ e_2}$$

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

The first three rules defining the transition relation are somtimes called *instructions*, since they correspond to the primitive execution steps of the

machine. Addition and multiplication are evaluated by adding and multiplying; `let` bindings are evaluated by substituting the definition for the variable in the body. In all three cases the *principal arguments* of the constructor are required to be numbers. Both arguments of an addition or multiplication are principal, but only the binding of the variable in a *let* expression is principal. We say that these primitives are evaluated *by value*, because the instructions apply only when the principal arguments have been fully evaluated.

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

For example, consider the following evaluation sequence:

$$\begin{aligned}
\texttt{let}\ x\ \texttt{be}\ 1\texttt{+}2\ \texttt{in}\ (x\texttt{+}3)\texttt{*}4 &\mapsto \texttt{let}\ x\ \texttt{be}\ 3\ \texttt{in}\ (x\texttt{+}3)\texttt{*}4 \\
&\mapsto (3\texttt{+}3)\texttt{*}4 \\
&\mapsto 6\texttt{*}4 \\
&\mapsto 24
\end{aligned}$$

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

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

7.2 Evaluation Semantics

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

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

$$E = \{\, e \mid \emptyset \vdash e\ \mathsf{ok} \,\}.$$

The set of values is defined by

$$V = \{\, \mathtt{num}[n] \mid n \geq 0 \,\}.$$

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

$$\frac{}{\mathtt{num}[n] \Downarrow \mathtt{num}[n]}$$

$$\frac{e_1 \Downarrow \mathtt{num}[n_1] \quad e_2 \Downarrow \mathtt{num}[n_2] \quad (n = n_1 + n_2)}{\mathtt{plus}(e_1, e_2) \Downarrow \mathtt{num}[n]}$$

$$\frac{e_1 \Downarrow \mathtt{num}[n_1] \quad e_2 \Downarrow \mathtt{num}[n_2] \quad (n = n_1 \times n_2)}{\mathtt{times}(e_1, e_2) \Downarrow \mathtt{num}[n]}$$

$$\frac{e_1 \Downarrow \mathtt{num}[n_1] \quad \{\mathtt{num}[n_1]/x\}e_2 \Downarrow v}{\mathtt{let}(e_1, x.e_2) \Downarrow v}$$

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

Since the evaluation relation is inductively defined, it has associated with it a principle of proof by rule induction. Specifically, to show that $(e, \mathtt{num}[n])\ J$ holds for some judgement J governing expressions and numbers, it is enough to show that J is closed under the rules given above. Specifically,

1. Show that $(\mathtt{num}[n], \mathtt{num}[n])\ J$.
2. Assume that $(e_1, \mathtt{num}[n_1])\ J$ and $(e_2, \mathtt{num}[n_2])\ J$. Show that $(\mathtt{plus}(e_1, e_2), \mathtt{num}[n_1 + n_2])\ J$ and that $(\mathtt{times}(e_1, e_2), \mathtt{num}[n_1 \times n_2])\ J$.
3. Assume that $(e_1, \mathtt{num}[n_1])\ J$ and $(\{\mathtt{num}[n_1]/x\}e_2, \mathtt{num}[n_2])\ J$. Show that $(\mathtt{let}(e_1, x.e_2), \mathtt{num}[n_2])\ J$.

7.3 Relating Transition and Evaluation Semantics

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

Theorem 7.1
For all well-formed, closed arithmetic expressions e and all natural numbers n, $e \stackrel{!}{\mapsto} \mathtt{num}[n]$ iff $e \Downarrow \mathtt{num}[n]$.

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

Lemma 7.2
If $e \Downarrow \mathtt{num}[n]$, then $e \stackrel{!}{\mapsto} \mathtt{num}[n]$.

Proof: By induction on the definition of the evaluation relation. For example, suppose that $\mathtt{plus}(e_1, e_2) \Downarrow \mathtt{num}[n]$ by the rule for evaluating additions. By induction we know that $e_1 \stackrel{!}{\mapsto} \mathtt{num}[n_1]$ and $e_2 \stackrel{!}{\mapsto} \mathtt{num}[n_2]$. We

reason as follows:

$$\begin{array}{rcl} \texttt{plus}(e_1, e_2) & \stackrel{*}{\mapsto} & \texttt{plus}(\texttt{num}[n_1], e_2) \\ & \stackrel{*}{\mapsto} & \texttt{plus}(\texttt{num}[n_1], \texttt{num}[n_2]) \\ & \mapsto & \texttt{num}[n_1 + n_2] \end{array}$$

Therefore $\texttt{plus}(e_1, e_2) \stackrel{!}{\mapsto} \texttt{num}[n_1 + n_2]$, as required. The other cases are handled similarly. ■

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

Lemma 7.3
If $e \mapsto e'$ and $e' \Downarrow \texttt{num}[n]$, then $e \Downarrow \texttt{num}[n]$.

Proof: By induction on the definition of the transition relation. For example, suppose that $\texttt{plus}(e_1, e_2) \mapsto \texttt{plus}(e_1', e_2)$, where $e_1 \mapsto e_1'$. Suppose further that $\texttt{plus}(e_1', e_2) \Downarrow \texttt{num}[n]$, so that $e_1' \Downarrow \texttt{num}[n_1]$, and $e_2 \Downarrow \texttt{num}[n_2]$ and $n = n_1 + n_2$. By induction $e_1 \Downarrow \texttt{num}[n_1]$, and hence $\texttt{plus}(e_1, e_2) \Downarrow n$, as required. ■

7.4 Exercises

1. Prove that if $e \mapsto e_1$ and $e \mapsto e_2$, then $e_1 \equiv e_2$.
2. Prove that if $e \in I$ and $e \mapsto e'$, then $e' \in I$. Proceed by induction on the definition of the transition relation.
3. Prove that if $e \in I \setminus F$, then there exists e' such that $e \mapsto e'$. Proceed by induction on the rules defining well-formedness given in Chapter 6.
4. Prove that if $e \Downarrow v_1$ and $e \Downarrow v_2$, then $v_1 \equiv v_2$.
5. Complete the proof of equivalence of evaluation and transition semantics.

Chapter 8

Relating Static and Dynamic Semantics

The static and dynamic semantics of a language *cohere* if the strictures of the static semantics ensure the well-behavior of the dynamic semantics. In the case of arithmetic expressions, this amounts to showing two properties:

1. **Preservation**: If $\emptyset \vdash e$ ok and $e \mapsto e'$, then $\emptyset \vdash e'$ ok.
2. **Progress**: If $\emptyset \vdash e$ ok, then either $e = \mathtt{num}[n]$ for some n, or there exists e' such that $e \mapsto e'$.

The first says that the steps of evaluation preserve well-formedness, the second says that well-formedness ensures that either we are done or we can make progress towards completion.

8.1 Preservation for Arithmetic Expressions

The preservation theorem is proved by induction on the rules defining the transition system for step-by-step evaluation of arithmetic expressions. We will write e ok for $\emptyset \vdash e$ ok to enhance readability. Consider the rule

$$\frac{e_1 \mapsto e_1'}{\mathtt{plus}(e_1, e_2) \mapsto \mathtt{plus}(e_1', e_2).}$$

By induction we may assume that if e_1 ok, then e_1' ok. Assume that $\texttt{plus}(e_1, e_2)$ ok. From the definition of the static semantics we have that e_1 ok and e_2 ok. By induction e_1' ok, so by the static semantics $\texttt{plus}(e_1', e_2)$ ok. The other cases are quite similar.

8.2 Progress for Arithmetic Expressions

A moment's thought reveals that if $e \not\mapsto$, then e must be a name, for otherwise e is either a number or some transition applies. Thus the content of the progress theorem for the language of arithmetic expressions is that evaluation of a well-formed expression cannot encounter an unbound variable.

The proof of progress proceeds by induction on the rules of the static semantics. The rule for variables cannot occur, because we are assuming that the context, Γ, is empty. To take a representative case, consider the rule

$$\frac{\Gamma \vdash e_1 \text{ ok} \quad \Gamma \vdash e_2 \text{ ok}}{\Gamma \vdash \texttt{plus}(e_1, e_2) \text{ ok}}$$

where $\Gamma = \emptyset$. Let $e = \texttt{plus}(e_1, e_2)$, and assume e ok. Since e is not a number, we must show that there exists e' such that $e \mapsto e'$. By induction we have that either e_1 is a number, $\texttt{num}[n_1]$, or there exists e_1' such that $e_1 \mapsto e_1'$. In the latter case it follows that $\texttt{plus}(e_1, e_2) \mapsto \texttt{plus}(e_1', e_2)$, as required. In the former we also have by induction that either e_2 is a number, $\texttt{num}[n_2]$, or there exists e_2' such that $e_1 \mapsto e_1'$. In the latter case we have $\texttt{plus}(\texttt{num}[n_1], e_2) \mapsto \texttt{plus}(\texttt{num}[n_2], e_2')$. In the former we have $\texttt{plus}(\texttt{num}[n_1], \texttt{num}[n_2]) \mapsto \texttt{num}[n_1 + n_2]$. The other cases are handled similarly.

8.3 Exercises

Part III

A Functional Language

Chapter 9

MinML, A Minimal Functional Language

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

9.1 Syntax

9.1.1 Concrete Syntax

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

These categories are defined by the following grammar:

$$
\begin{array}{llll}
\textit{Var's} & x & ::= & \ldots \\
\textit{Num's} & n & ::= & \ldots \\
\textit{Op's} & o & ::= & \texttt{+} \mid \texttt{*} \mid \texttt{-} \mid \texttt{=} \mid \texttt{<} \\
\textit{Types} & \tau & ::= & \texttt{int} \mid \texttt{bool} \mid \tau_1 \rightarrow \tau_2 \\
\textit{Expr's} & e & ::= & x \mid n \mid o(e_1, \ldots, e_n) \mid \texttt{true} \mid \texttt{false} \mid \\
 & & & \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \\
 & & & \texttt{fun}\ f\ (x:\tau_1):\tau_2\ \texttt{is}\ e \mid \\
 & & & \texttt{apply}(e_1, e_2) \\
\textit{Prog's} & p & ::= & e
\end{array}
$$

We do not specify precisely the sets of numbers or variables. We generally write x, y, *etc.* for variables, and we write numbers in ordinary decimal notation. As usual we do not bother to specify such niceties as parenthesization or the use of infix syntax for binary operators, both of which would be necessary in practice.

9.1.2 Abstract Syntax

The abstract syntax of MinML may be read off from its concrete syntax by interpreting the preceding grammar as a specification of a set of abstract binding trees, rather than as a set of strings. The only additional information we need, beyond what is provided by the context-free grammar, is a specification of the binding and scopes of names in an expression. Informally, these may be specified by saying that in the function expression $\texttt{fun}\, f\, \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\, \texttt{is}\, e$ the variables f and x are both bound within the *body* of the function, e. Written as an abt, a function expression has the form $\texttt{fun}(\tau_1, \tau_2, f, x.e)$, which has the virtue of making explicit that f and x are bound within e, and that the argument and result types of the function are part of the syntax.

The following signature constitutes a precise definition of the abstract syntax of MinML as a class of abt's:

Operator	*Arity*
`int`	$()$
`bool`	$()$
$\rightarrow$	$(0,0)$
`n`	$()$
o	$(\underbrace{0,\ldots,0}_{n})$
`fun`	$(0,0,2)$
`apply`	$(0,0)$
`true`	$()$
`false`	$()$
`if`	$(0,0,0)$

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

9.2 Static Semantics

Not all expressions in MinML are sensible. For example, the expression `if` 3 `then` 1 `else` 0 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*, or *well-formed*.

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 three-place *typing judgement*, written $\Gamma \vdash e : \tau$, states that e is a well-typed expression with type τ in the context Γ, which assigns types to some finte set of names that may occur free in e. 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 τ (if any) assigned to x by Γ. The function $\Gamma[x{:}\tau]$, where $x \notin \mathrm{dom}(\Gamma)$, is defined by the following equation

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

The typing relation is inductively defined by the following rules:

$$\frac{(\Gamma(x) = \tau)}{\Gamma \vdash x : \tau} \tag{9.1}$$

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

$$\frac{}{\Gamma \vdash n : \texttt{int}} \tag{9.2}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \tag{9.3}$$

$$\frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}} \tag{9.4}$$

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

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash +(e_1, e_2) : \texttt{int}} \tag{9.5}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash *(e_1, e_2) : \texttt{int}} \tag{9.6}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash -(e_1, e_2) : \texttt{int}} \tag{9.7}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash =(e_1, e_2) : \texttt{bool}} \tag{9.8}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash <(e_1, e_2) : \texttt{bool}} \tag{9.9}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : \tau} \tag{9.10}$$

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

$$\frac{\Gamma[f{:}\tau_1 \to \tau_2][x{:}\tau_1] \vdash e : \tau_2 \quad (f, x \notin \mathrm{dom}(\Gamma))}{\Gamma \vdash \texttt{fun}\, f\, (x : \tau_1) : \tau_2\, \texttt{is}\, e : \tau_1 \to \tau_2} \tag{9.11}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{apply}(e_1, e_2) : \tau} \tag{9.12}$$

9.3 Properties of Typing

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

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

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

$$\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : \tau,$$

it suffices to show that $\Gamma \vdash e : \texttt{bool}$, $\Gamma \vdash e_1 : \tau$, and $\Gamma \vdash e_2 : \tau$, because of Rule 9.10. Since there is exactly one typing rule for each expression, the typing rules also express *necessary* conditions for typing. For example, if $\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : \tau$, then $\Gamma \vdash e : \texttt{bool}$, $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$. That is, we can "invert" each rule to obtain a necessary typing condition. This is the content of the following theorem.

Theorem 9.1 (Inversion)

1. *If* $\Gamma \vdash x : \tau$, *then* $\Gamma(x) = \tau$.
2. *If* $\Gamma \vdash n : \tau$, *then* $\tau = \texttt{int}$.
3. *If* $\Gamma \vdash \texttt{true} : \tau$, *then* $\tau = \texttt{bool}$, *and similarly for* `false`.
4. *If* $\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : \tau$, *then* $\Gamma \vdash e : \texttt{bool}$, $\Gamma \vdash e_1 : \tau$ *and* $\Gamma \vdash e_2 : \tau$.
5. *If* $\Gamma \vdash \texttt{fun}\, f\, (x{:}\tau_1){:}\tau_2\, \texttt{is}\, e : \tau$, *then* $\Gamma[f{:}\tau_1 \to \tau_2][x{:}\tau_1] \vdash e : \tau_2$ *and* $\tau = \tau_1 \to \tau_2$.
6. *If* $\Gamma \vdash \texttt{apply}(e_1, e_2) : \tau$, *then there exists* τ_2 *such that* $\Gamma \vdash e_1 : \tau_2 \to \tau$ *and* $\Gamma \vdash e_2 : \tau_2$.

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

Lemma 9.2

1. *Typing is not affected by "junk" in the symbol table. If* $\Gamma \vdash e : \tau$ *and* $\Gamma' \supseteq \Gamma$, *then* $\Gamma' \vdash e : \tau$.
2. *Substitution for a variable with type* τ *by an expression of the same type doesn't affect typing. If* $\Gamma[x{:}\tau] \vdash e' : \tau'$, *and* $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash \{e/x\}e' : \tau'$.

Proof:

1. By induction on the typing rules. For example, consider the typing rule for applications. Inductively we may assume that if $\Gamma' \supseteq \Gamma$, then $\Gamma' \vdash e_1 : \tau_2 \to \tau$ and if $\Gamma' \supseteq \Gamma$, then $\Gamma' \vdash e_2 : \tau_2$. Consequently, if $\Gamma' \supseteq \Gamma$, then $\Gamma' \vdash \mathtt{apply}(e_1, e_2) : \tau$, 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 9.1) 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 9.11) We have that $e' = \mathtt{fun}\, f\ (y : \tau_1) : \tau_2\ \mathtt{is}\ e_2$ and $\tau' = \tau_1 \to \tau_2$. We may assume that f and y are chosen so that

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

   By definition of substitution,

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

   Applying the inductive hypothesis to the premise of Rule 9.11,

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

   it follows that

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

   Hence

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

   as required.

■

9.4 Dynamic Semantics

The dynamic semantics of MinML is given by an inductive definition of the *one-step evaluation* relation, $e \mapsto e'$, between *closed* expressions. Recall that we are modelling computation in MinML as a form of "in place" calculation; the relation $e \mapsto e'$ means that e' is the result of performing a single step of computation starting with e. To calculate the value of an expression e, we repeatedly perform single calculation steps until we reach a *value*, v, which is either a number, a boolean constant, or a function.

The rules defining the dynamic semantics of MinML may be classified into two categories: rules defining the fundamental computation steps (or, *instructions*) of the language, and rules for determining where the next instruction is to be executed. The purpose of the search rules is to ensure that the dynamic semantics is *deterministic*, which means that for any expression there is at most one "next instruction" to be executed.[1]

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

$$\frac{}{\mathtt{+}(m,n) \mapsto m+n} \tag{9.13}$$

The other primitive operations are defined similarly.

The primitive instructions for conditional expressions are as follows:

$$\frac{}{\mathtt{if\ true\ then}\ e_1\ \mathtt{else}\ e_2 \mapsto e_1} \tag{9.14}$$

$$\frac{}{\mathtt{if\ false\ then}\ e_1\ \mathtt{else}\ e_2 \mapsto e_2} \tag{9.15}$$

The primitive instruction for application is as follows:

$$\frac{(v = \mathtt{fun}\ f\ (x:\tau_1):\tau_2\ \mathtt{is}\ e)}{\mathtt{apply}(v, v_1) \mapsto \{v, v_1/f, x\}e} \tag{9.16}$$

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

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

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

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

$$\frac{e_1 \mapsto e_1'}{+(e_1, e_2) \mapsto +(e_1', e_2)} \tag{9.17}$$

$$\frac{e_2 \mapsto e_2'}{+(v_1, e_2) \mapsto +(v_1, e_2')} \tag{9.18}$$

The other primitive operations are handled similarly.

For the conditional, we evaluate the test expression.

$$\frac{e \mapsto e'}{\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \mapsto \texttt{if}\, e'\, \texttt{then}\, e_1\, \texttt{else}\, e_2} \tag{9.19}$$

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

$$\frac{e_1 \mapsto e_1'}{\texttt{apply}(e_1, e_2) \mapsto \texttt{apply}(e_1', e_2)} \tag{9.20}$$

$$\frac{e_2 \mapsto e_2'}{\texttt{apply}(v_1, e_2) \mapsto \texttt{apply}(v_1, e_2')} \tag{9.21}$$

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

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

$$\frac{}{e \mapsto^* e} \tag{9.22}$$

$$\frac{e \mapsto e' \quad e' \mapsto^* e''}{e \mapsto^* e''} \tag{9.23}$$

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

9.5 Properties of the Dynamic Semantics

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

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

Lemma 9.3
For every closed expression e, there exists at most one e' such that $e \mapsto e'$. In other words, the relation $\mapsto$ is a partial function.

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

It follows that evaluation to a value is deterministic:

Lemma 9.4
For every closed expression e, there exists at most one value v such that $e \mapsto^* v$.

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

9.6 Exercises

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

2. Prove by induction on the structure of e that for every e and every Γ there exists at most one τ such that $\Gamma \vdash e : \tau$. *Hint:* use rule induction for the rules defining the abstract syntax of expressions.

Chapter 10

Type Safety for MinML

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

10.1 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 $e \mapsto e'$, then e' is also a well-typed program.

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

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

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

10.2 Type Safety of MinML

Theorem 10.1 (Preservation)
If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.

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

(Rule 9.13) 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 9.14) Here $e = \texttt{if true then } e_1 \texttt{ else } e_2$ and $e' = e_1$. Since $e : \tau$, by inversion $e_1 : \tau$, as required.

(Rule 9.15) Here $e = \texttt{if false then } e_1 \texttt{ else } e_2$ and $e' = e_2$. Since $e : \tau$, by inversion $e_2 : \tau$, as required.

(Rule 9.16) Here $e = \texttt{apply}(v_1, v_2)$, where $v_1 = \texttt{fun}\, f\ (x{:}\tau_2){:}\tau\, \texttt{is}\, e_2$, and $e' = \{v_1, v_2/f, x\}e_2$. By inversion applied to e, we have $v_1 : \tau_2 \to \tau$ and $v_2 : \tau_2$. By inversion applied to v_1, we have $[f{:}\tau_2 \to \tau][x{:}\tau_2] \vdash e_2 : \tau$. Therefore, by substitution we have $\{v_1, v_2/f, x\}e_2 : \tau$, as required.

(Rule 9.17) Here $e = +(e_1, e_2)$, $e' = +(e_1', e_2)$, and $e_1 \mapsto e_1'$. By inversion $e_1 : \texttt{int}$, so that by induction $e_1' : \texttt{int}$, and hence $e' : \texttt{int}$, as required.

(Rule 9.18) Here $e = +(v_1, e_2)$, $e' = +(v_1, e_2')$, and $e_2 \mapsto e_2'$. By inversion $e_2 : \texttt{int}$, so that by induction $e_2' : \texttt{int}$, and hence $e' : \texttt{int}$, as required.

The other primitive operations are handled similarly.

(Rule 9.19) Here $e = \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3$ and $e' = \texttt{if}\, e_1'\, \texttt{then}\, e_2\, \texttt{else}\, e_3$. By inversion we have that $e_1 : \texttt{bool}$, $e_2 : \tau$ and $e_3 : \tau$. By inductive hypothesis $e_1' : \texttt{bool}$, and hence $e' : \tau$.

(Rule 9.20) Here $e = \texttt{apply}(e_1, e_2)$ and $e' = \texttt{apply}(e_1', e_2)$. By inversion $e_1 : \tau_2 \to \tau$ and $e_2 : \tau_2$, for some type τ_2. By induction $e_1' : \tau_2 \to \tau$, and hence $e' : \tau$.

(Rule 9.21) Here $e = \texttt{apply}(v_1, e_2)$ and $e' = \texttt{apply}(v_1, e_2')$. By inversion, $v_1 : \tau_2 \to \tau$ and $e_2 : \tau_2$, for some type τ_2. By induction $e_2' : \tau_2$, and hence $e' : \tau$.

■

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

Lemma 10.2 (Canonical Forms)
Suppose that $v : \tau$ is a closed, well-formed value.

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

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

Exercise 10.3
Give a proof of the canonical forms lemma.

Theorem 10.4 (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 9.1) Cannot occur, since e is closed.

(Rules 9.2, 9.3, 9.4, 9.11) In each case e is a value, which completes the proof.

(Rule 9.5) Here $e = +(e_1, e_2)$ and $\tau = \texttt{int}$, with $e_1 : \texttt{int}$ and $e_2 : \texttt{int}$. By induction we have either e_1 is a value, or there exists e_1' such that $e_1 \mapsto e_1'$ for some expression e_1'. In the latter case it follows that $e \mapsto e'$, where $e' = +(e_1', e_2)$. 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 9.10) Here $e = \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3$, with $e_1 : \texttt{bool}$, $e_2 : \tau$, and $e_3 : \tau$. By the first inductive hypothesis, either e_1 is a value, or there exists e_1' such that $e_1 \mapsto e_1'$. If e_1 is a value, then we have by the Canonical Forms Lemma, either $e_1 = \texttt{true}$ or $e_1 = \texttt{false}$. In the former case $e \mapsto e_2$, and in the latter $e \mapsto e_3$, as required. If e_1 is not a value, then $e \mapsto e'$, where $e' = \texttt{if}\, e_1'\, \texttt{then}\, e_2\, \texttt{else}\, e_3$, by Rule 9.19.

(Rule 9.12) Here $e = \texttt{apply}(e_1, e_2)$, 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 $e \mapsto \texttt{apply}(e_1', e_2)$ by Rule 9.20, as required. By the second inductive hypothesis, either e_2 is a value, or there exists e_2' such that $e_2 \mapsto e_2'$. If e_2 is not a value, then $e \mapsto e'$, where $e' = \texttt{apply}(e_1, e_2')$, as required. Finally, if both e_1 and e_2 are values, then by the Canonical Forms Lemma, $e_1 = \texttt{fun}\, f\, (x : \tau_2) : \tau\, \texttt{is}\, e''$, and $e \mapsto e'$, where $e' = \{e_1, e_2 / f, x\} e''$, by Rule 9.16.

■

Theorem 10.5 (Safety)
If e is closed and well-typed, then evaluation of e can only terminate with a value of the same type. In particular, evaluation cannot "get stuck" in an ill-defined state.

10.3 Run-Time Errors and Safety

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

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

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

But suppose that we add to MinML a primitive operation, such as quotient, that is undefined for certain arguments. An expression such as $3/0$

[1] It should be easy to convince yourself that it is undecidable whether a given C program can reach an implementation-dependent state.

would most-assuredly be "stuck", yet would be well-typed, at least if we take the natural typing rule for it:

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \,/\, e_2 : \texttt{int}}$$

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

To recover safety, we have two options:

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

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

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

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

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

are inherently portable, which explains the recent resurgence in interest in them.

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

$$\frac{}{\Gamma \vdash \texttt{error} : \tau} \tag{10.1}$$

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

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

$$\frac{}{v_1 \,/\, 0 \mapsto \texttt{error}} \tag{10.2}$$

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

$$\frac{}{\texttt{apply(error,}v_2\texttt{)} \mapsto \texttt{error}} \tag{10.3}$$

$$\frac{}{\texttt{apply(}v_1\texttt{,error)} \mapsto \texttt{error}} \tag{10.4}$$

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

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

Theorem 10.6 (Safety With Errors)
If an expression is well-typed, it can only evaluate to a value or evaluate to `error`*. It cannot "get stuck" in an ill-defined state.*

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

Part IV

Control and Data Flow

Chapter 11

Abstract Machines

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

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

[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.

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

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

11.1 Control Flow

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

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

[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 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.

by the following rules:

$$\frac{}{\bullet\ \mathsf{stack}} \tag{11.1}$$

$$\frac{f\ \mathsf{frame} \quad k\ \mathsf{stack}}{f \triangleright k\ \mathsf{stack}} \tag{11.2}$$

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

$$\frac{e_2\ \mathsf{expr}}{+(\square, e_2)\ \mathsf{frame}} \tag{11.3}$$

$$\frac{v_1\ \mathsf{value}}{+(v_1, \square)\ \mathsf{frame}} \tag{11.4}$$

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

$$\frac{e_1\ \mathsf{expr} \quad e_2\ \mathsf{expr}}{\mathtt{if}\ \square\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ \mathsf{frame}} \tag{11.5}$$

$$\frac{e_2\ \mathsf{expr}}{\mathtt{apply}(\square, e_2)\ \mathsf{frame}} \tag{11.6}$$

$$\frac{v_1\ \mathsf{value}}{\mathtt{apply}(v_1, \square)\ \mathsf{frame}} \tag{11.7}$$

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

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

$$(k, +(e_1, e_2)) \mapsto_{\mathsf{C}} (+(\square, e_2) \triangleright k, e_1) \tag{11.8}$$

$$(+(\square, e_2) \triangleright k, v_1) \mapsto_{\mathsf{C}} (+(v_1, \square) \triangleright k, e_2) \tag{11.9}$$

$$(+(n_1, \square) \triangleright k, n_2) \mapsto_{\mathsf{C}} (k, n_1 + n_2) \tag{11.10}$$

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 booleans.

$$(k, \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2) \mapsto_{\mathsf{C}} (\texttt{if}\ \square\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \triangleright k, e) \tag{11.11}$$

$$(\texttt{if}\ \square\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \triangleright k, \texttt{true}) \mapsto_{\mathsf{C}} (k, e_1) \tag{11.12}$$

$$(\texttt{if}\ \square\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \triangleright k, \texttt{false}) \mapsto_{\mathsf{C}} (k, e_2) \tag{11.13}$$

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

Finally, we consider the rules for application of functions.

$$(k, \texttt{apply}(e_1, e_2)) \mapsto_{\mathsf{C}} (\texttt{apply}(\square, e_2) \triangleright k, e_1) \tag{11.14}$$

$$(\texttt{apply}(\square, e_2) \triangleright k, v_1) \mapsto_{\mathsf{C}} (\texttt{apply}(v_1, \square) \triangleright k, e_2) \tag{11.15}$$

$$(\texttt{apply}(v_1, \square) \triangleright k, v_2) \mapsto_{\mathsf{C}} (k, \{v_1, v_2/f, x\}e) \tag{11.16}$$

The last rule applies in the case that $v_1 = \texttt{fun}\ f\ (x:\tau_1):\tau_2\ \texttt{is}\ e$. 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 MinML interpreter corresponding to premises of the evaluation rules. The runtime stack is required because the interpreter is *not* a tail recursive function. In contrast an implementation of the C machine *is* tail recursive, precisely because there are no premises on any of the transitions rules defining it.

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

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

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

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

Theorem 11.1

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

The first part of the Theorem states that the C 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.[3]

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.

   (a) Consider the transition

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

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

   (b) Consider the transition

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

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

   (c) Consider the transition

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

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

   $$\begin{array}{rcl} k \mathbin{@} e & = & k' \mathbin{@} \texttt{apply}(v_1, v_2) \\ & \mapsto & k' \mathbin{@} e' \end{array}$$

   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 $k' \mathbin{@} \texttt{apply}(v_1, v_2)$ must be the application $\texttt{apply}(v_1, v_2)$.

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

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

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

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

   From this, the result follows immediately.

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

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

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

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

   Now since

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

   the result follows immediately.

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

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

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

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

   Now since

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

   the result follows immediately.

■

Exercise 11.2
Finish the proof of the theorem by giving a complete proof of part (1), and filling in the missing cases in part (2).

Corollary 11.3

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

Proof:

1. By induction on the transition sequence, making use of part (1) of the theorem, then 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 $k @ e = v \mapsto^{*} v$, as required.
   (b) Reverse execution. Here we have $(k', e') \mapsto_{\mathsf{C}} (k, e) \mapsto_{\mathsf{C}}^{*} (\bullet, v)$. By induction $k @ e \mapsto_{\mathsf{M}}^{*} v$, and by Theorem 11.1 $k' @ e' \mapsto_{\mathsf{M}}^{*} k @ e$, so $k' @ e' \mapsto_{\mathsf{M}}^{*} v$.
2. By induction on transition sequence, making use of part (2) of the theorem, then taking $e' = v$ and $k = \bullet$. We have two cases:
   (a) Reflexivity. In this case $e = e'$ and the result is immediate.
   (b) Reverse execution. Here $e \mapsto_{\mathsf{M}} e'' \mapsto_{\mathsf{M}}^{*} e'$ and $(k, e') \mapsto_{\mathsf{C}}^{*} (\bullet, v)$. By induction $(k, e'') \mapsto_{\mathsf{C}}^{*} (\bullet, v)$ and by Theorem 11.1 we have $(k, e) \mapsto_{\mathsf{C}}^{*} (\bullet, v)$, as required.

■

To facilitate comparison with the E machine described below, it is useful to restructure the C machine in the following manner. First, we introduce an "auxiliary" state of the form (v, k), which represents the process

of passing the value v to the stack k. Second, we "link" these two states by the transition rule

$$(k, v) \mapsto_{\mathsf{C}} (v, k). \tag{11.17}$$

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

$$\begin{array}{lcll} (k, e) & \mapsto_{\mathsf{C}} & (k', e') & \textit{process expression} \\ (k, v) & \mapsto_{\mathsf{C}} & (v, k) & \textit{pass value to stack} \\ (v, k) & \mapsto_{\mathsf{C}} & (v', k') & \textit{pass value up stack} \\ (v, k) & \mapsto_{\mathsf{C}} & (k', e') & \textit{process pending expression} \end{array}$$

Exercise 11.4
Complete the suggested re-formulation of the C machine, and show that it is equivalent to the orginal formulation.

11.2 Environments

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

The basic idea is simple: rather than *replace* variables by their bindings when performing a function application, we instead *record* the bindings of variables in a data structure, and, correspondingly, *look up* the bindings of variables when they are used. In a sense we are performing substitution "lazily", rather than "eagerly", to avoid unnecessary duplication and to avoid modifying the program during execution. The main complication introduced by environments is that we must exercise considerable caution

to ensure that we do not confuse the scopes of variables.[4] It is remarkably easy, if we are not careful, to confuse the bindings of variables that happen to have the same name. We avoid difficulties by introducing *closures*, data structures that package an expression together with an environment.

To see the point, let's first sketch out the structure of the E machine. A state of the E machine has the form (K, E, e), where K is a *machine stack*, E is an *environment*, a finite function mapping variables to *machine values*, and e is an open expression such that $\mathrm{FV}(e) \subseteq \mathrm{dom}(E)$. Machine values are values "inside the machine", distinct from the syntactic notion of value used in the M and C machines. The reason for the distinction arises from the replacement of substitution by binding.

