Programming Languages: Theory and Practice

(Working Draft of February 4, 2006.)

Robert Harper
Carnegie Mellon University

Spring Semester, 2006
This is a collection of lecture notes for Computer Science 15–312 *Programming Languages*. I am grateful to Andrew Appel and Frank Pfenning for their advice, suggestions, and numerous examples. I am also grateful 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. Enjoy!
Contents

Preface iii

I Preliminaries 1

1 Inductive Definitions 3
  1.1 Judgements ........................................... 3
  1.2 Rules and Derivations ................................. 4
  1.3 Rule Induction ......................................... 6
  1.4 Iterated and Simultaneous Inductive Definitions .... 8
  1.5 Admissible and Derivable Rules ....................... 9
  1.6 Defining Functions by Rules ......................... 11
  1.7 Foundations .......................................... 12
  1.8 Exercises ............................................ 13

2 Higher Order Judgement Forms 15
  2.1 Hypothetical Judgements ............................... 15
  2.2 General Judgements ................................... 18
  2.3 Hypothetico-General Judgements ....................... 19
  2.4 Inductive Definitions, Revisited ..................... 20
  2.5 Exercises ............................................ 23

3 Transition Systems 25
  3.1 Transition Systems .................................... 25
  3.2 Exercises ............................................ 26

II Levels of Syntax 27

4 Concrete Syntax 29
## CONTENTS

4.1 Strings .................................................. 29
4.2 Context-Free Grammars ................................. 30
4.3 Ambiguity ................................................. 31
4.4 Resolving Ambiguity .................................... 33
4.5 Exercises ................................................ 34

5 Abstract Syntax Trees .................................... 35
  5.1 Abstract Syntax Trees ................................... 36
  5.2 Structural Induction .................................... 37
  5.3 Parsing ................................................ 38
  5.4 Exercises ................................................ 40

6 Abstract Binding Trees ................................... 41
  6.1 Names ................................................... 41
  6.2 Abstract Syntax With Names ........................... 42
  6.3 Abstract Binding Trees ................................ 42
  6.4 Renaming ................................................. 44
  6.5 Structural Induction With Binding and Scope ...... 46
  6.6 Substitution ............................................. 47
  6.7 Summary ................................................ 47
  6.8 Exercises ................................................ 48

7 Specifying Syntax .......................................... 49

III Static and Dynamic Semantics ......................... 51

8 The Phase Distinction ..................................... 53

9 Static Semantics ............................................ 57
  9.1 Static Semantics of Expressions ....................... 57
  9.2 Properties of the Static Semantics ................... 58
  9.3 Exercises ................................................ 59

10 Dynamic Semantics ....................................... 61
  10.1 Structured Operational Semantics .................... 61
  10.2 Evaluation Semantics .................................. 63
  10.3 Relating Transition and Evaluation Semantics ...... 64
  10.4 Environment Semantics ................................. 66
  10.5 Exercises ................................................ 67

WORKING DRAFT ................................. FEBRUARY 4, 2006
CONTENTS

17 Recursive Types 109
  17.1 Recursive Types .................................................. 109
  17.2 Unbounded Data Structures ..................................... 111
  17.3 Un(i)typed Languages ............................................ 112
  17.4 Exercises ............................................................. 114

18 Pattern Compilation 115

VI Control Flow 117

19 Abstract Machines 119
  19.1 The C Machine ....................................................... 120
  19.2 Correctness of the C Machine ................................. 123
    19.2.1 Proof of Completeness ...................................... 125
    19.2.2 Proof of Soundness .......................................... 126
  19.3 The E Machine ....................................................... 126
  19.4 Exercises ............................................................. 130

20 Continuations 131
  20.1 Informal Overview ................................................ 132
  20.2 Semantics of Continuations ..................................... 135
  20.3 Exercises ............................................................. 136

21 Exceptions 139
  21.1 Exercises ............................................................. 144

22 Coroutines and Cooperative Threads 147
  22.1 Coroutines ............................................................ 147
  22.2 Cooperative Threads ............................................... 149
  22.3 Exercises ............................................................. 152

VII Polymorphism and Data Abstraction 153

23 Polymorphism 155
  23.1 A Polymorphic Language ......................................... 156
  23.2 ML-style Type Inference .......................................... 162

WORKING DRAFT  FEBRUARY 4, 2006
## CONTENTS

35.1.3 Abstract Classes and Interfaces ............................................ 264
35.2 Subtyping in Java ................................................................. 266
  35.2.1 Subtyping ........................................................................ 266
  35.2.2 Subsumption .................................................................... 267
  35.2.3 Dynamic Dispatch ............................................................ 268
  35.2.4 Casting ............................................................................. 269
35.3 Methodology .......................................................... 271

XIII Program Equivalence ...................................................... 273

36 Functional Equivalence ..................................................... 275
  36.1 Expression Equivalence ..................................................... 276
  36.2 Some Laws of Equivalence ................................................. 278
    36.2.1 General Laws ............................................................ 278
    36.2.2 Symbolic Evaluation .................................................. 279
    36.2.3 Extensionality ............................................................ 280
    36.2.4 Strictness Properties .................................................. 280
    36.2.5 Arithmetic Laws ......................................................... 281
    36.2.6 Products ..................................................................... 281
    36.2.7 Lists .......................................................................... 282

37 Parametricity ........................................................................ 285
  37.1 Informal Overview ............................................................. 286
  37.2 Relational Parametricity ..................................................... 289

38 Representation Independence .............................................. 293

XIV Concurrency ....................................................................... 297

39 Process Calculus ............................................................. 299

40 Concurrent ML ............................................................... 301

XV Dynamic Typing .............................................................. 303

41 Dynamic Typing ............................................................... 305
  41.1 Dynamic Typing ................................................................. 307
  41.2 Implementing Dynamic Typing .......................................... 308

February 4, 2006 Working Draft
41.3 Dynamic Typing as Static Typing ............................. 310

XVI Storage Management ................................. 313

42 Storage Management ........................................ 315
  42.1 The A Machine ........................................ 315
  42.2 Garbage Collection ...................................... 319
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 Judgements

We start with the notion of a judgement, or assertion, about one or more objects of study. In general a judgement is a statement of knowledge, including assertions such as “it is raining outside”, “every natural number may be written as a product of primes”, or “the sum of 2 and 2 is 4.” In the study of programming languages we shall make use of many forms of judgement, including the following (informal) examples:

- $t$ ast $e$ is an abstract syntax tree
- $\tau$ type $\tau$ is a type
- $e : \tau$ expression $e$ has type $\tau$
- $e \Downarrow v$ expression $e$ has value $v$

Each of these forms of judgement makes an assertion about one or more objects.\(^1\)

The notation for writing judgements varies according to the situation, but when discussing the general case we use postfix notation, writing $x_1, \ldots, x_n \ J$, or, briefly, $\vec{x} \ J$, to assert that $J$ holds of the objects $\vec{x} = x_1, \ldots, x_n$. The judgement $\vec{x} \ J$ is sometimes called an instance of the judgement form $J$.

\(^1\)For the time being we do not specify what sort of objects may be the subject of a judgement, but see Section 1.7 for more on this.
1.2 Rules and Derivations

In the study of programming languages we make frequent use of inductive definitions of judgement forms. An inductive definition of an \( n \)-ary judgement form, \( J \), consists of a set of inference rules of the form

\[
\frac{\vec{x}_1 J \quad \cdots \quad \vec{x}_k J}{\vec{x} J}
\]

where \( \vec{x} \) and each \( \vec{x}_1 \ldots, \vec{x}_k \) are \( n \)-tuples of objects, and \( J \) is the judgement 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.

An inference rule may be read as an implication stating that if \( \vec{x}_1 J \) and \( \ldots \) and \( \vec{x}_k J \) are all valid, then \( \vec{x} J \) is valid as well. Thus each inference rule specifies that the premises are a sufficient condition for the conclusion: to show \( \vec{x} J \), it suffices to show that \( \vec{x}_1 J, \ldots, \vec{x}_k J \) hold. Notice that when \( k = 0 \) (i.e., when the rule is an axiom), nothing is required for the conclusion to hold — it must hold under all circumstances.

A set of rules may be regarded as an inductive definition of the judgement form, \( J \), by insisting that the rules are necessary, as well as sufficient, conditions for the validity of instances of \( J \). This is to say that \( \vec{x} J \) is valid only if there is some rule with that conclusion all of whose premises are also valid. The judgement, \( J \), inductively defined by the given set of rules is the strongest, or most restrictive, judgements form satisfying the given rules.

An example will help clarify these ideas. The following set, \( \mathcal{N} \), of rules constitute an inductive definition of the judgement form \( \text{nat} \).

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

According to these rules the judgement \( x \text{ nat} \) is defined to hold exactly when either \( x \) is \( \text{zero} \), or \( x \) is \( \text{succ}(y) \) for some \( y \) such that \( y \text{ nat} \). In other words, \( x \text{ nat} \) holds iff \( x \) is (the unary form of) a natural number.

Similarly, the following set, \( \mathcal{T} \), of rules constitute an inductive definition of the judgement form \( \text{tree} \):

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

\[
\frac{x \text{ tree} \quad y \text{ tree}}{x y \text{ tree}}
\]

April 4, 2006
According to these rules the judgement $x$ tree holds exactly when $x$ is a binary tree, either the empty tree, empty, or a node, node$(x_1, x_2)$, with children $x_1$ and $x_2$ such that $x_1$ tree and $x_2$ nat.

It is worth noting that the inductive definitions of natural numbers and trees each have infinitely many rules, not just two. In each case the variables $x$ and $y$ range over the universe of objects over which the definition takes place. What we have specified is a rule scheme with metavariables $x$ and $y$. A rule scheme stands for infinitely many rules, one for each choice of object for each of the metavariables involved. This raises the question of what is the universe of objects over which we are working — just which objects are permissible in an inductive definition? For now we rely on informal intuition, but see Section 1.7 for further discussion.

To show that an instance of an inductively defined judgement form is valid, it is enough to exhibit a derivation (tree) of it consisting of a composition of rules, starting from axioms and ending with that judgement. Derivations have a natural tree structure arising from stacking rules on top of the another. If

$$
\frac{x_1 \ J \ \cdots \ x_k \ J}{\hat{x} \ J}
$$

is an inference rule and $D_1, \ldots, D_k$ are derivations of its premises, then

$$
\frac{D_1 \ \cdots \ D_k}{\hat{x} \ J}
$$

is a derivation of its conclusion.

For example, here is a derivation of $\text{succ} \left( \text{succ} \left( \text{succ} \left( \text{zero} \right) \right) \right)$ nat according to the set of rules $\mathcal{N}$:

$$
\frac{\text{zero} \ \text{nat}}{\text{succ}(\text{zero}) \ \text{nat}}
\frac{\text{succ}(\text{zero}) \ \text{nat}}{\text{succ}(\text{succ}(\text{zero})) \ \text{nat}}
\frac{\text{succ}(\text{succ}(\text{zero})) \ \text{nat}}{\text{succ}(\text{succ}(\text{succ}(\text{zero}))) \ \text{nat}}
$$

Similarly, here is a derivation that $\text{node} \left( \text{node} \left( \text{empty}, \text{empty} \right), \text{empty} \right)$ tree according to the set of rules $\mathcal{T}$:

$$
\frac{\text{empty} \ \text{tree} \ \text{empty} \ \text{tree}}{\text{node} \left( \text{empty}, \text{empty} \right) \ \text{tree}}
\frac{\text{node} \left( \text{empty}, \text{empty} \right) \ \text{tree} \ \text{empty} \ \text{tree}}{\text{node} \left( \text{node} \left( \text{empty}, \text{empty} \right), \text{empty} \right) \ \text{tree}}
$$
In general, 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, or bottom-up construction, and backward chaining, or top-down construction. 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 Rule Induction

Suppose that the judgement form, \( J \), is inductively defined by a rule set \( S \). Since \( J \) is, by definition, the strongest (most restrictive) judgement for which the rules in \( S \) are valid, we may employ the important principle of rule induction to derive properties of those objects \( \vec{x} \) such that \( \vec{x} \vdash J \) is valid. Specifically, if we wish to show \( P \vec{x} \), it is enough to show that the property
1.3 Rule Induction

$P$ is closed under, or respects, the rules in $S$. More precisely, for every rule in $S$ of the form

$$
\frac{x_1 J \quad \cdots \quad x_k J}{\bar{x} J}
$$

we must show that $P \bar{x}$ is valid, under the assumptions that $P x_1, \ldots, P x_k$ are all valid. These assumptions are called the inductive assumptions of the inference, and the conclusion establishes the inductive step corresponding to that rule. Remember that we must consider all rules in the definition of $J$ in order to establish the desired conclusion!

Another way to justify the principle of rule induction is by analysis of the possible derivations of $\bar{x} J$ according to the rules. This amounts to a case-by-case analysis on the root of the derivation tree. For each rule that could occur at the root, we may inductively assume the result for each of the sub-derivations, and derive from these the result for the whole derivation. This process is sometimes known as induction on derivations; it is entirely equivalent to rule induction over the rules defining the judgement form under consideration.

The principle of rule induction associated with the rule set, $\mathcal{N}$, states that to show $P \; \bar{x}$ whenever $\bar{x}$ nat, it is enough to show

1. $P \; \text{zero}$.
2. $P \; \text{succ}(x)$, assuming $P \; x$.

This is just the familiar principle of mathematical induction. Similarly, the principle of rule induction associated with the rule set $\mathcal{T}$ states that to show that $x$ tree implies $P \; x$, it is enough to show

1. $P \; \text{empty}$.
2. $P \; \text{node}(x_1, x_2)$, assuming $P \; x_1$ and $P \; x_2$.

This is sometimes called the principle of tree induction.

As a simple example, let us show that every natural number is either zero, one, or two more than some other natural number. More precisely, the judgement $P \; x$ in this case states that either $x = \text{zero}$, or $x = \text{succ}(\text{zero})$, or $x = \text{succ}\left(\text{succ}(y)\right)$ for some $y$ nat. We prove by rule induction on the rule set $\mathcal{N}$ that if $x$ nat, then $P \; x$, as follows:

1. Show that $P \; \text{zero}$. This is immediate.
2. Assume that \( P \) and show that \( P \). We have by the inductive assumption that \( x = zero \), \( x = succ(zero) \), or \( x = succ(succ(y)) \) for some \( y \). We are to show that either \( succ(x) = zero \), or \( succ(x) = succ(zero) \), or there exists \( z \) such that \( succ(x) = succ(succ(z)) \). In the first case, observe that \( succ(x) = succ(zero) \); in the second, observe that \( succ(x) = succ(succ(zero)) \), so we may take \( z = zero \); in the third, observe that \( succ(x) = succ(succ(succ(y))) \), so we may take \( z = succ(y) \).

This completes the proof.

1.4 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, \( L \), defines the judgement form \( list \), which expresses that an object is a list of natural numbers:

\[
\begin{array}{c}
\text{nil list} \\
\text{cons(x, y) list}
\end{array}
\]

Notice that the second rule uses the judgement form \( nat \) defined earlier.

It is also common to give a simultaneous inductive definition of several judgements, \( J_1, \ldots, J_n \), by a collection of rules, each of which may use as premises any of the \( n \) judgement forms being defined. Thus each rule has the form

\[
\frac{\vec{x}_1 J_{i_1} \cdots \vec{x}_m J_{i_m}}{\vec{x} J_i}
\]

where \( 1 \leq i_j \leq n \) for each \( 1 \leq j \leq m \). As before, we may have \( m = 0 \) premises, in which case the rule is an axiom.

The difference between simultaneous and iterated inductive definitions is simply that in the iterated case we finish one inductive definition and then use it in a subsequent one, whereas in the simultaneous case two or more judgements are being defined in a mutually recursive manner, so that neither can be considered “finished” before the other.

The meaning of a simultaneous inductive definition is a generalization of that for a single inductive definition. It defines the strongest judgements \( J_1, \ldots, J_n \) closed under the rules. This gives rise to the principle of rule induction for simultaneous inductive definitions, which permits us to prove properties about such inductively defined families of judgements. If we
1.5 Admissible and Derivable Rules

Wish to show \( P_{i_1}, \ldots, P_{i_n} \) whenever \( \vec{x}_1 J_{i_1}, \ldots, \vec{x}_n J_{i_n} \), it is enough to show for each rule

\[
\frac{\vec{x}_1 J_{i_1}, \ldots, \vec{x}_m J_{i_m}}{\vec{x} J_i},
\]

that if \( \vec{x}_1 P_{i_1}, \ldots, \vec{x}_m P_{i_m} \), then \( \vec{x} P_i \).

For example, consider the following rule set, which constitutes a simultaneous inductive definition of the judgement forms \( x \) even, stating that \( x \) is an even natural number, and \( x \) odd, stating that \( x \) is an odd natural number:

\[
\begin{array}{ccc}
\text{zero} & \text{even} & \text{odd} \\
\text{succ}(x) & \text{even} & \text{odd}
\end{array}
\]

The associated principle of rule induction states that if we wish to show \( P x \), whenever \( x \) even, and \( Q x \), whenever \( x \) odd, it is enough to show

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

These proof obligations are derived by considering the rules defining the even and odd judgement forms.

1.5 Admissible and Derivable Rules

Let \( S \) be an inductive definition of the judgement \( J \). There are two senses in which a rule

\[
\frac{\vec{x}_1 J, \ldots, \vec{x}_k J}{\vec{x} J}
\]

may be thought of as being “valid” for \( S \): 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 \( \mathcal{N} \):

\[
\frac{x \text{ nat}}{\text{succ(succ(succ(x)))) \text{ nat}}}
\]
1.5 Admissible and Derivable Rules

Its derivation is as follows:

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

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

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

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

If a rule is derivable in a rule set \( S \), then it remains derivable in any rule set \( S' \supseteq S \). 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 \( S \), but inadmissible in some extension \( S' \supseteq S' \)! For example, suppose that we add to \( \mathcal{N} \) the rule

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

Now it is no longer the case that the rule

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

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

Since admissibility is sensitive to which rules are \textit{absent}, as well as to which are \textit{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).

1.6 Defining Functions by Rules

A common use of inductive definitions is to define inductively its graph, a judgement, which we then prove is a function. For example, one way to define the addition function on natural numbers is to define inductively the judgement \( A(m, n, p) \), with the intended meaning that \( p \) is the sum of \( m \) and \( n \), as follows:

\[
\frac{m \text{ nat}}{A(m, \text{zero}, m)} \quad \frac{A(m, n, p)}{A(m, \text{succ}(n), \text{succ}(p))}
\]

We then must show that \( p \) is uniquely determined as a function of \( m \) and \( n \). That is, we show that if \( m \text{ nat} \) and \( n \text{ nat} \), then there exists a unique \( p \) such that \( A(m, n, p) \) by rule induction on the rules defining the natural numbers.

1. From \( m \text{ nat} \) and \( \text{zero} \text{ nat} \), 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 \( m \text{ nat} \) and \( \text{succ}(n) \text{ nat} \) 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 \( A(m, \text{succ}(n), q) \). Taking \( q = \text{succ}(p) \) does the job.

This fact may be summarized by saying that the mode of this judgement is \((\forall, \forall, \exists!\))\), which means that the first two arguments may be thought of as inputs, and the third as a uniquely determined output. (It is implicit that the inputs and outputs are objects \( x \) such that \( x \text{ nat} \).) This mode specification states that \( A(m, n, p) \) determines a total function in which each pair \( m \text{ nat} \) and \( n \text{ nat} \) determines a unique \( p \text{ nat} \) such that \( A(m, n, p) \). Other modes for this judgement are \((\forall, \forall, \exists)\), which merely asserts that to every pair of inputs there is a (not necessarily unique) output, and \((\forall, \forall, \exists^{\leq 1})\), which asserts that to every pair of inputs there is at most one output. The former states that the judgement is a total relation, the latter that it is a partial function. Of course the property of being a total function is stronger than
either of these, but situations will arise in which only the weaker modes are available.

As another example, the following rules define the height of a binary tree, making use of an auxiliary "maximum" function on natural numbers that you may readily define yourself:

\[
\begin{align*}
H(\text{empty}, \text{zero}) & \quad H(t_1, n_1) \quad H(t_2, n_2) \quad M(n_1, n_2, n) \\
& \quad H(\text{node}(t_1, t_2), \text{succ}(n))
\end{align*}
\]

One may readily show by tree induction that the mode of this judgement is \((\forall, \exists)\) over inputs and outputs \(x\) such that \(x\) tree.

Whenever we are defining a judgement that is intended to be a function (i.e., one argument is determined as a function of the others), we often write the definition using equations. For example, we may re-state the inductive definition of addition above using equations as follows:

\[
\begin{align*}
m \text{ nat} & \quad m + \text{ zero} = m \text{ nat} \\
m + n = p \text{ nat} & \quad m + \text{ succ}(n) = \text{ succ}(p) \text{ nat}
\end{align*}
\]

When using this notation we tacitly incur the obligation to prove that the mode of the judgement is such that the object on the right-hand side of the equations is determined as a function of those on the left. Having done so, we abuse the notation by using the relation as function, writing just \(m + n\) for the unique \(p\) such that \(m + n = p\) nat.

### 1.7 Foundations

So far we have been vague about what sorts of "objects" may be the subjects of judgements. For example, the inductive definition of binary trees makes use of objects \(\text{empty}\) and \(\text{node}(x, y)\), where \(x\) and \(y\) are themselves objects, without saying precisely just what are these objects. More generally, we may ask, what sort of objects may we make judgements about? This is a delicate matter of foundations that we will only touch on briefly here.

One point of view is to simply take as given that the constructions we have mentioned so far are intuitively acceptable, and require no further justification or explanation. Roughly speaking, we admit as sensible any form of "finitary" construction in which finite entities are built up from other such finite entities by a finitely computable processes. Obviously this leaves quite a lot of room for interpretation, but in practice we never get into serious trouble and hence may safely adopt this rough-and-ready rule as a guide in the sequel.
If we’re really worried about nailing down what sorts of objects are admissible, then we must work within some presumed well-understood framework, such as set theory, in which to carry out our work.\(^2\) While this leads to an account that may be considered mathematically satisfactory, 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, since it provides no computational interpretation of sets.

A more reasonable choice for our purposes is to work within the universe of hereditarily finite sets, which are finite sets whose elements are finite sets, whose elements are finite sets, and so on. Any construction that can be carried out in this universe may be taken as computationally and foundationally meaningful. A more concrete, but technically awkward, approach is to admit only the natural numbers as objects — any other object of interest must be encoded as a natural number using the technique of Gödel numbering, which establishes a bijection between a set \(X\) of finitary objects and the set \(\mathbb{N}\) of natural numbers.\(^3\)

A natural universe of objects for programming purposes is provided by well-founded, finitely branching trees, or algebraic terms, which we will introduce in Chapter 5. These are quite convenient to use as a representation for a wide array of commonly occurring objects. Indeed, trees are easily represented in ML using data types and pattern matching.

### 1.8 Exercises

1. Let \(J\) be inductively defined by a rule set \(S\). Give an inductive definition of the judgement “\(D\) is a derivation of \(\vec{x} J\)” relative to the rule set \(S\).

2. Give an inductive definition of the forward-chaining and backward-chaining search strategies.

3. Introduce and discuss the internal and external consequence relations as two forms of hypothetical judgement . . . .

\(^2\)In effect we relegate all foundational questions to questions about the existence of appropriate sets.

\(^3\)One may then ask where the natural numbers come from. The answer is that they are taken as a primitive notion, rather than as being inductively defined. One has to start somewhere.
Chapter 2

Higher Order Judgement Forms

In Chapter 1 we introduced the concept of an inductively defined judgement expressing a relationship among a collection of objects. Such judgements are sometimes called categorical because they are unconditional assertions, such as the assertion that an object \( x \) is a binary tree. In this chapter we extend the framework of inductive definitions to permit two additional forms of judgement, the hypothetical and the general, which we combine to form the hypothetico-general judgement. These higher-order judgements play an important role in the theory of programming languages.

Note to the reader: Extending the framework of inductive definitions with hypothetical judgements requires no additional machinery beyond what was introduced in Chapter 1. However, the notion of a general judgement relies on material that we shall only present in Chapter 6. This chapter may be safely skipped, or lightly skimmed, on first reading; the concepts introduced here will not be needed until Chapter 9.

2.1 Hypothetical Judgements

The hypothetical judgement has the form

\[ J_1, \ldots, J_n \vdash J, \]

where each \( J_i \) (for \( 1 \leq i \leq n \)) and \( J \) are inductively defined categorical judgements. Informally, the hypothetical judgement expresses that \( J \) holds under the assumption that \( J_1, \ldots, J_n \) all hold. The judgements \( J_1, \ldots, J_n \) are called the hypotheses of the judgement, and \( J \) is called its conclusion. (The punctuation mark separating them is called a turnstile.) The hypothetical judgement form is also called an entailment relation, or a consequence relation.
For example, the hypothetical judgement

\[ x \text{ nat} \vdash \text{succ(succ}(x)) \text{ nat} \]

expresses the conditional judgement that, for any object \( x \), if \( x \) is a natural number, then so is its double successor. Intuitively, this is, of course, a valid judgement. Let us now make this intuition precise.

Recall from Chapter 1 that the evidence for an inductively defined categorical judgement is a derivation consisting of a composition of inference rules starting with axioms and ending that judgement. Evidence for a hypothetical judgement is defined similarly, except that the derivation of the conclusion may start with any or all of the hypotheses and end with the conclusion of the judgement. Put in other terms, evidence for a hypothetical judgement of the above form consists of evidence for \( J \) with respect to the extension of the rules defining the judgement form of \( J \) with \( J_1, \ldots, J_n \) as new axioms (rules without premise).

The crucial point is that the evidence for a hypothetical judgement consists of a uniform way to transform presumed evidence for the hypotheses into evidence for the conclusion, without regard to what that presumed evidence may actually be. The derivation may be specialized by “plugging in” whatever evidence for \( J_1, \ldots, J_n \) may arise, resulting in a derivation of \( J \) that does not make use of these new axioms. Of course, the evidence for the \( J_i \)'s may itself be hypothetical, say in hypotheses \( J'_1, \ldots, J'_m \), resulting in evidence for \( J'_1, \ldots, J'_m \vdash J \).

This interpretation leads to the following structural rules for the hypothetical judgement. Let \( \Gamma \) stand for any sequence \( J_1, \ldots, J_n \) of judgements under consideration. The structural rules governing the hypothetical judgement are as follows:

**Reflexivity** Every judgement is a consequence of itself: \( J \vdash J \).

**Weakening** If \( \Gamma \vdash J \), then \( \Gamma, J' \vdash J \) for any judgement \( J' \).

**Permutation** If \( \Gamma, J_1, J_2, \Gamma' \vdash J \), then \( \Gamma, J_2, J_1, \Gamma' \vdash J \).

**Contraction** If \( \Gamma, J, J' \vdash J' \), then \( \Gamma, J' \vdash J' \).

**Transitivity** If \( \Gamma, J' \vdash J \) and \( \Gamma', J' \vdash J' \), then \( \Gamma, \Gamma' \vdash J \).

These properties follow directly from the meaning of the hypothetical judgement:

**Reflexivity** The additional “axiom” \( J \) counts as a derivation of \( J \).
2.1 Hypothetical Judgements

Weakening  The derivation of \( J \) may make use of presumed derivations for the hypotheses, but need not do so.

Permutation  The order of hypotheses does not matter in the given interpretation.

Contraction  Since we may use the same hypothesis more than once, it does not matter if we repeat it.

Transitivity  If we plug in actual evidence for a hypothesis, then the evidence for the conclusion may be specialized to use it, leaving as hypotheses those used as evidence for it.

There is a close connection between the notion of a hypothetical judgement and the notion of a derived rule. Specifically, the inference rule

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

is derivable iff the hypothetical judgement

\[ J_1, \ldots, J_n \vdash J \]

is valid, for in both cases the meaning is that there exists a derivation of \( J \) from \( J_1, \ldots, J_n \).

There is another form of hypothetical judgement corresponding to admissible rules, written

\[ J_1, \ldots, J_n \models = J. \]

This means that \( J \) is valid with respect to the original set of inference rules whenever \( J_1, \ldots, J_n \) are also valid in the original set of rules. For example, with respect to the rule set defining natural numbers, given in Chapter 1, we have

\[ \text{succ}(x) \text{ nat} \models = x \text{ nat}. \]

This may be proved by rule induction, for if \( \text{succ}(x) \text{ nat} \), then this can only be by virtue of the rule

\[
\frac{x \text{ nat}}{\text{succ}(x) \text{ nat}},
\]

and hence the desired conclusion must hold (it is the premise of this inference). This is precisely the same as saying that the rule

\[
\frac{\text{succ}(x) \text{ nat}}{x \text{ nat}}
\]
is admissible.

Note that \( \text{succ}(x) \text{ nat} \not\vdash x \text{ nat} \) — there is no composition of rules that starts with the hypothesis and leads to the conclusion. Thus, the two forms of hypothetical judgement do not coincide, even though they satisfy the same structural properties. Here is a brief summary of why they are true for the second form:

**Reflexivity** If \( J \) is derivable from the original rules, then \( J \) is derivable from the original rules.

**Weakening** If \( J \) is derivable from the original rules assuming that \( J_1, \ldots, J_n \) are, then so it must also be derivable from an additional assumption.

**Permutation** Obviously the order of our assumptions does not matter.

**Contraction** Assuming the same thing twice is the same as assuming it once.

**Substitution** The assumption of \( J' \) used, in addition to \( \Gamma \), to derive \( J \) may be discharged by simply using the derivation of \( J' \) from \( \Gamma' \). This means that \( J \) is valid with respect to the original rule set, under the assumptions \( \Gamma \) and \( \Gamma' \).

When discussing the two forms of hypothetical judgement, we refer to the former as the *internal form*, and the latter as the *external form*. It should be immediately clear that the internal form is stronger than the external form (if \( \Gamma \vdash J \), then \( \Gamma \models J \)), but, as we have just seen, the converse does not hold. This is just a re-statement of the observation that every derivable rule is admissible, but the converse does not, in general, hold. Both forms of hypothetical judgement arise in the study of programming languages, but the internal form is far and away the more important — it arises in the definition of nearly every language we shall study.

### 2.2 General Judgements

The *general judgement* expresses a “parameterized assertion,” one that involves variables ranging over objects in the universe of discourse. The general judgement has the form

\[
\mid_{x_1, \ldots, x_n} \ J,
\]

where \( x_1, \ldots, x_n \) are variables that may occur in \( J \). Informally, the general judgement means that every instance of \( J \) obtained by choosing objects for the variables is valid.
2.3 Hypothetico-General Judgements

But what is a variable? And what is an instance? For these concepts to make sense, we restrict attention to inductively defined judgements over the universe of abstract binding trees (abt’s), which are introduced in Chapter 6. The reason for this restriction is that abt’s provide a notion of variable and substitution, which we now use to explain the meaning of the general judgement.