Since the M and C machines perform function application by substitution, there is never any need to consider expressions with free variables in them; the invariant that the expression part of the state is closed is maintained throughout evaluation. The whole point of the E 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 *data stack* for precisely this reason. However, a moment's thought reveals that this characterization is a tad too simplistic, because it overlooks a crucial issue in the implementation of functional languages, namely the ability to return functions as results of function applications. Suppose that f is a function of type $\texttt{int}\rightarrow\texttt{int}\rightarrow\texttt{int}$. When applied to an integer n, the result $\texttt{apply}(f,n)$ yields a function of type $\texttt{int}\rightarrow\texttt{int}$. For example, f might be the following function:

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

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

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

call and return, we will, upon calling f, bind x to 1, yielding the value

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

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 $\texttt{apply}(f, 1)$ is the closure[6]

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

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:

$$\frac{}{n\ \mathsf{mvalue}} \tag{11.18}$$

$$\frac{}{\texttt{true}\ \mathsf{mvalue}} \tag{11.19}$$

$$\frac{}{\texttt{false}\ \mathsf{mvalue}} \tag{11.20}$$

$$\frac{x\ \mathsf{var} \quad y\ \mathsf{var} \quad e\ \mathsf{expr}}{\texttt{fun}\ x\ (y\texttt{:}\tau_1)\texttt{:}\tau_2\ \texttt{is}\ e[E]\ \mathsf{mvalue}} \tag{11.21}$$

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

[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.

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

$$\frac{}{\bullet\ \mathsf{mstack}} \tag{11.22}$$

$$\frac{F\ \mathsf{mframe} \quad K\ \mathsf{mstack}}{F \triangleright K\ \mathsf{mstack}}\ . \tag{11.23}$$

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

$$\frac{e_2\ \mathsf{expr}}{+(\square, e_2)[E]\ \mathsf{mframe}} \tag{11.24}$$

$$\frac{V_1\ \mathsf{mvalue}}{+(V_1, \square)\ \mathsf{mframe}} \tag{11.25}$$

$$\frac{e_1\ \mathsf{expr} \quad e_2\ \mathsf{expr}}{\mathtt{if}\ \square\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2[E]\ \mathsf{mframe}} \tag{11.26}$$

$$\frac{e_2\ \mathsf{expr}}{\mathtt{apply}(\square, e_2)[E]\ \mathsf{mframe}} \tag{11.27}$$

$$\frac{V_1\ \mathsf{mvalue}}{\mathtt{apply}(V_1, \square)\ \mathsf{mframe}} \tag{11.28}$$

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

The E machine has two kinds of states: (K, E, e), described earlier, and "auxiliary" states of the form (V, K), where K is a machine stack and V is a machine value. The auxiliary state represents the passage of a machine value to the top frame of the machine stack. (In the C machine this is accomplished by simply filling the hole in the stack frame, but here a bit more work is required.)

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

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

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

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

$$(K, E, x) \mapsto_{\mathsf{E}} (E(x), K) \tag{11.29}$$

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

$$(K, E, n) \mapsto_{\mathsf{E}} (n, K) \tag{11.30}$$

$$(K, E, \mathtt{true}) \mapsto_{\mathsf{E}} (\mathtt{true}, K) \tag{11.31}$$

$$(K, E, \mathtt{false}) \mapsto_{\mathsf{E}} (\mathtt{false}, K) \tag{11.32}$$

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

$$(K, E, \mathtt{fun}\, f\ (x : \tau_1) : \tau_2\ \mathtt{is}\, e) \mapsto_{\mathsf{E}} (\mathtt{fun}\, f\ (x : \tau_1) : \tau_2\ \mathtt{is}\, e[E], K) \tag{11.33}$$

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

$$(K, E, +(e_1, e_2)) \mapsto_{\mathsf{E}} (+(\Box, e_2)[E] \triangleright K, E, e_1) \tag{11.34}$$

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

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

$$(K, E, \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2) \mapsto_{\mathsf{E}} (\texttt{if}\ \Box\ \texttt{then}\ e_1\ \texttt{else}\ e_2[E] \triangleright K, E, e) \quad (11.35)$$

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

$$(K, E, \texttt{apply}(e_1, e_2)) \mapsto_{\mathsf{E}} (\texttt{apply}(\Box, e_2)[E] \triangleright K, E, e_1) \quad (11.36)$$

To complete the definition of the E machine, we must define the transitions governing the auxiliary states.

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

$$(V_1, +(\Box, e_2)[E] \triangleright K) \mapsto_{\mathsf{E}} (+(V_1, \Box) \triangleright K, E, e_2) \quad (11.37)$$

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

$$(n_2, +(v_1, \Box) \triangleright K) \mapsto_{\mathsf{E}} (n_1 + n_2, K) \quad (11.38)$$

Pending conditional branches are handled in the obvious manner.

$$(\texttt{true}, \texttt{if}\ \Box\ \texttt{then}\ e_1\ \texttt{else}\ e_2[E] \triangleright K) \mapsto_{\mathsf{E}} (K, E, e_1) \quad (11.39)$$

$$(\texttt{false}, \texttt{if}\ \Box\ \texttt{then}\ e_1\ \texttt{else}\ e_2[E] \triangleright K) \mapsto_{\mathsf{E}} (K, E, e_2) \quad (11.40)$$

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

Pending function applications are handled as follows.

$$(V, \texttt{apply}(\Box, e_2)[E] \triangleright K) \mapsto_{\mathsf{E}} (\texttt{apply}(V, \Box) \triangleright K, E, e_2) \tag{11.41}$$

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

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

$$(V_2, \texttt{apply}(V, \Box) \triangleright K) \mapsto_{\mathsf{E}} (K, E[f \mapsto V][x \mapsto V_2], e) \tag{11.42}$$

where

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

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

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

A correspondence between the 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 (`int` or `bool`), then $(\bullet, e) \mapsto_{\mathsf{C}}^{*} (\bullet, v)$ iff $(\bullet, \emptyset, e) \mapsto_{\mathsf{E}}^{*} (v, \bullet)$. (The restriction to base type is necessary if we are to claim that both machines return the *same* value.)

Chapter 12

Continuations

The semantics of many control constructs (such as exceptions and co-routines) can be expressed in terms of *reified* control stacks, a representation of a control stack as an ordinary value. This is achieved by allowing a stack to be passed as a value within a program and to be restored at a later point, *even if* control has long since returned past the point of reification. Reified control stacks of this kind are called *first-class continuations*, where the qualification "first class" stresses that they are ordinary values with an indefinite lifetime that can be passed and returned at will in a computation. First-class continuations never "expire", and it is always sensible to reinstate a continuation without compromising safety. Thus first-class continuations support unlimited "time travel" — we can go back to a previous point in the computation and then return to some point in its future, at will.

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

mutate the data structure, preventing any form of time travel.

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

Why are first-class continuations useful? Fundamentally, they are representations of the control state of a computation at a given point in time. Using first-class continuations we can "checkpoint" the control state of a program, save it in a data structure, and return to it later. In fact this is precisely what is necessary to implement *threads* (concurrently executing programs) — the thread scheduler must be able to checkpoint a program and save it for later execution, perhaps after a pending event occurs or another thread yields the processor. In Section 12.3 we will show how to build a threads package for concurrent programming using continuations.

12.1 Informal Overview of Continuations

We will extend MinML with the type τ `cont` of continuations accepting values of type τ. A continuation will, in fact, be a control stack of type τ `stack`, but rather than expose this representation to the programmer, we will regard τ `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 is, as we've discussed, a reifi-

[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`".

cation of the current control stack, which represents the current point in the evaluation of the program. The type of x is τ `cont`, where τ 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 τ, which then determines how execution proceeds. Thus x is bound to a stack expecting a value of type τ, that is, a value of type τ `cont`. Note that this is the *only* way to obtain a value of type τ `cont`; there are no expressions that evaluate to continuations. (This is similar to our treatment of references — values of type τ `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 τ_1 `cont`, and e_1 must evaluate to a value of type τ_1. The current control stack is abandoned in favor of the reified control stack resulting from the evaluation of e_2; the value of e_1 is then passed to that stack.

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

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

Ignoring the `letcc` 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. 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 use of `letcc`. 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 `letcc` 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 eval-

uation proceeds normally. In the unusual case of encountering a `0` in the list, we branch directly to the return point, passing the value `0`, effecting an early return from the procedure with result value `0`.

Here's another formulation of the same function:

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

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

Let's look at another example: given a continuation k of type τ `cont` and a function f of type $\tau' \rightarrow \tau$, 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

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

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

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

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

```
fun compose (f, k) =
    letcc ret in
      throw (f (letcc r in throw r to ret)) to k
```

The type of `ret` is τ' `cont cont`, a continuation expecting a continuation expecting a value of type τ'!

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

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

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

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

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

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

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

and that the type of the second version becomes

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

These transformations are representative of the general case.

12.2 Semantics of Continuations

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

First, the abstract syntax. We extend the language of MinML types with continuation types of the form τ `cont`. We extend the language of MinML expressions with these additional forms:

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

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:

$$\frac{K\ \mathsf{stack}}{K\ \mathsf{value}} \tag{12.1}$$

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

$$\frac{e_2\ \mathsf{expr}}{\texttt{throw}\,\square\,\texttt{to}\,e_2\ \mathsf{frame}} \tag{12.2}$$

$$\frac{v_1\ \mathsf{value}}{\texttt{throw}\,v_1\,\texttt{to}\,\square\ \mathsf{frame}} \tag{12.3}$$

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

$$\frac{\Gamma[x{:}\tau\ \texttt{cont}] \vdash e : \tau}{\Gamma \vdash \texttt{letcc}\, x\, \texttt{in}\, e : \tau} \tag{12.4}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_1\ \texttt{cont}}{\Gamma \vdash \texttt{throw}\, e_1\, \texttt{to}\, e_2 : \tau'} \tag{12.5}$$

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

$$\frac{\vdash K : \tau\ \mathtt{stack}}{\Gamma \vdash K : \tau\ \mathtt{cont}} \tag{12.6}$$

That is, a continuation value K has type $\tau\ \mathtt{cont}$ exactly if it is a stack accepting values of type τ. This relation is defined below, when we consider type safety of this extension to MinML.

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

$$\overline{(K, \mathtt{letcc}\ x\ \mathtt{in}\ e) \mapsto (K, \{K/x\}e)} \tag{12.7}$$

$$\overline{(\mathtt{throw}\ v\ \mathtt{to}\ \Box \triangleright K, K') \mapsto (K', v)} \tag{12.8}$$

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

$$\overline{(K, \mathtt{throw}\ e_1\ \mathtt{to}\ e_2) \mapsto (\mathtt{throw}\ \Box\ \mathtt{to}\ e_2 \triangleright K, e_1)} \tag{12.9}$$

$$\overline{(\mathtt{throw}\ \Box\ \mathtt{to}\ e_2 \triangleright K, v_1) \mapsto (\mathtt{throw}\ v_1\ \mathtt{to}\ \Box \triangleright K, e_2)} \tag{12.10}$$

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

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

$$\frac{\vdash K : \tau\ \mathtt{stack} \qquad \vdash e : \tau}{(K, e)\ \mathsf{ok}} \tag{12.11}$$

That is, a state (K, e) is well-formed iff e is an expression of type τ and K is a τ-accepting control stack.

To define the judgement $\vdash K : \tau\ \mathtt{stack}$, we must first fix the type of the "ultimate answer" of a program, the value returned when evaluation is

completed. The particular choice of answer type is not important, but it is important that it be a fixed type, $\tau_{\mathtt{ans}}$.

$$\frac{}{\vdash \bullet : \tau_{\mathtt{ans}}\ \mathtt{stack}} \tag{12.12}$$

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

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

$$\frac{\vdash e_2 : \mathtt{int}}{\vdash +(\square, e_2) : (\mathtt{int}, \mathtt{int})\ \mathtt{frame}} \tag{12.14}$$

$$\frac{v_1\ \mathsf{value} \quad \vdash v_1 : \mathtt{int}}{\vdash +(v_1, \square) : (\mathtt{int}, \mathtt{int})\ \mathtt{frame}} \tag{12.15}$$

$$\frac{\vdash e_1 : \tau \quad \vdash e_2 : \tau}{\vdash \mathtt{if}\ \square\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : (\mathtt{bool}, \tau)\ \mathtt{frame}} \tag{12.16}$$

$$\frac{\vdash e_2 : \tau_2}{\vdash \mathtt{apply}(\square, e_2) : (\tau_2 \to \tau, \tau)\ \mathtt{frame}} \tag{12.17}$$

$$\frac{v_1\ \mathsf{value} \quad \vdash v_1 : \tau_2 \to \tau}{\vdash \mathtt{apply}(v_1, \square) : (\tau_2, \tau)\ \mathtt{frame}} \tag{12.18}$$

$$\frac{\vdash e_2 : \tau\ \mathtt{cont}}{\vdash \mathtt{throw}\ \square\ \mathtt{to}\ e_2 : (\tau, \tau')\ \mathtt{frame}} \tag{12.19}$$

$$\frac{\vdash v_1 : \tau}{\vdash \mathtt{throw}\ v_1\ \mathtt{to}\ \square : (\tau\ \mathtt{cont}, \tau')\ \mathtt{frame}} \tag{12.20}$$

Intuitively, a frame of type $(\tau_1, \tau_2)\ \mathtt{frame}$ takes an "argument" of type τ_1 and yields a "result" of type τ_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 hand, we may state the preservation theorem as follows:

Theorem 12.1 (Preservation)
If (K, e) ok *and* $(K, e) \mapsto (K', e')$*, then* (K', e') ok*.*

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

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

Lemma 12.2 (Canonical Forms)
If $\vdash v : \tau$ `cont`*, then* $v = K$ *for some control stack* K *such that* $\vdash K : \tau$ `stack`*.*

Finally, progress is stated as follows:

Theorem 12.3 (Progress)
If (K, e) ok *then either* $K = \bullet$ *and* e value*, or there exists* K' *and* e' *such that* $(K, e) \mapsto (K', e')$*.*

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

12.3 Coroutines

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

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

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

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

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

```
datatype state = S of state cont
fun resume (S k : state) : state =
    callcc (fn k' : state cont => throw k (S k'))
val buf : int ref = ref 0
fun produce (n:int, cons:state) =
    (buf := n; produce (n+1, resume cons))
fun consume (prod:state) =
    (print (Int.toString(!buf)); consume (resume prod))
```

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

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

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

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

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

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

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

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

User-level threads are naturally implemented as continuations. A thread is a value of type `unit cont`. The scheduler maintains a queue of threads

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

Here is a simple implementation of threads:

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

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

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

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

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

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

There is much more to be said about threads! We will return to this later in the course. For now, the main idea is to give a flavor of how first-class continuations can be used to implement a user-level threads package with very little difficulty. A more complete implementation is, of course,

somewhat more complex, but not much more. We can easily provide all that is necessary for sophisticated thread programming in a few hundred lines of ML code.

12.4 Exercises

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

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

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

$$\texttt{apply}(f,\Box) \triangleright \texttt{throw}\,\Box\,\texttt{to}\,k \triangleright \bullet \tag{12.21}$$

   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.

Chapter 13

Exceptions

Exceptions effects 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:

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

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

The static semantics of exceptions is quite straightforward:

$$\frac{}{\Gamma \vdash \mathtt{fail} : \tau} \tag{13.1}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{try}\, e_1\, \mathtt{ow}\, e_2 : \tau} \tag{13.2}$$

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:

$$\frac{e_2\ \mathsf{expr}}{\mathtt{try}\,\square\,\mathtt{ow}\, e_2\ \mathsf{frame}} \tag{13.3}$$

The evaluation rules are extended as follows:

$$\frac{}{(K, \mathtt{try}\, e_1\, \mathtt{ow}\, e_2) \mapsto (\mathtt{try}\,\square\,\mathtt{ow}\, e_2 \triangleright K, e_1)} \tag{13.4}$$

$$\frac{}{(\mathtt{try}\,\square\,\mathtt{ow}\, e_2 \triangleright K, v) \mapsto (K, v)} \tag{13.5}$$

$$\frac{}{(\mathtt{try}\,\square\,\mathtt{ow}\, e_2 \triangleright K, \mathtt{fail}) \mapsto (K, e_2)} \tag{13.6}$$

$$\frac{(F \neq \mathtt{try}\,\square\,\mathtt{ow}\, e_2)}{(F \triangleright K, \mathtt{fail}) \mapsto (K, \mathtt{fail})} \tag{13.7}$$

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

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

Theorem 13.1 (Preservation)
If (K, e) ok *and* $(K, e) \mapsto (K', e')$*, then* (K, e) ok*.*

Proof: By induction on evaluation. ■

Theorem 13.2 (Progress)
If (K, e) ok *then either*

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

Proof: By induction on typing. ■

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

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

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

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

$$\frac{}{\bullet \ \mathsf{hstack}} \tag{13.8}$$

$$\frac{K \ \mathsf{stack} \quad e \ \mathsf{expr} \quad H \ \mathsf{hstack}}{(K, e) \triangleright H \ \mathsf{hstack}} \tag{13.9}$$

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:

$$\overline{((K', e') \triangleright H, K, \mathtt{fail}) \mapsto (H, K', e')} \tag{13.10}$$

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

$$\overline{(\bullet, K, \mathtt{fail}) \mapsto (\bullet, \bullet, \mathtt{fail})} \tag{13.11}$$

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

$$\overline{(H, K, \mathtt{try}\, e_1\, \mathtt{ow}\, e_2) \mapsto ((K, e_2) \triangleright H, \mathtt{try}\, \square\, \mathtt{ow}\, e_2 \triangleright K, e_1)} \tag{13.12}$$

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

$$\overline{((K, e_2) \triangleright H, \mathtt{try}\, \square\, \mathtt{ow}\, e_2 \triangleright K, v_1) \mapsto (H, K, v_1)} \tag{13.13}$$

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

The prefix property may be taken as a formal justification of an implementation based on the `setjmp` and and `longjmp` constructs of the C language. Unlike `setjmp` and `longjmp`, the exception mechanism is completely safe — it is impossible to return past the "finger" yet later attempt to "pop" the control stack to that point. In C the fingers are kept as addresses (pointers) in memory, and there is no discipline for ensuring that the set point makes any sense when invoked later in a computation.

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

$$\frac{\Gamma \vdash e : \tau_{\mathsf{exn}}}{\Gamma \vdash \texttt{raise(}e\texttt{)} : \tau} \tag{13.14}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau_{\mathsf{exn}} \rightarrow \tau}{\Gamma \vdash \texttt{try}\, e_1\, \texttt{ow}\, e_2 : \tau} \tag{13.15}$$

These rules are parameterized by the type of values associated with exceptions, τ_{exn}.

The question is: what should be the type τ_{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 τ_{exn} that supports a flexible programming style.

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

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

However, consider the plight of the poor handler, which may wish to distinguish between division-by-zero and arithmetic overflow. How might it

do that? If exception values were strings, it would have to parse the string, relying on the message to be in a standard format, and dispatch based on the parse. This is manifestly unworkable. For similar reasons we wouldn't choose $\tau_{\texttt{exn}}$ 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 $\tau_{\texttt{exn}}$ 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 α-conversion) to be distinct from all other exceptions that may occur elsewhere in the program, even if they happen to have the same name. If two modules declare an exception named `Error`, then these are *different* exceptions; no confusion is possible.

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

13.1 Exercises

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

2. Prove Theorem 13.1.

3. Prove Theorem 13.2.

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

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

   (b) Invocation of a handler rolls back the memory to the state at the point of installation of the handler.

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

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

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

Part V

Imperative Functional Programming

Chapter 14

Mutable Storage

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

14.1 References

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

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

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 Λ is a *location typing*, a finite function mapping locations to types; the other components of the judgement are as for MinML. The location typing Λ records the

types of allocated locations during execution; this is critical for a precise statement and proof of type soundness.

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

$$\frac{(\Lambda(l) = \tau)}{\Lambda;\Gamma \vdash l : \tau\,\mathtt{ref}} \tag{14.1}$$

$$\frac{\Lambda;\Gamma \vdash e : \tau}{\Lambda;\Gamma \vdash \mathtt{ref}(e) : \tau\,\mathtt{ref}} \tag{14.2}$$

$$\frac{\Lambda;\Gamma \vdash e : \tau\,\mathtt{ref}}{\Lambda;\Gamma \vdash\ !e : \tau} \tag{14.3}$$

$$\frac{\Lambda;\Gamma \vdash e_1 : \tau_2\,\mathtt{ref} \qquad \Lambda;\Gamma \vdash e_2 : \tau_2}{\Lambda;\Gamma \vdash e_1 :\!= e_2 : \tau_2} \tag{14.4}$$

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

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

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

$$\frac{(M, e) \mapsto (M', e')}{(M, \mathtt{ref}(e)) \mapsto (M', \mathtt{ref}(e'))} \tag{14.5}$$

$$\frac{(l \notin \mathrm{dom}(M))}{(M, \mathtt{ref}(v)) \mapsto (M[l{:=}v], l)} \tag{14.6}$$

$$\frac{(M, e) \mapsto (M', e')}{(M, !e) \mapsto (M', !e')} \tag{14.7}$$

$$\frac{(l \in \mathrm{dom}(M))}{(M, !l) \mapsto (M, M(l))} \tag{14.8}$$

$$\frac{(M, e_1) \mapsto (M', e_1')}{(M, e_1 := e_2) \mapsto (M', e_1' := e_2)} \tag{14.9}$$

$$\frac{(M, e_2) \mapsto (M', e_2')}{(M, v_1 := e_2) \mapsto (M', v_1 := e_2')} \tag{14.10}$$

$$\frac{(l \in \mathrm{dom}(M))}{(M, l := v) \mapsto (M[l=v], v)} \tag{14.11}$$

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

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

$$\frac{\mathrm{dom}(M) = \mathrm{dom}(\Lambda) \quad \forall l \in \mathrm{dom}(\Lambda)\ \Lambda; \bullet \vdash M(l) : \Lambda(l)}{\vdash M : \Lambda} \tag{14.12}$$

It is very important to study this rule carefully! First, we require that Λ 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 Λ. This means, in particular, that memories may be "circular" in the sense that the value at location l may contain an occurrence of l, for example if that value is a function.

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

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

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

Exercise 14.1

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* $M(l_1) = l_2$, $M(l_2) = l_3$, $\ldots$, $M(l_n) = l_1$ *for some sequence* $l_1, \ldots, l_n$ *of locations. Show that there is no location typing* Λ *such that* $\vdash M : \Lambda$.

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

$$\frac{\vdash M : \Lambda \qquad \Lambda; \bullet \vdash e : \tau}{(M, e)\ \mathsf{ok}} \tag{14.13}$$

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

Theorem 14.2 (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 14.3
Prove Theorem 14.2. The strengthened form tells us that the location typing, and the memory, increase monotonically during evaluation — the type of a location never changes once it is established at the point of allocation. This is crucial for the induction.

Theorem 14.4 (Progress)
If (M, e) ok *then either* (M, e) *is a final state or there exists* (M', e') *such that* $(M, e) \mapsto (M', e')$.

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

Exercise 14.5
Prove Theorem 14.4 by induction on typing of machine states.

Chapter 15

Monads

As we saw in Chapter 14 one way to combine functional and imperative programming is to add a type of reference cells to MinML. This approach works well for call-by-value languages,[1] because we can easily predict where expressions are evaluated, and hence where references are allocated and assigned. For call-by-name languages this approach is problematic, because in such languages it is much harder to predict when (and how often) expressions are evaluated.

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

Another approach to introducing effects in a purely functional language is to make the use of effects explicit in the type system. Several methods have been proposed, but the most elegant and widely used is the concept of a *monad*. Roughly speaking, we distinguish between *pure* and *impure* expressions, and make a corresponding distinction between *pure* and *impure* function types. Then a function of type `int`→`int` is a pure function (has no effects when evaluated), whereas a function of type `int` ⇀ `int` may have an effect when applied. The monadic approach is

[1] We need to introduce cbv and cbn earlier, say in Chapter 9.

more popular for call-by-name languages, but is equally sensible for call-by-value languages.

15.1 Monadic MinML

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

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

$$
\begin{array}{llll}
\textit{Types} & \tau & ::= & \texttt{int} \mid \texttt{bool} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \rightharpoonup \tau_2 \\
\textit{Pure} & e & ::= & x \mid n \mid o(e_1,\ldots,e_n) \mid \\
 & & & \texttt{true} \mid \texttt{false} \mid \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \mid \\
 & & & \texttt{fun}\, f\, (x{:}\tau_1){:}\tau_2\, \texttt{is}\, e \mid \texttt{apply}(e_1, e_2) \\
 & & & \texttt{fun}\, f\, (x{:}\tau_1){:}\tau_2\, \texttt{is}\, m\, \texttt{end} \\
\textit{Impure} & m & ::= & \texttt{return}\, e \mid \texttt{bind}\, x{:}\tau \leftarrow m_1\, \texttt{in}\, m_2 \\
 & & & \texttt{if}_\tau\, e\, \texttt{then}\, m_1\, \texttt{else}\, m_2\, \texttt{fi} \mid \texttt{apply}(e_1, e_2)
\end{array}
$$

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

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

below.

$$\frac{\Gamma, f{:}\tau_1 \rightharpoonup \tau_2, x{:}\tau_1 \vdash m : \tau_2}{\Gamma \vdash \mathtt{fun}\, f\ (x:\tau_1):\tau_2\ \mathtt{is}\, m\, \mathtt{end} : \tau_1 \rightharpoonup \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \rightharpoonup \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{apply}(e_1, e_2) : \tau}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{return}\, e : \tau}$$

$$\frac{\Gamma \vdash m_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash m_2 : \tau_2}{\Gamma \vdash \mathtt{bind}\, x{:}\tau \leftarrow m_1\, \mathtt{in}\, m_2 : \tau_2}$$

$$\frac{\Gamma \vdash e : \mathtt{bool} \quad \Gamma \vdash m_1 : \tau \quad \Gamma \vdash m_2 : \tau}{\Gamma \vdash \mathtt{if}_\tau\, e\, \mathtt{then}\, m_1\, \mathtt{else}\, m_2\, \mathtt{fi} : \tau}$$

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

$$\mathit{Impure} \quad m \quad ::= \quad \mathtt{ref}(e) \mid\ !e \mid e_1 := e_2$$

Their typing rules are as follows:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{ref}(e) : \tau\, \mathtt{ref}}$$

$$\frac{\Gamma \vdash e : \tau\, \mathtt{ref}}{\Gamma \vdash\ !e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau\, \mathtt{ref} \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 := e_2 : \tau_2}$$

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

$$\frac{(\Gamma(l) = \tau)}{\Gamma \vdash l : \tau\, \mathtt{ref}}$$

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

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

$$\frac{e \mapsto e'}{(M, \mathtt{return}\, e) \mapsto (M, \mathtt{return}\, e')}$$

$$\frac{(M, m_1) \mapsto (M', m_1')}{(M, \mathtt{bind}\, x{:}\tau \leftarrow m_1 \,\mathtt{in}\, m_2) \mapsto (M', \mathtt{bind}\, x{:}\tau \leftarrow m_1' \,\mathtt{in}\, m_2)}$$

$$\frac{}{(M, \mathtt{bind}\, x{:}\tau \leftarrow \mathtt{return}\, v \,\mathtt{in}\, m_2) \mapsto (M, \{v/x\}m_2)}$$

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

15.2 Reifying Effects

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

The syntax required for this extension is as follows:

Types	τ	$::=$	$!\,\tau$
Pure	e	$::=$	$\mathtt{box}(m)$
Impure	m	$::=$	$\mathtt{unbox}(e)$

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

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

$$\frac{\Gamma \vdash m : \tau}{\Gamma \vdash \mathtt{box}(m) : !\,\tau}$$

$$\frac{\Gamma \vdash e : !\,\tau}{\Gamma \vdash \mathtt{unbox}(e) : \tau}$$

The dynamic semantics is given by the following transition rules:

$$\frac{}{(M, \mathtt{unbox}(\mathtt{box}(m))) \mapsto (M, m)}$$

$$\frac{e \mapsto e'}{(M, \mathtt{unbox}(e)) \mapsto (M, \mathtt{unbox}(e'))}$$

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

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

$$\mathtt{fun}\, f\ (x : \tau_1) : \tau_2\ \mathtt{is}\ m\ \mathtt{end}$$

is replaced by the pure function expression

$$\mathtt{fun}\, f\ (x : \tau_1) : \tau_2\ \mathtt{is}\ \mathtt{box}(m)\ \mathtt{end}.$$

The impure application,

$$\mathtt{apply}(e_1, e_2),$$

is replaced by

$$\mathtt{unbox}(\mathtt{apply}(e_1, e_2)),$$

which unboxes, hence executes, the suspended computation.

15.3 Exercises

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

Part VI

Cost Semantics and Parallelism

Chapter 16

Cost 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). One benefit of a transition semantics is that it provides a natural measure of the time complexity of an expression, namely the number of steps required to reach a value.

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

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

16.1 Evaluation Semantics

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

$$\frac{}{n \Downarrow n} \tag{16.1}$$

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{+(e_1, e_2) \Downarrow n_1 + n_2} \tag{16.2}$$

(and similarly for the other primitive operations).

$$\frac{}{\texttt{true} \Downarrow \texttt{true}} \qquad \frac{}{\texttt{false} \Downarrow \texttt{false}} \tag{16.3}$$

$$\frac{e \Downarrow \texttt{true} \quad e_1 \Downarrow v}{\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \Downarrow v} \tag{16.4}$$

$$\frac{e \Downarrow \texttt{false} \quad e_2 \Downarrow v}{\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \Downarrow v} \tag{16.5}$$

$$\frac{}{\texttt{fun}\, f\, (x:\tau_1):\tau_2\, \texttt{is}\, e \Downarrow \texttt{fun}\, f\, (x:\tau_1):\tau_2\, \texttt{is}\, e} \tag{16.6}$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad \{v_1, v_2/f, x\}e \Downarrow v}{\texttt{apply}(e_1, e_2) \Downarrow v} \tag{16.7}$$

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

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

16.2 Relating Evaluation Semantics to Transition Semantics

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

Theorem 16.1

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

Proof:

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

It follows that $\texttt{apply}(e_1, e_2) \mapsto^* \texttt{apply}(v_1, e_2) \mapsto^* \texttt{apply}(v_1, v_2) \mapsto \{v_1, v_2/f, x\}e \mapsto^* v$, as required. The other cases are handled similarly.

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

■

16.3 Cost Semantics

In this section we will give a cost semantics for MinML that reflects the number of steps required to complete evaluation according to the structured operational semantics given in Chapter 9.

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.

$$\frac{}{n \Downarrow^0 n} \tag{16.8}$$

$$\frac{e_1 \Downarrow^{k_1} n_1 \quad e_2 \Downarrow^{k_2} n_2}{+(e_1, e_2) \Downarrow^{k_1+k_2+1} n_1 + n_2} \tag{16.9}$$

(and similarly for the other primitive operations).

$$\frac{}{\texttt{true} \Downarrow^0 \texttt{true}} \qquad \frac{}{\texttt{false} \Downarrow^0 \texttt{false}} \tag{16.10}$$

$$\frac{e \Downarrow^{k} \texttt{true} \quad e_1 \Downarrow^{k_1} v}{\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \Downarrow^{k+k_1+1} v} \tag{16.11}$$

$$\frac{e \Downarrow^{k} \texttt{false} \quad e_2 \Downarrow^{k_2} v}{\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \Downarrow^{k+k_2+1} v} \tag{16.12}$$

$$\frac{}{\texttt{fun}\, f\, (x:\tau_1):\tau_2\, \texttt{is}\, e \Downarrow^{0} \texttt{fun}\, f\, (x:\tau_1):\tau_2\, \texttt{is}\, e} \tag{16.13}$$

$$\frac{e_1 \Downarrow^{k_1} v_1 \quad e_2 \Downarrow^{k_2} v_2 \quad \{v_1, v_2/f, x\}e \Downarrow^{k} v}{\texttt{apply}(e_1, e_2) \Downarrow^{k_1+k_2+k+1} v} \tag{16.14}$$

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

This completes the definition of the cost semantics for MinML.

16.4 Relating Cost Semantics to Transition Semantics

What is it that makes the cost semantics given above "correct"? Informally, we expect that if $e \Downarrow^{k} v$, then e should evaluate to v in k steps. Moreover, we also expect the converse to hold — the cost semantics should be completely faithful to the underlying execution model. This is captured by the following theorem.