The general judgement

$$| \ldots x_n \ J$$

is valid iff

$$[x_1, \ldots, x_n \leftarrow a_1, \ldots, a_n] J$$

is valid for every choice of abt’s $a_i$. Evidence for the validity of the general judgement $| \ldots x_n \ J$ consists of a derivation scheme, $D$, a derivation with parameters, such that

$$[x_1, \ldots, x_n \leftarrow a_1, \ldots, a_n] D$$

is a derivation of

$$[x_1, \ldots, x_n \leftarrow a_1, \ldots, a_n] J.$$

Like the hypothetical judgement, the general judgement also obeys a collection of structural rules. For the sake of concision in stating these rules, let $\Delta$ range over finite sets of variables, $x_1, \ldots, x_n$.

**Weakening** If $|\Delta J$, then $|\Delta, x J$.

**Permutation** If $|\Delta, x_1, x_2, \Delta' J$, then $|\Delta, x_2, x_1, \Delta' J$.

**Contraction** If $|\Delta, x J$, then $|\Delta, x J$.

**Instantiation** If $|\Delta, x J$, then $|\Delta [x \leftarrow a] J$, provided that the free variables of $a$ are among those in $\Delta$.

Note the strong similarity to the structural rules governing the hypothetical judgement.

2.3 Hypothetico-General Judgements

The general judgement is most often used in conjunction with the hypothetical judgement. For example, the following general, hypothetical judgement is valid:

$$| x \ (x \text{ nat } \vdash \text{succ(succ}(x))) \text{ nat}).$$
2.4 Inductive Definitions, Revisited

The parentheses are written here for emphasis, but we usually omit them by treating the hypothetical judgement as binding more tightly than the general. The hypothesis \( x \text{ nat} \) has the effect of constraining the parameter \( x \) to range over natural numbers, excluding "garbage" not of interest for the inference.

Note that this is a single judgement form expressing a family of hypothetical judgements

\[
a \text{ nat} \vdash \text{succ}(\text{succ}(a)) \text{ nat}.
\]

In this way the general judgement permits us to capture the informal idea of a rule scheme as a single concept.

Since the general judgement occurs so often in conjunction with the hypothetical, we often combine the notation, writing \( \Gamma \vdash J \) for \( |\Delta \Gamma \vdash J \). This combined form is called a hypothetico-general judgement for obvious reasons. For example, the hypothetico-general judgement

\[
x \text{ nat} \vdash_x \text{succ}(\text{succ}(x)) \text{ nat}
\]

is a short-hand for the general, hypothetical judgement given above.

Somewhat confusingly, the subscript on the turnstile is often omitted, writing just \( \Gamma \vdash J \) for \( \Delta \Gamma \vdash J \), where \( \Delta \) is the set of free variables occurring in \( \Gamma \) and \( J \). For example, we may write just

\[
x \text{ nat} \vdash \text{succ}(\text{succ}(x)) \text{ nat}
\]

for the preceding hypothetico-general judgement, it being understood that \( x \) is a parameter of the hypothetical judgement.

2.4 Inductive Definitions, Revisited

Hypothetical and general judgements permit a particularly elegant form of inductive definition that may be illustrated by the following example. The idea is to give an inductive definition of a function to compute the "depth" of a closed abt over some (unspecified) signature. Roughly speaking, the depth of a closed abt is defined to be the maximum nesting depth of abstractors within it. We will give a simultaneous inductive definition of two judgements, \( d(a \text{ abt}) = n \), specifying that the abt \( a \) has depth \( n \), and \( d(\beta \text{ abs}) = n \), specifying that the abstractor \( \beta \) has depth \( n \). Because abstractors bind variables, it is impossible to define the depth only for closed abt's, but instead we must consider open ones as well. The main "trick" is to use
a hypothetico-general judgement in the rules to handle the variables that are introduced during the recursion. Here are the rules:

\[
\begin{align*}
d(\beta_1 \text{ abs}) &= n_1 \\
&\vdots \\
d(\beta_k \text{ abs}) &= n_k \\
\overline{\text{d}(o(\beta_1, \ldots, \beta_k) \text{ abt})} &= \max(n_1, \ldots, n_k)
\end{align*}
\]

\[
\begin{align*}
d(a \text{ abt}) &= n \\
d(x \text{ abt}) &= 0 \vdash_x d(\beta \text{ abs}) = n \\
\overline{d(a \text{ abs})} &= n \\
\overline{d(x.\beta \text{ abs})} &= n + 1
\end{align*}
\]

Observe that the premise of the third rule is a hypothetico-general judgement, which expresses the idea that the depth of an abstractor is one more than the depth of its body, assuming that the bound variable has depth 0. This assumption is used to provide a definition for the depth of a variable whenever they are encountered, using the reflexivity property of the hypothetical judgement.

This example illustrates an important convention, called the freshness convention, that we shall use tacitly throughout the book. Whenever a rule, such as the last one above, introduces a name using a hypothetico-general judgement, it is assumed that this name is chosen so as not to otherwise occur in the parameter set. This requirement may always be met by suitably renaming the bound variable of the abstractor before applying the rule. This is one important benefit of always working “up to α-conversion,” which frees us from having to worry about the complications of name re-use within a given scope.

The use of (the internal form of) a hypothetical judgement in the premise of an inference rule goes beyond what we considered in Chapter 1, wherein only categorical judgements were permitted. This is a significant extension, and some justification is required to ensure that what we’re doing is sensible. The key is to regard the rules as a simultaneous inductive definition of an infinite family of judgements of the form

\[
d(x_1 \text{ abt}) = 0, \ldots, d(x_k \text{ abt}) = 0 \vdash_{x_1, \ldots, x_k} d(a \text{ abt}) = n
\]

and

\[
d(x_1 \text{ abt}) = 0, \ldots, d(x_k \text{ abt}) = 0 \vdash_{x_1, \ldots, x_k} d(\beta \text{ abs}) = n.
\]

The idea is that for each choice of parameters set \(x_1, \ldots, x_k\) and each set of corresponding assumptions \(d(x_1 \text{ abt}) = 0, \ldots, d(x_k \text{ abt}) = 0\), we have two judgements defining the depth of an abt or abstractor with free variables among those parameters. The rule for abstractors augments the parameter and hypothesis sets, and thereby refers to another member of the same family of judgements in its definition.
To clarify the situation, let us re-write the inductive definition of the depth judgements in a form that makes the parameter and hypothesis sets explicit. Let $\Delta$ range over finite sets of variables $x_1, \ldots, x_k$, and, for each choice of parameter set $\Delta$, let $\Gamma$ range over sets of assumptions of the form $d(x_1 \text{abt}) = 0, \ldots, d(x_k \text{abt}) = 0$. Here are the same rules, written in a more explicit form:

$$\Gamma \vdash_{\Delta} d(\beta_1 \text{abs}) = n_1 \quad \cdots \quad d(\beta_k \text{abs}) = n_k$$

$$\Gamma \vdash_{\Delta} d(o(\beta_1, \ldots, \beta_k) \text{abt}) = \max(n_1, \ldots, n_k)$$

$$\Gamma \vdash_{\Delta} d(a \text{abt}) = n \quad \Gamma, d(x \text{abt}) = 0 \vdash_{\Delta, x} d(\beta \text{abs}) = n$$

$$\Gamma \vdash_{\Delta} d(a \text{abs}) = n \quad \Gamma \vdash_{\Delta} d(x.\beta \text{abs}) = n + 1$$

This presentation makes clear the augmentation of the current parameter and hypothesis set in the inference. When we do so, we “switch” to another judgement in the same family, with the extended parameter and hypothesis sets.

What is the principle of rule induction associated with rules whose premises involve hypothetical and general judgements? An example will illustrate the general case. Suppose we wish to prove that the mode of the depth judgements is $(\forall, \exists!)$, expressing that they really do define functions.

**Theorem 2.1**

1. If $d(x_1 \text{abt}) = 0, \ldots, d(x_k \text{abt}) = 0 \vdash_{x_1, \ldots, x_k} d(a \text{abt}) = n$, then for every family $a_1, \ldots, a_k$ of abt’s such that $d(a_1 \text{abt}) = 0, \ldots, d(a_k \text{abt}) = 0$, there exists a unique $n$ such that $d([x_1, \ldots, x_k, x] \leftarrow a_1, \ldots, a_k]a \text{abt}) = n$.

2. If $d(x_1 \text{abt}) = 0, \ldots, d(x_k \text{abt}) = 0 \vdash_{x_1, \ldots, x_k} d(\beta \text{abs}) = n$, then for every family $a_1, \ldots, a_k$ of abt’s such that $d(a_1 \text{abt}) = 0, \ldots, d(a_k \text{abt}) = 0$, there exists a unique $n$ such that $d([x_1, \ldots, x_k, x] \leftarrow a_1, \ldots, a_k]\beta \text{abt}) = n$.

**Proof**: We may prove these facts simultaneously by rule induction. The most interesting case, of course, is the last. The inductive hypothesis states that if $d(b \text{abt}) = 0$, then there exists a unique $n$ such that

$$d([x_1, \ldots, x_k, x] \leftarrow a_1, \ldots, a_k,b]a \text{abt}) = n.$$ 

We are to show that there exists a unique $n'$ such that

$$d(x,[x_1, \ldots, x_k] \leftarrow a_1, \ldots, a_k]b \text{abs}) = n'.$$

The result follows immediately by taking, in the inductive hypothesis, $b = x$, and taking, in the conclusion, $n' = n + 1$. 

---

**WORKING DRAFT**

**FEBRUARY 4, 2006**
Finally, let us note that it is not permissible to use the external form of hypothetical judgement in an inference rule! The justification we have given for using the internal form in premises does not extend to the external form, precisely because its meaning is not defined by extending the rule set with new axioms. Indeed, admitting the external form in the premise of a rule destroys the mathematical underpinnings of the theory of inductive definitions, invalidating their use as a definitional tool.

2.5 Exercises

1. Investigate why the premise of an inference rule may not be taken to be the external form of hypothetical judgement. Show that there exists an “inductive definition” using the external form of hypothetical for which there is no strongest judgement closed under the given rules.
Chapter 3

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 judgement, $\rightarrow$. The transition judgement describes how the state of the machine evolves during execution.

3.1 Transition Systems

A transition system is specified by the following judgements:

1. $s$ state, asserting that $s$ is a state of the transition system.
2. $s$ final, asserting that $s$ is a final state.
3. $s$ init, asserting that $s$ is an initial state.
4. $s \mapsto s'$, where $s$ state and $s'$ state, asserting that state $s$ may transition to state $s'$.

We require that if $s$ final, then for no $s'$ do we have $s \mapsto s'$. In general, a state $s$ for which there is no $s' \in S$ such that $s \mapsto s'$ is said to be stuck. All final states are stuck, but not all stuck states need be final!

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

A transition system is deterministic iff for every state $s$ there exists at most one state $s'$ 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 judgement \( s \xrightarrow{*} s' \) is the reflexive, transitive closure of the transition
judgement. It is inductively defined by the following rules:

\[
\begin{align*}
&\frac{}{s \xrightarrow{*} s} \\
&\frac{s \xrightarrow{\delta} s' \quad s' \xrightarrow{*} s''}{s \xrightarrow{*} s''}
\end{align*}
\]

It is easy to prove by rule induction that \( \xrightarrow{*} \) is indeed reflexive and transi-
tive.

Since the multistep transition is inductively defined, we may prove that
\( P(s, s') \) holds whenever \( s \xrightarrow{*} s' \) by showing

1. \( P(s, s) \).

2. if \( s \xrightarrow{\delta} s' \) and \( P(s', s'') \), then \( P(s, s'') \).

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.

The \( n \)-times iterated transition judgement, \( s \xrightarrow{n} s' \), where \( n \) nat, is in-
ductively defined by the following rules:

\[
\begin{align*}
&\frac{}{s \xrightarrow{0} s} \\
&\frac{s \xrightarrow{\delta} s' \quad s' \xrightarrow{n} s''}{s \xrightarrow{n+1} s''}
\end{align*}
\]

It is easy to show that \( s \xrightarrow{*} s' \) iff \( s \xrightarrow{n} s' \) for some \( n \) nat.

Finally, the complete transition judgement, \( s \xrightarrow{\sim} s' \) is the restriction to
\( s \xrightarrow{*} s' \) so that \( s' \) final. That is, \( s \xrightarrow{\sim} s' \) iff \( s \xrightarrow{*} s' \) and \( s' \) final.

3.2 Exercises

1. Prove that \( s \xrightarrow{*} s' \) iff there exists \( n \) nat such that \( s \xrightarrow{n} s' \).
Part II

Levels of Syntax
Chapter 4

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.

4.1 Strings

To begin with we must define what we mean by characters and strings. An alphabet, $\Sigma$, is a set of characters, or symbols. Often $\Sigma$ 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 $\text{char}$ is inductively defined by the following rules (one per choice of $c \in \Sigma$):

\[
\begin{align*}
(c \in \Sigma) & \quad \frac{}{c \ \text{char}}
\end{align*}
\]

The judgment form $\text{string}_\Sigma$ states that $s$ is a string of characters from $\Sigma$. It is inductively defined by the following rules:

\[
\begin{align*}
\varepsilon \ \text{string}_\Sigma & \quad \frac{\text{char}}{c \ \cdot \ s \ \text{string}_\Sigma}
\end{align*}
\]

29
In most cases we omit explicit mention of the alphabet, Σ, and just write a string to indicate that s is a string over an implied choice of alphabet.

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

\section{4.2 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 \(\Sigma\) of terminals.
2. A finite set \(N\) of non-terminals that stand for the syntactic categories.
3. A set \(P\) of productions of the form \(A : : = \alpha\), where \(A\) is a non-terminal and \(\alpha\) is a string of terminals and non-terminals.

Whenever there is a set of productions

\begin{align*}
A & : : = \alpha_1 \\
& \vdots \\
A & : : = \alpha_n
\end{align*}

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

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

of the grammar, where \(A_1, \ldots, A_n\) are all of the non-terminals on the right-hand side of that production, and \(s_1, \ldots, s_{n+1}\) are strings of terminals, add a rule

\[s'_1 A_1 \ldots s'_n A_n \over s_1 s'_1 s_2 \ldots s_n s'_n s_{n+1} A\]
4.3 Ambiguity

to the rule set $R$. For each non-terminal $A$, we say that $s$ is a string of syntactic category $A$ iff $sA$ 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.

$$
\begin{align*}
\text{Digits} & : = 0 \mid 1 \mid \cdots \mid 9 \\
\text{Numbers} & : = D \mid ND \\
\text{Expressions} & : = N \mid E+E \mid E*E
\end{align*}
$$

Here is this grammar presented as a simultaneous inductive definition:

$$
\begin{align*}
0 \text{ dig} & \cdots 9 \text{ dig} \\
\text{ s dig} & \frac{\text{ s num} \text{ s num}}{s_{1} \text{ num} \text{ s_{2} dig}} \\
\text{ s num} & \frac{\text{ s exp}}{} \\
\text{ s_{1} exp} & \frac{\text{ s_{2} exp}}{s_{1}+s_{2} \text{ exp}} \\
\text{ s_{1} exp} & \frac{\text{ s_{2} exp}}{s_{1}*s_{2} \text{ exp}}
\end{align*}
$$

Each syntactic category of the grammar determines a judgement form. For example, the category of expressions corresponds to the judgement form $\text{exp}$, and so forth.

4.3 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 tell from the generated string which reading is intended. This causes numerous problems, an example of which arises from an attempt to define the value, a natural number, of an arithmetic expression, \( e \), represented as a string.