To state the theorem we need one additional bit of notation. Define $e \overset{k}{\mapsto} e'$ by induction on k as follows. For the basis, we define $e \overset{0}{\mapsto} e'$ iff $e = e'$; if $k = k' + 1$, we define $e \overset{k}{\mapsto} e'$ to hold iff $e \mapsto e'' \overset{k'}{\mapsto} e'$.

Theorem 16.2
For any closed expression e and closed value v of the same type, $e \Downarrow^{k} v$ iff $e \overset{k}{\mapsto} v$.

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 $e = \texttt{apply}(e_1, e_2)$ and $k = k_1 + k_2 + k + 1$, where

1. $e_1 \Downarrow^{k_1} v_1$,

2. $e_2 \Downarrow^{k_2} v_2$,
3. $v_1 = \mathtt{fun}\, f\ (x:\tau_1):\tau_2\ \mathtt{is}\, e$,
4. $\{v_1, v_2/f, x\}e \Downarrow^{k} v$.

By induction we have

1. $e_1 \stackrel{k_1}{\mapsto} v_1$,
2. $e_2 \stackrel{k_2}{\mapsto} v_2$,
3. $\{v_1, v_2/f, x\}e \stackrel{k}{\mapsto} v$,

and hence

$$\begin{array}{lcl} e_1(e_2) & \stackrel{k_1}{\mapsto} & v_1(e_2) \\ & \stackrel{k_2}{\mapsto} & v_1(v_2) \\ & \mapsto & \{v_1, v_2/f, x\}e \\ & \stackrel{k}{\mapsto} & v \end{array}$$

which is enough for the result.

From right to left we proceed by induction on k. For $k = 0$, we must have $e = v$. By inspection of the cost evaluation rules we may check that $v \Downarrow^0 v$ for every value v. For $k = k' + 1$, we must show that if $e \mapsto e'$ 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. $v_1 = \mathtt{fun}\, f\ (x:\tau_1):\tau_2\ \mathtt{is}\, e$,
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. ■

16.5 Exercises

Chapter 17

Implicit Parallelism

In this chapter we study the extension of MinML with *implicit data parallelism*, a means of speeding up computations by allowing expressions to be evaluated simultaneously. By "implicit" we mean that the use of parallelism is invisible to the programmer as far as the ultimate results of computation are concerned. By "data parallel" we mean that the parallelism in a program arises from the simultaneous evaluation of the components of a data structure.

Implicit parallelism is very natural in an effect-free language such as MinML. The reason is that in such a language it is not possible to determine the order in which the components of an aggregate data structure are evaluated. They might be evaluated in an arbitrary sequential order, or might even be evaluated simultaneously, without affecting the outcome of the computation. This is in sharp contrast to effect-ful languages, for then the order of evaluation, or the use of parallelism, is visible to the programmer. Indeed, dependence on the evaluation order must be carefully guarded against to ensure that the outcome is determinate.

17.1 Tuple Parallelism

We begin by considering a parallel semantics for tuples according to which all components of a tuple are evaluated simultaneously. For simplicity we consider only pairs, but the ideas generalize in a straightforward manner to tuples of any size. Since the "widths" of tuples are specified statically as part of their type, the amount of parallelism that can be induced in any

one step is bounded by a static constant. In Section 17.3 we will extend this to permit a statically unbounded degree of parallelism.

To facilitate comparison, we will consider two operational semantics for this extension of MinML, the *sequential* and the *parallel*. The sequential semantics is as in Chapter 19. However, we now write $e \mapsto_{seq} e'$ for the transition relation to stress that this is the sequential semantics. The sequential evaluation rules for pairs are as follows:

$$\frac{e_1 \mapsto_{seq} e_1'}{(e_1, e_2) \mapsto_{seq} (e_1', e_2)} \tag{17.1}$$

$$\frac{v_1 \text{ value} \quad e_2 \mapsto_{seq} e_2'}{(v_1, e_2) \mapsto_{seq} (v_1, e_2')} \tag{17.2}$$

$$\frac{v_1 \text{ value} \quad v_2 \text{ value}}{\texttt{split}\,(v_1, v_2)\,\texttt{as}\,(x, y)\,\texttt{in}\,e \mapsto_{seq} \{v_1, v_2/x, y\}e} \tag{17.3}$$

$$\frac{e_1 \mapsto_{seq} e_1'}{\texttt{split}\,e_1\,\texttt{as}\,(x, y)\,\texttt{in}\,e_2 \mapsto_{seq} \texttt{split}\,e_1'\,\texttt{as}\,(x, y)\,\texttt{in}\,e_2} \tag{17.4}$$

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:[1]

$$\frac{e_1 \mapsto_{par} e_1' \quad e_2 \mapsto_{par} e_2'}{(e_1, e_2) \mapsto_{par} (e_1', e_2')} \tag{17.5}$$

$$\frac{e_1 \mapsto_{par} e_1' \quad v_2 \text{ value}}{(e_1, v_2) \mapsto_{par} (e_1', v_2)} \tag{17.6}$$

$$\frac{v_1 \text{ value} \quad e_2 \mapsto_{par} e_2'}{(v_1, e_2) \mapsto_{par} (v_1, e_2')} \tag{17.7}$$

Three rules are required to account for the possibility that evaluation of one component may complete before the other.

[1] It might be preferable to admit progress on either e_1 or e_2 alone, without requiring the other to be a value.

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 17.1
For every closed, well-typed expression e, $e \mapsto^*_{seq} v$ iff $e \mapsto^*_{par} v$.

Proof: For the implication from left to right, it suffices to show that if $e \mapsto_{seq} e' \mapsto^*_{par} v$, then $e \mapsto^*_{par} v$. This is proved by induction on the sequential evaluation relation. For example, suppose that

$$(e_1, e_2) \mapsto_{seq} (e_1', e_2) \mapsto^*_{par} (v_1, v_2),$$

where $e_1 \mapsto_{seq} e_1'$. By inversion of the parallel evaluation sequence, we have $e_1' \mapsto^*_{par} v_1$ and $e_2 \mapsto^*_{par} v_2$. Hence, by induction, $e_1 \mapsto^*_{par} v_1$, from which it follows immediately that $(e_1, e_2) \mapsto^*_{par} (v_1, v_2)$. 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 $e \mapsto_{par} e' \mapsto^*_{seq} v$, then $e \mapsto^*_{seq} v$. We proceed by induction on the definition of the parallel evaluation relation. For example, suppose that we have

$$(e_1, e_2) \mapsto_{par} (e_1', e_2') \mapsto^*_{seq} (v_1, v_2)$$

with $e_1 \mapsto_{par} e_1'$ and $e_2 \mapsto_{par} e_2'$. We are to show that $(e_1, e_2) \mapsto^*_{seq} (v_1, v_2)$. Since $(e_1', e_2') \mapsto^*_{seq} (v_1, v_2)$, it follows that $e_1' \mapsto^*_{seq} v_1$ and $e_2' \mapsto^*_{seq} v_2$. By induction $e_1 \mapsto^*_{seq} v_1$ and $e_2 \mapsto^*_{seq} v_2$, which is enough for the result. The other cases of evaluation for pairs are handled similarly.

■

One important consequence of this theorem is that parallelism is *semantically invisible*: whether we use parallel or sequential evaluation of pairs, the result is the same. Consequently, parallelism may safely be left *implicit*, at least as far as *correctness* is concerned. However, as one might expect, parallelism effects the *efficiency* of programs.

17.2 Work and Depth

An operational semantics for a language induces a measure of time complexity for expressions, namely the number of steps required to evaluate that expression to a value. The *sequential complexity* of an expression is its time complexity relative to the sequential semantics; the *parallel complexity* is its time complexity relative to the paralle semantics. These can, in general, be quite different. Consider, for example, the following naïve implementation of the Fibonacci sequence in MinML with products:

```
fun fib (n:int):int is
    if n=0 then 1
    else if n=1 then 1
    else plus(fib(n-1),fib(n-2)) fi fi
```

where `plus` is the following function on ordered pairs:

```
fun plus (p:int*int):int is
    split p as (m:int,n:int) in m+n
```

The sequential complexity of `fib` n is $O(2^n)$, whereas the parallel complexity of the same expression is $O(n)$. The reason is that each recursive call spawns two further recursive calls which, if evaluated sequentially, lead to an exponential number of steps to complete. However, if the two recursive calls are evaluated in parallel, then the number of parallel steps to completion is bounded by n, since n is decreased by 1 or 2 on each call. Note that the same number of arithmetic operations is performed in each case! The difference is only in whether they are performed simultaneously.

This leads naturally to the concepts of *work* and *depth*. The *work* of an expression is the total number of primitive instruction steps required to complete evaluation. Since the sequential semantics has the property that each rule has at most one premise, each step of the sequential semantics amounts to the execution of exactly one instruction. Therefore the sequential complexity coincides with the work required. (Indeed, work and sequential complexity are often taken to be synonymous.) The work required to evaluate `fib` n is $O(2^n)$.

On the other hand the *depth* of an expression is the length of the longest chain of sequential dependencies in a complete evaluation of that expression. A sequential dependency is induced whenever the value of one expression depends on the value of another, forcing a sequential evaluation

ordering between them. In the Fibonacci example the two recursive calls have no sequential dependency among them, but the function itself sequentially depends on both recursive calls — it cannot return until both calls have returned. Since the parallel semantics evaluates both components of an ordered pair simultaneously, it exactly captures the independence of the two calls from each, but the dependence of the result on both. Thus the parallel complexity coincides with the depth of the computation. (Indeed, they are often taken to be synonymous.) The depth of the expression `fib` n is $O(n)$.

With this in mind, the cost semantics introduced in Chapter 16 may be extended to account for parallelism by specifying both the work and the depth of evaluation. The judgements of the parallel cost semantics have the form $e \Downarrow^{w,d} v$, where w is the work and d the depth. For all cases but evaluation of pairs the work and the depth track one another. The rule for pairs is as follows:

$$\frac{e_1 \Downarrow^{w_1,d_1} v_1 \quad e_2 \Downarrow^{w_2,d_2} v_2}{(e_1,e_2) \Downarrow^{w_1+w_2,\max(d_1,d_2)} (v_1,v_2)} \tag{17.8}$$

The remaining rules are easily derived from the sequential cost semantics, with both work and depth being additively combined at each step.[2]

The correctness of the cost semantics states that the work and depth costs are consistent with the sequential and parallel complexity, respectively, of the expression.

Theorem 17.2
For any closed, well-typed expression e, $e \Downarrow^{w,d} v$ iff $e \mapsto^{w}_{seq} v$ and $e \mapsto^{d}_{par} v$.

Proof: From left to right, we proceed by induction on the cost semantics. For example, we must show that if $e_1 \mapsto^{d_1}_{par} v_1$ and $e_2 \mapsto^{d_2}_{par} v_2$, then

$$(e_1,e_2) \mapsto^{d}_{par} (v_1,v_2),$$

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 $e_1 \mapsto^{d_1}_{par} v_1$ and $e_2 \mapsto^{d_1}_{par} e_2' \mapsto^{d'}_{par} v_2$.

[2] If we choose, we might evaluate arguments of primop's in parallel, in which case the depth complexity would be calculated as one more than the maximum of the depths of its arguments. We will not do this here since it would only complicate the development.

It follows that

$$\begin{array}{lll}(e_1,e_2) & \mapsto_{par}^{d_1} & (v_1,e_2') \\ & \mapsto_{par}^{d'} & (v_1,v_2).\end{array}$$

For the converse, we proceed by considering work and depth costs separately. For work, we proceed as in Chapter 16. For depth, it suffices to show that if $e \mapsto_{par} e'$ and $e' \Downarrow^{d} v$, then $e \Downarrow^{d+1} v$.[3] For example, suppose that $(e_1,e_2) \mapsto_{par} (e_1',e_2')$, with $e_1 \mapsto_{par} e_1'$ and $e_2 \mapsto_{par} e_2'$. Since $(e_1',e_2') \Downarrow^{d} v$, we must have $v = (v_1,v_2)$, $d = \max(d_1,d_2)$ with $e_1' \Downarrow^{d_1} v_1$ and $e_2' \Downarrow^{d_2} v_2$. By induction $e_1 \Downarrow^{d_1+1} v_1$ and $e_2 \Downarrow^{d_2+1} v_2$ and hence $(e_1,e_2) \Downarrow^{d+1} (v_1,v_2)$, as desired. ■

17.3 Vector Parallelism

To support *vector parallelism* we will extend MinML with a type of *vectors*, which are finite sequences of values of a given type whose length is not determined until execution time. The primitive operations on vectors are chosen so that they may be executed in parallel on a *shared memory multiprocessor*, or *SMP*, in constant depth for an arbitrary vector.

The following primitives are added to MinML to support vectors:

$$\begin{array}{llll}\textit{Types} & \tau & ::= & \tau\,\texttt{vector} \\ \textit{Expr's} & e & ::= & [e_0,\ldots,e_{n-1}] \mid \texttt{elt}(e_1,e_2) \mid \texttt{size}(e) \mid \texttt{index}(e) \mid \\ & & & \texttt{map}(e_1,e_2) \mid \texttt{update}(e_1,e_2) \\ \textit{Values} & v & ::= & [v_0,\ldots,v_{n-1}]\end{array}$$

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 $\texttt{elt}(e_1,e_2)$ retrieves the element of the vector given by e_1 at the index given by e_2. The operation $\texttt{size}(e)$ returns the number of elements in the vector given by e. The operation $\texttt{index}(e)$ creates a vector of length n (given by e) whose elements are $0,\ldots,n-1$. The operation $\texttt{map}(e_1,e_2)$ applies the function given by e_1 to every element of e_2 in parallel. Finally, the operation $\texttt{update}(e_1,e_2)$ yields a new vector of the same size, n, as the vector v given by e_1, but

[3] The work component of the cost is suppressed here for the sake of clarity.

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 ith element of v should be replaced by x, provided that $b = \texttt{true}$.

The static semantics of these primitives is given by the following typing rules:

$$\frac{\Gamma \vdash e_1 : \tau \quad \cdots \quad \Gamma \vdash e_n : \tau}{\Gamma \vdash \texttt{[}e_0\texttt{,} \ldots \texttt{,}e_{n-1}\texttt{]} : \tau\ \texttt{vector}} \tag{17.9}$$

$$\frac{\Gamma \vdash e_1 : \tau\ \texttt{vector} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{elt(}e_1\texttt{,}e_2\texttt{)} : \tau} \tag{17.10}$$

$$\frac{\Gamma \vdash e : \tau\ \texttt{vector}}{\Gamma \vdash \texttt{size(}e\texttt{)} : \texttt{int}} \tag{17.11}$$

$$\frac{\Gamma \vdash e : \texttt{int}}{\Gamma \vdash \texttt{index(}e\texttt{)} : \texttt{int vector}} \tag{17.12}$$

$$\frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau\ \texttt{vector}}{\Gamma \vdash \texttt{map(}e_1\texttt{,}e_2\texttt{)} : \tau'\ \texttt{vector}} \tag{17.13}$$

$$\frac{\Gamma \vdash e_1 : \tau\ \texttt{vector} \quad \Gamma \vdash e_2 : (\texttt{bool*int*}\tau)\ \texttt{vector}}{\Gamma \vdash \texttt{update(}e_1\texttt{,}e_2\texttt{)} : \tau\ \texttt{vector}} \tag{17.14}$$

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 parallelism:

$$\frac{\forall i \in I\ (e_i \mapsto_{par} e_i') \quad \forall i \notin I\ (e_i' = e_i\ \&\ e_i\ \mathsf{value})}{\texttt{[}e_0\texttt{,} \ldots \texttt{,}e_{n-1}\texttt{]} \mapsto_{par} \texttt{[}e_0'\texttt{,} \ldots \texttt{,}e_{n-1}'\texttt{]}} \tag{17.15}$$

where $\emptyset \neq I \subseteq \{0, \ldots, n-1\}$. This allows for the parallel evaluation of all components of the vector that have not yet been evaluated.

For each of the primitive operations of the language there is a rule specifying that its arguments are evaluated in left-to-right order. We omit these rules here for the sake of brevity. The primitive instructions are as follows:

$$\frac{}{\texttt{elt(}\texttt{[}v_0\texttt{,} \ldots \texttt{,}v_{n-1}\texttt{]}\texttt{,}i\texttt{)} \mapsto_{par} v_i} \tag{17.16}$$

$$\frac{}{\texttt{size}([v_0,\ldots,v_{n-1}]) \mapsto_{par} n} \tag{17.17}$$

$$\frac{}{\texttt{index}(n) \mapsto_{par} [0,\ldots,n-1]} \tag{17.18}$$

$$\frac{}{\texttt{map}(v,[v_0,\ldots,v_{n-1}]) \mapsto_{par} [\texttt{apply}(v,v_0),\ldots,\texttt{apply}(v,v_{n-1})]} \tag{17.19}$$

$$\frac{}{\begin{array}{c}\texttt{update}([v_0,\ldots,v_{n-1}],[(b_0,i_0,x_0),\ldots,(b_{k-1},i_{k-1},x_{k-1})]) \\ \mapsto_{par} \\ [v'_0,\ldots,v'_{n-1}]\end{array}} \tag{17.20}$$

where for each $i \in \{ i_0,\ldots,i_{k-1} \}$, 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.

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:

$$\frac{e_i \mapsto_{seq} e'_i}{[v_0,\ldots,v_{i-1},e_i,e_{i+1},\ldots,e_{n-1}] \mapsto [v_0,\ldots,v_{i-1},e'_i,e_{i+1},\ldots,e_{n-1}]} \tag{17.21}$$

We write $e \mapsto_{seq} e'$ 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.

$$\frac{\forall\, 0 \leq i < n\ (e_i \Downarrow^{w_i,d_i} v_i)}{[e_0,\ldots,e_{n-1}] \Downarrow^{w,d} [v_0,\ldots,v_{n-1}]} \tag{17.22}$$

where $w = \sum_{i=0}^{n-1} w_i$ and $d = \max_{i=0}^{n-1} d_i$.

Retrieving an element of a vector takes constant work and depth.

$$\frac{e_1 \Downarrow^{w_1,d_1} [v_0,\ldots,v_{n-1}] \quad e_2 \Downarrow^{w_2,d_2} i \quad (0 \leq i < n)}{\texttt{elt}(e_1,e_2) \Downarrow^{w_1+w_2+1,d_1+d_2+1} v_i} \tag{17.23}$$

Retrieving the size of a vector takes constant work and depth.

$$\frac{e \Downarrow^{w,d} [v_0,\ldots,v_{n-1}]}{\texttt{size}(e) \Downarrow^{w+1,d+1} n} \tag{17.24}$$

Creating an index vector takes linear work and constant depth.

$$\frac{e \Downarrow^{w,d} n}{\texttt{index}(e) \Downarrow^{w+n,d+1} [0,\ldots,n-1]} \tag{17.25}$$

Mapping a function across a vector takes constant work and depth beyond the cost of the function applications.

$$\frac{\begin{array}{c} e_1 \Downarrow^{w_1,d_1} v \quad e_2 \Downarrow^{w_2,d_2} [v_0,\ldots,v_{n-1}] \\ [\texttt{apply}(v,v_0),\ldots,\texttt{apply}(v,v_{n-1})] \Downarrow^{w,d} [v'_0,\ldots,v'_{n-1}] \end{array}}{\texttt{map}(e_1,e_2) \Downarrow^{w_1+w_2+w+1,d_1+d_2+d+1} [v'_0,\ldots,v'_{n-1}]} \tag{17.26}$$

Updating a vector takes linear work and constant depth.

$$\frac{e_1 \Downarrow^{w_1,d_1} [v_0,\ldots,v_{n-1}] \quad e_2 \Downarrow^{w_2,d_2} [(b_1,i_1,x_1),\ldots,(b_k,i_k,x_k)]}{\texttt{update}(e_1,e_2) \Downarrow^{w_1+w_2+k+n,d_1+d_2+1} [v'_0,\ldots,v'_{n-1}]} \tag{17.27}$$

where for each $i \in \{i_1,\ldots,i_k\}$, if b_i is `true`, then $v'_i = x_i$, and otherwise $v'_i = v_i$. If an index i appears more than once, the rightmost occurrence takes precedence over the others.

Theorem 17.3
For the extension of MinML with vectors, $e \Downarrow^{w,d} v$ iff $e \mapsto^{d}_{par} v$ and $e \mapsto^{w}_{seq} v$.

Chapter 18

A Parallel Abstract Machine

The parallel operational semantics described in Chapter 17 abstracts away some important aspects of the implementation of parallelism. For example, the parallel evaluation rule for ordered pairs

$$\frac{e_1 \mapsto_{par} e_1' \quad e_2 \mapsto_{par} e_2'}{(e_1, e_2) \mapsto_{par} (e_1', e_2')}$$

does not account for the overhead of allocating e_1 and e_2 to two (physical or virtual) processors, or for synchronizing with those two processors to obtain their results. In this chapter we will discuss a more realistic operational semantics that accounts for this overhead.

18.1 A Simple Parallel Language

Rather than specify which primitives, such as pairing, are to be evaluated in parallel, we instead introduce a "parallel let" construct that allows the programmer to specify the simultaneous evaluation of two expressions. Moreover, we restrict the language so that the arguments to all primitive operations must be values. This forces the programmer to decide for herself which constructs are to be evaluated in parallel, and which are to be evaluated sequentially.

$$
\begin{array}{lrcl}
\textit{Types} & \tau & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{unit} \mid \tau_1 \texttt{*} \tau_2 \mid \tau_1 \rightarrow \tau_2 \\
\textit{Expressions} & e & ::= & v \mid \texttt{let}\ x_1 : \tau_1\ \texttt{be}\ e_1\ \texttt{and}\ x_2 : \tau_2\ \texttt{be}\ e_2\ \texttt{in}\ e\ \texttt{end} \mid \\
 & & & o(v_1, \ldots, v_n) \mid \texttt{if}\ \tau\ \texttt{then}\ v\ \texttt{else}\ e_1 e_2 \mid \\
 & & & \texttt{apply}(v_1, v_2) \mid \texttt{split}\ v\ \texttt{as}\ (x_1, x_2)\ \texttt{in}\ e \\
\textit{Values} & v & ::= & x \mid n \mid \texttt{true} \mid \texttt{false} \mid () \mid (v_1, v_2) \mid \\
 & & & \texttt{fun}\ x\ (y : \tau_1) : \tau_2\ \texttt{is}\ e
\end{array}
$$

The binding conventions are as for MinML with product types, with the additional specification that the variables x_1 and x_2 are bound within the body of a `let` expression. Note that variables are regarded as values only for the purpose of defining the syntax of the language; evaluation is, as ever, defined only on closed terms.

As will become apparent when we specify the dynamic semantics, the "sequential let" is definable from the "parallel let":

$$\texttt{let}\ \tau_1 : x_1\ \texttt{be}\ e_1\ \texttt{in}\ e_2 := \texttt{let}\ x_1 : \tau_1\ \texttt{be}\ e_1\ \texttt{and}\ x : \texttt{unit}\ \texttt{be}\ ()\ \texttt{in}\ e_2\ \texttt{end}$$

where x does not occur free in e_2. Using these, the "parallel pair" is definable by the equation

$$(e_1, e_2)_{par} := \texttt{let}\ x_1 : \tau_1\ \texttt{be}\ e_1\ \texttt{and}\ x_2 : \tau_2\ \texttt{be}\ e_2\ \texttt{in}\ (x_1, x_2)\ \texttt{end}$$

whereas the "(left-to-right) sequential pair" is definable by the equation

$$(e_1, e_2)_{seq} := \texttt{let}\ \tau_1 : x_1\ \texttt{be}\ e_1\ \texttt{in}\ \texttt{let}\ \tau_2 : x_2\ \texttt{be}\ e_2\ \texttt{in}\ (x_1, x_2).$$

The static semantics of this language is essentially that of MinML with product types, with the addition of the following typing rule for the parallel let construct:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma, x_1{:}\tau_1, x_2{:}\tau_2 \vdash e : \tau}{\Gamma \vdash \texttt{let}\ x_1 : \tau_1\ \texttt{be}\ e_1\ \texttt{and}\ x_2 : \tau_2\ \texttt{be}\ e_2\ \texttt{in}\ e\ \texttt{end} : \tau} \tag{18.1}$$

It is a simple exercise to give a parallel structured operational semantics to this language in the style of Chapter 17. In particular, it would employ the following rules for the parallel `let` construct.

$$\frac{e_1 \mapsto_{par} e_1' \quad e_2 \mapsto_{par} e_2'}{\begin{array}{c}\texttt{let}\ x_1 : \tau_1\ \texttt{be}\ e_1\ \texttt{and}\ x_2 : \tau_2\ \texttt{be}\ e_2\ \texttt{in}\ e\ \texttt{end} \\ \mapsto_{par} \\ \texttt{let}\ x_1 : \tau_1\ \texttt{be}\ e_1'\ \texttt{and}\ x_2 : \tau_2\ \texttt{be}\ e_2'\ \texttt{in}\ e\ \texttt{end}\end{array}} \tag{18.2}$$

$$\frac{e_1 \mapsto_{par} e_1'}{\begin{array}{c}\texttt{let}\ x_1:\tau_1\ \texttt{be}\ e_1\ \texttt{and}\ x_2:\tau_2\ \texttt{be}\ v_2\ \texttt{in}\ e\ \texttt{end}\\ \mapsto_{par}\\ \texttt{let}\ x_1:\tau_1\ \texttt{be}\ e_1'\ \texttt{and}\ x_2:\tau_2\ \texttt{be}\ v_2\ \texttt{in}\ e\ \texttt{end}\end{array}} \tag{18.3}$$

$$\frac{e_2 \mapsto_{par} e_2'}{\begin{array}{c}\texttt{let}\ x_1:\tau_1\ \texttt{be}\ v_1\ \texttt{and}\ x_2:\tau_2\ \texttt{be}\ e_2\ \texttt{in}\ e\ \texttt{end}\\ \mapsto_{par}\\ \texttt{let}\ x_1:\tau_1\ \texttt{be}\ v_1\ \texttt{and}\ x_2:\tau_2\ \texttt{be}\ e_2'\ \texttt{in}\ e\ \texttt{end}\end{array}} \tag{18.4}$$

However, these rules ignore the overhead associated with allocating the sub-expression to processors. In the next section we will consider an abstract machine that accounts for this overhead.

Exercise 18.1
Prove preservation and progress for the static and dynamic semantics just given.

18.2 A Parallel Abstract Machine

The essence of parallelism is the simultaneous execution of several programs. Each execution is called a *thread of control*, or *thread*, for short. The problem of devising a parallel abstract machine is how to represent multiple threads of control, in particular how to represent the creation of new threads and synchronization between threads. The P-machine is designed to represent a parallel computer with an unbounded number of processors in a simple and elegant manner.

The main idea of the P-machine is represent the state of a parallel computer by a nested composition of parallel `let` statements representing the *active* threads in a program. Each step of the machine consists of executing *all* of the active instructions in the program, resulting in a new P-state.

In order to account for the activation of threads and the synchronization of their results we make explicit the process of *activating* an expression, which corresponds to assigning it to a processor for execution. Execution of a parallel `let` instruction whose constituent expressions have not yet been activated consists of the activation of these expressions. Execution of a parallel `let` whose constituents are completely evaluated consists

of substituting the values of these expressions into the body of the `let`, which is itself then activated. Execution of all other instructions is exactly as before, with the result being made active in each case.

This can be formalized using *parallelism contexts*, which capture the tree structure of nested parallel computations. Let l and variants range over a countable set of *labels*. These will serve to identify the abstract processors assigned to the execution of an active expression. The set of parallelism contexts $\mathcal{L}$ is defined by the following grammar:

$$\begin{aligned}\mathcal{L} \quad ::= \quad & l:\Box \mid l:\texttt{let}\ x_1:\tau_1\ \texttt{be}\ \mathcal{L}_1\ \texttt{and}\ x_2:\tau_2\ \texttt{be}\ \mathcal{L}_2\ \texttt{in}\ e \\ & l:\texttt{let}\ x_1:\tau_1\ \texttt{be}\ \mathcal{L}_1\ \texttt{and}\ x_2:\tau_2\ \texttt{be}\ v_2\ \texttt{in}\ e\ \texttt{end} \mid \\ & l:\texttt{let}\ x_1:\tau_1\ \texttt{be}\ v_1\ \texttt{and}\ x_2:\tau_2\ \texttt{be}\ \mathcal{L}_2\ \texttt{in}\ e\ \texttt{end}\end{aligned}$$

A parallelism context is *well-formed* only if all labels occurring within it are distinct; hereafter we will consider only well-formed parallelism contexts.

A labelled "hole" in a parallelism context represents an active computation site; a labelled `let` expression represents a pending computation that is awaiting completion of its child threads. We have arranged things so that all active sites are children of pending sites, reflecting the intuition that an active site must have been spawned by some (now pending) site.

The *arity* of a context is defined to be the number of "holes" occurring within it. The arity is therefore the number of active threads within the context. If $\mathcal{L}$ is a context with arity n, then the expression $\mathcal{L}[l = e]_{i=1}^n$ represents the result of "filling" the hole labelled l_i with the expression e_i, for each $1 \leq i \leq n$. Thus the e_i's represent the active expressions within the context; the label l_i represents the "name" of the processor assigned to execute e_i.

Each step of the P-machine consists of executing *all* of the active instructions in the current state. This is captured by the following evaluation rule:

$$\frac{e_1 \longrightarrow e_1' \quad \cdots \quad e_n \longrightarrow e_n'}{\mathcal{L}[l = e]_{i=1}^n \mapsto_{\mathsf{P}} \mathcal{L}[l = e']_{i=1}^n}$$

The relation $e \longrightarrow e'$ defines the atomic instruction steps of the P-machine. These are defined by a set of axioms. The first is the *fork* axiom, which initiates execution of a parallel `let` statement:

$$\frac{}{\begin{array}{c}\texttt{let}\ x_1:\tau_1\ \texttt{be}\ e_1\ \texttt{and}\ x_2:\tau_2\ \texttt{be}\ e_2\ \texttt{in}\ e\ \texttt{end} \\ \longrightarrow \\ \texttt{let}\ x_1:\tau_1\ \texttt{be}\ l_1:e_1\ \texttt{and}\ x_2:\tau_2\ \texttt{be}\ l_2:e_2\ \texttt{in}\ e\ \texttt{end}\end{array}} \tag{18.5}$$

Here l_1 and l_2 are "new" labels that do not otherwise occur in the computation. They serve as the labels of the processors assigned to execute e_1 and e_2, respectively.

The second instruction is the *join* axiom, which completes execution of a parallel `let`:

$$\frac{v_1 \text{ value} \quad v_2 \text{ value}}{\texttt{let}\ x_1 : \tau_1\ \texttt{be}\ l_1 : v_1\ \texttt{and}\ x_2 : \tau_2\ \texttt{be}\ l_2 : v_2\ \texttt{in}\ e\ \texttt{end} \longrightarrow \{v_1, v_2/x_1, x_2\}e} \tag{18.6}$$

The other instructions are inherited from the M-machine. For example, function application is defined by the following instruction:

$$\frac{v_1 \text{ value} \quad v_2 \text{ value} \quad (v_1 = \texttt{fun}\, f\ (x : \tau_1) : \tau_2\ \texttt{is}\ e)}{\texttt{apply}(v_1, v_2) \longrightarrow \{v_1, v_2/f, x\}e} \tag{18.7}$$

This completes the definition of the P-machine.

Exercise 18.2
State and prove preservation and progress relative to the P*-machine.*

18.3 Cost Semantics, Revisited

A primary motivation for introducing the P-machine was to achieve a proper accounting for the cost of creating and synchronizing threads. In the simplified model of Chapter 17 we ignored these costs, but here we seek to take them into account. This is accomplished by taking the following rule for the cost semantics of the parallel `let` construct:

$$\frac{e_1 \Downarrow^{w_1,d_1} v_1 \quad e_2 \Downarrow^{w_2,d_2} v_2 \quad \{v_1, v_2/x_1, x_2\}e \Downarrow^{w,d} v}{\texttt{let}\ x_1 : \tau_1\ \texttt{be}\ e_1\ \texttt{and}\ x_2 : \tau_2\ \texttt{be}\ e_2\ \texttt{in}\ e\ \texttt{end} \Downarrow^{w',d'} v} \tag{18.8}$$

where $w' = w_1 + w_2 + w + 2$ and $d' = \max(d_1, d_2) + d + 2$. Since the remaining expression forms are all limited to values, they have unit cost for both work and depth.