We will give an inductive definition of the following three forms of judgement:

\[
egin{align*}
   s \text{ exp } &\downarrow k \text{ nat } & \text{ expression } s \text{ has value } k \\
   s \text{ num } &\downarrow k \text{ nat } & \text{ numeral } s \text{ has value } k \\
   s \text{ dig } &\downarrow k \text{ nat } & \text{ digit } s \text{ has value } k
\end{align*}
\]

The rules defining these judgements are as follows:

\[
\begin{align*}
   &0 \text{ dig } \downarrow \text{ zero } \text{ nat } \\
   &1 \text{ dig } \downarrow \text{ succ(zero) } \text{ nat } \\
   &s \text{ dig } \downarrow k \text{ nat } \\
   &s \text{ num } \downarrow k \text{ nat } \\
   &s_1 \text{ num } \downarrow k_1 \text{ nat } \\
   &s_2 \text{ dig } \downarrow k_2 \text{ nat } \\
   &k = 10 \times k_1 + k_2 \\
   &s_1 \text{ s num } \downarrow k \text{ nat }
\end{align*}
\]

\[
\begin{align*}
   &s \text{ num } \downarrow k \text{ nat } \\
   &s \text{ exp } \downarrow k \text{ nat }
\end{align*}
\]

\[
\begin{align*}
   &s_1 \text{ exp } \downarrow k_1 \text{ nat } \\
   &s_2 \text{ exp } \downarrow k_2 \text{ nat } \\
   &k = k_1 + k_2 \\
   &s_1+s_2 \text{ exp } \downarrow k \text{ nat }
\end{align*}
\]

\[
\begin{align*}
   &s_1 \text{ exp } \downarrow k_1 \text{ nat } \\
   &s_2 \text{ exp } \downarrow k_2 \text{ nat } \\
   &k = k_1 \times k_2 \\
   &s_1*s_2 \text{ exp } \downarrow k \text{ nat }
\end{align*}
\]

(We have taken the liberty of assuming that rules for computing with natural numbers have already been defined.)

Given the intended interpretation of these judgements, it is natural to consider whether they have the mode \((\forall, \exists!)\), over the domain of expressions/numbers/digits (strings formed according to the grammar given earlier) as input and natural numbers as output. The all-important question is whether this is a valid mode for these judgements — do they determine a partial function from input to output, as might be expected?

Perhaps surprisingly, the answer is no! Informally, 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, there does not exist a unique value for every string of the appropriate grammatical class.

It is instructive to see how an attempt to prove that the evaluation judgements have the specified mode breaks down. First, let us be precise about what we need to prove. We must show that for every $s$, if $s \exp$, there is a unique $k$ such that $k \nat$ and $s \exp \downarrow k \nat$, and similarly for the other two judgement forms. It is natural to proceed by rule induction on the rules defining the judgement $s \exp$. We consider each rule in turn. The crucial cases are when $s = s_1 + s_2$ and when $s = s_1 * s_2$, and we have by induction that $s_1 \exp \downarrow k_1 \nat$ and $s_2 \exp \downarrow k_2 \nat$ for some uniquely determined $k_1$ and $k_2$ such that $k_1 \nat$ and $k_2 \nat$. And in that case we may take as the sum and product, respectively, of $k_1$ and $k_2$. Since the sum and product of $k_1$ and $k_2$ are uniquely determined, we seem to have completed the proof (the other cases being handled similarly).

But have we? The problem is that a given string $s$ can be both of the form $s_1 + s_2$ and $s_1 \times s_2$ at the same time, and we have no way to know which interpretation is intended! The preceding argument, which proceeds by rule induction on the rules defining $s \exp$, tells us that the value of $s$ is uniquely determined if we are given the rule used to form $s$ — that is, we are told how to interpret it (as a sum or as a product). If we are given $s$ alone, with no information about how it was generated by the grammar, then the result is ambiguous — one string can have many values according to these rules.

### 4.4 Resolving Ambiguity

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 interpretation.

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

3. Introduce associativity conventions that determine how to resolve ambiguities between operators of the same precedence (e.g., by stipulating that addition is right-associative).
Using these techniques, we arrive at the following revised grammar for arithmetic expressions.

\[
\begin{align*}
\text{Digits} & \quad D \, : \, :: \, 0 \mid 1 \mid \cdots \mid 9 \\
\text{Numbers} & \quad N \, : \, :: \, D \mid ND \\
\text{Factors} & \quad F \, : \, :: \, N \mid (E) \\
\text{Terms} & \quad T \, : \, :: \, F \mid F \cdot T \\
\text{Expressions} & \quad E \, : \, :: \, T \mid T+E
\end{align*}
\]

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

Re-writing this as an inductive definition, we obtain the following rules:

\[
\begin{align*}
0 \text{dig} & \cdots 9 \text{dig} \\
\text{s num} & \quad \text{s exp} \\
\text{s fct} & \quad \text{s trm} \\
\end{align*}
\]

Using these rules, it is then possible to prove that the evaluation judgements have mode \((\forall, \exists!\)) over grammatically correct strings as inputs and natural numbers as outputs. The crucial difference from our first attempt is that each string admits at most one decomposition consistent with the rules of the grammar. For example, the string \(s = 1+2*3\) may be derived as \(s\exp\) only by decomposing \(e\) as \(s_1+s_2\), where \(s_1 = 1\) and \(s_2 = 2*3\); no other decomposition is possible (be sure you understand why). If we consider the input to be any strings at all, not just those that are grammatical according to the specified rules, then the best mode possible is \((\forall, \exists \leq 1)\), since an ungrammatical string has no value, but the value is uniquely determined for every grammatical string.

### 4.5 Exercises
Chapter 5

Abstract Syntax Trees

The concrete syntax of a language consists of a linear presentation of it as a set of strings — sequences of characters that reflect the conventional modes of reading and writing programs. The main job of concrete syntax design is to ensure the convenient readability and writability of the language, based on subjective criteria such as similarity to other languages, ease of editing, and so forth.

But languages are also the subject of analysis, for example to define what it means to evaluate an arithmetic expression we must analyze the structure of expressions. For this purpose the concrete syntax introduces a level of bureaucracy that we would like to avoid. For example, we must ensure that the syntax is presented in unambiguous form, and we are forced to deal with details of presentation that have no effect on its meaning, but rather are conveniences for reading and writing phrases in the language.

To avoid this clutter it is conventional to define an abstract syntax of a language that abstracts away from the concrete presentation of a phrase as a string, and instead focuses on the essential structure of the phrase in a manner amenable to analysis. Parsing is the process of translating the concrete to the abstract syntax; once parsed, we need never worry about the concrete presentation of a phrase again.

The abstract syntax of a language consists of an inductively-defined set of abstract syntax trees, or ast’s. An ast is a tree structure whose nodes are labeled with operators of a specified arity, the number of children of a node labeled with that operator. The tree structure makes evident the overall form of a piece of abstract syntax, avoiding the need for any machinery to disambiguate.
5.1 Abstract Syntax Trees

Abstract syntax trees are constructed from other abstract syntax trees by combining them with a 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 \( o \in \text{dom}(\Omega) \) its arity \( \Omega(o) \). The judgement form \( \text{ast}_\Omega \) is inductively defined by the following rules:

\[
\frac{a_1 \text{ast}_\Omega \cdots a_n \text{ast}_\Omega \quad (\Omega(o) = n)}{o(a_1, \ldots, a_n) \text{ast}_\Omega}
\]

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, \( \Omega_{\text{expr}} \), specifies an abstract syntax for the language of arithmetic expressions:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Arity</th>
</tr>
</thead>
<tbody>
<tr>
<td>num([n])</td>
<td>0</td>
</tr>
<tr>
<td>plus</td>
<td>2</td>
</tr>
<tr>
<td>times</td>
<td>2</td>
</tr>
</tbody>
</table>

Here \( n \) ranges over the natural numbers; the operator \( \text{num}[n] \) is the \( n \)th numeral, which takes no arguments. The operators plus and times take two arguments each, as might be expected. The abstract syntax of our language consists of those \( a \) such that \( a \text{ast}_{\Omega_{\text{expr}}} \).

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

\[
\frac{(n \in \mathbb{N})}{\text{num}[n] \text{ast}} \quad (5.1)
\]

\[
\frac{a_1 \text{ast} \quad a_2 \text{ast}}{\text{plus}(a_1, a_2) \text{ast}} \quad (5.2)
\]

\[
\frac{a_1 \text{ast} \quad a_2 \text{ast}}{\text{times}(a_1, a_2) \text{ast}} \quad (5.3)
\]

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.
5.2 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 general case to show that $P \ a$ holds whenever $a \ ast \ \Omega$, it is enough to show that for each operator $o$ such that $\Omega(o) = n$, if $P \ a_1, \ldots, P \ a_n$, then $P \ o(a_1,\ldots,a_n)$.

In the special case of arithmetic expressions the principal of structural induction states that to show $P \ a$ whenever $a \ ast$, it is enough to show the following three facts:

1. $P \ \text{num}[n]$ for every $n \in \mathbb{N}$.
2. if $P \ a_1$ and $P \ a_2$, then $P \ \text{plus}(a_1,a_2)$.
3. if $P \ a_1$ and $P \ a_2$, then $P \ \text{times}(a_1,a_2)$.

To illustrate the use of structural induction let us inductively define the evaluation judgement $a \ ast \ \Downarrow \ k \ nat$ by the following rules:

$$
\text{num}[n] \ ast \ \Downarrow \ n \ nat \quad (5.4)
$$

$$
\begin{array}{c}
\frac{a_1 \ ast \ \Downarrow \ k_1 \ nat \quad a_2 \ ast \ \Downarrow \ k_2 \ nat \quad k = k_1 + k_2 \ nat}{\text{plus}(a_1,a_2) \ ast \ \Downarrow \ k \ nat} \\
\end{array} \quad (5.5)
$$

$$
\begin{array}{c}
\frac{a_1 \ ast \ \Downarrow \ k_1 \ nat \quad a_2 \ ast \ \Downarrow \ k_2 \ nat \quad k = k_1 \times k_2 \ nat}{\text{times}(a_1,a_2) \ ast \ \Downarrow \ k \ nat} \\
\end{array} \quad (5.6)
$$

The evaluation judgement has mode $(\forall, \exists!)$, which is to say that for every $a \ ast$ there exists a unique $k \ nat$ such that $a \ ast \ \Downarrow \ k \ nat$. This is easily proved by structural induction on $a$, showing that in each case there is a uniquely determined $k$ such that $a \ ast \ \Downarrow \ k \ nat$.

In the above presentation of the evaluation judgement we chose the output domain to be the natural numbers. But it would be equally easy and natural to choose the output domain to be the same as the input domain, so that the output is also an ast, albeit one in fully evaluated form. Here are the revised rules written in this style.
5.3 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.

As an example, consider the following inductive definition of several mutually parsing judgements that relate the concrete to the abstract syntax.

\[
\begin{align*}
\text{0 dig} & \leftrightarrow \text{num}[0] \text{ ast} & & \ldots & & \text{9 dig} & \leftrightarrow \text{num}[9] \text{ ast} \\
\text{s dig} & \leftrightarrow \text{a ast} & & \text{s num} & \leftrightarrow \text{a ast} & & \text{s}_1 \text{ num} & \leftrightarrow \text{num}[k_1] \text{ ast} & & \text{s}_2 \text{ dig} & \leftrightarrow \text{num}[k_2] \text{ ast} & & \text{s}_1 \text{s}_2 \text{ num} & \leftrightarrow \text{num}[10 \times k_1 + k_2] \text{ ast}
\end{align*}
\]
Observe, first of all, that a successful parse implies that the string must have been derived according to the unambiguous grammar and that the result is a valid ast.

**Theorem 5.1**

1. If $s \text{dig} \rightarrow a \text{ast}$, then $s \text{dig}$ and $a \text{ast}$.

2. If $s \text{num} \rightarrow a \text{ast}$, then $s \text{num}$ and $a \text{ast}$.

3. If $s \text{fct} \rightarrow a \text{ast}$, then $s \text{fct}$ and $a \text{ast}$.

4. If $s \text{trm} \rightarrow a \text{ast}$, then $s \text{trm}$ and $a \text{ast}$.

5. If $s \text{exp} \rightarrow a \text{ast}$, then $s \text{exp}$ and $a \text{ast}$.

These may be proved by induction on the rules defining the parser.

If a string is generated according to the rules of the grammar, then it has a parse as an ast.

**Theorem 5.2**

1. If $s \text{dig}$, then there is a unique $a$ such that $s \text{dig} \rightarrow a \text{ast}$.

2. If $s \text{num}$, then there is a unique $a$ such that $s \text{num} \rightarrow a \text{ast}$.

3. If $s \text{fct}$, then there is a unique $a$ such that $s \text{fct} \rightarrow a \text{ast}$.

4. If $s \text{trm}$, then there is a unique $a$ such that $s \text{trm} \rightarrow a \text{ast}$.

5. If $s \text{exp}$, then there is a unique $a$ such that $s \text{exp} \rightarrow a \text{ast}$.

These are proved simultaneously by induction on the rules defining the unambiguous grammar.
5.4 Exercises

1. Give a right-recursive grammar for numbers, and show how to parse it. Discuss the relevance of this variation to writing a recursive descent parser.

2. Show that the parser may be “run backwards” to obtain an unparser, or pretty printer. Introduce judgements that characterize those ast’s that unparse to a string of each grammatical class. Then show that the unparser also has mode (∀, ∃!) over appropriate domains.
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 designators.

6.1 Names

Names are widely used in programming languages: names of variables, names of fields in structures, names of types, 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 intrinsic significance, but serves only to distinguish one name from another. Consequently, we shall treat names as atoms, and abstract away their internal structure. We shall assume that we have a judgement \( x \text{ name} \) expressing that \( x \) is a name, and a judgement \( x \neq y \text{ name} \) stating that \( x \) and \( y \) are distinct names. We shall also assume that there are infinitely many \( x \) such that \( x \text{ name} \). The judgement \( [x \leftrightarrow y]z = z' \text{ name} \) is inductively defined by the following rules:

\[
\begin{align*}
[x \leftrightarrow y]x &= y \text{ name} \\
[x \leftrightarrow y]y &= x \text{ name} \\
x \neq z \text{ name} & \quad y \neq z \text{ name} \\
\hline
[x \leftrightarrow y]z &= z' \text{ name}
\end{align*}
\]
6.2 Abstract Syntax With Names

Suppose that we enrich the language of arithmetic expressions given in Chapter 5 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:

\[
\text{id}(x) \text{ ast}_\Omega \quad \text{let}(x, a_1, a_2) \text{ ast}_\Omega
\]

The ast \(\text{id}(x)\) represents a use of a name, \(x\), as a variable, and the ast \(\text{let}(x, a_1, a_2)\) introduces a name, \(x\), that is to be bound to (the value of) \(a_1\) for use within \(a_2\).

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 \(\text{let}(x, a_1, a_2)\), the name \(x\) is available for use within \(a_2\), but not within \(a_1\). That is, the name \(x\) is bound by the \(\text{let}\) construct for use within its scope, the sub-tree \(a_2\). 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?

\[\text{let}(x, a_1, \text{let}(x, \text{id}(x), \text{id}(x)))\]

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

6.3 Abstract Binding Trees

Abstract binding trees are a generalization of abstract syntax trees that provide intrinsic support for binding and scope of names. Just as with ast’s, operators may be used to combine several (possibly none) abt’s to form another. In addition there are two other forms of abt: a name, and an abstractor. An abstractor binds a name within a specified abt. That name may be used within that abt to refer to the binding site represented by the abstractor. Since various operators may bind various names in various argument positions, we must generalize the arity of an operator to be a finite sequence of natural numbers specifying the valence of each constituent abt of the operator. The valence of an abt is simply the number of abstractors at the root.

\[\text{let}(x, a_1, \text{let}(x, \text{id}(x), \text{id}(x)))\]

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

---

1One may also devise a concrete syntax, for example writing \(\text{let } x = e_1 \text{ in } e_2\) for the binding construct, and a parser to translate from the concrete to the abstract syntax.
of that abt, specifying how many variables are bound within it. This notion of arity generalizes that in Chapter 5 by taking the arity \( k \) in that chapter to mean the arity \((0,0,\ldots,0)\) of length \( k \) in this chapter.

This informal description can be made precise by giving an inductive definition of the judgement \( a \ \text{abt}_\Omega^k \) stating that \( a \) is a well-formed abt of valence \( k \) with respect to the signature \( \Omega \) assigning an arity to each of a finite set of operators.

\[
\frac{x \ \text{name}}{x \ \text{abt}_\Omega^0} \quad \frac{a_1 \ \text{abt}_\Omega^{n_1} \ \cdots \ a_k \ \text{abt}_\Omega^{n_k}}{o(a_1,\ldots,a_k) \ \text{abt}_\Omega^0} \quad \text{(s)} \quad \frac{x \ \text{name}}{x.a \ \text{abt}_\Omega^{n+1}}
\]

The condition marked \((s)\) states that \( \Omega(o) = (n_1,\ldots,n_k) \).

An abt of valence \( n \) has the form \( x_1.x_2.\ldots.x_n.a \), which we often write as \( x_1,\ldots,x_n.a \). We tacitly assume that no name is repeated in such a sequence, since doing so serves no useful purpose. We usually omit explicit mention of the signature \( \Omega \) when it is clear from context, and we often write just \( a \ \text{abt} \) to mean \( a \ \text{abt}_\Omega^0 \).

The language of arithmetic expressions consists of the abstract binding trees over the following signature.

\[
\begin{array}{ll}
\text{Operator} & \text{Arity} \\
\text{num}\ [n] & () \\
\text{plus} & (0,0) \\
\text{times} & (0,0) \\
\text{let} & (0,1)
\end{array}
\]

The arity of the “let” operator indicates that no name is bound in the first position, but that one name is bound in the second.

This class of abt’s over this signature may be explicitly defined by the following rules:

\[
\frac{x \ \text{name}}{x \ \text{abt}} \quad \frac{n \ \text{nat}}{\text{num}\ [n] \ \text{abt}}
\]

\[
\frac{a_1 \ \text{abt} \ a_2 \ \text{abt}}{\text{plus}(a_1,a_2) \ \text{abt}} \quad \frac{a_1 \ \text{abt} \ a_2 \ \text{abt}}{\text{times}(a_1,a_2) \ \text{abt}}
\]

\[
\frac{a_1 \ \text{abt} \ x \ \text{name} \ a_2 \ \text{abt}}{\text{let}(a_1,x.a_2) \ \text{abt}}
\]

By specializing the definition to a particular signature we avoid explicit mention of abt’s of non-zero valence, these being only of auxiliary interest.
6.4 Renaming

A fundamental concept is the notion of a name, \( x \), lying apart from an abt, \( a \). This is expressed by the judgement \( x \# a \ abt^n \), which is inductively defined by the following rules:\(^2\)

\[
\begin{align*}
  & x \# y \ name \quad x \# a_1 \ abt^{n_1} \quad \cdots \quad x \# a_k \ abt^{n_k} \\
  & \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \44

\begin{align*}
  & x \# y \ name \\
  & \frac{x \# y \ abt^0}{x \# a \ abt^n} \\
  & x \# y \ name \quad x \# a \ abt^n \\
  & \frac{x \# y \ abt^0}{x \# a \ abt^n} \\
  & x \# x.a \ abt^{n+1}
\end{align*}

We say that a name, \( x \), lies within, or is free in, an abt, \( a \), written \( x \in a \ abt \), iff it is not the case that \( x \# a \ abt \). We leave as an exercise to give a direct inductive definition of this judgement.

The result, \( a' \), of swapping one name, \( x \), for another, \( y \), within an abt, \( a \), written \([x \leftrightarrow y]a = a' \ abt \) is inductively defined by the following rules:

\[
\begin{align*}
  & [x \leftrightarrow y]z = z' \ name \\
  & [x \leftrightarrow y]z = z' \ abt^0 \\
  & [x \leftrightarrow y]a_1 = a'_1 \ abt^{n_1} \\
  & \cdots \\
  & [x \leftrightarrow y]a_k = a'_k \ abt^{n_k} \\
  & [x \leftrightarrow y]o(a_1, \ldots, a_k) = o(a'_1, \ldots, a'_k) \ abt^0 \\
  & [x \leftrightarrow y]z = z' \ name \\
  & [x \leftrightarrow y]z.a = z'.a' \ abt^{n+1}
\end{align*}
\]

It is easy to check that the swapping judgement has mode \((\forall, \forall, \forall, \exists)\), and so we will henceforth use it as a function. Note that name-swapping is self-inverse in that applying it twice leaves the term invariant.

A chief characteristic of a binding operator is that the choice of bound names does not matter. This is captured by treating as equivalent any two abt’s that differ only in the choice of bound names, but are otherwise identical. This relation is called, for historical reasons, \( \alpha \)-equivalence. It is in-

\(^2\)Here and elsewhere in this chapter, the side condition marked (\( \ast \)) on rules is as described in the preceding section.
6.4 Renaming

ductively defined by the following rules:

\[
\begin{align*}
  x &=_a x \cdot \text{abt}^0 \\
  a &=_a b \cdot \text{abt}^n & a_1 &=_a b_1 \cdot \text{abt}^n_1 & \cdots & a_k &=_a b_k \cdot \text{abt}^n_k \\
  o(a_1, \ldots, a_k) &=_a o(b_1, \ldots, b_k) \cdot \text{abt}^0 \\
  x \cdot a &=_a x \cdot b \cdot \text{abt}^{n+1} & x \cdot y \cdot \text{abt}^n & [x \leftrightarrow y]a &=_a b \cdot \text{abt}^n \\
  x \cdot a &=_a y \cdot b \cdot \text{abt}^{n+1}
\end{align*}
\]

In practice we abbreviate these relations to \(a =_a b\) and \(\beta =_a \gamma\), respectively.

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

\[
\begin{align*}
  \text{let}(x, x.x) &=_a \text{let}(x, y.y) \\
  \text{let}(y, x.x) &=_a \text{let}(y, y.y) \\
  \text{let}(x, x.x) \neq_a \text{let}(y, y.y) \\
  \text{let}(x, x.\text{plus}(x, y)) &=_a \text{let}(x, z.\text{plus}(z, y)) \\
  \text{let}(x, x.\text{plus}(x, y)) \neq_a \text{let}(x, y.\text{plus}(y, y))
\end{align*}
\]

The following rule of \(\alpha\)-equivalence, which is often stated as a basic axiom, is derivable from the preceding rules:

\[
\begin{align*}
  x \cdot y \cdot \text{abt}^n & y \cdot a \cdot \text{abt}^n \\
  x.a &=_a y.\text{let}(x, y.y)
\end{align*}
\]

The following variation on the rule of \(\alpha\)-equivalence for abstractors is also derivable, and, moreover, includes the rule we have given as a special case:

\[
\begin{align*}
  x \cdot y \cdot \text{abt}^n & y \cdot b \cdot \text{abt}^n & z \cdot a \cdot \text{abt}^n & z \cdot b \cdot \text{abt}^n \\
  x.a &=_a y.\text{let}(y, y.y) \\
  x.a &=_a y.\text{let}(x, z.\text{plus}(z, y)) \\
  x.a &=_a y.\text{let}(x, y.\text{plus}(y, y))
\end{align*}
\]

Apartness respects \(\alpha\)-equivalence:

**Lemma 6.1**

If \(a =_a b\) and \(x \neq b\), then \(x \neq a\).

It may be shown by rule induction that \(\alpha\)-equivalence is, in fact, an equivalence relation (i.e., it is reflexive, symmetric, and transitive).

**Theorem 6.2**

The \(\alpha\)-equivalence relation is reflexive, symmetric, and transitive.
From this point onwards we identify any two abt's \( a \) and \( b \) such that 
\[ a =_\alpha b \text{ } \text{abt}. \]
This means that an abt implicitly stands for its \( \alpha \)-equivalence class, and that we tacitly assert that all operations and relations on abt's respect \( \alpha \)-equivalence. Put the other way around, any operation or relation on abt’s that fails to respect \( \alpha \)-equivalence is illegitimate, and therefore ruled out of consideration. One consequence of this policy on abt’s is that whenever we encounter an abstractor \( x.a \), we may assume that \( x \) is fresh in the sense that it is distinct from any given finite set of names.

### 6.5 Structural Induction With Binding and Scope

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 (for all \( n \geq 0 \)) that \( a \text{ } \text{abtn} \) implies \( P_n \text{ } a \) holds, it is enough to show the following:

1. For any name \( x \), the judgement \( P_0 \text{ } x \) holds.
2. For each operator, \( o \), of arity \( (m_1, \ldots, m_k) \), if \( P_{m_1} \text{ } a_1 \) and \( \ldots \) and \( P_{m_k} \text{ } a_k \), then \( P_0 \text{ } o(a_1, \ldots, a_k) \).
3. For some and any “fresh” name \( x \), if \( P_n \text{ } a \), then \( P_{n+1} \text{ } x.a \).

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.

Specializing this to arithmetic expressions as defined earlier, the principle of structural induction states that to show \( P \text{ } a \) for every \( a \text{ } \text{abt} \), it is enough to show the following facts:

1. If \( x \) name, then \( P \text{ } x \).
2. If \( n \) nat, then \( P \text{ } \text{num}[n] \).
3. If \( P \text{ } a_1 \) and \( P \text{ } a_2 \), then \( P \text{ } \text{plus}(a_1, a_2) \).
4. If \( P \text{ } a_1 \) and \( P \text{ } a_2 \), then \( P \text{ } \text{times}(a_1, a_2) \).
5. If \( P \text{ } a_1 \) and, for some/every \( x \) name, \( P \text{ } a_2 \), then \( P \text{ } \text{let}(a_1, x.a_2) \).

Here again the choice of bound variable name is irrelevant, provided that it is “fresh” in the sense of not clashing with any other name in \( P \).
6.6 Substitution

Substitution is the process of replacing free occurrences of a name with a specified abt (of valence 0). The judgement \([x \leftarrow a]b = b'\) abt states that \(c\) is the result of substituting \(a\) for all free occurrences of \(x\) in \(b\). It is inductively defined by the following rules:

\[
\begin{align*}
[x \leftarrow a]x &= a \text{ abt}^0 & [x \leftarrow a]y &= y \text{ abt}^0 \\
[x \leftarrow a]b_1 = b'_1 \text{ abt}^{n_1} & \cdots & [x \leftarrow a]b_k = b'_k \text{ abt}^{n_k} \\
[x \leftarrow a]o(b_1, \ldots, b_k) &= o(b'_1, \ldots, b'_k) \text{ abt}^0 \quad (\star) \\
x \# y \text{ name} & \quad y \# a \text{ abt}^0 \\
[x \leftarrow a]b &= c \text{ abt}^n \\
[x \leftarrow a]y.b &= y.c \text{ abt}^{n+1}
\end{align*}
\]

The apartness conditions on the last rule imply no loss of generality, because they can always be satisfied by appropriate choice of bound variable name, \(y\), in the target of the substitution.

Substitution defines a function up to \(\alpha\)-equivalence.

**Theorem 6.3**

1. If \(a \text{ abt}^0\), \(x\) name, and \(b \text{ abt}^n\), there exists \(b' \text{ abt}^n\) such that \(b =_\alpha b' \text{ abt}^n\) and \([x \leftarrow a]b' = c \text{ abt}^n\) for some \(c \text{ abt}^n\).

2. If \(a \text{ abt}^0\), \(x\) name, \(b =_\alpha b' \text{ abt}^n\), \([x \leftarrow a]b = c \text{ abt}^n\) and \([x \leftarrow a]b' = c' \text{ abt}^n\), then \(c =_\alpha c' \text{ abt}^n\).

6.7 Summary

Let us now take stock of what we have accomplished. The definition of \(\alpha\)-equivalence for abt’s makes clear the nature of bound names. Briefly, a bound name is merely a reference to a binding site; bound names have no intrinsic identity. This is enforced by treating abt’s modulo \(\alpha\)-equivalence, which is to say that we do not distinguish two abt’s \(a\) and \(b\) such that \(a =_\alpha b\).

The significance of this identification may be briefly summarized in several equivalent ways.

1. A bound variable name may always be chosen to be different from any given finite set of names. This is because one representative, \(a\), of an \(\alpha\)-equivalence class is as good as any other, \(b\).
2. It is illegitimate to rely upon the choice of a bound variable name, since it "changes under one’s feet" without explicit mention. This is just to say that a property of an \( \alpha \)-equivalence class is only well-defined if it respects \( \alpha \)-equivalence — that is, if its meaning is independent of the choice of representative.

3. One choice of bound variable name is as good as any other; if we do not like the one we have, we may rename it at will without changing the abt in any material way. That is, we may always replace an abt by an \( \alpha \)-equivalent one; they designate the same \( \alpha \)-equivalence class.

4. Bound variable names “automatically” evade confusion with any other variable name in a given context. This is because we may always choose another representative in the case that one choice is inconvenient in a given context.

We will freely make use of these and similar conveniences afforded by \( \alpha \)-equivalence throughout this book.

### 6.8 Exercises

1. Give a direct inductive definition of the judgements \( x \in a \) abt.

2. Give a proof that substitution defines a function up to \( \alpha \)-equivalence.

3. Give an inductive definition of *simultaneous substitution*,

\[
[x_1, \ldots, x_k \leftarrow a_1, \ldots, a_k]b = c \text{ abt},
\]

which states that \( c \) is the result of replacing all free occurrences of \( x_i \) by \( a_i \) in \( b \) (for each \( 1 \leq i \leq k \)).
Chapter 7

Specifying Syntax

Having defined the three levels of syntax, we may now summarize how we shall make use of them in the rest of this book. The focus of our work will be on the abstract syntax and binding structure of languages. We will not concern ourselves with the concrete syntax of the languages we consider. However, it will be necessary for us to write down examples, so we must establish conventions for defining the syntax of a language in a concise form. While the inference rule format always suffices, it is often more concise to use a modified form of grammar notation to define the abstract syntax and binding structure of a language.

This notation is best illustrated by example. Here is a presentation of the abstract syntax and binding structure of a language of expressions using grammar notation.

\[
\text{Types} \quad \tau ::= \text{num} | \text{str} \\
\text{Expr's} \quad e ::= x | \text{num}[n] | \text{str}[s] | \text{plus}(e_1,e_2) | \text{cat}(e_1,e_2) | \text{let}(e_1,x.e_2)
\]

The important point about this form of specification is that it specifies two categories of abstract binding trees, the *types* and the *terms*. The specification is given “by example”, using meta-variables that range over the syntactic categories to illustrate the pattern. In this case the meta-variable \( \tau \) ranges over the category of types, and the meta-variable \( e \) ranges over the category of expressions. In addition, the meta-variable \( x \) ranges over names of variables, and \( n \) ranges over natural numbers. (This convention is often used without explicit mention.)

The notation used in the grammar makes clear the intended binding and scope of variables. For example, it is clear from the notation used that
let is an operator with arity \((0, 1)\), specifying that it binds one variable in the second position. We take all such conventions to be tacitly understood without explicit mention, and treat all abt’s module \(a\)-equivalence, which ensures that the names of bound variables may be chosen at will.