The calculation of work and depth for the parallel `let` construct is justified by relating the cost semantics to the P-machine. The work performed

in an evaluation sequence $e \mapsto_{\mathsf{P}}^{*} v$ is the total number of primitive instruction steps performed in the sequence; it is the sequential cost of executing the expression e.

Theorem 18.3
If $e \Downarrow^{w,d} v$, then $l : e \mapsto_{\mathsf{P}}^{d} l : v$ with work w.

Proof: The proof from left to right proceeds by induction on the cost semantics. For example, consider the cost semantics of the parallel `let` construct. By induction we have

1. $l_1 : e_1 \mapsto_{\mathsf{P}}^{d_1} l_1 : v_1$ with work w_1;
2. $l_2 : e_2 \mapsto_{\mathsf{P}}^{d_2} l_2 : v_2$ with work w_2;
3. $l : \{v_1, v_2 / x_1, x_2\} e \mapsto_{\mathsf{P}}^{d} l : v$ with work w.

We therefore have the following P-machine evaluation sequence:

$$
\begin{array}{ll}
l : \texttt{let}\ x_1 : \tau_1\ \texttt{be}\ e_1\ \texttt{and}\ x_2 : \tau_2\ \texttt{be}\ e_2\ \texttt{in}\ e\ \texttt{end} & \mapsto_{\mathsf{P}} \\
l : \texttt{let}\ x_1 : \tau_1\ \texttt{be}\ l_1 : e_1\ \texttt{and}\ x_2 : \tau_2\ \texttt{be}\ l_2 : e_2\ \texttt{in}\ e\ \texttt{end} & \mapsto_{\mathsf{P}}^{\max(d_1,d_2)} \\
l : \texttt{let}\ x_1 : \tau_1\ \texttt{be}\ l_1 : v_1\ \texttt{and}\ x_2 : \tau_2\ \texttt{be}\ l_2 : v_2\ \texttt{in}\ e\ \texttt{end} & \mapsto_{\mathsf{P}} \\
l : \{v_1, v_2 / x_1, x_2\} e & \mapsto_{\mathsf{P}}^{d} \\
l : v &
\end{array}
$$

The total length of the evaluation sequence is $\max(d_1, d_2) + d + 2$, as required by the depth cost, and the total work is $w_1 + w_2 + w + 2$, as required by the work cost. ■

18.4 Provable Implementations (Summary)

The semantics of parallelism given above is based on an idealized parallel computer with an unlimited number of processors. In practice this idealization must be simulated using some fixed number, p, of physical processors. In practice p is on the order of 10's of processors, but may even rise (at the time of this writing) into the 100's. In any case p does not vary with input size, but is rather a fixed parameter of the implementation platform. The important question is how efficiently can one simulate

unbounded parallelism using only p processors? That is, how realistic are the costs assigned to the language by our semantics? Can we make accurate predictions about the running time of a program on a real parallel computer based on the idealized cost assigned to it by our semantics?

The answer is *yes*, through the notion of a *provably efficient implementation*. While a full treatment of these ideas is beyond the scope of this book, it is worthwhile to summarize the main ideas.

Theorem 18.4 (Blelloch and Greiner)
If $e \Downarrow^{w,d} v$, then e can be evaluated on an SMP with p-processors in time $O(w/p + d \lg p)$.

For our purposes, an SMP is any of a wide range of parallel computers, including a CRCW PRAM, a hypercube, or a butterfly network. Observe that for $p = 1$, the stated bound simplifies to $O(w)$, as would be expected.

To understand the significance of this theorem, observe that the definition of work and depth yields a lower bound of $\Omega(\max(w/p, d))$ on the execution time on p processors. We can never complete execution in fewer than d steps, and can, at best, divide the total work evenly among the p processors. The theorem tells us that we can come within a constant factor of this lower bound. The constant factor, $\lg p$, represents the overhead of scheduling parallel computations on p processors.

The goal of parallel programming is to maximize the use of parallelism so as to minimize the execution time. By the theorem this will occur if the term w/p dominates, which occurs if the ratio w/d of work to depth is at least $p \lg p$. This ratio is sometimes called the *parallelizability* of the program. For highly sequential programs, d is directly proportional to w, yielding a low parallelizability — increasing the number of processors will not speed up the computation. For highly parallel programs, d might be constant or proportional to $\lg w$, resulting in a large parallelizability, and good utilization of the available computing resources. It is important to keep in mind that *it is not known* whether there are inherently sequential problems (for which no parallelizable solution is possible), or whether, instead, all problems can benefit from parallelism. The best that we can say at the time of this writing is that there are problems for which no parallelizable solution is known.

To get a sense of what is involved in the proof of Blelloch and Greiner's theorem, let us consider the assumption that the index operation on vec-

tors (given in Chapter 17) has constant depth. The theorem implies that index is implementable on an SMP in time $O(n/p + \lg p)$. We will briefly sketch a proof for this one case. The main idea is that we may assume that every processor is assigned a unique number from 0 to $p - 1$. To implement index, we simply allocate, but do not initialize, a region of memory of the appropriate size, and ask each processor to simultaneously store its identifying number i into the ith element of the allocated array. This works directly if the size of the vector is no more than the number of processors. Otherwise, we may divide the problem in half, and recursively build two index vectors of half the size, one starting with zero, the other with $n/2$. This process need proceed at most $\lg p$ times before the vectors are small enough, leaving n/p sub-problems of size at most p to be solved. Thus the total time required is $O(n/p + \lg p)$, as required by the theorem.

The other primitive operations are handled by similar arguments, justifying the cost assignments made to them in the operational semantics. To complete the proof of Blelloch and Greiner's theorem, we need only argue that the total work w can indeed be allocated to p processors with a cost of only $\lg p$ for the overhead. This is a consequence of Brent's Theorem, which states that a total workload w divided into d parallel steps may be implemented on p processors in $O(n/p + d \lg p)$ time. The argument relies on certain assumptions about the SMP, including the ability to perform a parallel fetch-and-add operation in constant time.

Part VII

Data Structures and Abstraction

Chapter 19

Aggregate Data Structures

It is interesting to add to MinML support for programming with aggregate data structures such as n-tuples, lists, and tree structures. We will decompose these familiar data structures into three types:

1. Product (or tuple) types. In general these are types whose values are n-tuples of values, with each component of a specified type. We will study two special cases that are sufficient to cover the general case: 0-tuples (also known as the unit type) and 2-tuples (also known as ordered pairs).

2. Sum (or variant or union) types. These are types whose values are values of one of n specified types, with an explicit "tag" indicating which of the n choices is made.

3. Recursive types. These are "self-referential" types whose values may have as constituents values of the recursive type itself. Familiar examples include lists and trees. A non-empty list consists of a value at the head of the list together with another value of list type.

19.1 Products

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

$$\begin{array}{lrcl} \textit{Types} & \tau & ::= & \mathtt{unit} \mid \tau_1 * \tau_2 \\ \textit{Expressions} & e & ::= & \mathtt{()} \mid \mathtt{check}\, e_1\, \mathtt{is}\, \mathtt{()}\, \mathtt{in}\, e_2 \mid \mathtt{(}e_1\mathtt{,}e_2\mathtt{)} \mid \\ & & & \mathtt{split}\, e_1\, \mathtt{as}\, \mathtt{(}x\mathtt{,}y\mathtt{)}\, \mathtt{in}\, e_2 \\ \textit{Values} & v & ::= & \mathtt{()} \mid \mathtt{(}v_1\mathtt{,}v_2\mathtt{)} \end{array}$$

The higher-order abstract syntax is given by stipulating that in the expression $\mathtt{split}\, e_1\, \mathtt{as}\, \mathtt{(}x\mathtt{,}y\mathtt{)}\, \mathtt{in}\, e_2$ 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:

$$\frac{}{\Gamma \vdash \mathtt{()} : \mathtt{unit}} \tag{19.1}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{unit} \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{check}\, e_1\, \mathtt{is}\, \mathtt{()}\, \mathtt{in}\, e_2 : \tau_2} \tag{19.2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{(}e_1\mathtt{,}e_2\mathtt{)} : \tau_1 * \tau_2} \tag{19.3}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 * \tau_2 \quad \Gamma, x{:}\tau_1, y{:}\tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathtt{split}\, e_1\, \mathtt{as}\, \mathtt{(}x\mathtt{,}y\mathtt{)}\, \mathtt{in}\, e_2 : \tau} \tag{19.4}$$

The dynamic semantics is given by these rules:

$$\frac{}{\mathtt{check}\, \mathtt{()}\, \mathtt{is}\, \mathtt{()}\, \mathtt{in}\, e \mapsto e} \tag{19.5}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{check}\, e_1\, \mathtt{is}\, \mathtt{()}\, \mathtt{in}\, e_2 \mapsto \mathtt{check}\, e_1'\, \mathtt{is}\, \mathtt{()}\, \mathtt{in}\, e_2} \tag{19.6}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{(}e_1\mathtt{,}e_2\mathtt{)} \mapsto \mathtt{(}e_1'\mathtt{,}e_2\mathtt{)}} \tag{19.7}$$

$$\frac{e_2 \mapsto e_2'}{(v_1, e_2) \mapsto (v_1, e_2')} \tag{19.8}$$

$$\overline{\texttt{split}\,(v_1, v_2)\,\texttt{as}\,(x, y)\,\texttt{in}\,e \mapsto \{v_1, v_2/x, y\}e} \tag{19.9}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{split}\,e_1\,\texttt{as}\,(x, y)\,\texttt{in}\,e_2 \mapsto \texttt{split}\,e_1'\,\texttt{as}\,(x, y)\,\texttt{in}\,e_2} \tag{19.10}$$

$$\frac{e \mapsto e'}{\texttt{case}_\tau\,e\,\texttt{of}\,\texttt{inl}(x_1 : \tau_1) \Rightarrow e_1 \mid \texttt{inr}(x_2 : \tau_2) \Rightarrow e_2 \mapsto \texttt{case}_\tau\,e'\,\texttt{of}\,\texttt{inl}(x_1 : \tau_1) \Rightarrow e_1 \mid \texttt{inr}(x_2 : \tau_2) \Rightarrow e} \tag{19.11}$$

Exercise 19.1
State and prove the soundness of this extension to MinML.

Exercise 19.2
A variation is to treat any pair (e_1, e_2) as a value, regardless of whether or not e_1 or e_2 are values. Give a precise formulation of this variant, and prove it sound.

Exercise 19.3
It is also possible to formulate a direct treatment of n-ary product types (for $n \geq 0$), rather than to derive them from binary and nullary products. Give a direct formalization of n-ary products. Be careful to get the cases $n = 0$ and $n = 1$ right!

Exercise 19.4
Another variation is to considered labelled products *in which the components are accessed directly by referring to their labels (in a manner similar to C* `struct`*'s). Formalize this notion.*

19.2 Sums

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

$$\begin{array}{lrcl} \textit{Types} & \tau & ::= & \tau_1 \texttt{+} \tau_2 \\ \textit{Expressions} & e & ::= & \texttt{inl}_{\tau_1+\tau_2}(e_1) \mid \texttt{inr}_{\tau_1+\tau_2}(e_2) \mid \\ & & & \texttt{case}_\tau\, e_0\, \texttt{of}\, \texttt{inl}(x:\tau_1) \texttt{=>} e_1 \mid \texttt{inr}(y:\tau_2) \texttt{=>} e_2 \\ \textit{Values} & v & ::= & \texttt{inl}_{\tau_1+\tau_2}(v_1) \mid \texttt{inr}_{\tau_1+\tau_2}(v_2) \end{array}$$

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

$$\frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \texttt{inl}_{\tau_1+\tau_2}(e_1) : \tau_1+\tau_2} \tag{19.12}$$

$$\frac{\Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{inr}_{\tau_1+\tau_2}(e_2) : \tau_1+\tau_2} \tag{19.13}$$

$$\frac{\Gamma \vdash e_0 : \tau_1+\tau_2 \quad \Gamma, x_1{:}\tau_1 \vdash e_1 : \tau \quad \Gamma, x_2{:}\tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{case}_\tau\, e_0\, \texttt{of}\, \texttt{inl}(x_1:\tau_1) \texttt{=>} e_1 \mid \texttt{inr}(x_2:\tau_2) \texttt{=>} e_2 : \tau} \tag{19.14}$$

The evaluation rules are as follows:

$$\frac{e \mapsto e'}{\texttt{inl}_{\tau_1+\tau_2}(e) \mapsto \texttt{inl}_{\tau_1+\tau_2}(e')} \tag{19.15}$$

$$\frac{e \mapsto e'}{\texttt{inr}_{\tau_1+\tau_2}(e) \mapsto \texttt{inr}_{\tau_1+\tau_2}(e')} \tag{19.16}$$

$$\frac{}{\texttt{case}_\tau\, \texttt{inl}_{\tau_1+\tau_2}(v)\, \texttt{of}\, \texttt{inl}(x_1:\tau_1) \texttt{=>} e_1 \mid \texttt{inr}(x_2:\tau_2) \texttt{=>} e_2 \mapsto \{v/x_1\}e_1} \tag{19.17}$$

$$\frac{}{\texttt{case}_\tau\, \texttt{inr}_{\tau_1+\tau_2}(v)\, \texttt{of}\, \texttt{inl}(x_1:\tau_1) \texttt{=>} e_1 \mid \texttt{inr}(x_2:\tau_2) \texttt{=>} e_2 \mapsto \{v/x_2\}e_2} \tag{19.18}$$

Exercise 19.5
State and prove the soundness of this extension.

Exercise 19.6
Consider these variants: $\texttt{inl}_{\tau_1+\tau_2}(e)$ *and* $\texttt{inr}_{\tau_1+\tau_2}(e)$ *are values, regardless of whether or not* e *is a value; n-ary sums; labelled sums.*

19.3 Recursive Types

Recursive types are somewhat less familiar than products and sums. Few well-known languages provide direct support for these. Instead the programmer is expected to simulate them using pointers and similar low-level representations. Here instead we'll present them as a fundamental concept.

As mentioned in the introduction, the main idea of a recursive type is similar to that of a recursive function — self-reference. The idea is easily illustrated by example. Informally, a list of integers may be thought of as either the empty list, `nil`, or a non-empty list, `cons`(h,t), where h is an integer and t is another list of integers. The operations `nil` and `cons`$(-,-)$ are *value constructors* for the type `ilist` of integer lists. We may program with lists using a form of case analysis, written

$$\texttt{listcase}\ e\ \texttt{of nil =>}\ e_1\ \texttt{|\ cons}(x,y)\ \texttt{=>}\ e_2,$$

where x and y are bound in e_2. This construct analyses whether e is the empty list, in which case it evaluates e_1, or a non-empty list, with head x and tail y, in which case it evaluates e_2 with the head and tail bound to these variables.

Exercise 19.7
Give a formal definition of the type `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 19.8
Consider the type of binary trees with integers at the nodes. To what sum type would such a type be isomorphic?

This motivates the following general definition of recursive types. The first-order abstract syntax is given by the following grammar:

$$\begin{array}{lrcl} \textit{Types} & \tau & ::= & t \mid \texttt{rec}\, t\, \texttt{is}\, \tau \\ \textit{Expressions} & e & ::= & \texttt{roll}(e) \mid \texttt{unroll}(e) \\ \textit{Values} & v & ::= & \texttt{roll}(v) \end{array}$$

Here t ranges over a set of *type variables*, which are used to stand for the recursive type itself, in much the same way that we give a name to recursive functions to stand for the function itself. For the present we will insist that type variables are used only for this purpose; they may occur only inside of a recursive type, where they are bound by the recursive type constructor itself.

For example, the type $\tau = \texttt{rec}\, t\, \texttt{is}\, \texttt{unit+}(\texttt{int*}t)$ is the recursive type of lists of integers. It is isomorphic to its *unrolling*, the type

$$\texttt{unit+}(\texttt{int*}\tau).$$

This is the isomorphism described informally above.

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

$$\frac{\Gamma \vdash e : \{\texttt{rec}\, t\, \texttt{is}\, \tau/t\}\tau}{\Gamma \vdash \texttt{roll}(e) : \texttt{rec}\, t\, \texttt{is}\, \tau} \tag{19.19}$$

$$\frac{\Gamma \vdash e : \texttt{rec}\, t\, \texttt{is}\, \tau}{\Gamma \vdash \texttt{unroll}(e) : \{\texttt{rec}\, t\, \texttt{is}\, \tau/t\}\tau} \tag{19.20}$$

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

The dynamic semantics is given by the following rules:

$$\frac{}{\texttt{unroll}(\texttt{roll}(v)) \mapsto v} \tag{19.21}$$

$$\frac{e \mapsto e'}{\texttt{unroll}(e) \mapsto \texttt{unroll}(e')} \tag{19.22}$$

$$\frac{e \mapsto e'}{\texttt{roll}(e) \mapsto \texttt{roll}(e')} \tag{19.23}$$

Exercise 19.9
State and prove the soundness of this extension of MinML.

Exercise 19.10
Consider the definition of the type `ilist` *as a recursive type given above. Give definitions of* `nil`*,* `cons`*, and* `listcase` *in terms of the operations on recursive types, sums, and products.*

Chapter 20

Polymorphism

MinML is an *explicitly typed* language. The abstract syntax is defined to have sufficient type information to ensure that all expressions have a unique type. In particular the types of the parameters of a function must be chosen when the function is defined.

While this is not itself a serious problem, it does expose a significant weakness in the MinML type system. For example, there is no way to define a generic procedure for composing two functions whose domain and range match up appropriately. Instead we must define a separate composition operation for each choice of types for the functions being composed. Here is one composition function

```
fun _ (f:string->int):(char->string)->(string->int) is
  fun _ (g:char->string):string->int is
    fun _ (x:string):int is apply(f, apply(g, x)),
```

and here is another

```
fun _ (f:float->double):(int->float)->(int->double) is
  fun _ (g:int->float):int->double is
    fun _ (x:int):double is apply(f, 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.

20.1 Polymorphic MinML

Polymorphic MinML, or PolyMinML, is an extension of MinML with the ability to define *polymorphic functions*. Informally, a polymorphic function is a function that takes a *type* as argument and yields a *value* as result. The type parameter to a polymorphic function represents an *unknown*, or *generic*, type, which can be instantiated by applying the function to a specific type. The types of polymorphic functions are called *polymorphic types*, or *polytypes*.

A significant design decision is whether to regard polymorphic types as "first-class" types, or whether they are, instead, "second-class" citizens. Polymorphic functions in ML are second-class — they cannot be passed as arguments, returned as results, or stored in data structures. The only thing we may do with polymorphic values is to bind them to identifiers with a `val` or `fun` binding. Uses of such identifiers are automatically instantiated by an implicit polymorphic instantiation. The alternative is to treat polymorphic functions as first-class values, which can be used like any other value in the language. Here there are no restrictions on how they can be used, but you should be warned that doing so precludes using type inference to perform polymorphic abstraction and instantiation automatically.

We'll set things up for second-class polymorphism by explicitly distinguishing polymorphic types from monomorphic types. The first-class case can then be recovered by simply conflating polytypes and monotypes.

Abstract Syntax

The abstract syntax of PolyMinML is defined by the following extension to the MinML grammar:

$$\begin{array}{llll} \textit{Polytypes} & \sigma & ::= & \tau \mid \forall t(\sigma) \\ \textit{Monotypes} & \tau & ::= & \ldots \mid t \\ \textit{Expressions} & e & ::= & \ldots \mid \texttt{Fun}\, t\, \texttt{in}\, e \mid \texttt{inst(}e\texttt{,}\tau\texttt{)} \\ \textit{Values} & v & ::= & \ldots \mid \texttt{Fun}\, t\, \texttt{in}\, e \end{array}$$

The variable t ranges over a set of *type variables*, which are written ML-style `'a`, `'b`, and so on in examples. In the polytype $\forall t(\sigma)$ the type variable t is bound in σ; 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 $\texttt{Fun}\, t\, \texttt{in}\, e$ is a *polymorphic function* with parameter t and body e. The variable t is bound within e. An expression of the form $\texttt{inst(}e\texttt{,}\tau\texttt{)}$ is a *polymorphic instantiation* of the polymorphic function e at *monotype* τ. Notice that we may *only* instantiate polymorphic functions with monotypes. In examples we write $f\,\texttt{[}\tau\texttt{]}$ for polymorphic instantiation, rather than the more verbose $\texttt{inst(}f\texttt{,}\tau\texttt{)}$.

We write $\mathrm{FTV}(\tau)$ (respectively, $\mathrm{FTV}(\sigma)$, $\mathrm{FTV}(e)$) for the set of free type variables occurring in τ (respectively, σ, e). Capture-avoiding substitution of a monotype τ for free occurrences of a type variable t in a polytype σ (resp., monotype τ', 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 Γ is a context assigning types to ordinary variables. Only those variables in $\mathrm{dom}\,\Gamma$ may legally occur in e. For PolyMinML we must introduce an additional context, Δ, which is a set of type variables, those that may legally occur in the types and expression of the judgement.

The static semantics consists of rules for deriving the following two

judgements:

$\Delta \vdash \sigma\ \mathsf{ok}$ *σ is a well-formed type in Δ*
$\Gamma \vdash_\Delta e : \sigma$ *e is a well-formed expression of type σ in Γ and Δ*

The rules for validity of types are as follows:

$$\frac{t \in \Delta}{\Delta \vdash t\ \mathsf{ok}} \tag{20.1}$$

$$\frac{}{\Delta \vdash \mathtt{int}\ \mathsf{ok}} \tag{20.2}$$

$$\frac{}{\Delta \vdash \mathtt{bool}\ \mathsf{ok}} \tag{20.3}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{ok} \quad \Delta \vdash \tau_2\ \mathsf{ok}}{\Delta \vdash \tau_1 \rightarrow \tau_2\ \mathsf{ok}} \tag{20.4}$$

$$\frac{\Delta \cup \{\, t \,\} \vdash \sigma\ \mathsf{ok} \quad t \notin \Delta}{\Delta \vdash \forall t(\sigma)\ \mathsf{ok}} \tag{20.5}$$

The auxiliary judgement $\Delta \vdash \Gamma$ is defined by the following rule:

$$\frac{\Delta \vdash \Gamma(x)\ \mathsf{ok}\ (\forall x \in \mathrm{dom}(\Gamma))}{\Delta \vdash \Gamma\ \mathsf{ok}}. \tag{20.6}$$

The rules for deriving typing judgements $\Gamma \vdash_\Delta e : \sigma$ are as follows. We assume that $\Delta \vdash \Gamma\ \mathsf{ok}$, $\Delta \vdash \sigma\ \mathsf{ok}$, $\mathrm{FV}(e) \subseteq \mathrm{dom}(\Gamma)$, and $\mathrm{FTV}(e) \subseteq \Delta$. We give only the rules specific to PolyMinML; the remaining rules are those of MinML, augmented with a set Δ of type variables.

$$\frac{\Gamma \vdash_{\Delta \cup \{\, t \,\}} e : \sigma \quad t \notin \Delta}{\Gamma \vdash_\Delta \mathtt{Fun}\, t\, \mathtt{in}\, e : \forall t(\sigma)} \tag{20.7}$$

$$\frac{\Gamma \vdash_\Delta e : \forall t(\sigma) \quad \Delta \vdash \tau\ \mathsf{ok}}{\Gamma \vdash_\Delta \mathtt{inst}(e, \tau) : \{\tau/t\}\sigma} \tag{20.8}$$

For example, here is the polymorphic composition function in PolyMinML:

```
Fun t in
  Fun u in
    Fun v in
      fun _(f:u->v):(t->u)->(t->v) is
        fun _(g:t->u):t->v is
          fun _(x:t):v is apply(f, apply(g, x))
```

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 20.1 (Instantiation)
If $\Gamma \vdash_{\Delta \cup \{t\}} e : \sigma$*, where* $t \notin \Delta$*, and* $\Delta \vdash \tau$ ok*, then* $\{\tau/t\}\Gamma \vdash_{\Delta} \{\tau/t\}e : \{\tau/t\}\sigma$*.*

The proof is by induction on typing, and involves no new ideas beyond what we have already seen.

We will also have need of the following canonical forms lemma:

Lemma 20.2 (Canonical Forms)
If $v : \forall t(\sigma)$*, then* $v =$ `Fun` t `in` e *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:

$$\frac{}{\texttt{inst(Fun}\, t\, \texttt{in}\, e, \tau\texttt{)} \mapsto \{\tau/t\}e} \tag{20.9}$$

$$\frac{e \mapsto e'}{\texttt{inst}(e, \tau) \mapsto \texttt{inst}(e', \tau)} \tag{20.10}$$

It is then a simple matter to prove safety for this language.

Theorem 20.3 (Preservation)
If $e : \sigma$ and $e \mapsto e'$, then $e' : \sigma$.

The proof is by induction on evaluation.

Theorem 20.4 (Progress)
If $e : \sigma$, then either e is a value or there exists e' such that $e \mapsto e'$.

As before, this is proved by induction on typing.

First-Class Polymorphism

The syntax given above describes an ML-like treatment of polymorphism, *albeit* one in which polymorphic abstraction and instantiation is explicit, rather than implicit, as it is in ML. To obtain the first-class variant of PolyMinML, we simply ignore the distinction between poly- and mono-types, regarding them all as simply types. Everything else remains unchanged, including the proofs of progress and preservation.

With first-class polymorphism we may consider types such as

$$\forall t(t \rightarrow t) \rightarrow \forall t(t \rightarrow t),$$

which cannot be expressed in the ML-like fragment. This is the type of functions that accept a polymorphic function as argument and yield a polymorphic function (of the same type) as result. If f has the above type, then $f\,(\texttt{Fun}\ t\ \texttt{in}\ \texttt{fun}\ \_\ (x\!:\!t)\!:\!t\ \texttt{is}\ x)$ is well-formed. However, the application $f\,(\texttt{fun}\ \_\ (x\!:\!\texttt{int})\!:\!\texttt{int}\ \texttt{is}\ +(x,1))$ 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 `inst(`g`,int)(succ)` is well-formed, since we first instantiate g at `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 $\forall t(\sigma)\ \texttt{list}$ and $\forall t(\sigma\ \texttt{list})$ for various choices

of σ. 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:

```
let
    val r : ('a -> 'a) ref = ref (fn x:'a => x)
 in
    r := (fn x:int => x+1) ; (!r)(true)
end
```

A simple check of the polymorphic typing rules reveals that this is a well-formed expression, provided that the value restriction is suspended. Of course, it "gets stuck" during evaluation by attempting to add `1` to `true`.

Using the framework of explicit polymorphism, I will argue that the superficial plausibility of this example (which led to the unsoundness in the language) stems from a failure to distinguish between these two types:

1. The type $\forall t(t \rightarrow t\,\texttt{ref})$ of polymorphic functions yielding reference cells containing a function from a type to itself.

2. The type $\forall t(t \rightarrow t)\,\texttt{ref}$ 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 `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 `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:

```
let
    val r : All 'a (('a -> 'a) ref) =
      Fun 'a in ref (fn x:'a => x) end
 in
    r[int] := (fn x:int => x+1) ; (!(r[bool]))(true)
end
```

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 polymporphic functions are values, and the binding of `r` is just such a value. Each of the two instances of `r` executes the body of this function separately, each time allocating a new reference cell. Hence the unsoundness goes away!

Here's another rendering that is, in fact, ill-typed (and should be, since it "gets stuck"!).

```
let
    val r : (All 'a ('a -> 'a)) ref =
      ref (Fun 'a in fn x:'a => x end)
 in
    r := (fn x:int => x+1) ; (!r)[bool](true)
end
```

The assignment to `r` is ill-typed because the successor is not sufficiently polymorphic. The retrieval and subsequent instantiation and application is type-correct, however. If we change the program to

```
let
    val r : (All 'a ('a -> 'a)) ref =
      ref (Fun 'a in fn x:'a => x end)
 in
    r := (Fun 'a in fn x:'a => x end) ; (!r)[bool](true)
end
```

then the expression is well-typed, and behaves sanely, precisely because we have assigned to `r` a sufficiently polymorphic function.

20.2 ML-style Type Inference

ML-style type inference may be viewed as a translation from the implicitly typed syntax of ML to the explicitly-typed syntax of PolyMinML. Specifically, the type inference mechanism performs the following tasks:

- Attaching type labels to function arguments and results.

- Inserting polymorphic abstractions for declarations of polymorphic type.
- Inserting polymorphic instantiations whenever a polymorphic declared variable is used.

Thus in ML we may write

```
val I : 'a -> 'a = fn x => x
val n : int = I(I)(3)
```

This stands for the PolyMinML declarations[1]

```
val I : ∀t(t→t) = Fun t in fun _(x:t):t is x
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 $\forall t((t\rightarrow t)\,\texttt{ref})$. 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.[2] However, if we limit ourselves to values

[1] We've not equipped PolyMinML with a declaration construct, but you can see from the example how this might be done.

[2] One could argue that the ML semantics is incorrect, which leads to a different language.

in the first place, then the polymorphic abstraction is only ever wrapped around a value, and no change of semantics occurs. Therefore[3], the insertion of polymorphic abstraction doesn't change the semantics, and everything is safe. The example above involving reference cells is ruled out, because the expression `ref (fn x => x)` is not a value, but such is the nature of the value restriction.

20.3 Parametricity

Our original motivation for introducing polymorphism was to enable more programs to be written — those that are "generic" in one or more types, such as the composition function give above. The idea is that if the behavior 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 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

[3] This would need to be proved, of course.

languages with mutable references!

20.3.1 Informal Discussion

We will begin with an informal discussion of parametricity based on a "seat of the pants" understanding of the set of well-formed programs of a type.

Suppose that a function value f has the type $\forall t(t \to t)$. What function could it be?

1. It could diverge when instantiated — f `[`τ`]` goes into an infinite loop. Since f is polymorphic, its behavior cannot depend on the choice of τ, so in fact f `[`τ'`]` diverges for all τ' if it diverges for τ.

2. It could converge when instantiated at τ to a function g of type $\tau \to \tau$ that loops when applied to an argument v of type τ — *i.e.*, g`(`v`)` runs forever. Since f is polymorphic, g must diverge on *every* argument v of type τ if it diverges on *some* argument of type τ.

3. It could converge when instantiated at τ to a function g of type $\tau \to \tau$ that, when applied to a value v of type τ returns a value v' of type τ. 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 $\forall t(t \to t)$ is the polymorphic identity function.

Suppose that f is an interesting function of type $\forall t(t)$. What function could it be? A moment's thought reveals that it cannot be interesting! That is, every function f of this type must diverge when instantiated, and hence is uninteresting. In other words, there are no interesting values of this type — it is essentially an "empty" type.

For a final example, suppose that f is an interesting function of type $\forall t(t$ `list` $\to t$ `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 τ, and values $z : \tau$ and $s : \tau{\to}\tau$, the expression

$$f\ [\tau]\,(z)\,(s)$$

must yield a value of type τ. Moreover, it must behave uniformly with respect to the choice of τ. What values could it yield? The only way to build a value of type τ 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 $\overline{n}$ for the polymorphic function of type N representing the natural number n, namely the function

```
Fun t in
    fn z:t in
       fn s:t->t in
          s(s(... s)...))
       end
    end
end
```

where there are n occurrences of s in the expression. Observe that if we instantiate $\overline{n}$ 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 $\overline{m}$ and $\overline{n}$ of type N, we wish to compute their sum $\overline{m+n}$, also of type N. That is, the addition function must look as follows:

```
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
```

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:

```
fn m:N in
   fn n:N in
      Fun t in
          fn z:t in
             fn s:t->t in
                n[t](m[t](z)(s))(s)
             end
          end
      end
   end
end
```

To see that it works, instantiate the result at τ, apply it to z and s, and observe the result.

20.3.2 Relational Parametricity

In this section we give a more precise formulation of parametricity. The main idea is that polymorphism implies that certain equations between expressions must hold. For example, if $f : \forall t(t \to t)$, then f must be *equal to* the identity function, and if $f : N$, then f must be *equal to* some Church numeral $\overline{n}$. 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 τ 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 $\tau_1 \to \tau_2$ are

equal iff whenever they are applied to equal arguments of type τ_1, they yield equal results of type τ_2.

More formally, we make the following definitions. First, we define equality of closed expressions of type τ as follows:

$$e \cong_{exp} e' : \tau \quad \textit{iff} \quad e \mapsto^* v \Leftrightarrow e' \mapsto^* v' \quad \textit{and} v \cong_{val} v' : \tau.$$

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:

$$\begin{array}{rcl} v \cong_{val} v' : \texttt{bool} & \textit{iff} & v = v' = \texttt{true} \textit{ or } v = v' = \texttt{false} \\ v \cong_{val} v' : \texttt{int} & \textit{iff} & v = v' = n \textit{ for some } n \geq 0 \\ v \cong_{val} v' : \tau_1 \rightarrow \tau_2 & \textit{iff} & v_1 \cong_{val} v_1' : \tau_1 \textit{ implies } v(v_1) \cong_{exp} v'(v_1') : \tau_2 \end{array}$$

The following lemma states two important properties of this notion of equality.