For the sake of writing examples, we will sometimes take liberties with the syntax for the sake of readability, relying on conventions of concrete syntax that are familiar to the reader from other contexts. For example, we may introduce parentheses to emphasize grouping, or use infix or other standard forms of notation that enhance the readability of the examples without being explicit about the intended meaning. For example, we might write \(e_1 + e_2\) for \(\text{plus}(e_1, e_2)\), or \(\text{let } x \text{ be } e_1 \text{ in } e_2\) for \(\text{let}(e_1, x. e_2)\), leaving it to the reader to interpret these in the evident manner.
Part III

Static and Dynamic Semantics
Chapter 8

The Phase Distinction

We will distinguish two *phases* of processing, the *static phase* and the *dynamic phase*. Roughly speaking, the static phase consists of ensuring that the program is well-formed, and the dynamic phase consists of executing well-formed programs. Depending on the level of detail one wishes to consider, one may make finer distinctions, considering, for example, parsing to be a separate phase from code generation, or linking to be separate from execution.

We will, in fact, find it useful to draw such fine distinctions later in the development, but before we can do that we must first answer these two fundamental questions:

1. Which are the well-formed programs?

2. What is the execution behavior of a well-formed programs?

The first question is answered by the *static semantics* of a language, and the second is answered by its *dynamic semantics*.

The central organizing principle for static semantics is the concept of a *type*. A type is characterized by two, closely related notions:

1. The primitive operations that *create*, or *introduce*, values of that type.

2. The primitive operations that *compute with*, or *eliminate*, values of that type.

The terminology of introduction and elimination is rooted in history, and over time the words have sometime acquired other meanings. In particular, please note that the “elimination” operations having nothing to do with storage reclamation!
As an illustrative example, the type of natural numbers may be characterized by the following primitive notions:

1. The primitives zero and succ(−) for introducing natural numbers.

2. Operations such as addition and multiplication for computing with natural numbers, or, more generally, the ability to define a function by induction on the natural numbers.

Anything we wish to do with natural numbers can be accomplished using these operations alone. So, in particular, we never need to know what the natural numbers “really are” — they are an abstraction characterized by the introduction and elimination operations on them.

The dynamic semantics of a language determines how to execute the programs given by the static semantics. The key to the dynamic semantics is the inverse principle, which states that the elimination operations are inverse to the introduction operations. (Here again the terminology cannot be taken too seriously, but is only suggestive of the general idea.) The elimination operations compute with the values created by the introduction operations, taking apart what was introduced in order to obtain their result. Since the elimination operations take apart what the introduction operations put together, they may be seen as a kind of inverse relationship between them.

An example will help make this clear. The addition operation takes as input two natural numbers and computes their sum. Looking at, say, the first of the two numbers, it can have been created in only one of two ways. Either it is zero, in which case the sum is the second of the two numbers. Otherwise, it is succ(x), in which case the sum is the successor of the sum of x and the second number. Thus we see that addition “takes apart” what the introduction operations created in order to compute its result.

Interesting languages have more than one type. For example, we might also have a type string of strings of characters whose introduction operations include character strings enclosed in quotation marks, and whose elimination operations might include string concatenation and a length calculation. Once we have more than one type, we then have the potential for a type error, a combination of operations that violates the type structure. For example, it makes no sense to add a number to a string, or to concatenate two numbers. The role of a static semantics is to ensure that such erroneous programs are ruled out of consideration so that the dynamic semantics may concern itself only with well-formed combinations such as additions of two numbers or concatenations of two strings, and never have to be concerned with giving meaning to ill-typed phrases.
This leads to the central concept of type safety, which ensures that the static semantics and the dynamic semantics “fit together” properly. More precisely, we may think of the static semantics as imposing strictures that are tacitly assumed to hold by the dynamic semantics. But what if they don’t? How do we know that the assumptions of the dynamic semantics match the strictures imposed by the static semantics? The answer is that we do not know this until we prove it. That is, we must prove a theorem, called the type safety theorem, that states that the static and dynamic semantics cohere. In practical terms this ensures that a whole class of run-time faults, which manifest themselves as “bus errors” or “core dumps” in familiar languages, cannot arise. One theorem about a language implies infinitely many theorems, one for each program written in it.

This establishes the pattern for the remainder of this book. Programming languages are organized as a collection of types — the “features” of the language emerge as the operations associated with a particular type. These types are given meaning by the static and dynamic semantics, and we ensure that the whole is well-defined by proving type safety. This simple methodology is surprisingly powerful, both as a tool for language design and as a tool for language implementation — for the theory and practice of programming languages.
Chapter 9

Static Semantics

In this chapter we will illustrate the specification of a static semantics for a simple language of expressions defined as follows:

Types \( \tau \) := num | str

Expr’s \( e \) := \( x \) | \( \text{num}[n] \) | \( \text{str}[s] \) | \( \text{plus}(e_1, e_2) \) | \( \text{cat}(e_1, e_2) \) | \( \text{let}(e_1, x, e_2) \)

The introduction forms for \( \text{num} \) are the numerals, \( \text{num}[n] \), and the elimination form is \( \text{plus}(e_1, e_2) \). The introduction forms for \( \text{str} \) are the string literals, \( \text{str}[s] \), and the elimination form is \( \text{cat}(e_1, e_2) \). Finally, we have a variable binding construct, \( \text{let}(e_1, x, e_2) \).

9.1 Static Semantics of Expressions

The static semantics is defined by the typing judgement \( e : \tau \), where \( \tau \) is a type (either \( \text{num} \) or \( \text{str} \)), and \( e \) is an expression, which is an abstract binding tree over the preceding signature. The typing judgement is defined using higher-order judgement forms of the kind considered in Chapter 2, with an explicit representation of the typing hypotheses. We write \( \Gamma \vdash e : \tau \), where \( \Gamma \) is of the form \( x_1 : \tau_1, \ldots, x_n : \tau_n \) with no two \( x_i \)'s being equal, and the free names occurring in \( e \) are among the set \( x_1, \ldots, x_n \).

The static semantics of expressions is inductively defined by the follow-

\(^1\)Officially, the turnstile is indexed by this set of variables, but we omit explicit mention of this in our notation for typing judgements.
9.2 Properties of the Static Semantics

The rule for variables is implicit in the meaning of the hypothetical judgement:

\[ \Gamma, x : \tau \vdash x : \tau \]

We often state this rule explicitly for the sake of emphasis.

In the rule for let’s we tacitly assume that \( x \) is not otherwise declared in \( \Gamma \); this may always be achieved by suitable renaming of bound variables. Also, if \( x : \tau \) occurs anywhere in \( \Gamma \), we may tacitly regard \( \Gamma \) as having the form \( \Gamma', x : \tau \), in which the designated declaration occurs “last”. This is justified by the admissibility of permutation for the hypothetical judgement form, as discussed in Chapter 2.

**9.2 Properties of the Static Semantics**

The structural rules governing the hypothetical judgement ensure that the typing judgement obeys the following weakening and substitution properties:

**Theorem 9.1 (Structural Properties of Typing)**

1. If \( \Gamma \vdash e' : \tau' \), then \( \Gamma, x : \tau \vdash e' : \tau' \), provided that \( x \) is not already declared in \( \Gamma \).

2. If \( \Gamma, x : \tau \vdash e' : \tau' \) and \( \Gamma \vdash e : \tau \), then \( \Gamma \vdash [x \leftarrow e] e' : \tau' \).

**Proof:**

1. By induction on the derivation of \( \Gamma \vdash e' : \tau' \).

2. By induction on the derivation of \( \Gamma, x : \tau \vdash e' : \tau' \).
The typing rules are syntax-directed in the sense that there is exactly one rule for each form of expression. Consequently, we obtain the following inversion principles for typing.

**Theorem 9.2 (Inversion for Typing)**

If $\Gamma \vdash e : \tau$, then

1. if $e = x$ for some variable $x$, then $\Gamma = \Gamma', x : \tau$.
2. if $e = \text{plus}(e_1, e_2)$, then $\tau = \text{num}$, $e_1 : \text{num}$, and $e_2 : \text{num}$.
3. if $e = \text{cat}(e_1, e_2)$, then $\tau = \text{str}$, $e_1 : \text{str}$, and $e_2 : \text{str}$.
4. if $e = \text{let}(e_1, x.e_2)$, then for some $\tau_1$, $e_1 : \tau_1$, and $\Gamma, x : \tau_1 \vdash e_2 : \tau$.

**Proof:** By induction on the derivation of $\Gamma \vdash e : \tau$. ■

A value, $v$, is either num[$n$] for some $n$ nat or str[$s$] for some $s$ string. We may characterize the closed values of a type as follows.

**Theorem 9.3 (Canonical Forms)**

If $v : \tau$, where $v$ is a value, then

1. If $\tau = \text{num}$, then $v = \text{num}[n]$ for some natural number $n$.
2. If $\tau = \text{str}$, then $v = \text{str}[s]$ for some string $s$.

**Proof:** By induction on the derivation of $v : \tau$, taking account of the fact that $v$ is a value. ■

### 9.3 Exercises

1. Show that the expression $e = \text{plus}(\text{num}[7], \text{str}[abc])$ is ill-typed in that there is no $\tau$ such that $e : \tau$.
2. Show that if $\Gamma \vdash e : \tau$ and $x \in e$, then $\Gamma = \Gamma', x : \tau$ for some $\tau$. 

FEBRUARY 4, 2006 WORKING DRAFT
Chapter 10

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 relation between a phrase and its value, without detailing how it is to be determined in a step-by-step manner. Each presentation has its uses, so we discuss both forms of dynamic semantics, as well as their relation to one another.

10.1 Structured Operational Semantics

A structured operational semantics for a language consists of an inductively defined transition system whose states are closed, well-formed expressions. Every state is an initial state, and the final states are the values, defined by the following rules:

<table>
<thead>
<tr>
<th>num[n] value</th>
<th>str[s] value</th>
</tr>
</thead>
</table>

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

What if the principal arguments have not (yet) been fully evaluated? Then we must evaluate them! In the case of 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 judgement is defined so that one step of evaluation is made at a time, reconstructing the entire expression as necessary.
10.2 Evaluation Semantics

For example, consider the following evaluation sequence.

\[
\begin{align*}
\text{let}(\text{plus}(\text{num}[1],\text{num}[2]), x.\text{plus}(\text{plus}(x,\text{num}[3]), \text{num}[4])) & \rightarrow \text{let}(\text{num}[3], x.\text{plus}(\text{plus}(x,\text{num}[3]), \text{num}[4])) \\
& \rightarrow \text{plus}(\text{plus}(\text{num}[3], \text{num}[3]), \text{num}[4]) \\
& \rightarrow \text{plus}(\text{num}[6], \text{num}[4]) \\
& \rightarrow \text{num}[10]
\end{align*}
\]

Each step is justified by a rule defining the transition judgement. 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

\[
\text{plus}(\text{num}[1], \text{num}[2]) \rightarrow \text{num}[3],
\]

which is justified by the first instruction rule defining the transition judgement. Each of the subsequent steps is justified similarly.

Observe that the expression \(e = \text{cat}(\text{num}[3], \text{str}[abc])\) is not a final state, but there is no \(e'\) such that \(e \rightarrow e'\) — it is a stuck state. Fortunately it is also ill-typed! We shall prove in the next chapter that no well-typed expression is stuck.

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

10.2 Evaluation Semantics

Another method for defining the dynamic semantics of a language, called evaluation semantics, consists of an inductive definition of the evaluation judgement, \(e \Downarrow v\), specifying the value, \(v\), of a closed expression, \(e\). This
10.3 Relating Transition and Evaluation Semantics

The value of a λet 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 λet, the rules are not syntax-directed.

Since the evaluation judgement is inductively defined, it has associated with it a principle of proof by rule induction. Specifically, to show that the property \( P(e, v) \) holds, it is enough to show that \( P \) is closed under the rules defining the evaluation judgement. Specifically, our proof obligations are:

1. Show that \( P(\text{num}[n], \text{num}[n]) \).
2. Show that \( P(\text{str}[s], \text{str}[s]) \).
3. Show that \( P(\text{plus}(e_1, e_2), \text{num}[n]), \) assuming \( n = n_1 + n_2 \), \( P(e_1, \text{num}[n_1]) \) and \( P(e_2, \text{num}[n_2]) \).
4. Show that \( P(\text{plus}(e_1, e_2), \text{str}[s]), \) assuming \( s = s_1 \cdot s_2 \), \( P(e_1, \text{str}[s_1]) \) and \( P(e_2, \text{str}[s_2]) \).
5. Show that \( P(\text{let}(e_1, x.e_2), v_2), \) assuming \( P(e_1, v_1) \) and \( P([x←v_1]e_2, v_2) \).

This induction principle is not the same as structural induction on \( e \), because the evaluation rules are not syntax-directed.

10.3 Relating Transition and Evaluation Semantics

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

**Theorem 10.1**
For all closed expressions $e$ and natural numbers $n$, $e \mapsto^\dagger \text{num}[n]$ iff $e \Downarrow \text{num}[n]$.

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

**Lemma 10.2**
If $e \Downarrow \text{num}[n]$, then $e \mapsto^\dagger \text{num}[n]$.

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

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

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

For the converse, recall from Chapter 3 the definitions of multi-step evaluation and complete evaluation. Since $\text{num}[n] \Downarrow \text{num}[n]$, it suffices to show that evaluation is closed under head expansion.

**Lemma 10.3**
If $e \mapsto e'$ and $e' \Downarrow \text{num}[n]$, then $e \Downarrow \text{num}[n]$.

**Proof:** By induction on the definition of the transition judgement. For example, suppose that $\text{plus}(e_1, e_2) \mapsto \text{plus}(e'_1, e_2)$, where $e_1 \mapsto e'_1$. Suppose further that $\text{plus}(e'_1, e_2) \Downarrow \text{num}[n]$, so that $e'_1 \Downarrow \text{num}[n_1]$, and $e_2 \Downarrow \text{num}[n_2]$ and $n = n_1 + n_2$. By induction $e_1 \Downarrow \text{num}[n_1]$, and hence $\text{plus}(e_1, e_2) \Downarrow \text{num}[n]$, as required. ■
10.4 Environment Semantics

Both the transition semantics and the evaluation semantics given earlier rely on substitution to replace let-bound variables by their bindings during evaluation. This approach maintains the invariant that only closed expressions are ever considered, and, as we shall see in the next chapter, facilitates proving properties of the language. However, in practice, we do not perform substitution, but rather record the bindings of variables in some sort of data structure. In this section we show how this can be elegantly modeled using hypothetical judgements.

The basic idea is to consider hypotheses of the form \( x \Downarrow v \), where \( x \) is a variable and \( v \) is a value, such that no two hypotheses govern the same variable. Let \( \eta \) range over finite sets of such hypotheses, which we call an environment. We will consider judgements of the form \( \eta \vdash X e \Downarrow v \), where \( X \) is the finite set of variables appearing on the left of a hypothesis in \( \eta \). As usual, we will suppress explicit mention of the parameter set \( X \), and simply write \( \eta \vdash e \Downarrow v \). The rules defining this judgement are as follows:

\[
\begin{align*}
\eta, x & \vdash x \Downarrow v \\
\eta \vdash e_1 \Downarrow \text{num}[n_1] & \quad \eta \vdash e_2 \Downarrow \text{num}[n_2] \\
\eta \vdash \text{plus}(e_1, e_2) \Downarrow \text{num}[n_1 + n_2] & \\
\eta \vdash e_1 \Downarrow \text{str}[s_1] & \quad \eta \vdash e_2 \Downarrow \text{str}[s_2] \\
\eta \vdash \text{cat}(e_1, e_2) \Downarrow \text{num}[s_1^s_2] & \\
\eta \vdash e_1 \Downarrow v_1 & \quad \eta, x \Downarrow v_1 \vdash e_2 \Downarrow v_2 \\
\eta \vdash \text{let}(e_1, x. e_2) \Downarrow v_2
\end{align*}
\]

The variable rule is an instance of the reflexivity rule for hypothetical judgements, and therefore need not be explicitly stated. We nevertheless include it here for clarity. The let rule augments the environment with a new assumption governing the bound variable (which, by \( \alpha \)-conversion, may be chosen to be distinct from any other variable currently in \( \eta \) to preserve the invariant that no two assumptions govern the same variable).