Lemma 20.5

1. *Expression and value equivalence are reflexive, symmetric, and transitive.*

2. *Expression equivalence is a* congruence*: we may replace any sub-expression of an expression e by an equivalent sub-expression to obtain an equivalent expression.*

So far we've considered only equality of closed expressions of monomorphic type. The definition is made so that it readily generalizes to the polymorphic case. The idea is that when we quantify over a type, we are not able to say *a priori* what we mean by equality at that type, precisely because it is "unknown". Therefore we *also* quantify over all possible notions of equality to cover all possible interpretations of that type. Let us write $R : \tau \leftrightarrow \tau'$ to indicate that R is a binary relation between values of type τ and τ'.

Here is the definition of equality of polymorphic values:

$$v \cong_{val} v' : \forall t(\sigma) \quad \textit{iff} \quad \textit{for all } \tau \textit{ and } \tau', \textit{ and all } R : \tau \leftrightarrow \tau',\ v\ [\tau] \cong_{exp} v'\ [\tau'] : \sigma$$

where we take equality at the type variable t to be the relation R (*i.e.*, $v \cong_{val} v' : t$ iff $v\, R\, v'$).

There is one important *proviso*: when quantifying over relations, we must restrict attention to what are called *admissible* relations, a sub-class of relations that, in a suitable sense, respects computation. Most natural choices of relation are admissible, but it is possible to contrive examples that are not. The rough-and-ready rule is this: a relation is admissible iff it is closed under "partial computation". Evaluation of an expression e to a value proceeds through a series of intermediate expressions $e \mapsto e_1 \mapsto e_2 \mapsto \cdots e_n$. The expressions e_i may be thought of as "partial computations" of e, stopping points along the way to the value of e. If a relation relates corresponding partial computations of e and e', then, to be admissible, it must also relate e and e' — it cannot relate all partial computations, and then refuse to relate the complete expressions. We will not develop this idea any further, since to do so would require the formalization of partial computation. I hope that this informal discussion suffices to give the idea.

The following is Reynolds' Parametricity Theorem:

Theorem 20.6 (Parametricity)
If $e : \sigma$ is a closed expression, then $e \cong_{exp} e : \sigma$.

This may seem obvious, until you consider that the notion of equality between expressions of polymorphic type is very strong, requiring equivalence under *all possible* relational interpretations of the quantified type.

Using the Parametricity Theorem we may prove a result we stated informally above.

Theorem 20.7
If $f : \forall t(t \to t)$ is an interesting value, then $f \cong_{val} id : \forall t(t \to t)$, where id is the polymorphic identity function.

Proof: Suppose that τ and τ' are monotypes, and that $R : \tau \leftrightarrow \tau'$. We wish to show that

$$f\ [\tau] \cong_{exp} id\ [\tau'] : t \to t,$$

where equality at type t is taken to be the relation R.

Since f (and id) are interesting, there exists values f_τ and $id_{\tau'}$ such that

$$f\ [\tau] \mapsto^* f_\tau$$

and

$$id\ [\tau'] \mapsto^* id_{\tau'}.$$

We wish to show that

$$f_\tau \cong_{val} id_{\tau'} : t \to t.$$

Suppose that $v_1 \cong_{val} v_1' : t$, which is to say $v_1 \, R \, v_1'$ since equality at type t is taken to be the relation R. We are to show that

$$f_\tau(v_1) \cong_{exp} id_{\tau'}(v_1') : t$$

By the assumption that f is interesting (and the fact that id is interesting), there exists values v_2 and v_2' such that

$$f_\tau(v_1) \mapsto^* v_2$$

and

$$id_{\tau'}(v_1') \mapsto^* v_2'.$$

By the definition of id, it follows that $v_2' = v_1'$ (it's the identity function!). We must show that $v_2 \, R \, v_1'$ to complete the proof.

Now define the relation $R' : \tau \leftrightarrow \tau$ to be the set $\{ (v, v) \mid v \, R \, v_1' \}$. Since $f : \forall t(t \to t)$, we have by the Parametricity Theorem that $f \cong_{val} f : \forall t(t \to t)$, where equality at type t is taken to be the relation R'. Since $v_1 \, R \, v_1'$, we have by definition $v_1 \, R' \, v_1$. Using the definition of equality of polymorphic type, it follows that

$$f_\tau(v_1) \cong_{exp} id_{\tau'}(v_1) : t.$$

Hence $v_2 \, R \, v_1'$, as required. ■

You might reasonably wonder, at this point, what the relationship $f \cong_{val} id : \forall t(t \to t)$ 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 $f : \forall t(t \to t)$, we can show that for every τ and every $v : \tau$, $f\,[\tau]\,(v)$ evaluates to v. By the preceding theorem $f \cong_{val} id : \forall t(t \to t)$. Suppose that τ 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 21

Data Abstraction

Data abstraction is perhaps the most fundamental technique for structuring programs to ensure their robustness over time and to facilitate team development. The fundamental idea of data abstraction is the separation of the *client* from the *implementor* of the abstraction by an *interface*. The interface is a form of "contract" between the client and implementor. It specifies the operations that may be performed on values of the abstract type by the client and, at the same time, imposes the obligation on the implementor to provide these operations with the specified functionality. By limiting the client's view of the abstract type to a specified set of operations, the interface protects the client from depending on the details of the implementation of the abstraction, most especially its representation in terms of well-known constructs of the programming language. Doing so ensures that the implementor is free to change the representation (and, correspondingly, the implementation of the operations) of the abstract type without affecting the behavior of a client of the abstraction.

Our intention is to develop a rigorous account of data abstraction in an extension of PolyMinML with *existential types*. Existential types provide the fundamental linguistic mechanisms for defining interfaces, implementing them, and using the implementation in client code. Using this extension of PolyMinML we will then develop a formal treatment of representation independence based on Reynolds's Parametricity Theorem for PolyMinML. The representation independence theorem will then serve as the basis for proving the correctness of abstract type implementations using bisimulation relations.

21.1 Existential Types

21.1.1 Abstract Syntax

The syntax of PolyMinML is extended with the following constructs:

Polytypes	σ	$::=$	$\ldots$
		$\mid$	$\exists t(\sigma)$
Expressions	e	$::=$	$\ldots$
		$\mid$	$\texttt{pack}\,\tau\,\texttt{with}\,e\,\texttt{as}\,\sigma$
		$\mid$	$\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x\!:\!\sigma\,\texttt{in}\,e_2$
Values	v	$::=$	$\ldots$
		$\mid$	$\texttt{pack}\,\tau\,\texttt{with}\,v\,\texttt{as}\,\sigma$

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 $\texttt{pack}\,\tau\,\texttt{with}\,v\,\texttt{as}\,\exists t(\sigma)$ consisting of a monotype τ together with a value v of type $\{\tau/t\}\sigma$. The monotype τ 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 $\texttt{open}\,e_i\,\texttt{as}\,t\,\texttt{with}\,x\!:\!\sigma\,\texttt{in}\,e_c$, 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 σ (standing for the unknown operations).

In an existential type $\exists t(\sigma)$ the type variable t is bound in σ, and may be renamed at will to satisfy uniqueness requirements. In an expression of the form $\texttt{open}\,e_i\,\texttt{as}\,t\,\texttt{with}\,x\!:\!\sigma\,\texttt{in}\,e_c$ 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 `open` "generates" a "new" type for use within the body of the client. In reality this informal notion of generativity comes down to renaming of bound variables to ensure their uniqueness in a context.

21.1.2 Correspondence With ML

To fix ideas, it is worthwhile to draw analogies between the present formalism and (some aspects of) the Standard ML module system. We have the following correspondences:

PolyMinML + Existentials	*Standard ML*
Existential type	Signature
Package	Structure, with opaque ascription
Opening a package	`open` declaration

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:

```
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

$$\tau_q := q*((\texttt{int}*q)\rightarrow q)*(q\rightarrow(\texttt{int}*q))$$

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:

```
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 $e_q := \texttt{pack}\ \texttt{int}\ \texttt{list}\ \texttt{with}\ v_q\ \texttt{as}\ \sigma_q$, where

$$v_q := \texttt{(nil,(}v_i\texttt{,}v_r\texttt{))}$$

where v_i and v_r are the obvious function abstractions corresponding to the ML code given above.

Finally, a client of an abstraction in ML might typically open it within a scope:

```
local
  open QL
in
  ...
end
```

This corresponds to writing

```
open QL as q with <n,i,r> : τq in ... end
```

in the existential type formalism, using pattern matching syntax for tuples.

21.1.3 Static Semantics

The static semantics is an extension of that of PolyMinML with rules governing the new constructs. The rule of formation for existential types is as follows:

$$\frac{\Delta \cup \{ t \} \vdash \sigma\ \mathsf{ok} \quad t \notin \Delta}{\Delta \vdash \exists t(\sigma)\ \mathsf{ok}} \tag{21.1}$$

The requirement $t \notin \Delta$ may always be met by renaming the bound variable.

The typing rule for packages is as follows:

$$\frac{\Delta \vdash \tau \ \mathsf{ok} \quad \Delta \vdash \exists t(\sigma) \ \mathsf{ok} \quad \Gamma \vdash_\Delta e : \{\tau/t\}\sigma}{\Gamma \vdash_\Delta \mathtt{pack}\,\tau\,\mathtt{with}\,e\,\mathtt{as}\,\exists t(\sigma)} \tag{21.2}$$

The implementation, e, of the operations "knows" the representation type, τ, of the ADT.

The typing rule for opening a package is as follows:

$$\frac{\Delta \vdash \tau_c \ \mathsf{ok} \quad \Gamma, x{:}\sigma \vdash_{\Delta\cup\{t\}} e_c : \tau_c \quad \Gamma \vdash_\Delta e_i : \exists t(\sigma) \quad t \notin \Delta}{\Gamma \vdash_\Delta \mathtt{open}\,e_i\,\mathtt{as}\,t\,\mathtt{with}\,x{:}\sigma\,\mathtt{in}\,e_c : \tau_c} \tag{21.3}$$

This is a complex rule, so study it carefully! Two things to note:

1. The type of the client, τ_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 Δ. It is in this sense that abstract types are "new"! Whenever a client opens a package, it introduces a local name for the representation type, which is bound within the body of the client. By our general conventions on bound variables, this local name may be chosen to ensure that it is distinct from any other such local name that may be in scope, which ensures that the "new" type is different from any other type currently in scope. At an informal level this ensures that the representation type is "held abstract"; we will make this intuition more precise in Section 21.2 below.

21.1.4 Dynamic Semantics

We will use structured operational semantics (SOS) to specify the dynamic semantics of existential types. Here is the rule for evaluating package expressions:

$$\frac{e \mapsto e'}{\mathtt{pack}\,\tau\,\mathtt{with}\,e\,\mathtt{as}\,\sigma \mapsto \mathtt{pack}\,\tau\,\mathtt{with}\,e'\,\mathtt{as}\,\sigma} \tag{21.4}$$

Opening a package begins by evaluating the package expressions:

$$\frac{e_i \mapsto e_i'}{\texttt{open}\, e_i\, \texttt{as}\, t\, \texttt{with}\, x\!:\!\sigma\, \texttt{in}\, e_c \mapsto \texttt{open}\, e_i'\, \texttt{as}\, t\, \texttt{with}\, x\!:\!\sigma\, \texttt{in}\, e_c} \tag{21.5}$$

Once the package is fully evaluated, we bind t to the representation type and x to the implementation of the operations within the client code:

$$\frac{(\sigma = \exists t(\sigma'))}{\texttt{open}\, (\texttt{pack}\, \tau\, \texttt{with}\, v\, \texttt{as}\, \sigma)\, \texttt{as}\, t\, \texttt{with}\, x\!:\!\sigma'\, \texttt{in}\, e_c \mapsto \{\tau, v/t, x\}e_c} \tag{21.6}$$

Observe that *there are no abstract types at run time*! During execution of the client, the representation type is *fully exposed*. It is held abstract *only* during type checking to ensure that the client does not (accidentally or maliciously) depend on the implementation details of the abstraction. Once the program type checks there is no longer any need to enforce abstraction. The dynamic semantics reflects this intuition directly.

21.1.5 Safety

The safety of the extension is stated and proved as usual. The argument is a simple extension of that used for PolyMinML to the new constructs.

Theorem 21.1 (Preservation)
If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.

Lemma 21.2 (Canonical Forms)
If $v : \exists t(\sigma)$ is a value, then $v = \texttt{pack}\, \tau\, \texttt{with}\, v'\, \texttt{as}\, \exists t(\sigma)$ for some monotype τ and some value $v' : \{\tau/t\}\sigma$.

Theorem 21.3 (Progress)
If $e : \tau$ then either e value or there exists e' such that $e \mapsto e'$.

21.2 Representation Independence

Parametricity is the essence of representation independence. The typing rules for `open` given above ensure that the client of an abstract type is

polymorphic in the representation type. According to our informal understanding 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

$$\texttt{pack}\,\tau_1\,\texttt{with}\,v_1\,\texttt{as}\,\exists t(\sigma)$$

and

$$\texttt{pack}\,\tau_2\,\texttt{with}\,v_2\,\texttt{as}\,\exists t(\sigma)$$

of type $\exists t(\sigma)$ iff $v_1 \cong_{val} v_2 : \sigma$, taking equality at type t to be the relation R. The reason for calling such a relation R a bisimulation will become apparent shortly. Two packages are said to be *bisimilar* whenever there is a bisimulation between them.

Since the client e_c of a data abstraction of type $\exists t(\sigma)$ is essentially a polymorphic function of type $\forall t(\sigma \to \tau_c)$, where $t \notin \mathrm{FTV}(\tau_c)$, it follows from the Parametricity Theorem that

$$\{\tau_1, v_1/t, x\}e_c \cong_{exp} \{\tau_2, v_2/t, x\}e_c : \tau_c$$

whenever R is such a bisimulation. Consequently,

$$\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x{:}\sigma\,\texttt{in}\,e_c \cong_{exp} \texttt{open}\,e_2\,\texttt{as}\,t\,\texttt{with}\,x{:}\sigma\,\texttt{in}\,e_c : \tau_c.$$

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:

```
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 : `int list` $\leftrightarrow$ `int list*int list` as follows:

$$R = \{\, (l, (b, f)) \mid l \cong_{val} b\texttt{@rev}(f) \,\}$$

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.empty` R `QFB.empty`.
2. Assuming that $m \cong_{val} n$: `int` and $l\ R\ (b, f)$, show that
$$\texttt{QL.insert}((m, l))\ R\ \texttt{QFB.insert}((n, (b, f))).$$
3. Assuming that $l\ R\ (b, f)$, show that
$$\texttt{QL.remove}(l) \cong_{exp} \texttt{QFB.remove}((b, f)) : \texttt{int}*t,$$
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.empty` R `QFB.empty`, it suffices to show that

   $$\texttt{nil@rev(nil)} \cong_{exp} \texttt{nil} : \texttt{int list},$$

   which is obvious from the definitions of append and reverse.

2. For `insert`, we assume that $m \cong_{val} n : \texttt{int}$ and $l\ R\ (b, f)$, and prove that

   $$\texttt{QL.insert}(m, l)\ R\ \texttt{QFB.insert}(n, (b, f)).$$

   By the definition of `QL.insert`, the left-hand side is equivalent to $m :: l$, and by the definition of `QR.insert`, the right-hand side is equivalent to $(n :: b, f)$. It suffices to show that

   $$m :: l \cong_{exp} (n :: b)\texttt{@rev}(f) : \texttt{int list}.$$

   Calculating, we obtain

   $$\begin{array}{lll} (n :: b)\texttt{@rev}(f) & \cong_{exp} & n :: (b\texttt{@rev}(f)) \\ & \cong_{exp} & n :: l \end{array}$$

   since $l \cong_{exp} b\texttt{@rev}(f)$. Since $m \cong_{val} n : \texttt{int}$, it follows that $m = n$, which completes the proof.

3. For `remove`, we assume that l is related by R to (b, f), which is to say that $l \cong_{exp} b\texttt{@rev}(f)$. We are to show

   $$\texttt{QL.remove}(l) \cong_{exp} \texttt{QFB.remove}((b, f)) : \texttt{int} * t,$$

   taking t equality to be the relation R. Assuming that the queue is non-empty, so that the `remove` is defined, we have $l \cong_{exp} l'\texttt{@}[m]$ for some l' and m. We proceed by cases according to whether or not f is empty. If f is non-empty, then $f \cong_{exp} n :: f'$ for some n and f'. Then by the definition of `QFB.remove`,

   $$\texttt{QFB.remove}((b, f)) \cong_{exp} (n, (b, f')) : \texttt{int} * t,$$

relative to R. We must show that

$$(m, l') \cong_{exp} (n, (b, f')) : \texttt{int} * t,$$

relative to R. This means that we must show that $m = n$ and $l' \cong_{exp} b\texttt{@rev}(f') : \texttt{int list}$.

Calculating from our assumptions,

$$\begin{array}{rcl} l & = & l'\texttt{@}[m] \\ & = & b\texttt{@rev}(f) \\ & = & b\texttt{@rev}(n :: f') \\ & = & b\texttt{@}(\texttt{rev}(f')\texttt{@}[n]) \\ & = & (b\texttt{@rev}(f'))\texttt{@}[n] \end{array}$$

From this the result follows. Finally, if f is empty, then $b \cong_{exp} b'\texttt{@}[n]$ for some b' and n. But then $\texttt{rev}(b) \cong_{exp} n :: \texttt{rev}(b')$, which reduces to the case for f non-empty.

This completes the proof — by Representation Independence the reference and candidate implementations are equivalent.

Part VIII

Lazy Evaluation

Chapter 22

Lazy Types

The language MinML is an example of an *eager*, or *strict*, functional language. Such languages are characterized by two, separable features of their operational semantics.

1. *Call-by-value*. The argument to a function is evaluated before control is passed to the body of the function. Function parameters are only ever bound to values.

2. *Strict data types*. A value of a data type is constructed, possibly from other values, at the point at which the constructor is used.

Since most familiar languages are eager, this might seem to be the most natural, or even the only possible, choice. The subject of this chapter is to explore an alternative, *lazy evaluation*, that seeks to delay evaluation of expressions as long as possible, until their value is actually required to complete a computation. This strategy is called "lazy" because we perform only the evaluation that is actually required to complete a computation. If the value of an expression is never required, it is never (needlessly) computed. Moreover, the lazy evaluation strategy memoizes delayed computations so that they are never performed more than once. Once (if ever) the value has been determined, it is stored away to be used in case the value is ever needed again.

Lazy languages are characterized by the following features of their operational semantics.

1. *Call-by-need*. The argument to a function is passed to the body of the function without evaluating it. The argument is only evaluated if it

is needed in the computation, and then its value is saved for future reference in case it is needed again.

2. *Lazy data types.* An expression yielding a value of a data type is not evaluated until its value is actually required to complete a computation. The value, once obtained, is saved in case it is needed again.

While it might seem, at first glance, that lazy evaluation would lead to more efficient programs (by avoiding unnecessary work), it is not at all obvious that this is the case. In fact it's not the case. The main issue is that memoization is costly, because of the bookkeeping overhead required to manage the transition from unevaluated expression to evaluated value. A delayed computation must store the *code* that determines the value of an expression (should it be required), together with some means of triggering its evaluation once it is required. If the value is ever obtained, the value determined by the code must be stored away, and we must somehow ensure that this value is returned on subsequent access. This can slow down many programs. For example, if we know that a function will inspect the value of every element of a list, it is much more efficient to simply evaluate these elements when the list is created, rather than fruitlessly delaying the computation of each element, only to have it be required eventually anyway. *Strictness analysis* is used in an attempt to discover such cases, so that the overhead can be eliminated, but in general it is impossible (for decidability reasons) to determine completely and accurately whether the value of an expression is surely needed in a given program.

The real utility of lazy evaluation lies not in the possible efficiency gains it may afford in some circumstances, but rather in a substantial increase in expressive power that it brings to a language. By delaying evaluation of an expression until it is needed, we can naturally model situations in which the value *does not even exist* until it is required. A typical example is interactive input. The user can be modelled as a "delayed computation" that produces its values (*i.e.*, enters its input) only upon demand, not all at once before the program begins execution. Lazy evaluation models this scenario quite precisely.

Another example of the use of lazy evaluation is in the representation of infinite data structures, such as the sequence of all natural numbers. Obviously we cannot hope to compute the entire sequence at the time that it is created. Fortunately, only a finite initial segment of the sequence is ever needed to complete execution of a program. Using lazy evaluation we can

compute this initial segment on demand, avoiding the need to compute the part we do not require.

Lazy evaluation is an important and useful concept to have at your disposal. The question that we shall explore in this chapter is how best to provide such a feature in a programming language. Historically, there has been a division between eager and lazy *languages*, exemplified by ML and Haskell, respectively, which impose one or the other evaluation strategy globally, leaving no room for combining the best of both approaches.

More recently, it has come to be recognized by both communities that it is important to support both forms of evaluation. This has led to two, distinct approaches to supporting laziness:

1. *Lazy types in a strict language*. The idea is to add support for lazy data types to a strict language by providing a means of defining such types, and for creating and destroying values of these types. Constructors are implicitly memoized to avoid redundant re-computation of expressions. The call-by-value evaluation strategy for functions is maintained.

2. *Strict types in a lazy language*. The idea is to add support for constructors that forcibly evaluate their arguments, avoiding the overhead of managing the bookkeeping associated with delayed, memoized computation. The call-by-need evaluation strategy for function calls is maintained.

We will explore both alternatives.

22.1 Lazy Types in MinML

We will first explore the addition of lazy data types to a strict functional language. We will focus on a specific example, the type of lazy lists. For the sake of simplicity we'll consider only lazy lists of integers, but nothing hinges on this assumption.[1] For the rest of this section we'll drop the modifier "lazy", and just write "list", instead of "lazy list".

The key idea is to treat a *computation* of a list element as a *value* of list type, where a computation is simply a memoized, delayed evaluation of

[1] It simply allows us to avoid forward-referencing the concept of polymorphism.

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 list element, which is then analyzed and deconstructed. It is very important to keep in mind the distinction between *evaluation* of an expression of list type, and *forcing* a value of list type. The former simply yields a computation as value, whereas the latter evaluates and memoizes the delayed computation.

One consequence of laziness is that the tail of a (non-empty) lazy list, need not "exist" at the time the non-empty list is created. Being itself a lazy list, the tail need only be produced "on demand", by forcing a computation. This is the key to using lazy lists to model interactive input and to represent infinite data structures. For example, we might define the infinite list of natural numbers by the equation

```
nats = iterate successor 0
```

where the function `iterate` is defined (informally) by the equation

```
iterate f x = lcons (x, 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 succcessive recursive calls to `iterate`, which are only made as we explore the list.

Another approach to defining the infinite list of natural numbers is to make use of *self-reference*, as illustrated by the following example. The infinite sequence of natural numbers may be thought as a solution to the recursion equation

```
nats = lcons (0, lmap successor nats),
```

where `successor` and `lmap` are the evident functions. Here again we must think of `nats` as being created on demand. Successive elements of `nats` are created as follows. When we inspect the first element of `nats`, it is immediately revealed to be `0`, as specified. When we inspect the second element, we apply `lmap successor` to `nats`, then inspect the head element

of the result. This is `successor(0)`, or `1`; it's tail is the result of mapping `successor` over that list — that is, the result of adding `2` to every element of the original list, and so on.

22.1.1 Lazy Lists in an Eager Language

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

Types	τ	::=	`llist`
Expressions	e	::=	`lnil` $\mid$ `lcons(`e_1`,`e_2`)` $\mid$ `lazy` x `is` e $\mid$
			`lcase` e `of lnil =>` e_0 `| lcons(`x`,`y`) =>` e_1

In the expression `lazy` x `is` e the variable x is bound within e; in the expression `lcase` e `of lnil =>` e_0 `| lcons(`x`,`y`) =>` e_1 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 `lnil` and `lcons` — or by a recursion equation — using `lazy` x `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

```
lazy x is lcons(1,x).
```

More interesting examples can be expressed using recursive definitions such as the following definition of the list of all natural numbers:

```
lazy x is lcons (1, lmap successor x).
```

To complete this definition we must define `lmap`. This raises a subtle issue that is very easy to overlook. A natural choice is as follows:

```
fun map(f:int->int):llist->llist is
  fun lmapf(l:llist) is
    lcase l
       of lnil => lnil
        | lcons(x,y) => lcons (f x, lmapf y).
```

Unfortunately this definition doesn't work as expected! Suppose that f is a function of type `int->int` and that l is a non-empty lazy list. Consider what happens when we evaluate the expression `map` $f\ l$. The `lcase` forces evaluation of l, which leads to a recursive call to the internal function `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 `map` $f\ l$ should be represented by a *computation* of a list, in which subsequent calls to `map` on the tail(s) of that list are delayed until they are needed. This is achieved by the following coding trick:

```
fun map(f:int->int):llist->llist is
  fun lmapf(l:llist) is
    lazy _ is
      lcase l
         of lnil => lnil
          | lcons(x,y) => lcons (f x, lmapf y).
```

All we have done is to interpose a `lazy` constructor (with no name, indicated by writing an underscore) to ensure that the evaluation of the `lcase` expression is deferred until it is needed. Check for yourself that `map` $f\ l$ terminates even if l is an infinite list, precisely because of the insertion of the use of `lazy` in the body of `lmapf`. This usage is so idiomatic that we sometimes write instead the following definition:

```
fun map(f:int->int):llist->llist is
  fun lazy lmapf(l:llist) is
    lcase l
       of lnil => lnil
        | lcons(x,y) => lcons (f x, lmapf y).
```

The keyword `lazy` on the inner `fun` binding ensures that the body is evaluated lazily.

Exercise 22.1
Give a formal definition of `nats` in terms of `iterate` according to the informal equation given earlier. You will need to make use of `lazy` function definitions.

The static semantics of these lazy list expressions is given by the following typing rules:

$$\frac{}{\Gamma \vdash \texttt{lnil} : \texttt{llist}} \tag{22.1}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{llist}}{\Gamma \vdash \texttt{lcons}(e_1, e_2) : \texttt{llist}} \tag{22.2}$$

$$\frac{\Gamma, x{:}\texttt{llist} \vdash e : \texttt{llist}}{\Gamma \vdash \texttt{lazy}\, x\, \texttt{is}\, e : \texttt{llist}} \tag{22.3}$$

$$\frac{\Gamma \vdash e : \texttt{llist} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x{:}\texttt{int}, y{:}\texttt{llist} \vdash e_1 : \tau}{\Gamma \vdash \texttt{lcase}\, e\, \texttt{of}\, \texttt{lnil} \Rightarrow e_0 \mid \texttt{lcons}(x, y) \Rightarrow e_1 : \tau} \tag{22.4}$$

In Rule 22.2 the body, e, of the lazy list expression $\texttt{lazy}\, x\, \texttt{is}\, e$ 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 $\texttt{lnil}$, $\texttt{lcons}(e_1, e_2)$, and $\texttt{lazy}\, x\, \texttt{is}\, e$ to be values, independently of whether their constituent expressions are values. Second, we evaluate case analyses according to the following transition rules:

$$\frac{}{\texttt{lcase}\, \texttt{lnil}\, \texttt{of}\, \texttt{lnil} \Rightarrow e_0 \mid \texttt{lcons}(x, y) \Rightarrow e_1 \mapsto e_0} \tag{22.5}$$

$$\frac{}{\begin{array}{c}\texttt{lcase}\, \texttt{lcons}(e_h, e_t)\, \texttt{of}\, \texttt{lnil} \Rightarrow e_0 \mid \texttt{lcons}(x, y) \Rightarrow e_1 \\ \mapsto \\ \texttt{let}\, x{:}\texttt{int}\, \texttt{be}\, e_h\, \texttt{in}\, \texttt{let}\, y{:}\texttt{llist}\, \texttt{be}\, e_t\, \texttt{in}\, e_1\end{array}} \tag{22.6}$$

$$\frac{}{\begin{array}{c}\texttt{lcase}\, (\texttt{lazy}\, z\, \texttt{is}\, e)\, \texttt{of}\, \texttt{lnil} \Rightarrow e_0 \mid \texttt{lcons}(x, y) \Rightarrow e_1 \\ \mapsto \\ \texttt{lcase}\, \{\texttt{lazy}\, z\, \texttt{is}\, e/z\}e\, \texttt{of}\, \texttt{lnil} \Rightarrow e_0 \mid \texttt{lcons}(x, y) \Rightarrow e_1\end{array}} \tag{22.7}$$

$$\frac{e \mapsto e'}{\begin{array}{c}\texttt{lcase}\, e\, \texttt{of}\, \texttt{lnil} \Rightarrow e_0 \mid \texttt{lcons}(x,y) \Rightarrow e_1 \\ \mapsto \\ \texttt{lcase}\, e'\, \texttt{of}\, \texttt{lnil} \Rightarrow e_0 \mid \texttt{lcons}(x,y) \Rightarrow e_1\end{array}} \tag{22.8}$$

Observe that lazy list expressions are evaluated only when they appear as the subject of a case analysis expression. In the case of a non-empty list evaluation proceeds by first evaluating the head and tail of the list, then continuing with the appropriate clause. In the case of a recursively-defined list the expression is "unrolled" once before continuing analysis. This exposes the outermost structure of the list for further analysis.

Exercise 22.2
Define the functions `lhd:llist->int` *and* `ltl:llist->llist`*. Trace the evaluation of* `lhd(ltl(...(ltl(nats))...))`*, with* n *iterations of* `ltl`*, and verify that it evaluates to the number* n*.*

Exercise 22.3
State and prove the soundness of the non-memoizing dynamic semantics with respect to the static semantics given above.

Consider the lazy list value $v = \texttt{lazy}\, x\, \texttt{is}\, x$. It is easy to verify that e is well-typed, with type `llist`. It is also easy to see that performing a case analysis on v leads to an infinite regress, since $\{v/x\}x = v$. The value v is an example of a "black hole", a value that, when forced, will lead back to the value itself, and, moreover, is easily seen to lead to divergence. Another example of a black hole is the value

```
lazy x is (lmap succ x)
```

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

What is it that makes the recursive list

```
lazy nats is lcons (0, lmap succ nats)
```

well-defined? This expression is *not* a black hole because the occurrence of `nats` in the body of the recursive list expression is "guarded" by the call to `lcons`.

Exercise 22.4
Develop a type discipline that rules out black holes as ill-formed. Hint: *Define a judgement* $\Gamma \vdash e \downarrow x$, *which means that* x *is guarded within* e. *Ensure that* `lazy` x `is` e *is well-typed only if* x *is guarded within* e.

Exercise 22.5
It is often convenient to define several lists simultaneously by mutual recursion. Generalize `lazy` x `is` e *to admit simultaneous recursive definition of several lists at once.*

The foregoing dynamic semantics neglects the "evaluate at most once" aspect of laziness — if a lazy list expression is ever evaluated, its value should be stored so that re-evaluation is avoided should it ever be analyzed again. This can be modeled by introducing a memory that holds delayed computations whenever they are created. The memory is updated if (and only if) the value of that computation is ever required. Thus no evaluation is ever repeated, and some pending evaluations may never occur at all. This is called *memoization*.

The memoizing dynamic semantics is specified by an abstract machine with states of the form (M, e), where M is a memory, a finite mapping of variables to values, and e is an expression whose free variables are all in the domain of M. Free variables are used to stand for the values of list expressions; they are essentially pointers into the memory, which stores the value of the expression. We therefore regard free variables as values; these are in fact the only values of list type in this semantics.

The transition rules for the memoizing dynamic semantics are as follows:

$$\frac{(x \notin \mathrm{dom}(M))}{(M, \texttt{lazy}\, z\, \texttt{is}\, e) \mapsto (M[x{=}\texttt{lazy}\, z\, \texttt{is}\, e], x)} \tag{22.9}$$

$$\frac{(x \notin \mathrm{dom}(M))}{(M, \texttt{lnil}) \mapsto (M[x{=}\texttt{lnil}], x)} \tag{22.10}$$

$$\frac{(x \notin \mathrm{dom}(M))}{(M, \texttt{lcons(}e_1\texttt{,}e_2\texttt{)}) \mapsto (M[x{=}\texttt{lcons(}e_1\texttt{,}e_2\texttt{)}], x)} \tag{22.11}$$

$$\frac{(M(z) = \texttt{lnil})}{\begin{array}{c}(M, \texttt{lcase}\, z\, \texttt{of}\, \texttt{lnil} \Rightarrow e_0 \mid \texttt{lcons}(x,y) \Rightarrow e_1) \\ \mapsto \\ (M, e_0)\end{array}} \tag{22.12}$$

$$\frac{(M(z) = \texttt{lcons}(v_h, v_t))}{(M, \texttt{lcase}\, z\, \texttt{of}\, \texttt{lnil} \Rightarrow e_0 \mid \texttt{lcons}(x,y) \Rightarrow e_1) \mapsto (M, \{v_h, v_t/x, y\}e_1)} \tag{22.13}$$

$$\frac{(M(z) = \texttt{lcons}(e_h, e_t)) \quad (M[z{=}\bullet], e_h) \mapsto^* (M', v_h) \quad (M'[z{=}\bullet], e_t) \mapsto^* (M'', v_t)}{\begin{array}{c}(M, \texttt{lcase}\, z\, \texttt{of}\, \texttt{lnil} \Rightarrow e_0 \mid \texttt{lcons}(x,y) \Rightarrow e_1) \\ \mapsto \\ (M''[z{=}\texttt{lcons}(v_h, v_t)], \{v_h, v_t/x, y\}e_1)\end{array}} \tag{22.14}$$

$$\frac{(M(z) = \texttt{lazy}\, z\, \texttt{is}\, e) \quad (M[z{=}\bullet], e) \mapsto^* (M', v)}{\begin{array}{c}(M, \texttt{lcase}\, z\, \texttt{of}\, \texttt{lnil} \Rightarrow e_0 \mid \texttt{lcons}(x,y) \Rightarrow e_1) \\ \mapsto \\ (M'[z{=}v], \texttt{lcase}\, v\, \texttt{of}\, \texttt{lnil} \Rightarrow e_0 \mid \texttt{lcons}(x,y) \Rightarrow e_1)\end{array}} \tag{22.15}$$

$$\frac{(M, e) \mapsto (M', e')}{\begin{array}{c}(M, \texttt{lcase}\, e\, \texttt{of}\, \texttt{lnil} \Rightarrow e_0 \mid \texttt{lcons}(x,y) \Rightarrow e_1) \\ \mapsto \\ (M', \texttt{lcase}\, e'\, \texttt{of}\, \texttt{lnil} \Rightarrow e_0 \mid \texttt{lcons}(x,y) \Rightarrow e_1)\end{array}} \tag{22.16}$$

Warning: These rules are very subtle! Here are some salient points to keep in mind when studying them.

First, observe that the list-forming constructs are no longer values, but instead have evaluation rules associated with them. These rules simply store a pending computation in the memory and return a "pointer" to it as result. Thus a value of lazy list type is always a variable referring to a pending computation in the store.

Second, observe that the rules for case analysis inspect the contents of memory to determine how to proceed. The case for `lnil` is entirely straightforward, but the other two cases are more complex. Suppose that location z contains $\texttt{lcons}(e_1, e_2)$. First, we check whether we've already evaluated this list cell. If so, we continue by evaluating e_1, with x and y replaced by the previously-computed values of the head and tail of the list.

Otherwise, the time has come to evaluate this cell. We evaluate the head and tail *completely* to obtain their values, then continue by substituting these values for the appropriate variables in the clause for non-empty lists. Moreover, we update the memory to record the values of the head and tail of the list so that subsequent accesses avoid re-evaluation. Similarly, if z contains a recursively-defined list, we fully evaluate its body, continuing with the result and updating the memory to reflect the result of evaluation.

Third, we explicitly check for "black holes" by ensuring that a run-time error occurs whenever they are encountered. This is achieved by temporarily setting the contents of a list cell to the special "black hole" symbol, $\bullet$, during evaluation of a list expression, thereby ensuring the evaluation "gets stuck" (*i.e.*, incurs a run-time error) in the case that evaluation of a list expression requires the value of the list itself.

Exercise 22.6
Convince yourself that the replacement of z by $\bullet$ in the second premise of Rule 22.14 is redundant — the location z is already guaranteed to be bound to $\bullet$.

Exercise 22.7
State and prove the soundness of the memoizing dynamic semantics with respect to the static semantics given above. Be certain that your treatment of the memory takes account of cyclic dependencies.

Exercise 22.8
Give an evaluation semantics for memoized lazy lists by a set of rules for deriving judgements of the form $(M, e) \Downarrow (M', v)$.

Exercise 22.9
Consider once again the augmented static semantics in which black holes are ruled out. Prove that evaluation never "gets stuck" by accessing a cell that contains the black hole symbol.

Exercise 22.10
Consider again the definition of the natural numbers as the lazy list

```
lazy nats is (lcons (0, lmap succ nats)).
```

Prove that, for the non-memoized semantics, that accessing the nth element requires $O(n^2)$ time, whereas in the memoized semantics the same

computation requires $O(n)$ time. This shows that memoization can improve the asymptotic complexity of an algorithm (not merely lower the constant factors).

22.1.2 Delayed Evaluation and Lazy Data Structures

Another approach to lazy evaluation in the context of a strict language is to isolate the notion of a delayed computation as a separate concept. The crucial idea is that a delayed computation is a *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:

$$\begin{array}{lrcl} \textit{Types} & \tau & ::= & \tau\,\texttt{computation} \\ \textit{Expressions} & e & ::= & \texttt{delay}\,x\,\texttt{is}\,e \mid \texttt{eval}\,e_1\,\texttt{as}\,x\,\texttt{in}\,e_2 \end{array}$$

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

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{delay}\,x\,\texttt{is}\,e : \tau\,\texttt{computation}} \tag{22.17}$$

$$\frac{\Gamma \vdash e_1 : \tau_1\,\texttt{computation} \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{eval}\,e_1\,\texttt{as}\,x\,\texttt{in}\,e_2 : \tau_2} \tag{22.18}$$

A memoizing dynamic semantics for computations is given as follows. We admit, as before, variables as values; they serve as references to memo

cells that contain delayed computations. The evaluation rules are as follows:

$$\frac{(x \notin \mathrm{dom}(M))}{(M, \texttt{delay}\, x\, \texttt{is}\, e) \mapsto (M[x{=}\texttt{delay}\, x\, \texttt{is}\, e], x)} \tag{22.19}$$

$$\frac{(M(z) = \texttt{delay}\, z\, \texttt{is}\, e) \quad (M[z{=}\bullet], e) \mapsto^* (M', v)}{(M, \texttt{eval}\, z\, \texttt{as}\, x\, \texttt{in}\, e) \mapsto (M'[z{=}v], \{v/x\}e)} \tag{22.20}$$

$$\frac{(M(z) = v)}{(M, \texttt{eval}\, z\, \texttt{as}\, x\, \texttt{in}\, e) \mapsto (M', \{v/x\}e)} \tag{22.21}$$

$$\frac{(M, e_1) \mapsto (M', e_1')}{(M, \texttt{eval}\, e_1\, \texttt{as}\, x\, \texttt{in}\, e_2) \mapsto (M', \texttt{eval}\, e_1'\, \texttt{as}\, x\, \texttt{in}\, e_2)} \tag{22.22}$$

Exercise 22.11
State and prove the soundness of this extension to MinML.

One advantage of such a type of memoized, delayed computations is that it isolates the machinery of lazy evaluation into a single type constructor that can be used to define many different lazy data structures. For example, the type `llist` of lazy lists may be defined to be the type `lcell computation`, where `lcell` has the following constructors and destructors:

$$\frac{}{\Gamma \vdash \texttt{cnil} : \texttt{lcell}} \tag{22.23}$$

$$\frac{\Gamma \vdash e_h : \texttt{int} \quad \Gamma \vdash e_t : \texttt{llist}}{\Gamma \vdash \texttt{ccons}(e_h, e_t) : \texttt{lcell}} \tag{22.24}$$

$$\frac{\Gamma \vdash e : \texttt{lcell} \quad \Gamma \vdash e_n : \tau \quad \Gamma, x{:}\texttt{int}, y{:}\texttt{llist} \vdash e_c : \tau}{\Gamma \vdash \texttt{ccase}\, e\, \texttt{of}\, \texttt{cnil} \Rightarrow e_n \mid \texttt{ccons}(x, y) \Rightarrow e_c : \tau} \tag{22.25}$$

Observe that the "tail" of a `ccons` is of type `llist`, not `lcell`. Using these primitives we may define the lazy list constructors as follows:

```
lnil = lazy _ is cnil
lcons(e_h,e_t) = lazy _ is ccons(e_h, e_t)
lcase e of nil => e_n | cons(x, y) => e_c =
  force z=e in
    case z of cnil => e_n | ccons(x,y) => e_c
```

Observe that case analysis on a lazy list forces the computation of that list, then analyzes the form of the outermost lazy list cell.

This "two-stage" construction of lazy lists in terms of lazy cells is often short-circuited by simply identifying `llist` with `lcell`. However, this is a mistake! The reason is that according to this definition every lazy list expression must immediately determine whether the list is empty, and, if not, must determine its first element. But this conflicts with the "computation on demand" interpretation of laziness, according to which a lazy list might not even have a first element at the time that the list is defined, but only at the time that the code inspects it. It is therefore imperative to distinguish, as we have done, between the type `llist` of lazy lists (delayed computations of cells) and the type `lcell` of lazy cells (which specify emptiness and define the first element of non-empty lists).

Chapter 23

Lazy Languages

So far we've been considering the addition of *lazy types* to *eager languages*. Now we'll consider the alternative, the notion of a *lazy lanuage* and, briefly, the addition of *eager types* to a lazy language.

As we said in the introduction the main features of a lazy language are the *call-by-need* argument-passing discipline together with *lazy value constructors* that construct values of a type from delayed computations. Under call-by-value the arguments to functions and constructors are evaluated before the function is called or the constructor is applied. Variables are only ever bound to fully-evaluated expressions, or values, and constructors build values out of other values. Under call-by-need arguments are passed to functions in delayed, memoized form, without evaluating them until they are needed. Moreover, value constructors build delayed, memoized computations out of other delayed, memoized computations, without evaluation. Variables are, in general, bound to pending computations that are only forced when (and if) that value is required. Once forced, the binding is updated to record the computed value, should it ever be required again.

The interesting thing is that the static typing rules for the lazy variant of MinML are exactly the same as those for the eager version. What is different is how those types are interpreted. In an eager language values of type `int` are integer values (*i.e.*, numbers); in a lazy language they are integer computations, some of which might not even terminate when evaluated. Similarly, in an eager language values of list type are finite sequences of values of the element type; in a lazy language values of list type are computations of such sequences, which need not be finite. And

so on. The important point is that the types have different meanings in lazy languages than they do in strict languages.

One symptom of this difference is that lazy languages are very liberal in admitting recursive definitions compared to eager languages. In an eager language it makes no sense to admit recursive definitions such as

```
val x : int = 1+x
```

or

```
val x : int list = cons (1, x).
```

Roughly speaking, neither of these recursion equations has a solution. There is no integer value x satisfying the equation $x = 1 + x$, nor is there any finite list satisfying the equation $x = \texttt{cons}(1, x)$.

However, as we've already seen, equations such as

```
val x : int delayed = delay (1 + x)
```

and

```
val x : int list delayed = delay (lcons (1, x))
```

do make sense, precisely because they define recursive computations, rather than values. The first example defines a computation of an integer that, when forced, diverges; the second defines a computation of a list that, when forced, computes a non-empty list with 1 as first element and the list itself as tail.

In a lazy language *every* expression stands for a computation, so it is always sensible to make a recursive definition such as

```
val rec x : int = 1+x.
```

Syntactically this looks like the inadmissible definition discussed above, but, when taken in the context of a lazy interpretation, it makes perfect sense as a definition of a recursive computation — the value of `x` is the divergent computation of an integer.

The downside of admitting such a liberal treatment of computations is that it leaves no room in the language for ordinary values! Everything's a computation, with values emerging as those computations that happen to have a trivial evaluation (*e.g.*, numerals are trivial computations in the

sense that no work is required to evaluate them). This is often touted as an advantage of lazy languages — the "freedom" to ignore whether something is a value or not. But this appearance of freedom is really bondage. By admitting only computations, you are deprived of the ability to work with plain values. For example, lazy languages do not have a type of natural numbers, but rather only a type of computations of natural numbers. Consequently, elementary programming techniques such as definition by mathematical induction are precluded. The baby's been thrown out with the bathwater.

In recognition of this most lazy languages now admit eager types as well as lazy types, moving them closer in spirit to eager languages that admit lazy types, but biased in the opposite direction. This is achieved in a somewhat unsatisfactory manner, by relying on data abstraction mechanisms to ensure that the only values of a type are those that are generated by specified strict functions (those that evaluate their arguments). The reason it is unsatisfactory is that this approach merely limits the possible set of computations of a given type, but still admits, for example, the undefined computation as an element of every type.

23.0.3 Call-by-Name and Call-by-Need

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

$$\frac{\Gamma, x{:}\tau \vdash e : \tau}{\Gamma \vdash \texttt{rec}\,x:\tau\,\texttt{is}\,e : \tau} \tag{23.1}$$

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

$$\texttt{rec}\,f:\tau_1\rightarrow\tau_2\,\texttt{is}\,\texttt{fn}\,\tau_1:x\,\texttt{in}\,e.$$

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:

$$\overline{\texttt{fn}\,\tau\texttt{:}x\,\texttt{in}\,e(e') \mapsto \{\texttt{fn}\,\tau\texttt{:}x\,\texttt{in}\,e, e'/x\}e} \tag{23.2}$$

This is known as the *call-by-name*[1] 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:

$$\overline{\texttt{rec}\,x\texttt{:}\tau\,\texttt{is}\,e \mapsto \{\texttt{rec}\,x\texttt{:}\tau\,\texttt{is}\,e/x\}e} \tag{23.3}$$

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

Exercise 23.1
Show that the behavior of the recursive function expression $\texttt{fun}\,f\,\texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,e$ *is correctly defined by*

$$\texttt{rec}\,f\texttt{:}\tau_1\texttt{->}\tau_2\,\texttt{is}\,\texttt{fn}\,\tau_1\texttt{:}x\,\texttt{in}\,e$$

in the sense that an application of the latter mimicks the behavior of the former (under call-by-name).

To model the "at most once" aspect of lazy evaluation we introduce, as before, a memory in which we store computations, initially in their unevaluated, and later, if ever, in their evaluated forms. The difference here is that all expressions define computations that must be stored. Since the main ideas are similar to those used to define lazy lists, we simply give the evaluation rules here.

The state of computation is a pair (M, e) where M is a finite memory mapping variables to values, and e is an expression whose free variables lie within the domain of M. Final states have the form (M, v), where v is a closed value. In particular, v is *not* a variable.

[1] The terminology is well-established, but not especially descriptive. As near as I can tell the idea is that we pass the "name" of the computation (*i.e.*, the expression that engenders it), rather than its value.

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:

$$\frac{(M, e_1) \mapsto (M', e_1')}{(M, +(e_1, e_2)) \mapsto (M', +(e_1', e_2))} \tag{23.4}$$

$$\frac{(M, e_2) \mapsto (M', e_2')}{(M, +(v_1, e_2)) \mapsto (M', +(v_1, e_2'))} \tag{23.5}$$

$$\frac{}{(M, +(n_1, n_2)) \mapsto (M, n_1 + n_2)} \tag{23.6}$$

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

$$\frac{(x \notin \mathrm{dom}(M))}{(M, \mathtt{fn}\,\tau\mathtt{:}x\,\mathtt{in}\,e(e')) \mapsto (M[x = e'], e)} \tag{23.7}$$

$$\frac{(M(x) = v)}{(M, x) \mapsto (M, v)} \tag{23.8}$$

$$\frac{(M(x) = e) \quad (M[x = \bullet], e) \mapsto^* (M', v)}{(M, x) \mapsto (M'[x = v], v)} \tag{23.9}$$

$$\frac{(x \notin \mathrm{dom}(M))}{(M, \mathtt{rec}\,x\mathtt{:}\tau\,\mathtt{is}\,e) \mapsto (M[x = e], e)} \tag{23.10}$$

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

23.0.4 Strict Types in a Lazy Language

As discussed above, lazy languages are committed to the fundamental principle that the elements of a type are computations, which include values, and not just values themselves. This means, in particular, that every

type contains a "divergent" element, the computation that, when evaluated, goes into an infinite loop.[2]

One consequence, alluded to above, is that recursive type equations have overly rich solutions. For example, in this setting the recursive type equation

```
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 `cons` evaluate its tail whenever it is applied. This is called a *strictness* annotation. If `cons` is strict in its seond 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 `lnil` as a value, and regard `lcons`(e_1,e_2) as a value, regardless of whether

[2] This is often called "bottom", written $\bot$, for largely historical reasons. I prefer to avoid this terminology because so much confusion has been caused by it. In particular, it is *not* always correct to identify the least element of a domain with the divergent computation of that type! The domain of values of partial function type contains a least element, the totally undefined function, but this element does not correspond to the divergent computation of that type.

e_1 or e_2 are values. Then we define the transition rules for case analysis as follows:

$$\frac{}{\texttt{lcase}\ \texttt{lnil}\ \texttt{of}\ \texttt{lnil}\ \texttt{=>}\ e_n\ \texttt{|}\ \texttt{lcons}(x,y)\ \texttt{=>}\ e_c \mapsto e_n} \tag{23.11}$$

$$\frac{}{\texttt{lcase}\ \texttt{lcons}(e_1,e_2)\ \texttt{of}\ \texttt{lnil}\ \texttt{=>}\ e_n\ \texttt{|}\ \texttt{lcons}(x,y)\ \texttt{=>}\ e_c \mapsto \{e_1,e_2/x,y\}e_c} \tag{23.12}$$

If instead we wish to rule out infinite lists, then we may choose to regard $\texttt{lcons}(e_1,e_2)$ 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 $\texttt{lcons}(e_1,e_2)$ 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 IX

Dynamic Typing

Chapter 24

Dynamic Typing

The formalization of type safety given in Chapter 10 states that a language is type safe iff it satisfies both *preservation* and *progress*. According to this account, "stuck" states — non-final states with no transition — must be rejected by the static type system as ill-typed. Although this requirement seems natural for relatively simple languages such as MinML, it is not immediately clear that our formalization of type safety scales to larger languages, nor is it entirely clear that the informal notion of safety is faithfully captured by the preservation and progress theorems.

One issue that we addressed in Chapter 10 was how to handle expressions such as `3 div 0`, which are well-typed, yet stuck, in apparent violation of the progress theorem. We discussed two possible ways to handle such a situation. One is to enrich the type system so that such an expression is ill-typed. However, this takes us considerably beyond the capabilities of current type systems for practical programming languages. The alternative is to ensure that such ill-defined states are not "stuck", but rather make a transition to a designated error state. To do so we introduced the notion of a checked error, which is explicitly detected and signalled during execution. Checked errors are constrasted with unchecked errors, which are ruled out by the static semantics.

In this chapter we will concern ourselves with question of why there should unchecked errors at all. Why aren't all errors, including type errors, checked at run-time? Then we can dispense with the static semantics entirely, and, in the process, execute more programs. Such a language is called *dynamically typed*, in contrast to MinML, which is *statically typed*.

One advantage of dynamic typing is that it supports a more flexible

treatment of conditionals. For example, the expression

```
(if true then 7 else "7")+1
```

is statically ill-typed, yet it executes successfully without getting stuck or incurring a checked error. Why rule it out, simply because the type checker is unable to "prove" that the `else` branch cannot be taken? Instead we may shift the burden to the programmer, who is required to maintain invariants that ensure that no run-time type errors can occur, even though the program may contain conditionals such as this one.

Another advantage of dynamic typing is that it supports *heterogeneous data structures*, which may contain elements of many different types. For example, we may wish to form the "list"

```
[true, 1, 3.4, fn x=>x]
```

consisting of four values of distinct type. Languages such as ML preclude formation of such a list, insisting instead that all elements have the *same* type; these are called *homogenous* lists. The argument for heterogeneity is that there is nothing inherently "wrong" with such a list, particularly since its constructors are insensitive to the types of the components — they simply allocate a new node in the heap, and initialize it appropriately.

Note, however, that the additional flexibility afforded by dynamic typing comes at a cost. Since we cannot accurately predict the outcome of a conditional branch, nor the type of a value extracted from a heterogeneous data structure, we must program defensively to ensure that nothing bad happens, even in the case of a type error. This is achieved by turning type errors into checked errors, thereby ensuring progress and hence safety, even in the absence of a static type discipline. Thus dynamic typing catches type errors as late as possible in the development cycle, whereas static typing catches them as early as possible.

In this chapter we will investigate a dynamically typed variant of MinML in which type errors are treated as checked errors at execution time. Our analysis will reveal that, rather than being opposite viewpoints, dynamic typing is a *special case* of static typing! In this sense static typing is *more expressive* than dynamic typing, despite the superficial impression created by the examples given above. This viewpoint illustrates the *pay-as-you-go* principle of language design, which states that a program should only incur overhead for those language features that it actually uses. By viewing

dynamic typing as a special case of static typing, we may avail ourselves of the benefits of dynamic typing whenever it is required, but avoid its costs whenever it is not.

24.1 Dynamic Typing

The fundamental idea of dynamic typing is to regard type clashes as *checked*, rather than *unchecked*, errors. Doing so puts type errors on a par with division by zero and other checked errors. This is achieved by augmenting the dynamic semantics with rules that explicitly check for stuck states. For example, the expression `true+7` is such an ill-typed, stuck state. By checking that the arguments of an addition are integers, we can ensure that progress may be made, namely by making a transition to `error`.

The idea is easily illustrated by example. Consider the rules for function application in MinML given in Chapter 9, which we repeat here for convenience:

$$\frac{v\ \mathsf{value} \quad v_1\ \mathsf{value} \quad (v = \mathtt{fun}\, f\ (x:\tau_1):\tau_2\ \mathtt{is}\, e)}{\mathtt{apply}(v, v_1) \mapsto \{v, v_1/f, x\}e}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{apply}(e_1, e_2) \mapsto \mathtt{apply}(e_1', e_2)}$$

$$\frac{v_1\ \mathsf{value} \quad e_2 \mapsto e_2'}{\mathtt{apply}(v_1, e_2) \mapsto \mathtt{apply}(v_1, e_2')}$$

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

$$\frac{v\ \mathsf{value} \quad v_1\ \mathsf{value} \quad (v \neq \mathtt{fun}\, f\ (x:\tau_1):\tau_2\ \mathtt{is}\, e)}{\mathtt{apply}(v, v_1) \mapsto \mathtt{error}}$$

$$\frac{}{\mathtt{apply}(\mathtt{error}, e_2) \mapsto \mathtt{error}}$$

$$\frac{v_1\ \mathsf{value}}{\mathtt{apply}(v_1, \mathtt{error}) \mapsto \mathtt{error}}$$

The first rule states that a run-time error arises from any attempt to apply a non-function to an argument. The other two define the propagation of

such errors through other expressions — once an error occurs, it propagates throughout the entire program.

By entirely analogous means we may augment the rest of the semantics of MinML with rules to check for type errors at run time. Once we have done so, it is safe to eliminate the static semantics in its entirety.[1] Having done so, every expression is well-formed, and hence preservation holds vacuously. More importantly, the progress theorem also holds because we have augmented the dynamic semantics with transitions from every ill-typed expression to `error`, ensuring that there are no "stuck" states. Thus, the dynamically typed variant of MinML is safe in same sense as the statically typed variant. The meaning of safety does not change, only the means by which it is achieved.

24.2 Implementing Dynamic Typing

Since both the statically- and the dynamically typed variants of MinML are safe, it is natural to ask which is better. The main difference is in how early errors are detected — at compile time for static languages, at run time for dynamic languages. Is it better to catch errors early, but rule out some useful programs, or catch them late, but admit more programs? Rather than attempt to settle this question, we will sidestep it by showing that the apparent dichotomy between static and dynamic typing is illusory by showing that dynamic typing is a *mode of use* of static typing. From this point of view static and dynamic typing are matters of design for a particular program (which to use in a *given* situation), rather than a doctrinal debate about the design of a programming language (which to use in *all* situations).

To see how this is possible, let us consider what is involved in implementing a dynamically typed language. The dynamically typed variant of MinML sketched above includes rules for run-time type checking. For example, the dynamic semantics includes a rule that explicitly checks for an attempt to apply a non-function to an argument. How might such a check be implemented? The chief problem is that the natural representations of data values on a computer do not support such tests. For example,

[1] We may then simplify the language by omitting type declarations on variables and functions, since these are no longer of any use.

a function might be represented as a word representing a pointer to a region of memory containing a sequence of machine language instructions. An integer might be represented as a word interpreted as a two's complement integer. But given a word, you cannot tell, in general, whether it is an integer or a code pointer.

To support run-time type checking, we must adulterate our data representations to ensure that it is possible to implement the required checks. We must be able to tell by looking at the value whether it is an integer, a boolean, or a function. Having done so, we must be able to recover the underlying value (integer, boolean, or function) for direct calculation. Whenever a value of a type is created, it must be marked with appropriate information to identify the sort of value it represents.

There are many schemes for doing this, but at a high level they all amount to attaching a *tag* to a "raw" value that identifies the value as an integer, boolean, or function. Dynamic typing then amounts to checking and stripping tags from data during computation, transitioning to `error` whenever data values are tagged inappropriately. From this point of view, we see that dynamic typing should *not* be described as "run-time type checking", because we are not checking *types* at run-time, but rather *tags*. The difference can be seen in the application rule given above: we check only that the first argument of an application is some function, not whether it is well-typed in the sense of the MinML static semantics.

To clarify these points, we will make explicit the manipulation of tags required to support dynamic typing. To begin with, we revise the grammar of MinML to make a distinction between *tagged* and *untagged* values, as follows:

Expressions	e	$::=$	$x \mid v \mid o(e_1,\ldots,e_n) \mid \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid$ $\texttt{apply}(e_1,e_2)$
TaggedValues	v	$::=$	$\texttt{Int}\,(n) \mid \texttt{Bool}\,(\texttt{true}) \mid \texttt{Bool}\,(\texttt{false}) \mid$ $\texttt{Fun}\,(\texttt{fun}\,x\,(y:\tau_1):\tau_2\,\texttt{is}\,e)$
UntaggedValues	u	$::=$	$\texttt{true} \mid \texttt{false} \mid n \mid \texttt{fun}\,x\,(y:\tau_1):\tau_2\,\texttt{is}\,e$

Note that *only* tagged values arise as expressions; untagged values are used strictly for "internal" purposes in the dynamic semantics. Moreover, we do not admit general tagged expressions such as $\texttt{Int}\,(e)$, but only explicitly-tagged values.

Second, we introduce *tag checking* rules that determine whether or not a tagged value has a given tag, and, if so, extracts its underlying untagged

value. In the case of functions these are given as rules for deriving judgements of the form v is_fun u, which checks that v has the form `Fun (`u`)`, and extracts u from it if so, and for judgements of the form v isnt_fun, that checks that v does not have the form `Fun (`u`)` for any untagged value u.

$$\frac{}{\texttt{Fun}\ (u)\ \text{is\_fun}\ u}$$

$$\frac{}{\texttt{Int}\ (\_)\ \text{isnt\_fun}} \qquad \frac{}{\texttt{Bool}\ (\_)\ \text{isnt\_fun}}$$

Similar judgements and rules are used to identify integers and booleans, and to extract their underlying untagged values.

Finally, the dynamic semantics is re-formulated to make use of these judgement forms. For example, the rules for application are as follows:

$$\frac{v_1\ \text{value} \quad v\ \text{is\_fun}\ \texttt{fun}\ f\ (x:\tau_1):\tau_2\ \texttt{is}\ e}{\texttt{apply}(v, v_1) \mapsto \{v, v_1/f, x\}e}$$

$$\frac{v\ \text{value} \quad v\ \text{isnt\_fun}}{\texttt{apply}(v, v_1) \mapsto \texttt{error}}$$

Similar rules govern the arithmetic primitives and the conditional expression. For example, here are the rules for addition:

$$\frac{v_1\ \text{value} \quad v_2\ \text{value} \quad v_1\ \text{is\_int}\ n_1 \quad v_2\ \text{is\_int}\ n_2 \quad (n = n_1 + n_2)}{+(v_1, v_2) \mapsto \texttt{Int}\ (n)}$$

Note that we must explicitly check that the arguments are tagged as integers, and that we must apply the integer tag to the result of the addition.

$$\frac{v_1\ \text{value} \quad v_2\ \text{value} \quad v_1\ \text{isnt\_int}}{+(v_1, v_2) \mapsto \texttt{error}}$$

$$\frac{v_1\ \text{value} \quad v_2\ \text{value} \quad v_1\ \text{is\_int}\ n_1 \quad v_2\ \text{isnt\_int}}{+(v_1, v_2) \mapsto \texttt{error}}$$

These rules explicitly check for non-integer arguments to addition.

24.3 Dynamic Typing as Static Typing

Once tag checking is made explicit, it is easier to see its hidden costs in both time and space — time to check tags, to apply them, and to extract

the underlying untagged values, and space for the tags themselves. This is a significant overhead. Moreover, this overhead is imposed *whether or not* the original program is statically type correct. That is, even if we can prove that no run-time type error can occur, the dynamic semantics nevertheless dutifully performs tagging and untagging, just as if there were no type system at all.

This violates a basic principle of language design, called the *pay-as-you-go* principle. This principle states that a language should impose the cost of a feature only to the extent that it is actually used in a program. With dynamic typing we pay for the cost of tag checking, even if the program is statically well-typed! For example, if all of the lists in a program are homogeneous, we should not have to pay the overhead of supporting heterogeneous lists. The choice should be in the hands of the programmer, not the language designer.

It turns out that we can eat our cake and have it too! The key is a simple, but powerful, observation: dynamic typing is but a mode of use of static typing, provided that our static type system includes a type of *tagged data*! Dynamic typing emerges as a particular style of programming with tagged data.

The point is most easily illustrated using ML. The type of tagged data values for MinML may be introduced as follows:

```
(* The type of tagged values. *)
datatype tagged =
  Int of int |
  Bool of bool |
  Fun of tagged -> tagged
```

Values of type `tagged` are marked with a value constructor indicating their outermost form. Tags may be manipulated using pattern matching.

Second, we introduce operations on tagged data values, such as addition or function call, that explicitly check for run-time type errors.

```
exception TypeError
fun checked_add (m:tagged, n:tagged):tagged =
    case (m,n) of
      (Int a, Int b) => Int (a+b)
    | (_, _) => raise TypeError
fun checked_apply (f:tagged, a:tagged):tagged =
    case f of
      Fun g => g a
    | _ => raise TypeError
```

Observe that these functions correspond precisely to the instrumented dynamic semantics given above.

Using these operations, we can then build heterogeneous lists as values of type `tagged list`.