The environment semantics is related to the evaluation semantics by the following theorem:

**Theorem 10.4**

\( x_1 \Downarrow v_1, \ldots, x_n \Downarrow v_n \vdash e \Downarrow v \) iff \( [x_1, \ldots, x_n \leftarrow v_1, \ldots, v_n]e \Downarrow v \).

**Proof:** The left to right direction is proved by induction on the rules defining the evaluation semantics, making use of the definition of substitution and the definition of the evaluation semantics for closed expressions. The
converse is proved by induction on the structure of $e$, again making use of the definition of substitution. Note that we must induct on $e$ in order to detect occurrences of variables $x_i$ in $e$, which are governed by a hypothesis in the environment semantics.

10.5 Exercises

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

2. Prove that if $e \Downarrow v$, then $v$ value.

3. Prove that if $e \Downarrow v_1$ and $e \Downarrow v_2$, then $v_1 \equiv v_2$.

4. Complete the proof of equivalence of evaluation and transition semantics.

5. Is it possible to use environments in a structured operational semantics? What difficulties do you encounter?
Chapter 11

Type Safety

Many programming languages, including 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. What is remarkable is that we will be able to clarify the idea of type safety without making reference to an implementation. Consequently, the notion of type safety is extremely robust — it is shared by all correct implementations of the language.

Type safety states that the static and dynamic semantics of a language cohere in that the strictures of the type system ensure that execution is well-behaved. Simply put, the type system ensures that evaluation cannot “go off into the weeds” into an ill-defined state for which no definite result can be obtained. This is proved by showing that a transition from a well-defined state leads only to well-defined states, and that if a state is well-defined then it is either in a valid final state, or is capable of making a transition. The static semantics specifies what we mean by well-defined, and the dynamic semantics specifies what it means to make a transition. This leads to the following formal statements that, together, express the safety of the language:

1. **Preservation**: If $e : \tau$ and $e \rightarrow e'$, then $e' : \tau$.

2. **Progress**: If $e : \tau$, then either $e$ value, or there exists $e'$ such that $e \rightarrow e'$.

The first says that the steps of evaluation preserve well-typedness (indeed, preserves typing), and the second says that well-typedness ensures that either we are done or we can make progress towards completion.
11.1 Preservation for Expressions

The preservation theorem is proved by rule induction on the definition of the transition system of expressions.

**Theorem 11.1 (Preservation)**
If \( e : \tau \) and \( e \xrightarrow{} e' \), then \( e' : \tau \).

**Proof:** Consider the rule

\[
\frac{e_1 \xrightarrow{} e'_1}{\text{plus}(e_1, e_2) \xrightarrow{} \text{plus}(e'_1, e_2)}.
\]

Assume that \( \text{plus}(e_1, e_2) : \tau \). By inversion for typing, we have that \( \tau = \text{num} \), \( e_1 : \text{num} \), and \( e_2 : \text{num} \). By induction we have that \( e'_1 : \text{num} \), and hence \( \text{plus}(e_1, e_2) : \text{num} \). The cases for multiplication and concatenation are handled similarly.

Now consider the rule

\[
\frac{\text{let}(e_1, x.e_2) \xrightarrow{} [x\leftarrow e_1]e_2}{\text{let}(e_1, x.e_2) \xrightarrow{} [x\leftarrow e_1]e_2}.
\]

Assume that \( \text{let}(e_1, x.e_2) : \tau_2 \). By inversion for typing, \( e_1 : \tau_1 \) for some \( \tau_1 \) such that \( x : \tau_1 \vdash e_2 : \tau_2 \). By substitution \( [x\leftarrow e_1]e_2 : \tau_2 \), as desired.

We leave the remaining cases to the reader. \( \blacksquare \)

The proof of preservation must proceed by rule induction on the rules defining the transition judgement. It cannot, for example, proceed by induction on the structure of \( e \), for in most cases there is more than one transition rule for each expression form. Nor can it be proved by induction on the typing rules, for in the case of the \text{let} rule, the context is enriched to consider an open term, to which no dynamic semantics is assigned.

11.2 Progress for Expressions

The progress theorem captures the idea that well-typed programs cannot "get stuck".

**Theorem 11.2 (Progress)**
If \( e : \tau \), then either \( e \) value, or there exists \( e' \) such that \( e \xrightarrow{} e' \).
Proof: The proof is by induction on the typing derivation. The rule for variables cannot arise, because we are only considering closed typing judgments. Consider the typing rule

\[
\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num} \\
\Gamma \vdash \text{plus}(e_1, e_2) : \text{num}
\]

where \(\Gamma\) is empty. By induction we have that either \(e_1\) value, or there exists \(e'_1\) such that \(e_1 \rightarrow e'_1\). In the latter case it follows that \(\text{plus}(e_1, e_2) \rightarrow \text{plus}(e'_1, e_2)\), as required. In the former we also have by induction that either \(e_2\) value, or there exists \(e'_2\) such that \(e_1 \rightarrow e'_1\). In the latter case we have that \(\text{plus}(e_1, e_2) \rightarrow \text{plus}(e_1, e'_2)\), which is enough. In the former case we have, by canonical forms, that \(e_1 = \text{num}[n_1]\) and \(e_2 = \text{num}[n_2]\), and hence \(\text{plus}(\text{num}[n_1], \text{num}[n_2]) \rightarrow \text{num}[n_1 + n_2]\).

The other cases are handled similarly, and are left to the reader. ■

Since the typing rules for expressions are syntax-directed, the progress theorem could equally well be proved by induction on the structure of \(e\), appealing to the inversion theorem at each step to characterize the types of the parts of \(e\). But this approach breaks down when the typing rules are no longer syntax-directed, that is, when there may be more than one rule for a given expression form. In such cases it becomes clear that the most direct approach is to consider the typing rules one-by-one.

Summing up, the combination of preservation and progress together constitute a proof of safety. The progress theorem ensures that well-typed expressions do not “get stuck” in an ill-defined state, and the preservation theorem ensures that if a step is taken, the result remains well-typed (with the same type). Thus the two parts work hand-in-hand to ensure that the static and dynamic semantics are coherent, and that no ill-defined states can ever be encountered while evaluating a well-typed expression.

### 11.3 Exercises

1. Complete the proof of preservation.
2. Complete the proof of progress.
3. Do something similar, in detail.
Part IV

A Functional Language
Chapter 12

A Functional Language

The \( \lambda \)-calculus is a fundamental building block in the study of programming language concepts. In contrast to machine models, such as Turing machines or random-access machines, the \( \lambda \)-calculus is a linguistic foundation for computation that takes as primitive the notion of a function. In its barest form the entire language consists of nothing but functions — even data structures arise as functions in the \( \lambda \)-calculus!

Although elegant in its spartan simplicity, the \( \lambda \)-calculus is remarkably subtle and is best approached gradually as the culmination of the development of several key ideas. We will therefore begin our study with a simple functional language that, like the \( \lambda \)-calculus, takes the notion of function as a starting point, but which, unlike the \( \lambda \)-calculus, is not the only concept in the language. Later on we will see how the pure \( \lambda \)-calculus may be reconstructed on this foundation once we have developed a few more ideas.

12.1 Syntax

The syntax of MinML is given as follows:

\[
\begin{align*}
\text{Types} & \quad \tau : := \text{nat} \mid \text{arrow}(\tau_1, \tau_2) \\
\text{Expr's} & \quad e : := x \mid \text{num}[n] \mid \text{plus}(e_1, e_2) \mid \text{times}(e_1, e_2) \\
& \quad \quad \quad \mid \text{ifz}(e, e_1, e_2) \mid \text{lambda}(\tau, x. e) \mid \text{app}(e_1, e_2) \\
& \quad \quad \quad \mid \text{let}(\tau, e_1, x. e_2)
\end{align*}
\]

As discussed in Chapter 7, this grammar implicitly specifies a signature that determines the set of abt's described by the above grammar.
The constructs of the language may be classified by type. Associated with the type nat are the numerals, num\[n\], the arithmetic operations, \texttt{plus}(e_1, e_2) and \texttt{times}(e_1, e_2), and the zero-test, \texttt{ifz}(e, e_1, e_2). Associated with the function type \texttt{arrow}(\tau_1, \tau_2) are the \texttt{\lambda}-abstractions, \texttt{lambda}(\tau, x.e), and the applications, \texttt{app}(e_1, e_2). Finally, we have a generic construct for binding expressions to names, \texttt{let}(\tau, e_1, x.e_2).

### 12.2 Static Semantics

The typing judgement, \( e : \tau \), states that expression \( e \) has type \( \tau \). More generally, we will consider hypothetical judgements of the form

\[
\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau,
\]

stating that \( e \) is of type \( \tau \) under the assumptions that each variable \( x_i \) is of type \( \tau_i \). We let \( \Gamma \) stand for any such sequence of typing assumptions.

The typing judgement is inductively defined by the following rules:

\[
\begin{align*}
& \frac{}{\Gamma, x : \tau \vdash x : \tau} \quad (12.1) \\
& \frac{}{\Gamma \vdash \text{num}[n] : \text{nat}} \quad (12.2) \\
& \frac{\Gamma \vdash e_1 : \text{nat} \quad \Gamma \vdash e_2 : \text{nat}}{\Gamma \vdash \text{plus}(e_1, e_2) : \text{nat}} \quad (12.3) \\
& \frac{\Gamma \vdash e_1 : \text{nat} \quad \Gamma \vdash e_2 : \text{nat}}{\Gamma \vdash \text{plus}(e_1, e_2) : \text{nat}} \quad (12.4) \\
& \frac{\Gamma \vdash e : \text{nat} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{ifz}(e, e_1, e_2) : \tau} \quad (12.5) \\
& \frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \text{lambda}(\tau, x.e) : \text{arrow}(\tau_1, \tau_2)}{\Gamma \vdash \text{lambda}(\tau, x.e) : \text{arrow}(\tau_1, \tau_2)} \quad (12.6) \\
& \frac{\Gamma \vdash e_1 : \text{arrow}(\tau_2, \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \text{app}(e_1, e_2) : \tau} \quad (12.7)
\end{align*}
\]
12.3 Basic Properties of the Static Semantics

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 numeric constants and numeric primitives, one rule for the conditional, and so forth. The typing rules are therefore said to be *syntax-directed* in that the form of the expression determines the only possible typing rule that may apply to it. This observation leads to a useful lemma, called the *inversion lemma*, which states that the typing rules are necessary, as well as sufficient.

**Theorem 12.1 (Inversion)**

Suppose that $\Gamma \vdash e : \tau$.

1. If $e = x$, then $\Gamma = \Gamma', x : \tau$.

2. If $e = \text{num}[n]$, then $\tau = \text{nat}$.

3. If $e = \text{plus}(e_1, e_2)$ or $e = \text{times}(e_1, e_2)$, then $\tau = \text{nat}$ and $\Gamma \vdash e_1 : \text{nat}$ and $\Gamma \vdash e_2 : \text{nat}$.

4. If $e = \text{ifz}(e_0, e_1, e_2)$, then $\Gamma \vdash e_0 : \text{nat}$, $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$.

5. If $e = \lambda \text{mda}(\tau_1, x.e)$, then $\tau = \text{arrow}(\tau_1, \tau_2)$ and $\Gamma, x : \tau_1 \vdash e : \tau_2$.

6. If $e = \text{app}(e_1, e_2)$, then there exists $\tau_2$ such that $\Gamma \vdash e_1 : \text{arrow}(\tau_2, \tau)$ and $\Gamma \vdash e_2 : \tau_2$.

7. If $e = \text{let}(\tau_1, e_1, x, e_2)$, then $\Gamma \vdash e_1 : \tau_1$, $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$, and $\tau = \tau_2$.

**Proof:** The proof proceeds by rule induction on the typing rules. Observe that for each rule, exactly one case applies, and that the premises of the rule in question provide the required result. ■

The following substitution property for typing follows immediately from the meaning of the hypothetical judgement.
Lemma 12.2
1. If $\Gamma, x : \tau \vdash e' : \tau'$, and $\Gamma \vdash e : \tau$, then $\Gamma \vdash [x \leftarrow e]e' : \tau'$.

12.4 Dynamic Semantics

The dynamic semantics of MinML is given by a transition system whose states are closed expressions. All states are initial, and the final states are the values, inductively defined by the following axioms:

<table>
<thead>
<tr>
<th>$\text{num}[n]$ value</th>
<th>$\text{lambda}(\tau, x.e)$ value</th>
</tr>
</thead>
</table>

We often use the meta-variable $v$ in situations where we expect $v$ to be an expression such that $v$ value.

The transition judgement is inductively defined by the following rules.

\[
\begin{align*}
    e_1 \mapsto e'_1 & \quad \text{plus}(e_1, e_2) \mapsto \text{plus}(e'_1, e_2) \\
    v_1 \text{ value} e_2 \mapsto e'_2 & \quad \text{plus}(v_1, e_2) \mapsto \text{plus}(v_1, e'_2) \\
    (n = n_1 + n_2) & \quad \text{plus}(\text{num}[n_1], \text{num}[n_2]) \mapsto \text{num}[n] \\
    e_0 \mapsto e'_0 & \quad \text{ifz}(e_0, e_1, e_2) \mapsto \text{ifz}(e'_0, e_1, e_2) \\
    \text{ifz}(\text{num}[0], e_1, e_2) \mapsto e_1 & \quad (n \neq 0) \quad \text{ifz}(\text{num}[n], e_1, e_2) \mapsto e_2 \\
    e_1 \mapsto e'_1 & \quad \text{app}(e_1, e_2) \mapsto \text{app}(e'_1, e_2) \\
    v_1 \text{ value} e_2 \mapsto e'_2 & \quad \text{app}(v_1, e_2) \mapsto \text{app}(v_1, e'_2) \\
    \text{v value} & \quad \text{app}(\text{lambda}(\tau, x.e), v) \mapsto [x \leftarrow v]e \\
    e_1 \mapsto e'_1 & \quad \text{let}(\tau_1, e_1, e_2) \mapsto \text{let}(\tau_1, e'_1, e_2) \\
    v_1 \text{ value} & \quad \text{let}(\tau_1, v_1, e_2) \mapsto [x \leftarrow v_1]e_2
\end{align*}
\]

(The rules for multiplication are very similar to those for addition, and are omitted here.)

Observe that the argument of a function must be simplified to a value before the application can occur. This is called the call-by-value evaluation.
strategy for function applications. The alternative, known as call-by-name for historical reasons, is to pass the argument to the function in unevaluated form, so that it is evaluated only if its value is actually necessary to compute the value of the call. This can be more efficient, because the argument is not evaluated unless it is needed, but it can also be less efficient, because the argument is repeatedly evaluated on each use.

12.5 Basic Properties of the Dynamic Semantics

Let us prove that evaluation is deterministic, which implies that the value of any expression, if it has one, is uniquely determined by the expression alone. In other words, the transition judgement has mode \((\forall, \exists^1)\).