```
val het_list : tagged list =
    [Int 1, Bool true, Fun (fn x => x)]
val f : tagged = hd(tl(tl het_list))
val x : tagged = checked_apply (f, Int 5)
```

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

It is enlightening to consider a dynamically typed version of the factorial function:

```
fun dyn_fact (n : tagged) =
    let fun loop (n, a) =
        case n
          of Int m =>
             (case m
                of 0 => a
                 | m => loop (Int (m-1),
                              checked_mult (m, a)))
           | _ => raise RuntimeTypeError
      in loop (n, Int 1)
    end
```

Notice that tags must be manipulated within the loop, even though we can prove (by static typing) that they are not necessary! Ideally, we would like to hoist these checks out of the loop:

```
fun opt_dyn_fact (n : tagged) =
    let fun loop (0, a) = a
          | loop (n, a) = loop (n-1, n*a)
     in case n
          of Int m => Int (loop (m, 1))
           | _ => raise RuntimeTypeError
    end
```

It is *very hard* for a compiler to do this hoisting reliably. But if you consider dynamic typing to be a special case of static typing, as we do here, there is no obstacle to doing this optimization yourself, as we have illustrated here.

Chapter 25

Featherweight Java

We will consider a tiny subset of the Java language, called *Featherweight Java*, or FJ, that models subtyping and inheritance in Java. We will then discuss design alternatives in the context of FJ. For example, in FJ, as in Java, the subtype relation is tightly coupled to the subclass relation. Is this necessary? Is it desirable? We will also use FJ as a framework for discussing other aspects of Java, including interfaces, privacy, and arrays.

25.1 Abstract Syntax

The abstract syntax of FJ is given by the following grammar:

Classes	C	$::=$	$\texttt{class}\, c\, \texttt{extends}\, c\, \{\underline{c}\, \underline{f}\texttt{;}\, k\, \underline{d}\}$
Constructors	k	$::=$	$c\texttt{(}\underline{c}\, \underline{x}\texttt{)}\, \{\texttt{super(}\underline{x}\texttt{);}\, \texttt{this.}\underline{f}\texttt{=}\underline{x}\texttt{;}\}$
Methods	d	$::=$	$c\, m\texttt{(}\underline{c}\, \underline{x}\texttt{)}\, \{\texttt{return}\, e\texttt{;}\}$
Types	τ	$::=$	c
Expressions	e	$::=$	$x \mid e\texttt{.}f \mid e\texttt{.}m\texttt{(}\underline{e}\texttt{)} \mid \texttt{new}\, c\texttt{(}\underline{e}\texttt{)} \mid \texttt{(}c\texttt{)}\, e$

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, `Object`, standing for the root of the class hierarchy. It's role will become clear below. We assume that there is a distinguished variable `this` that cannot otherwise be declared in a program.

As a notational convenience we use "underbarring" to stand for sequences of phrases. For example, $\underline{d}$ 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 $\underline{c}\ \underline{f}$ for the sequence $c_1\ f_1, \ldots, c_k\ f_k$, where k is the length of the sequences $\underline{c}$ and $\underline{f}$. Similar conventions govern the other uses of sequence notation.

The class expression

$$\texttt{class}\ c\ \texttt{extends}\ c'\ \{\underline{c}\ \underline{f}\,;\ k\ \underline{d}\}$$

declares the class c to be a subclass of the class c'. The subclass has additional fields $\underline{c}\ \underline{f}$, single constructor k, and method suite $\underline{d}$. The methods of the subclass may override those of the superclass, or may be new methods specific to the subclass.

The constructor expression

$$c(\underline{c'}\ \underline{x'}, \underline{c}\ \underline{x})\ \{\texttt{super(}\underline{x'}\texttt{)}\,;\ \texttt{this.}\underline{f}\texttt{=}\underline{x}\texttt{;}\}$$

declares the constructor for class c with arguments $\underline{c'}\ \underline{x'}, \underline{c}\ \underline{x}$, corresponding to the fields of the superclass followed by those of the subclass. The variables $\underline{x'}$ and $\underline{x}$ are bound in the body of the constructor. The body of the constructor indicates the initialization of the superclass with the arguments $\underline{x'}$ and of the subclass with arguments $\underline{x}$.

The method expression

$$c\ m(\underline{c}\ \underline{x})\ \{\texttt{return}\ e\texttt{;}\}$$

declares a method m yielding a value of class c, with arguments $\underline{x}$ of class $\underline{c}$ and body returning the value of the expression e. The variables $\underline{x}$ 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 $e\,.\,m(\underline{e})$ invokes the method m of instance e with arguments $\underline{e}$. The expression $\texttt{new}\ c(\underline{e})$ creates a new instance of class c, passing arguments $\underline{e}$ to the constructor for c. The expression $(c)\ e$ casts the value of e to class c.

```
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 25.1: A Sample FJ Program

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.

A *class table* T is a finite function assigning classes to class names. The classes declared in the class table are bound within the table so that all classes may refer to one another via the class table.

A *program* is a pair (T, e) consisting of a class table T and an expression e. We generally suppress explicit mention of the class table, and consider programs to be expressions.

A small example of FJ code is given in Figure 25.1. In this example we assume given a class `Object` of all objects and make use of types `int` and `color` that are not, formally, part of FJ.

25.2 Static Semantics

The static semantics of FJ is defined by a collection of judgments of the following forms:

$\tau <: \tau'$	*subtyping*
$\Gamma \vdash e : \tau$	*expression typing*
d ok in c	*well-formed method*
C ok	*well-formed class*
T ok	*well-formed class table*
$\mathsf{fields}(c) = \underline{c}\,\underline{f}$	*field lookup*
$\mathsf{type}(m, c) = \underline{c} \to c$	*method type*

The rules defining the static semantics follow.
Every variable must be declared:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \tag{25.1}$$

The types of fields are defined in the class table.

$$\frac{\Gamma \vdash e_0 : c_0 \quad \mathsf{fields}(c_0) = \underline{c}\,\underline{f}}{\Gamma \vdash e_0.f_i : c_i} \tag{25.2}$$

The argument and result types of methods are defined in the class table.

$$\frac{\Gamma \vdash e_0 : c_0 \quad \Gamma \vdash \underline{e} : \underline{c} \quad \mathsf{type}(m, c_0) = \underline{c'} \to c \quad \underline{c} <: \underline{c'}}{\Gamma \vdash e_0.m(\underline{e}) : c} \tag{25.3}$$

Instantiation must provide values for all instance variables as arguments to the constructor.

$$\frac{\Gamma \vdash \underline{e} : \underline{c} \quad \underline{c} <: \underline{c'} \quad \mathsf{fields}(c) = \underline{c'}\,\underline{f}}{\Gamma \vdash \mathtt{new}\, c(\underline{e}) : c} \tag{25.4}$$

All casts are statically valid, but must be checked at run-time.

$$\frac{\Gamma \vdash e_0 : d}{\Gamma \vdash \texttt{(}c\texttt{)}\ e_0 : c} \tag{25.5}$$

The subtyping relation is read directly from the class table. Subtyping is the smallest reflexive, transitive relation containing the subclass relation:

$$\frac{}{\tau <: \tau} \tag{25.6}$$

$$\frac{\tau <: \tau' \quad \tau' <: \tau''}{\tau <: \tau''} \tag{25.7}$$

$$\frac{T(c) = \texttt{class}\ c\ \texttt{extends}\ c'\ \{\ldots\ ;\ \ldots\}}{c <: c'} \tag{25.8}$$

A well-formed class has zero or more fields, a constructor that initializes the superclass and the subclass fields, and zero or more methods. To account for method override, the typing rules for each method are relative to the class in which it is defined.

$$\frac{\begin{array}{c} k = c(\underline{c'}\ \underline{x'}, \underline{c}\ \underline{x})\ \{\texttt{super(}\underline{x'}\texttt{)}\,;\ \texttt{this.}\underline{f}\texttt{=}\underline{x}\,;\} \\ \mathsf{fields}(c') = \underline{c'}\ \underline{f'} \quad \underline{d}\ \mathsf{ok\ in}\ c \end{array}}{\texttt{class}\ c\ \texttt{extends}\ c'\ \{\underline{c}\ \underline{f}\,;\ k\ \underline{d}\}\ \mathsf{ok}} \tag{25.9}$$

Method overriding takes account of the type of the method in the superclass. The subclass method must have the same argument types and result type as in the superclass.

$$\frac{\begin{array}{c} T(c) = \texttt{class}\ c\ \texttt{extends}\ c'\ \{\ldots\ ;\ \ldots\} \\ \mathsf{type}(m, c') = \underline{c} \to c_0 \quad \underline{x}\texttt{:}\underline{c}, \texttt{this:}c \vdash e_0 : c_0 \end{array}}{c_0\ m\texttt{(}\underline{c}\ \underline{x}\texttt{)}\ \{\texttt{return}\ e_0\,;\}\ \mathsf{ok\ in}\ c} \tag{25.10}$$

A class table is well-formed iff all of its classes are well-formed:

$$\frac{\forall c \in \mathrm{dom}(T)\ T(c)\ \mathsf{ok}}{T\ \mathsf{ok}} \tag{25.11}$$

Note that well-formedness of a class is relative to the class table!

A program is well-formed iff its method table is well-formed and the expression is well-formed:

$$\frac{T\,\mathsf{ok} \quad \emptyset \vdash e : \tau}{(T, e)\,\mathsf{ok}} \tag{25.12}$$

The auxiliary lookup judgments determine the types of fields and methods of an object. The types of the fields of an object are determined by the following rules:

$$\frac{}{\mathsf{fields}(\texttt{Object}) = \bullet} \tag{25.13}$$

$$\frac{T(c) = \texttt{class}\,c\,\texttt{extends}\,c'\,\{\underline{c}\,\underline{f}\,;\,\ldots\} \quad \mathsf{fields}(c') = \underline{c'}\,\underline{f'}}{\mathsf{fields}(c) = \underline{c'}\,\underline{f'}, \underline{c}\,\underline{f}} \tag{25.14}$$

The type of a method is determined by the following rules:

$$\frac{\begin{array}{c}T(c) = \texttt{class}\,c\,\texttt{extends}\,c'\,\{\ldots\,;\,\ldots\,\underline{d}\} \\ d_i = c_i\,m(\underline{c_i}\,\underline{x})\,\{\texttt{return}\,e\,;\}\end{array}}{\mathsf{type}(m_i, c) = \underline{c_i} \to c_i} \tag{25.15}$$

$$\frac{\begin{array}{c}T(c) = \texttt{class}\,c\,\texttt{extends}\,c'\,\{\ldots\,;\,\ldots\,\underline{d}\} \\ m \notin \underline{d} \quad \mathsf{type}(m, c') = \underline{c_i} \to c_i\end{array}}{\mathsf{type}(m, c) = \underline{c_i} \to c_i} \tag{25.16}$$

25.3 Dynamic Semantics

The dynamic semantics of FJ may be specified using SOS rules similar to those for MinML. The transition relation is indexed by a class table T, which governs the semantics of casting and dynamic dispatch (which see below). In the rules below we omit explicit mention of the class table for the sake of brevity.

An instance of a class has the form $\texttt{new}\,c(\underline{e})$, where each e_i is a value.

$$\frac{\underline{e}\ \mathsf{value}}{\texttt{new}\,c(\underline{e})\ \mathsf{value}} \tag{25.17}$$

Since we arrange that there be a one-to-one correspondence between instance variables and constructor arguments, an instance expression of this form carries all of the information required to determine the values of the fields of the instance. This makes clear that an instance is essentially just a labelled collection of fields. Each instance is labelled with its class, which is used to guide method dispatch.

Field selection retrieves the value of the named field from either the subclass or its superclass, as appropriate.

$$\frac{\mathsf{fields}(c) = \underline{c'}\ \underline{f'}, \underline{c}\ \underline{f} \quad \underline{e'}\ \mathsf{value} \quad \underline{e}\ \mathsf{value}}{\mathtt{new}\, c(\underline{e'}, \underline{e})\,.\, f'_i \mapsto e'_i} \tag{25.18}$$

$$\frac{\mathsf{fields}(c) = \underline{c'}\ \underline{f'}, \underline{c}\ \underline{f} \quad \underline{e'}\ \mathsf{value} \quad \underline{e}\ \mathsf{value}}{\mathtt{new}\, c(\underline{e'}, \underline{e})\,.\, f_i \mapsto e_i} \tag{25.19}$$

Message send replaces `this` by the instance itself, and replaces the method parameters by their values.

$$\frac{\mathsf{body}(m, c) = \underline{x} \to e_0 \quad \underline{e}\ \mathsf{value} \quad \underline{e'}\ \mathsf{value}}{\mathtt{new}\, c(\underline{e})\,.\, m(\underline{e'}) \mapsto \{\underline{e'}/\underline{x}\}\{\mathtt{new}\, c(\underline{e})/\mathtt{this}\}e_0} \tag{25.20}$$

Casting checks that the instance is of a sub-class of the target class, and yields the instance.

$$\frac{c <: c' \quad \underline{e}\ \mathsf{value}}{(c')\ \mathtt{new}\, c(\underline{e}) \mapsto \mathtt{new}\, c(\underline{e})} \tag{25.21}$$

These rules determine the order of evaluation:

$$\frac{e_0 \mapsto e'_0}{e_0\,.\, f \mapsto e'_0\,.\, f} \tag{25.22}$$

$$\frac{e_0 \mapsto e'_0}{e_0\,.\, m(\underline{e}) \mapsto e'_0\,.\, m(\underline{e})} \tag{25.23}$$

$$\frac{e_0\ \mathsf{value} \quad \underline{e} \mapsto \underline{e'}}{e_0\,.\, m(\underline{e}) \mapsto e_0\,.\, m(\underline{e'})} \tag{25.24}$$

$$\frac{\underline{e} \mapsto \underline{e'}}{\texttt{new}\, c\,(\underline{e}) \mapsto \texttt{new}\, c\,(\underline{e'})} \tag{25.25}$$

$$\frac{e_0 \mapsto e_0'}{(c)\, e_0 \mapsto (c)\, e_0'} \tag{25.26}$$

Dynamic dispatch makes use of the following auxiliary relation to find the correct method body.

$$\frac{T(c) = \texttt{class}\, c\, \texttt{extends}\, c'\, \{\ldots;\ \ldots\ \underline{d}\} \quad d_i = c_i\, m_i(\underline{c_i\ x})\ \{\texttt{return}\, e;\}}{\mathsf{body}(m_i, c) = \underline{x} \to e} \tag{25.27}$$

$$\frac{T(c) = \texttt{class}\, c\, \texttt{extends}\, c'\, \{\ldots;\ \ldots\ \underline{d}\} \quad m \notin \underline{d} \quad \mathsf{body}(m, c') = \underline{x} \to e}{\mathsf{body}(m, c) = \underline{x} \to e} \tag{25.28}$$

Finally, we require rules for evaluating sequences of expressions from left to right, and correspondingly defining when a sequence is a value (*i.e.*, consists only of values).

$$\frac{e_1\ \mathsf{value} \quad \ldots \quad e_{i-1}\ \mathsf{value} \quad e_i \mapsto e_i'}{e_1, \ldots, e_{i-1}, e_i, e_{i+1}, \ldots, e_n \mapsto e_1, \ldots, e_{i-1}, e_i', e_{i+1}, \ldots, e_n} \tag{25.29}$$

$$\frac{e_1\ \mathsf{value} \quad \ldots \quad e_n\ \mathsf{value}}{\underline{e}\ \mathsf{value}} \tag{25.30}$$

This completes the dynamic semantics of FJ.

25.4 Type Safety

The safety of FJ is stated in the usual manner by the Preservation and Progress Theorems.

Since the dynamic semantics of casts preserves the "true" type of an instance, the type of an expression may become "smaller" in the subtype ordering during execution.

Theorem 25.1 (Preservation)
Assume that T is a well-formed class table. If $e : \tau$ and $e \mapsto e'$, then $e' : \tau'$ for some τ' such that $\tau' <: \tau$.

The statement of Progress must take account of the possibility that a cast may fail at execution time. Note, however, that field selection or message send can never fail — the required field or method will always be present.

Theorem 25.2 (Progress)
Assume that T is a well-formed class table. If $e : \tau$ then either

1. *v* value*, or*
2. *e contains an instruction of the form $(c)\,\texttt{new}\,c'(e_0)$ with e_0* value *and $c' \not<: c$, 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 25.3 (Canonical Forms)
If $e : c$ and e value*, then e has the form $\texttt{new}\,c'(e_0)$ with e_0* value *and $c' <: c$.*

25.5 Acknowledgement

This chapter is based on "Featherweight Java: A Minimal Core Calculus for Java and GJ" by Atsushi Igarashi, Benjamin Pierce, and Philip Wadler.

Part X

Subtyping and Inheritance

Chapter 26

Subtyping

A *subtype* relation is a pre-order[1] on types that validates the *subsumption principle*: if σ is a subtype of τ, then a value of type σ may be provided whenever a value of type τ is required. This means that a value of the subtype should "act like" a value of the supertype when used in supertype contexts.

26.1 MinML With Subtyping

We will consider two extensions of MinML with subtyping. The first, *MinML with implicit subtyping*, is obtained by adding the following rule of *implicit subsumption* to the typing rules of MinML:

$$\frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash e : \tau}$$

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:

$$\frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash (\tau)\, e : \tau}$$

[1] A pre-order is a reflexive and transitive binary relation.

The typing rules remain syntax-directed, but all uses of subtyping must be explicitly indicated.

We will refer to either variation as $\mathsf{MinML}_{<:}$ when the distinction does not matter. When it does, the implicit version is designated $\mathsf{MinML}^{i}_{<:}$, the implicit $\mathsf{MinML}^{e}_{<:}$.

To obtain a complete instance of $\mathsf{MinML}_{<:}$ 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:

$$\frac{}{\tau <: \tau} \qquad \frac{\rho <: \sigma \quad \sigma <: \tau}{\rho <: \tau}$$

Note that pure MinML is obtained as an instance of $\mathsf{MinML}^{i}_{<:}$ by giving no subtyping rules beyond these two, so that $\sigma <: \tau$ iff $\sigma = \tau$.

The dynamic semantics of an instance of $\mathsf{MinML}_{<:}$ 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 $\mathsf{MinML}_{<:}$, in either formulation, is assured, provided that the following *subtyping safety conditions* are met:

- For $\mathsf{MinML}^{e}_{<:}$, if $\sigma <: \tau$, then casting a value of the subtype σ to the supertype τ must yield a value of type τ.
- For $\mathsf{MinML}^{i}_{<:}$, 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 $\mathsf{MinML}_{<:}$.

Theorem 26.1 (Preservation)
For either variant of $\mathsf{MinML}_{<:}$*, under the assumption that the subtyping safety conditions hold, if* $e : \tau$ *and* $e \mapsto e'$*, then* $e' : \tau$*.*

Proof: By induction on the dynamic semantics, appealing to the casting condition in the case of the explicit subsumption rule of $\mathsf{MinML}^{e}_{<:}$. ■

Theorem 26.2 (Progress)
For either variant of $\mathsf{MinML}_{<:}$*, 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. ■

26.2 Varieties of Subtyping

In this section we will explore several different forms of subtyping in the context of extensions of MinML. To simplify the presentation of the examples, we tacitly assume that the dynamic semantics of casts is defined so that $(\tau)\, v \mapsto v$, unless otherwise specified.

26.2.1 Arithmetic Subtyping

In informal mathematics we tacitly treat integers as real numbers, even though $\mathbb{Z} \not\subseteq \mathbb{R}$. This is justified by the observation that there is an injection $\iota : \mathbb{Z} \hookrightarrow \mathbb{R}$ that assigns a canonical representation of an integer as a real number. This injection preserves the ordering, and commutes with the arithmetic operations in the sense that $\iota(m + n) = \iota(m) + \iota(n)$, where m and n are integers, and the relevant addition operation is determined by the types of its arguments.

In most cases the real numbers are (crudely) approximated by floating point numbers. Let us therefore consider an extension of MinML with an additional base type, `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]

[2] This convention is borrowed from O'Caml.

By analogy with mathematical practice, we will consider taking the type `int` to be a subtype of `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 $(\texttt{float})\,n$. We simply postulate that this is possible, writing $n.0$ for the floating point representation of the integer n, and noting that $n.0$ has type `float`.[3]

To ensure the safety of implicit subsumption we must ensure that the floating point arithmetic operations are well-defined for integer arguments. For example, we must ensure that an expression such as $+.(3,4)$ has a well-defined value as a floating point number. To achieve this, we simply require that floating point operations implicitly convert any integer arguments to floating point before performing the operation. In the foregoing example evaluation proceeds as follows:

$$+.(3,4) \mapsto +.(3.0,4.0) \mapsto 7.0.$$

This strategy requires that the floating point operations detect the presence of integer arguments, and that it convert any such arguments to floating point before carrying out the operation. We will have more to say about this inefficiency in Section 26.4 below.

26.2.2 Function Subtyping

Suppose that `int <: float`. What subtyping relationships, if any, should hold among the following four types?

1. `int`$\to$`int`
2. `int`$\to$`float`
3. `float`$\to$`int`
4. `float`$\to$`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.

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 : `int`$\rightarrow$`int`. If we apply f to x : `int`, the result has type `int`, and hence, by the arithmetic subtyping axiom, has type `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

$$\texttt{float}\rightarrow\texttt{int} <: \texttt{float}\rightarrow\texttt{float}$$

is also valid.

Now suppose that f : `float`$\rightarrow$`int`. If x : `int`, then x : `float` by subsumption, and hence we may apply f to x to obtain a result of type `int`. This suggests that

$$\texttt{float}\rightarrow\texttt{int} <: \texttt{int}\rightarrow\texttt{int}$$

is a valid subtype relationship. Since `int`$\rightarrow$`int` <: `int`$\rightarrow$`float`, it follows that

$$\texttt{float}\rightarrow\texttt{int} <: \texttt{int}\rightarrow\texttt{float}$$

is also valid.

Subtyping rules that specify how a type constructor interacts with subtyping are called *variance* principles. If a type constructor *preserves* subtyping in a given argument position, it is said to be *covariant* in that position. If, instead, it *inverts* subtyping in a given position it is said to be *contravariant* in that position. The discussion above suggests that the function space constructor is covariant in the range position and contravariant in the domain position. This is expressed by the following rule:

$$\frac{\tau_1 <: \sigma_1 \quad \sigma_2 <: \tau_2}{\sigma_1\rightarrow\sigma_2 <: \tau_1\rightarrow\tau_2}$$

Note well the inversion of subtyping in the domain, where the function constructor is contravariant, and the preservation of subtyping in the range, where the function constructor is covariant.

To ensure safety in the explicit case, we define the dynamic semantics of a cast operation by the following rule:

$$\overline{(\tau_1\rightarrow\tau_2)\, v \mapsto \texttt{fn}\, x{:}\tau_1\, \texttt{in}\, (\tau_2)\, v((\sigma_1)\, x)}$$

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.

26.2.3 Product and Record Subtyping

In Chapter 19 we considered an extension of MinML with product types. In this section we'll consider equipping this extension with subtyping. We will work with n-ary products of the form $\tau_1 * \cdots * \tau_n$ 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 ith 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 lth 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:

$$\frac{m > n}{\tau_1 * \cdots * \tau_m <: \tau_1 * \cdots * \tau_n}$$

Given a value of type $\tau_1 * \cdots * \tau_n$, we can access its ith component, for any $1 \leq i \leq n$. If $m > n$, then we can equally well access the ith component of an m-tuple of type $\tau_1 * \cdots * \tau_m$, 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:

$$\frac{m > n}{\{l_1 : \tau_1, \ldots, l_m : \tau_m\} <: \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}$$

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,

$$\frac{\forall 1 \leq i \leq n\ \sigma_i <: \tau_i}{\sigma_1 * \cdots * \sigma_n <: \tau_1 * \cdots * \tau_n}$$

and

$$\frac{\forall 1 \leq i \leq n\ \sigma_i <: \tau_i}{\{l_1 : \sigma_1, \ldots, l_n : \sigma_n\} <: \{l_1 : \tau_1, \ldots, l_n : \tau_n\}.}$$

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:

$$\frac{m \geq n}{(\tau_1 * \cdots * \tau_n) \langle v_1, \ldots, v_m \rangle \mapsto \langle (\tau_1)\, v_1, \ldots, (\tau_n)\, v_n \rangle.}$$

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 ith 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:

$$\langle v_1, \ldots, v_i, \ldots \rangle . i \mapsto v_i.$$

The ellision indicates that fields beyond the ith are not relevant to the operation. Similarly, for records we postulate that selection of the lth field is insensitive to the presence of any other fields:

$$\{l : v, \ldots\} . l \mapsto v.$$

The ellision expresses the independence of field selection from any "extra" fields.

26.2.4 Reference Subtyping

Finally, let us consider the reference types of Chapter 14. What should be the variance rule for reference types? Suppose that r has type $\sigma\,\texttt{ref}$. We can do one of two things with r:

1. Retrieve its contents as a value of type σ.
2. Replace its contents with a value of type σ.

If $\sigma <: \tau$, then retrieving the contents of r yields a value of type τ, by subsumption. This suggests that references are covariant:

$$\frac{\sigma <: \tau}{\sigma\,\texttt{ref} \stackrel{?}{<:} \tau\,\texttt{ref}.}$$

On the other hand, if $\tau <: \sigma$, then we may store a value of type τ into r. This suggests that references are contravariant:

$$\frac{\tau <: \sigma}{\sigma\,\texttt{ref} \stackrel{?}{<:} \tau\,\texttt{ref}.}$$

Given that we may perform either operation on a reference cell, we must insist that reference types are *invariant*:

$$\frac{\sigma <: \tau \quad \tau <: \sigma}{\sigma\,\texttt{ref} <: \tau\,\texttt{ref}\,.}$$

The premise of the rule is often strengthened to the requirement that σ and τ be equal:

$$\frac{\sigma = \tau}{\sigma\,\texttt{ref} <: \tau\,\texttt{ref}}$$

since there are seldom situations where distinct types are mutual subtypes.

A similar analysis may be applied to any mutable data structure. For example, *immutable* sequences may be safely taken to be covariant, but *mutable* sequences (arrays) must be taken to be invariant, lest safety be compromised.

26.3 Type Checking With Subtyping

Type checking for $\mathsf{MinML}_{<:}$, in either variant, clearly requires an algorithm for deciding subtyping: given σ and τ, 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 $\mathsf{MinML}_{<:}$, under the assumption that we can check the subtype relation.

Consider first the explicit variant of $\mathsf{MinML}_{<:}$. Since the typing rules are syntax directed, we can proceed as for MinML, with one additional case to consider. To check whether $(\sigma)\, e$ has type τ, we must check two things:

1. Whether e has type σ.
2. Whether $\sigma <: \tau$.

The former is handled by a recursive call to the type checker, the latter by a call to the subtype checker, which we assume given.

This discussion glosses over an important point. Even in pure MinML it is not possible to determine directly whether or not $\Gamma \vdash e : \tau$. For suppose that e is an application $e_1(e_2)$. To check whether $\Gamma \vdash e : \tau$, we must find the domain type of the function, e_1, against which we must check the type of the argument, e_2. To do this we define a *type synthesis* function that determines the unique (if it exists) type τ of an expression e in a context Γ, written $\Gamma \vdash e \Rightarrow \tau$. To check whether e has type τ, we synthesize the unique type for e and check that it is τ.

This methodology applies directly to $\mathsf{MinML}^{e}_{<:}$ by using the following rule to synthesize a type for a cast:

$$\frac{\Gamma \vdash e \Rightarrow \sigma \quad \sigma <: \tau}{\Gamma \vdash (\tau)\, e \Rightarrow \tau}$$

Extending this method to $\mathsf{MinML}^{i}_{<:}$ is a bit harder, because expressions no longer have unique types! The rule of subsumption allows us to weaken the type of an expression at will, yielding many different types for the same expression. A standard approach is define a type synthesis function that determines the *principal* type, rather than the *unique* type, of an expression in a given context. The principal type of an expression e in context Γ is the *least* type (in the subtyping pre-order) for e in Γ. 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 $\mathsf{MinML}^{i}_{<:}$ are as follows:

$$\frac{(\Gamma(x) = \tau)}{\Gamma \vdash x \Rightarrow \tau} \qquad \frac{}{\Gamma \vdash n \Rightarrow \mathtt{int}}$$

$$\frac{}{\Gamma \vdash \mathtt{true} \Rightarrow \mathtt{bool}} \qquad \frac{}{\Gamma \vdash \mathtt{false} \Rightarrow \mathtt{bool}}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \sigma_1 \quad \sigma_1 <: \tau_1 \quad \cdots \quad \Gamma \vdash e_n \Rightarrow \sigma_n \quad \sigma_n <: \tau_n}{\Gamma \vdash o(e_1, \ldots, e_n) \Rightarrow \tau}$$

where o is an n-ary primitive operation with arguments of type $\tau_1, \ldots, \tau_n$, and result type τ. We use subsumption to ensure that the argument types are subtypes of the required types.

$$\frac{\Gamma \vdash e \Rightarrow \sigma \quad \sigma <: \mathtt{bool} \quad \Gamma \vdash e_1 \Rightarrow \tau_1 \quad \tau_1 <: \tau \quad \Gamma \vdash e_2 \Rightarrow \tau_2 \quad \tau_2 <: \tau}{\Gamma \vdash \mathtt{if}\, e\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2 \Rightarrow \tau}$$

We use subsumption to ensure that the type of the test is a subtype of `bool`. Moreover, we rely on explicit specification of the type of the two clauses of the conditional.[4]

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

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_2 \to \tau \quad \Gamma \vdash e_2 \Rightarrow \sigma_2 \quad \sigma_2 <: \tau_2}{\Gamma \vdash e_1(e_2) \Rightarrow \tau}$$

We use subsumption to check that the argument type is a subtype of the domain type of the function.

Theorem 26.3

1. *If* $\Gamma \vdash e \Rightarrow \sigma$, *then* $\Gamma \vdash e : \sigma$.
2. *If* $\Gamma \vdash e : \tau$, *then there exists* σ *such that* $\Gamma \vdash e \Rightarrow \sigma$ *and* $\sigma <: \tau$.

Proof:

1. By a straightforward induction on the definition of the type synthesis relation.

[4] This may be avoided by requiring that the subtype relation have least upper bounds "whenever necessary"; we will not pursue this topic here.

2. By induction on the typing relation.

■

26.4 Implementation of Subtyping

26.4.1 Coercions

The dynamic semantics of subtyping sketched above suffices to ensure type safety, but is in most cases rather impractical. Specifically,

- Arithmetic subtyping relies on run-time type recognition and conversion.
- Tuple projection depends on the insensitivity of projection to the existence of components after the point of projection.
- Record field selection depends on being able to identify the lth 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 `int` to `float`. This means that type conversions are required exactly insofar as subsumption is used during type checking — a use of subsumption corresponds to a type conversion.

Since the subsumption rule is part of the static semantics, we can insert the appropriate conversions during type checking, and omit entirely the need to check for mixed-mode expressions during execution. This is called a *coercion interpretation* of subsumption. It is expressed formally by augmenting each subtype relation $\sigma <: \tau$ with a function value v of type $\sigma \rightarrow \tau$ (in pure MinML) that *coerces* values of type σ to values of type τ. The augmented subtype relation is written $\sigma <: \tau \rightsquigarrow v$.

Here are the rules for arithmetic subtyping augmented with coercions:

$$\frac{}{\tau <: \tau \rightsquigarrow \mathtt{id}_\tau} \qquad \frac{\rho <: \sigma \rightsquigarrow v \quad \sigma <: \tau \rightsquigarrow v'}{\rho <: \tau \rightsquigarrow v;v'}$$

$$\frac{}{\mathtt{int} <: \mathtt{float} \rightsquigarrow \mathtt{to\_float}} \qquad \frac{\tau_1 <: \sigma_1 \rightsquigarrow v_1 \quad \sigma_2 <: \tau_2 \rightsquigarrow v_2}{\sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2 \rightsquigarrow v_1 \to v_2}$$

These rules make use of the following auxiliary functions:

1. Primitive conversion: `to_float`.
2. Identity: $\mathtt{id}_\tau = \mathtt{fn}\, x : \tau\, \mathtt{in}\, x$.
3. Composition: $v;v' = \mathtt{fn}\, x : \tau\, \mathtt{in}\, v'(v(x))$.
4. Functions: $v_1 \to v_2 =$
   $\mathtt{fn}\, f : \sigma_1 \to \sigma_2\, \mathtt{in}\, \mathtt{fn}\, x : \tau_1\, \mathtt{in}\, v_2(f(v_1(x)))$.

The coercion interpretation is type correct. Moreover, there is at most one coercion between any two types:

Theorem 26.4

1. *If* $\sigma <: \tau \rightsquigarrow v$, *then* $\vdash^- v : \sigma \to \tau$.
2. *If* $\sigma <: \tau \rightsquigarrow v_1$ *and* $\sigma <: \tau \rightsquigarrow v_2$, *then* $\vdash^- v_1 \cong v_2 : \sigma \to \tau$.

Proof:

1. By a simple induction on the rules defining the augmented subtyping relation.
2. Follows from these equations:
   (a) Composition is associative with `id` as left- and right-unit element.
   (b) $\mathtt{id} \to \mathtt{id} \cong \mathtt{id}$.
   (c) $(v_1 \to v_2);(v_1' \to v_2') \cong (v_1';v_1) \to (v_2;v_2')$.

■

The type checking relation is augmented with a translation from $\mathsf{MinML}^{i}_{<:}$ to pure MinML that eliminates uses of subsumption by introducing coercions:

$$\frac{\Gamma \vdash e : \sigma \rightsquigarrow e' \quad \sigma <: \tau \rightsquigarrow v}{\Gamma \vdash e : \tau \rightsquigarrow v(e')}$$

The remaining rules simply commute with the translation. For example, the rule for function application becomes

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \rightsquigarrow e_1' \quad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow e_2'}{\Gamma \vdash e_1(e_2) : \tau \rightsquigarrow e_1'(e_2')}$$

Theorem 26.5

1. *If* $\Gamma \vdash e : \tau \rightsquigarrow e'$, *then* $\Gamma \vdash e' : \tau$ *in pure MinML.*
2. *If* $\Gamma \vdash e : \tau \rightsquigarrow e_1$ *and* $\Gamma \vdash e : \tau \rightsquigarrow e_2$, *then* $\Gamma \vdash e_1 \cong e_2 : \tau$ *in pure MinML.*
3. *If* $e : \texttt{int} \rightsquigarrow e'$ *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:

$$\frac{m > n}{\{l_1:\tau_1, \ldots, l_m:\tau_m\} <: \{l_1:\tau_1, \ldots, l_n:\tau_n\} \rightsquigarrow \mathtt{drop}_{m,n,l,\tau}}$$

$$\frac{\sigma_1 <: \tau_1 \rightsquigarrow v_1 \quad \ldots \quad \sigma_n <: \tau_n \rightsquigarrow v_n}{\{l_1:\sigma_1, \ldots, l_n:\sigma_n\} <: \{l_1:\tau_1, \ldots, l_n:\tau_n\} \rightsquigarrow \mathtt{copy}_{n,l,\sigma,v}}$$

These rules make use of the following coercion functions:

$$\mathtt{drop}_{m,n,l,\sigma} = \mathtt{fn}\, x{:}\{l_1:\sigma_1, \ldots, l_m:\sigma_m\}\, \mathtt{in}\, \{l_1:x.l_1, \ldots, l_n:x.l_n\}$$

$$\mathtt{copy}_{n,l,\sigma,v} = \mathtt{fn}\, x{:}\{l_1:\sigma_1, \ldots, l_n:\sigma_n\}\, \mathtt{in}\, \{l_1:v_1(x.l_1), \ldots, l_n:v_n(x.l_n)\}$$

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

$$\{l_1:\mathtt{int}, \ldots, l_n:\mathtt{int}\}*(\tau\, \mathtt{array}).$$

The field selection $l.e$ becomes a two-stage process:

$$\mathtt{snd}(e)[\mathtt{fst}(e).l]$$

Finally, coercions copy the view, without modifying the contents. If $\sigma = \{l_1:\sigma_1, \ldots, l_n:\sigma_n\}$ and $\tau = \{l_1:\mathtt{int}, \ldots, l_n:\mathtt{int}\}$, then

$$\mathtt{drop}_{m,n,l,\sigma} = \mathtt{fn}\, x\, \mathtt{in}\, (\mathtt{drop}_{m,n,l,\tau}(\mathtt{fst}(x)), \mathtt{snd}(x)).$$

Chapter 27

Inheritance and Subtyping in Java

In this note we discuss the closely-related, but conceptually distinct, notions of *inheritance*, or *subclassing*, and *subtyping* as exemplified in the Java language. Inheritance is a mechanism for supporting *code re-use* through incremental extension and modification. Subtyping is a mechanism for expressing *behavioral relationships* between types that allow values of a subtype to be provided whenever a value of a supertype is required.

In Java inheritance relationships give rise to subtype relationships, but not every subtype relationship arises via inheritance. Moreover, there are languages (including some extensions of Java) for which subclasses do not give rise to subtypes, and there are languages with no classes at all, but with a rich notion of subtyping. For these reasons it is best to keep a clear distinction between subclassing and subtyping.

27.1 Inheritance Mechanisms in Java

27.1.1 Classes and Instances

The fundamental unit of inheritance in Java is the *class*. A class consists of a collection of *fields* and a collection of *methods*. Fields are assignable variables; methods are procedures acting on these variables. Fields and methods can be either *static* (per-class) or *dynamic* (per-instance).[1] Static fields are per-class data. Static methods are just ordinary functions acting on static fields.

[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 `new` $C(e_1,\ldots,e_n)$, where C is a class and $e_1,\ldots,e_n$ are arguments to the constructor.[2] Static methods have access only to the static fields (and methods) of its class; dynamic methods have access to both the static and dynamic fields and methods of the class.

The components of a class have a designated *visibility* attribute, either `public`, `private`, or `protected`. The public components are those that are accessible by all clients of the class. Public static components are accessible to any client with access to the class. Public dynamic components are visible to any client of any instance of the class. Protected components are "semi-private; we'll have more to say about protected components later.

The components of a class also have a *finality* attribute. Final fields are not assignable — they are read-only attributes of the class or instance. Actually, final dynamic fields can be assigned exactly once, by a constructor of the class, to initialize their values. Final methods are of interest in connection with inheritance, to which we'll return below.

The components of a class have *types*. The type of a field is the type of its binding as a (possibly assignable) variable. The type of a method specifies the types of its arguments (if any) and the type of its results. The type of a constructor specifies the types of its arguments (if any); its "result type" is the instance type of the class itself, and may not be specified explicitly. (We will say more about the type structure of Java below.)

The public static fields and methods of a class C are accessed using "dot notation". If f is a static field of C, a client may refer to it by writing $C.f$. Similarly, if m is a static method of C, a client may invoke it by writing $C.m(e_1,\ldots,e_n)$, where $e_1,\ldots,e_n$ 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 c of a class C are similarly accessed using "dot notation", *albeit* from the instance, rather than the class. That is, if f is a public dynamic field of C, then $c.f$ refers

[2] Classes can have multiple constructors that are distinguished by overloading. We will not discuss overloading here.

to the f field of the instance c. Since distinct instances have distinct fields, there is no essential connection between $c.f$ and $c'.f$ when c and c' are 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.

27.1.2 Subclasses

A class may be defined by *inheriting* the visible fields and methods of another class. The new class is said to be a *subclass* of the old class, the *superclass*. Consequently, inheritance is sometimes known as *subclassing*. Java supports *single inheritance* — every class has at most one superclass. That is, one can only inherit from a single class; one cannot combine two classes by inheritance to form a third. In Java the subclass is said to `extend` the superclass.

There are two forms of inheritance available in Java:

1. *Enrichment*. The subclass enriches the superclass by providing additional fields and methods not present in the superclass.

2. *Overriding*. The subclass may re-define a method in the superclass by giving it a new implementation in the subclass.

Enrichment is a relatively innocuous aspect of inheritance. The true power of inheritance lies in the ability to override methods.

Overriding, which is also known as *method specialization*, is used to "specialize" the implementation of a superclass method to suit the needs of the subclass. This is particularly important when the other methods of the class invoke the overridden method by sending a message to `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 publically 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`,...,`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.

[3] Actually, Java assigns `protected` components "package scope", but since we are not discussing packages here, we will ignore this issue.

27.1.3 Abstract Classes and Interfaces

An *abstract class* is a class in which one or more methods are declared, but left unimplemented. Abstract methods may be invoked by the other methods of an abstract class by sending a message to `this`, but since their implementation is not provided, abstract classes do not themselves have instances. Instead the obligation is imposed on a subclass of the abstract class to provide implementations of the abstract methods to obtain a *concrete* class, which does have instances. Abstract classes are useful for setting up "code templates" that are instantiated by inheritance. The abstract class becomes the locus of code sharing for all concretions of that class, which inherit the shared code and provide the missing non-shared code.

Taking this idea to the extreme, an *interface* is a "fully abstract" class, which is to say that

- All its fields are `public static final` (*i.e.*, they are constants).
- All its methods are `abstract public`; they must be implemented by a subclass.

Since interfaces are a special form of abstract class, they have no instances.

The utility of interfaces stems from their role in `implements` declarations. As we mentioned above, a class may be declared to extend a *single* class to inherit from it.[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.

[4] Classes that do not specify a superclass implicitly extend the class `Object` of all objects.

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.

27.2 Subtyping in Java

The Java type system consists of the following types:

1. *Base types*, including `int`, `float`, `void`, and `boolean`.
2. *Class types* C, which classify the instances of a class C.
3. *Array types* of the form τ `[]`, where τ is a type, representing mutable arrays of values of type τ.

The basic types behave essentially as one would expect, based on previous experience with languages such as C and C++. Unlike C or C++ Java has true array types, with operations for creating and initializing an array and for accessing and assigning elements of an array. All array operations are safe in the sense that any attempt to exceed the bounds of the array results in a checked error at run-time.

Every class, whether abstract or concrete, including interfaces, has associated with it the type of its instances, called (oddly enough) the *instance type* of the class. Java blurs the distinction between the class as a program structure and the instance type determined by the class — class names serve not only to identify the class but also the instance type of that class. It may seem odd that abstract classes, and interfaces, all define instance types, even though they don't have instances. However, as will become clear below, even abstract classes have instances, indirectly through their concrete subclasses. Similarly, interfaces may be thought of as possessing instances, namely the instances of concrete classes that implement that interface.

27.2.1 Subtyping

To define the Java subtype relation we need two auxiliary relations. The *subclass* relation, $C \lhd C'$, 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, $C \lhd C'$ 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:

$$\frac{}{\tau <: \tau} \tag{27.1}$$

$$\frac{\tau <: \tau' \quad \tau' <: \tau''}{\tau <: \tau''} \tag{27.2}$$

Arrays are *covariant* type constructors, in the sense of this rule:

$$\frac{\tau <: \tau'}{\tau\,\texttt{[]} <: \tau'\,\texttt{[]}} \tag{27.3}$$

Inheritance implies subtyping:

$$\frac{C \lhd C'}{C <: C'} \tag{27.4}$$

Implementation implies subtyping:

$$\frac{C \blacktriangleleft I}{C <: I} \tag{27.5}$$

Every class is a subclass of the distinguished "root" class `Object`:

$$\frac{}{\tau <: \texttt{Object}} \tag{27.6}$$

The array subtyping rule is a structural subtyping principle — one need not explicitly declare subtyping relationships between array types for them to hold. On the other hand, the inheritance and implementation rules of subtyping are examples of nominal subtyping — they hold when they are declared to hold at the point of definition (or are implied by further subtyping relations).

27.2.2 Subsumption

The subsumption principle tells us that if e is an expression of type τ and $\tau <: \tau'$, then e is also an expression of type τ'. In particular, if a method is declared with a parameter of type τ, then it makes sense to provide an argument of any type τ' 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 τ, then it is legitimate to return a value of any subtype of τ.

This brings up an awkward issue in the Java type system. What should be the type of a conditional expression $e\,?\,e_1 : e_2$? Clearly e should have type `boolean`, and e_1 and e_2 should have the same type, since we cannot in general predict the outcome of the condition e. In the presence of subtyping, this amounts to the requirement that the types of e_1 and e_2 have an *upper bound* in the subtype ordering. To avoid assigning an excessively weak type, and to ensure that there is a unique choice of type for the conditional, it would make sense to assign the conditional the *least upper bound* of the types of e_1 and e_2. Unfortunately, two types need not have a least upper bound! For example, if an interface I extends incomparable interfaces K and L, and J extends both K and L, then I and J do not have a least upper bound — both K and L are upper bounds of both, but neither is smaller than the other. To deal with this Java imposes the rather *ad hoc* requirement that either the type of e_1 be a subtype of the type of e_2, or *vice versa*, to avoid the difficulty.

A more serious difficulty with the Java type system is that the array subtyping rule, which states that the array type constructor is *covariant* in the type of the array elements, violates the subsumption principle. To understand why, recall that we can do one of two things with an array: retrieve an element, or assign to an element. If $\tau <: \tau'$ and A is an array of type τ `[]`, then retrieving an element of A yields a value of type τ, which is by hypothesis an element of type τ'. 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 τ `[]`. Then A is also an array of type τ' `[]`, according to the Java rule for array subtyping. This means we can assign a value x of type τ' to an element of A. But this violates the assumption that A is an array of type τ `[]` — one of its elements is of type τ'.

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 run-

time type error ("gets stuck"). Java avoids this by a simple, but expensive, device — every array assignment incurs a "run-time type check" that ensures that the assignment does not create an unsafe situation. In the next subsection we explain how this is achieved.

27.2.3 Dynamic Dispatch

According to Java typing rules, if C is a sub-class of D, then C is a sub-type of D. Since the instances of a class C have type C, they also, by 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

[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.

Java *every single array assignment incurs a run-time check* whenever the array contains objects.[6]

27.2.4 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 `Object`, a general container is essentially a container whose elements are of type `Object`. This allows the container operations to be defined once for all element types. However, when retrieving an element from a container its static type is `Object`; we lost track of its dynamic type during type checking. If we wish to use such an object in any meaningful way, we must recover its dynamic type so that message sends are not rejected at compile time.

Java supports a safe form of *casting*, or *change of type*. A cast is written $(\tau)\, e$. The expression e is called the *subject* of the cast, and the type τ is the *target type* of the cast. The type of the cast is τ, provided that the cast makes sense, and its value is that of e. In general we cannot determine whether the cast makes sense until execution time, when the dynamic type of the expression is available for comparison with the target type. For example, every instance in Java has type `Object`, but its true type will usually be some class further down the type hierarchy. Therefore a cast applied to an expression of type `Object` cannot be validated until execution time.

Since the static type is an attenuated version of the dynamic type of an object, we can classify casts into three varieties:

1. *Up casts*, in which the static type of the expression is a subtype of the target type of the cast. The type checker accepts the cast, and no run-time check is required.

2. *Down casts*, in which the static type of the expression is a *supertype* of the target type. The true type may or may not be a subtype of the

[6] Arrays of integers and floats do not incur this overhead, because numbers are not objects.

target, so a run-time check is required.

3. *Stupid casts*, in which the static type of the expression rules out the possibility of its dynamic type matching the target of the cast. The cast is rejected.

Similar checks are performed to ensure that array assignments are safe.

Note that it is up to the programmer to maintain a sufficiently strong invariant to ensure that down casts do not fail. For example, if a container is intended to contain objects of a class C, then retrieved elements of that class will typically be down cast to a sub-class of C. It is entirely up to the programmer to ensure that these casts do not fail at execution time. That is, the programmer must maintain the invariant that the retrieved element really contains an instance of the target class of the cast.

27.3 Methodology

With this in hand we can (briefly) discuss the methodology of inheritance in object-oriented languages. As we just noted, in Java subclassing entails subtyping — the instance type of a subclass is a subtype of the instance type of the superclass. It is important to recognize that this is a methodological commitment to certain uses of inheritance.

Recall that a subtype relationship is intended to express a form of behavioral equivalence. This is expressed by the subsumption principle, which states that subtype values may be provided whenever a supertype value is required. In terms of a class hierarchy this means that a value of the subclass can be provided whenever a value of the superclass is required. For this to make good sense the values of the subclass should "behave properly" in superclass contexts — they should not be distinguishable from them.

But this isn't necessarily so! Since inheritance admits overriding of methods, we can make almost arbitrary[7] 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

[7] Limited only by finality declarations in the superclass.

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.[8]. 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.

[8]Nor is the type system of any other language that I am aware of, including ML

Part XI

Concurrency

Chapter 28

Concurrent ML

Part XII

Storage Management

Chapter 29

Storage Management

The dynamic semantics for MinML given in Chapter 9, and even the C-machine given in Chapter 11, ignore questions of storage management. In particular, all values, be they integers, booleans, functions, or tuples, are treated the same way. But this is unrealistic. Physical machines are capable of handling only rather "small" values, namely those that can fit into a word. Thus, while it is reasonable to treat, say, integers and booleans as values directly, it is unreasonable to do the same with "large" objects such as tuples or functions.

In this chapter we consider an extension of the C-machine to account for storage management. We proceed in two steps. First, we give an abstract machine, called the A-machine, that includes a *heap* for allocating "large" objects. This introduces the problem of *garbage*, storage that is allocated for values that are no longer needed by the program. This leads to a discussion of *automatic storage management*, or *garbage collection*, which allows us to reclaim unused storage in the heap.

29.1 The A Machine

The A-machine is defined for an extension of MinML in which we add an additional form of expression, a *location*, l, which will serve as a "reference" or "pointer" into the heap.

Values are classified into two categories, *small* and *large*, by the follow-

ing rules:

$$\frac{(l \in \mathit{Loc})}{l \text{ svalue}} \tag{29.1}$$

$$\frac{(n \in \mathbb{Z})}{n \text{ svalue}} \tag{29.2}$$

$$\frac{}{\texttt{true} \text{ svalue}} \tag{29.3}$$

$$\frac{}{\texttt{false} \text{ svalue}} \tag{29.4}$$

$$\frac{x \text{ var} \quad y \text{ var} \quad e \text{ expr}}{\texttt{fun}\, x\, (y : \tau_1) : \tau_2\, \texttt{is}\, e \text{ lvalue}} \tag{29.5}$$

A state of the A-machine has the form (H, k, e), where H is a *heap*, a finite function mapping locations to large values, k is a *control stack*, and e is an expression. A heap H is said to be *self-contained* iff $\mathrm{FL}(H) \subseteq \mathrm{dom}(H)$, where $\mathrm{FL}(H)$ is the set of locations occuring free in any location in H, and $\mathrm{dom}\, H$ is the domain of H.

Stack frames are similar to those of the C-machine, but refined to account for the distinction between small and large values.

$$\frac{e_2 \text{ expr}}{+(\square, e_2) \text{ frame}} \tag{29.6}$$

$$\frac{v_1 \text{ svalue}}{+(v_1, \square) \text{ frame}} \tag{29.7}$$

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

$$\frac{e_1 \text{ expr} \quad e_2 \text{ expr}}{\texttt{if}\, \square\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \text{ frame}} \tag{29.8}$$

$$\frac{e_2 \text{ expr}}{\texttt{apply}(\square, e_2) \text{ frame}} \tag{29.9}$$

$$\frac{v_1 \text{ svalue}}{\texttt{apply}(v_1, \square) \text{ frame}} \tag{29.10}$$

Notice that v_1 is required to be a *small* value; a function is represented by a location in the heap, which is small.

As with the C-machine, a stack is a sequence of frames:

$$\frac{}{\bullet\ \mathsf{stack}} \tag{29.11}$$

$$\frac{f\ \mathsf{frame} \quad k\ \mathsf{stack}}{f \triangleright k\ \mathsf{stack}} \tag{29.12}$$

The dynamic semantics of the A-machine is given by a set of rules defining the transition relation $(H, k, e) \mapsto_{\mathsf{A}} (H', k', e')$. The rules are similar to those for the C-machine, except for the treatment of functions.

Arithmetic expressions are handled as in the C-machine:

$$(H, k, +(e_1, e_2)) \mapsto_{\mathsf{A}} (H, +(\Box, e_2) \triangleright k, e_1) \tag{29.13}$$

$$(H, +(\Box, e_2) \triangleright k, v_1) \mapsto_{\mathsf{A}} (H, +(v_1, \Box) \triangleright k, e_2) \tag{29.14}$$

$$(H, +(n_1, \Box) \triangleright k, n_2) \mapsto_{\mathsf{A}} (H, k, n_1 + n_2) \tag{29.15}$$

Note that the heap is simply "along for the ride" in these rules.

Booleans are also handled similarly to the C-machine:

$$\begin{array}{c}(H, k, \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2) \\ \mapsto_{\mathsf{A}} \\ (H, \mathtt{if}\ \Box\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \triangleright k, e)\end{array} \tag{29.16}$$

$$(H, \mathtt{if}\ \Box\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \triangleright k, \mathtt{true}) \mapsto_{\mathsf{A}} (H, k, e_1) \tag{29.17}$$

$$(H, \mathtt{if}\ \Box\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \triangleright k, \mathtt{false}) \mapsto_{\mathsf{A}} (H, k, e_2) \tag{29.18}$$

Here again the heap plays no essential role.

The real difference between the C-machine and the A-machine is in the treatment of functions. A function expression is no longer a (small) value, but rather requires an execution step to allocate it on the heap.

$$\begin{array}{c}(H, k, \mathtt{fun}\ x\ (y : \tau_1) : \tau_2\ \mathtt{is}\ e) \\ \mapsto_{\mathsf{A}} \\ (H[l \mapsto \mathtt{fun}\ x\ (y : \tau_1) : \tau_2\ \mathtt{is}\ e], k, l)\end{array} \tag{29.19}$$

where l is chosen so that $l \notin \mathrm{dom}\, H$.

Evaluation of the function and argument position of an application is handled similarly to the C-machine.

$$(H, k, \mathtt{apply}(e_1, e_2)) \mapsto_{\mathsf{A}} (H, \mathtt{apply}(\Box, e_2) \triangleright k, e_1) \tag{29.20}$$

$$(H, \mathtt{apply}(\Box, e_2) \triangleright k, v_1) \mapsto_{\mathsf{A}} (H, \mathtt{apply}(v_1, \Box) \triangleright k, e_2) \tag{29.21}$$

Execution of a function call differs from the corresponding C-machine instruction in that the function must be retrieved from the heap in order to determine the appropriate instance of its body. Notice that the *location* of the function, and not the function itself, is substituted for the function variable!

$$\frac{v_1\ \mathsf{loc} \quad H(v_1) = \mathtt{fun}\, f\ (x{:}\tau_1){:}\tau_2\ \mathtt{is}\, e}{(H, \mathtt{apply}(v_1, \Box) \triangleright k, v_2) \mapsto_{\mathsf{A}} (H, k, \{v_1, v_2/f, x\}e)} \tag{29.22}$$

The A-machine preserves self-containment of the heap. This follows from observing that whenever a location is allocated, it is immediately given a binding in the heap, and that the bindings of heap locations are simply those functions that are encountered during evaluation.

Lemma 29.1
If H is self-contained and $(H, k, e) \mapsto_{\mathsf{A}} (H', k', e')$, then H' is also self-contained. Moreover, if $\mathrm{FL}(k) \cup \mathrm{FL}(e) \subseteq \mathrm{dom}\, H$, *then* $\mathrm{FL}(k') \cup \mathrm{FL}(e') \subseteq \mathrm{dom}\, H'$.

It is not too difficult to see that the A-machine and the C-machine have the same "observable behavior" in the sense that both machines determine the same value for closed expressions of integer type. However, it is somewhat technically involved to develop a precise correspondence. The main idea is to define the *heap expansion* of an A-machine state to be the C-machine state obtained by replacing all locations in the stack and expression by their values in the heap. (It is important to take care that the locations occurring in a value stored are themselves replaced by their values in the heap!) We then prove that an A-machine state reaches a final

state in accordance with the transition rules of the A-machines iff its expansion does in accordance with the rules of the C-machine. Finally, we observe that the value of a final state of integer type is the same for both machines.

Formally, let $\widehat{H}(e)$ stand for the substitution

$$\{H(l_1), \ldots, H(l_n)/l_1, \ldots, l_n\}e,$$

where $\mathrm{dom}\, H = \{ l_1, \ldots, l_n \}$. Similarly, let $\widehat{H}(k)$ denote the result of performing this substitution on every expression occurring in the stack k.

Theorem 29.2
If $(H, k, e) \mapsto_{\mathsf{A}} (H', k', e')$, *then* $(\widehat{H}(k), \widehat{H}(e)) \mapsto_{\mathsf{C}}^{0,1} (\widehat{H'}(k'), \widehat{H'}(e'))$.

Notice that the allocation of a function in the A-machine corresponds to zero steps of execution on the C-machine, because in the latter case functions are values.

29.2 Garbage Collection

The purpose of the A-machine is to model the memory allocation that would be required in an implementation of MinML. This raises the question of *garbage*, storage that is no longer necessary for a computation to complete. The purpose of a *garbage collector* is to reclaim such storage for further use. Of course, in a purely abstract model there is no reason to perform garbage collection, but in practice we must contend with the limitations of finite, physical computers. For this reason we give a formal treatment of garbage collection for the A-machine.

The crucial issue for any garbage collector is to determine which locations are unnecessary for computation to complete. These are deemed garbage, and are reclaimed so as to conserve memory. But when is a location unnecessary for a computation to complete? Consider the A-machine state (H, k, e). A location $l \in \mathrm{dom}(H)$ is *unnecessary*, or *irrelevant*, for this machine state iff execution can be completed without referring to the contents of l. That is, $l \in \mathrm{dom}\, H$ is unnecessary iff $(H, k, e) \mapsto_{\mathsf{A}}^{*} (H', \bullet, v)$ iff $(H_l, k, e) \mapsto_{\mathsf{A}}^{*} (H'', \bullet, v)$, where H_l is H with the binding for l removed, and H'' is some heap.

Unfortunately, a machine cannot decide whether a location is unnecessary!

Theorem 29.3
It is mechanically undecidable whether or not a location l is unnecessary for a given state of the A*-machine.*

Intuitively, we cannot decide whether l is necessary without actually running the program. It is not hard to formulate a reduction from the halting problem to prove this theorem: simply arrange that l is used to complete a computation iff some given Turing machine diverges on blank input.

Given this fundamental limitation, practical garbage collectors must employ a *conservative approximation* to determine which locations are unnecessary in a given machine state. The most popular criterion is based on *reachability*. A location l_n is *unreachable*, or *inaccessible*, iff there is no sequence of locations $l_1, \ldots, l_n$ such that l_1 occurs in either the current expression or on the control stack, and l_i occurs in l_{i+1} for each $1 \leq i < n$.

Theorem 29.4
If a location l is unreachable in a state (H, k, e), then it is also unnecessary for that state.

Each transition depends only on the locations occurring on the control stack or in the current expression. Some steps move values from the heap onto the stack or current expression. Therefore in a multi-step sequence, execution can depend only on reachable locations in the sense of the definition above.

The set of unreachable locations in a state may be determined by *tracing*. This is easily achieved by an iterative process that maintains a finite set of of locations, called the *roots*, containing the locations that have been found to be reachable up to that point in the trace. The root set is initialized to the locations occurring in the expression and control stack. The tracing process completes when no more locations can be added. Having found the reachable locations for a given state, we then deem all other heap locations to be unreachable, and hence unnecessary for computation to proceed. For this reason the reachable locations are said to be *live*, and the unreachable are said to be *dead*.

Essentially *all* garbage collectors used in practice work by tracing. But since reachability is only a conservative approximation of necessity, *all practical collectors are conservative*! So-called conservative collectors are, in fact, *incorrect* collectors that may deem as garbage storage that is actually

necessary for the computation to proceed. Calling such a collector "conservative" is misleading (actually, wrong), but it is nevertheless common practice in the literature.

The job of a garbage collector is to dispose of the unreachable locations in the heap, freeing up memory for later use. In an abstract setting where we allow for heaps of unbounded size, it is never necessary to collect garbage, but of course in practical situations we cannot afford to waste unlimited amounts of storage. We will present an abstract model of a particular form of garbage collection, called *copying collection*, that is widely used in practice. The goal is to present the main ideas of copying collection, and to prove that garbage collection is semantically "invisible" in the sense that it does not change the outcome of execution.

The main idea of copying collection is to simultaneously determine which locations are reachable, and to arrange that the contents of all reachable locations are preserved. The rest are deemed garbage, and are reclaimed. In a copying collector this is achieved by partitioning storage into two parts, called *semi-spaces*. During normal execution allocation occurs in one of the two semi-spaces until it is completely filled, at which point the collector is invoked. The collector proceeds by copying all reachable storage from the current, filled semi-space, called the *from space*, to the other semi-space, called the *to space*. Once this is accomplished, execution continues using the "to space" as the new heap, and the old "from space" is reclaimed in bulk. This exchange of roles is called a *flip*.

By copying all and only the reachable locations the collector ensures that unreachable locations are reclaimed, and that no reachable locations are lost. Since reachability is a conservative criterion, the collector may preserve more storage than is strictly necessary, but, in view of the fundamental undecidability of necessity, this is the price we pay for mechanical collection. Another important property of copying collectors is that their execution time is proportion to the size of the live data; no work is expended manipulating reclaimable storage. This is the fundamental motivation for using semi-spaces: once the reachable locations have been copied, the unreachable ones are eliminated by the simple measure of "flipping" the roles of the spaces. Since the amount of work performed is proportional to the live data, we can amortize the cost of collection across the allocation of the live storage, so that garbage collection is (asymptotically) "free". However, this benefit comes at the cost of using only half

of available memory at any time, thereby doubling the overall storage required.

Copying garbage collection may be formalized as an abstract machine with states of the form (H_f, S, H_t), where H_f is the " from" space, H_t is the "to" space, and S is the *scan set*, the set of reachable locations. The initial state of the collector is $(H, S, \emptyset)$, where H is the "current" heap and $\emptyset \neq S \subseteq \mathrm{dom}(H_f)$ is the set of locations occurring in the program or control stack. The final state of the collector is $(H_f, \emptyset, H_t)$, with an empty scan set.

The collector is invoked by adding the following instruction to the A-machine:

$$\frac{(H, \mathrm{FL}(k) \cup \mathrm{FL}(e), \emptyset) \mapsto_{\mathsf{G}}^{*} (H'', \emptyset, H')}{(H, k, e) \mapsto_{\mathsf{A}} (H', k, e)} \tag{29.23}$$

The scan set is initialized to the set of free locations occurring in either the current stack or the current expression. These are the locations that are immediately reachable in that state; the collector will determine those that are transitively reachable, and preserve their bindings. Once the collector has finished, the "to" space is installed as the new heap.

Note that a garbage collection can be performed at any time! This correctly models the unpredictability of collection in an implementation, but avoids specifying the exact criteria under which the collector is invoked. As mentioned earlier, this is typically because the current heap is exhausted, but in an abstract setting we impose no fixed limit on heap sizes, preferring instead to simply allow collection to be performed spontaneously according to unspecified criteria.

The collection machine is defined by the following two rules:

$$\frac{}{(H_f[l = v], S \cup \{\, l \,\}, H_t) \mapsto_{\mathsf{G}} (H_f, S \cup \mathrm{FL}(v), H_t[l = v])} \tag{29.24}$$

$$\frac{}{(H_f, S \cup \{\, l \,\}, H_t[l = v]) \mapsto_{\mathsf{G}} (H_f, S, H_t[l = v])} \tag{29.25}$$

The first rule copies a reachable binding in the "from" space to the "to" space, and extends the scan set to include those locations occurring in the copied value. This ensures that we will correctly preserve those locations that occur in a reachable location. The second rule throws away any location in the scan set that has already been copied. This rule is necessary because when the scan set is updated by the free locations of a heap value,

we may add locations that have already been copied, and we do not want to copy them twice!

The collector is governed by a number of important invariants.

1. The scan set contains only "valid" locations: $S \subseteq \mathrm{dom}\, H_f \cup \mathrm{dom}\, H_t$;
2. The "from" and "to" space are disjoint: $\mathrm{dom}\, H_f \cap \mathrm{dom}\, H_t = \emptyset$;
3. Every location in "to" space is either in "to" space, or in the scan set: $\mathrm{FL}(H_t) \subseteq S \cup \mathrm{dom}\, H_t$;
4. Every location in "from" space is either in "from" or "to" space: $\mathrm{FL}(H_f) \subseteq \mathrm{dom}\, H_f \cup \mathrm{dom}\, H_t$.

The first two invariants are minimal "sanity" conditions; the second two are crucial to the operation of the collector. The third states that the "to" space contains only locations that are either already copied into "to" space, or will eventually be copied, because they are in the scan set, and hence in "from" space (by disjointness). The fourth states that locations in "from" space contain only locations that either have already been copied or are yet to be copied.

These invariants are easily seen to hold of the initial state of the collector, since the "to" space is empty, and the "from" space is assumed to be self-contained. Moreover, if these invariants hold of a final state, then $\mathrm{FL}(H_t) \subseteq \mathrm{dom}\, H_t$, since $S = \emptyset$ in that case. Thus the heap remains self-contained after collection.

Theorem 29.5 (Preservation of Invariants)
If the collector invariants hold of (H_f, S, H_t) *and* $(H_f, S, H_t) \mapsto_{\mathsf{G}} (H'_f, S', H'_t)$, *then the same invariants hold of* (H'_f, S', H'_t).

The correctness of the collector follows from the following lemma.

Lemma 29.6
If $(H_f, S, H_t) \mapsto_{\mathsf{G}} (H'_f, S', H'_t)$, *then* $H_f \cup H_t = H'_f \cup H'_t$ *and* $S \cup \mathrm{dom}\, H_t \subseteq S' \cup \mathrm{dom}\, H'_t$.

The first property states that the union of the semi-spaces never changes; bindings are only copied from one to the other. The second property states that the domain of the "to" space together with the scan set does not change.

From this lemma we obtain the following crucial facts about the collector. Let $S = \mathrm{FL}(k) \cup \mathrm{FL}(e)$, and suppose that

$$(H, S, \emptyset) \mapsto_{\mathsf{G}}^{*} (H'', \emptyset, H').$$

Then we have the following properties:

1. The reachable locations are bound in H': $\mathrm{FL}(k) \cup \mathrm{FL}(e) \subseteq \mathrm{dom}\, H'$. This follows from the lemma, since the inital "to" space and the final scan set are empty.

2. The reachable data is correctly copied: $H' \subseteq H$. This follows from the lemma, which yields $H = H'' \cup H'$.

Bibliography