Lemma 12.3

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

Proof: By induction on the structure of \(e\). For example, if \(e = \text{app}(e_1, e_2)\), then by induction applied to \(e_1\), there is at most one \(e'_1\) such that \(e_1 \rightarrow e'_1\). If such a transition is possible, then \(e \rightarrow \text{app}(e'_1, e_2)\), and this is the only possible transition. Otherwise, if \(e_1\) is not a value, there is no transition from \(e\). If \(e_1\) is a value, then there is at most one transition \(e_2 \rightarrow e'_2\). If there is such a transition, then \(e \rightarrow \text{app}(e_1, e'_2)\), because \(e_1\) value. If not, then \(e_1\) may or may not be a value. If not, there is no transition from \(e\). If so, there is at most one transition, according to whether or not \(e_1\) is a function. The other cases are handled similarly. ■

12.6 Iteration and Recursion

So far MinML is extremely weak because it lacks any form of iteration or recursion, which is necessary even to express the most rudimentary computations on natural numbers. For example, not even the factorial function is definable without some additional machinery!

One extension, which corresponds to “for” loops in familiar programming languages, is to replace the ifz construct by a more general iteration construct that allows us to perform an operation \(n\) times for any natural number \(n\). This is achieved using an iterator, whose syntax is

\[
\text{rec}(\tau, e_0, e_1, x.e_2).
\]
We may explain the meaning of this construct informally as follows. Let $n_0$ be the value of $e_0$, a natural number. If $n_0$ is 0, then the result is the value of $e_1$. Otherwise, $n_0 = n'_0 + 1$, and we recursively evaluate the same iterator on $n'_0$, then substitute this result for $x$ in $e_2$ to obtain the final result. Put in other terms, if $n \geq 0$, we iterate $e_2$ for $n$ times, starting with $e_1$.

To make this precise, here is the typing rule for the iterator:

$$
\Gamma \vdash e_0 : \text{nat} \quad \Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau \vdash e_2 : \tau
$$

$$
\Gamma \vdash \text{rec}(\tau, e_0, e_1, x.e_2).
$$

As with the conditional, both “branches” must have the same type, $\tau$.

The dynamic semantics of the iterator is given by the following rules:

$$
e_0 \mapsto e'_0
$$

$$
\text{rec}(\tau, e_0, e_1, x.e_2) \mapsto \text{rec}(\tau, e'_0, e_1, x.e_2)
$$

$$
\text{rec}(\tau, \text{num}[0], e_1, x.e_2) \mapsto e_1
$$

$$
\text{rec}(\tau, \text{num}[n + 1], e_1, x.e_2) \mapsto \text{let}(\tau, \text{rec}(\tau, \text{num}[n], e_1, x.e_2), x.e_2)
$$

The use of the let in the third rule ensures that the recursive call is evaluated before it is passed to $e_2$.

A significantly more powerful extension is to admit recursive functions into the language. These are functions that may “call themselves” recursively, on any argument we wish to pass. Such functions are not guaranteed to terminate, but this is the price we pay for generality. In order for a function to call itself, it must have a name for itself. This can be arranged by generalizing the function abstraction to give itself a name:

$$
\text{fun}(\tau_1, \tau_2, f.x.e).
$$

The variable $f$ stands for the function itself, and the variable $x$ stands for its argument. If the function $e$ does not call itself, then the name $f$ is superfluous, and may be omitted by writing “_” in place of $f$.

The static semantics for recursive functions is defined so that the function may call itself under the assumption that it has the type it will turn out to have, namely $\text{arrow}(\tau_1, \tau_2)$:

$$
\Gamma, f : \text{arrow}(\tau_1, \tau_2), x : \tau_1 \vdash e : \tau_2
$$

$$
\Gamma \vdash \text{fun}(\tau_1, \tau_2, f.x.e) : \text{arrow}(\tau_1, \tau_2)
$$
12.7 Exercises

Notice that the typing rule is seemingly “circular” in that the assumption governing \( f \) states that it has the type of the function itself, which is checked by asserting that the body has type \( \tau_2 \) under this assumption and the additional assumption that the argument has type \( \tau_1 \).

The dynamic semantics for function applications changes so as to replace the name, \( f \), of the function itself by the function itself, thereby “tying the knot” in the recursion:

\[
\text{app}(v_1, v_2) \mapsto [f, x \leftarrow v_1, v_2]e \quad (v_1 = \text{fun}(\tau_1, \tau_2, f, x, e))
\]

The use of a variable to stand for the function itself is a common “trick” in programming languages. For example, in Java the identifier this stands for the object itself in exactly the same sense.

12.7 Exercises

1. Show that the mode of the typing judgement is \((\forall, \exists^{\leq 1})\) — for every (closed) expression there is at most one type for it. To prove this you must generalize the induction hypothesis to account for open expressions.

2. Formulate the call-by-name evaluation strategy for function applications in which arguments are passed unevaluated to functions.

3. Show that \( \text{let}(\tau_1, e_1, x. e_2) \) is definable in the sense that there is a translation of this construct in terms of the other constructs in the language such that its typing rule is derivable under this translation.

4. Define the evaluation judgement \( e \Downarrow v \), where \( e \) is a closed expression and \( e \) value, and show that \( e \Downarrow v \) iff \( \mapsto^{*} ev \).

5. Show that the conditional \( \text{ifz}(e_0, e_1, e_2) \) is definable in terms of the iterator.

6. Show that addition and multiplication are definable in terms of the iterator by giving a term \( e : \text{arrow}(\text{nat}, \text{arrow}(\text{nat}, \text{nat})) \) that implements these two arithmetic operations.

7. Show that the predecessor is definable in terms of the iterator, provided that we define the predecessor of 0 to be 0 and the predecessor of \( n + 1 \) to be \( n \).
8. Investigate the trade-offs between the call-by-name and call-by-value evaluation strategies for function applications.
Chapter 13

Type Safety for MinML

We will use the methodology described in Chapter 11 to prove the type safety of MinML. As discussed there, type safety is the combination of two key relationships between the static and dynamic semantics of the language, preservation and progress.

13.1 Safety for MinML

Theorem 13.1 (Preservation)
If $e : \tau$ and $e \rightarrow e'$, 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 rule induction on the definition of one-step evaluation. We will consider each rule in turn.

Consider the rule

\[
\begin{array}{c}
  e_1 \rightarrow e'_1 \\
  \text{plus}(e_1, e_2) \rightarrow \text{plus}(e'_1, e_2).
\end{array}
\]

Assume that $\text{plus}(e_1, e_2) : \tau$. By inversion $\tau = \text{nat}$, $e_1 : \text{nat}$, and $e_2 : \text{nat}$. By induction $e'_1 : \text{nat}$, and hence $\text{plus}(e'_1, e_2) : \text{nat}$, as was to be shown.

Consider the rule

\[
\begin{array}{c}
  v_1 \text{ value } e_2 \rightarrow e'_2 \\
  \text{plus}(v_1, e_2) \rightarrow \text{plus}(v_1, e'_2).
\end{array}
\]

Suppose that $\text{plus}(v_1, e_2) : \tau$. Then, by inversion, $\tau = \text{nat}$, $v_1 : \text{nat}$, and $e_2 : \text{nat}$. By induction $e'_2 : \text{nat}$, and hence $\text{plus}(v_1, e'_2) : \text{nat}$, as required.
Consider the rule
\[
(n = n_1 + n_2) \\
\text{plus}(\text{num}[n_1], \text{num}[n_2]) \longrightarrow \text{num}[n].
\]

Assume that \(\text{plus}(\text{num}[n_1], \text{num}[n_2]) : \tau\). Clearly, \(\text{num}[n] : \text{nat}\), which is enough for the result.

The rules governing multiplication are handled similarly.

Consider the rule
\[
e_0 \longmapsto e'_0 \\
\text{ifz}(e_0, e_1, e_2) \longmapsto \text{ifz}(e'_0, e_1, e_2).
\]

Suppose that \(\text{ifz}(e_0, e_1, e_2) : \tau\). By inversion, \(e_0 : \text{nat}\), \(e_1 : \tau\), and \(e_2 : \tau\). By induction \(e'_0 : \text{nat}\), and hence \(\text{ifz}(e'_0, e_1, e_2) : \tau\), as required.

The other rules governing the conditional test are handled similarly.

Consider the rule
\[
v \text{ value} \\
\text{app}(\lambda(\tau, x.e), v) \longmapsto [x \leftarrow v]e.
\]

Suppose that \(\text{app}(\lambda(\tau, x.e), v) : \tau'\). By inversion \(v : \tau\) and \(x : \tau \vdash e : \tau'\). By substitution \([x \leftarrow v]e : \tau'\).

The other two rules governing application are handled similarly to the rules for the arithmetic operations. The rules for the let construct are left to the reader.

A critical ingredient in the safety proof is the canonical forms lemma, which characterizes the form of values of a given type.

Lemma 13.2 (Canonical Forms)
Suppose that \(v : \tau\) is a closed, well-formed value.

1. If \(\tau = \text{nat}\), then \(v = \text{num}[n]\) for some \(n : \text{nat}\).
2. If \(\tau = \text{arrow}(\tau_1, \tau_2)\), then \(v = \lambda(\tau_1, x.e)\) for some \(x\) and \(e\) such that \(x : \tau_1 \vdash e : \tau_2\).

Proof: By induction on the typing rules, using the assumption \(v \text{ value}\).

Theorem 13.3 (Progress)
If \(e : \tau\), then either \(e\) is a value, or there exists \(e'\) such that \(e \longrightarrow e'\).
**Proof:** The proof is by rule induction on the definition of the typing judgement.

Consider the rule

$$
\Gamma \vdash \text{num}[n] : \text{nat}.
$$

By definition \text{num}[n] value, which is sufficient for the conclusion.

Consider the rule

$$
\Gamma \vdash e_1 : \text{nat} \quad \Gamma \vdash e_2 : \text{nat}
\quad \Gamma \vdash \text{plus}(e_1, e_2) : \text{nat}.
$$

By induction either \(e_1\) value or there exists \(e'_1\) such that \(e_1 \rightarrow e'_1\). In the latter case we have \(\text{plus}(e_1, e_2) \rightarrow \text{plus}(e'_1, e_2)\). In the former, we have by induction that either \(e_2\) value or there exists \(e'_2\) such that \(e_2 \rightarrow e'_2\). In the latter case we have \(\text{plus}(v_1, e_2) \rightarrow \text{plus}(v_1, e'_2)\). In the former we appeal to the canonical forms lemma (twice) to obtain that \(v_1 = \text{num}[n_1]\) for some \(n_1\) nat and \(v_2 = \text{num}[n_2]\) for some \(n_2\) nat. But then \(\text{plus}(v_1, v_2) \rightarrow \text{num}[n]\), where \(n = n_1 + n_2\) nat, as required.

Consider the rule

$$
\Gamma \vdash e : \text{nat} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau
\quad \Gamma \vdash \text{ifz}(e, e_1, e_2) : \tau.
$$

By induction either \(e\) value or there exists \(e'\) such that \(e \rightarrow e'\). In the latter case \(\text{ifz}(e, e_1, e_2) \rightarrow \text{ifz}(e', e_1, e_2)\). In the former we have by the canonical forms lemma that \(e = \text{num}[n]\) for some \(n\) nat. If \(n = \text{zero}\), then \(\text{ifz}(e, e_1, e_2) \rightarrow e_1\), otherwise \(\text{ifz}(e, e_1, e_2) \rightarrow e_2\).

Consider the rule

$$
\Gamma \vdash e_1 : \text{arrow}(\tau_2, \tau) \quad \Gamma \vdash e_2 : \tau
\quad \Gamma \vdash \text{app}(e_1, e_2) : \tau.
$$

By induction either \(e_1\) value or \(e_1 \rightarrow e'_1\). In the latter case we have \(\text{app}(e_1, e_2) \rightarrow \text{app}(e'_1, e_2)\). Otherwise we have by induction either \(e_2\) value or \(e_2 \rightarrow e'_2\). In the latter case we have \(\text{app}(e_1, e_2) \rightarrow \text{app}(e_1, e'_2)\) (bearing in mind \(e_1\) value). Otherwise, by the canonical forms lemma \(e_1 = \lambda bda(\tau_2, x, e)\) for some \(x\) and \(e\). But then \(\text{app}(e_1, e_2) \rightarrow [x\gets e_2]e\), again bearing in mind that \(e_2\) value.

The remaining cases are left to the reader.
13.2 Run-Time Errors and Safety

Type safety for MinML ensures that “stuck” states (those from which no transition is possible, yet are not values) are always ill-typed. But suppose that we wish to extend MinML with, say, a quotient operation that is undefined in some situations. For example, $3/0$, being undefined, would be “stuck”, yet is not a value and is well-typed — provided that we use the following typing rule for quotient.

$$
\Gamma \vdash e_1 : \text{nat} \quad \Gamma \vdash e_2 : \text{nat} \\
\Gamma \vdash \text{div}(e_1, e_2) : \text{nat}.
$$

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

To ensure safety of MinML extended with quotients 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 regarded as “stuck”, but rather as a checked error.

Either option is, in principle, viable, but the most common approach is the second. The first requires that the type checker prove that an expression be non-zero before permitting it to be used in the denominator of a quotient. It is difficult to do this without ruling out too many programs. For now we consider the second option, which is widely used.

The general idea is to distinguish **checked** from **unchecked** errors. An unchecked error is one that is ruled out by the type system. No run-time checking is performed to ensure that such an error does not occur, because the type system rules out the possibility of it arising. For example, the dynamic semantics of MinML need not check, when performing an addition, that its two arguments are, in fact, natural numbers, as opposed to, say, functions, because the type system ensures that this is the case. On the other hand the dynamic semantics for quotient must check for a zero denominator, because the type system does not rule out this possibility.

This may be achieved by adding to the language a new construct, **error**, which signals the occurrence of a checked error. The typing rule for a checked error permits it to be regarded as having any type at all:

$$
\Gamma \vdash \text{error} : \tau
$$

(13.1)
To ensure that this is safe requires that we augment the dynamic semantics with rules that *propagate* errors — once an error arises, it aborts the entire computation.

Partially defined operations, such as quotient, give rise to errors:

\[ \text{div}(v_1, \text{num}[0]) \rightarrow \text{error}. \] (13.2)

Once an error arises, it propagates through all other constructs. For example, we add the following rules to the definition of the transition relation for MinML:

\[ \text{app}(	ext{error}, e_2) \rightarrow \text{error} \] (13.3)
\[ \text{app}(v_1, \text{error}) \rightarrow \text{error} \] (13.4)

Similar rules propagate errors through the other constructs of the language.

The preservation theorem remains the same, and is proved similarly, bearing in mind that error has any type we like. The progress theorem must be modified as follows:

**Theorem 13.4 (Progress With Error)**

*If* \( e : \tau \), *then either* \( e = \text{error} \) *or* \( e \) *value or there exists* \( e' \) *such that* \( e \rightarrow e' \).

**Proof:** The proof is by induction on typing, and proceeds similarly to the proof given earlier, except that there are now three cases to consider at each point in the proof.

13.3 Exercises

1. Complete the proof of preservation and progress for MinML.

2. Complete the proof of progress for MinML extended with checked errors.
Chapter 14

Environments and Functions

In Chapter 10 we introduced the concept of an environment semantics, in which substitution is avoided in favor of maintaining a list of hypotheses specifying the bindings of the free variables of an expression. This corresponds more closely to practical implementations, which associate bindings to variables during execution, rather than perform substitution.

In this chapter we investigate the extensions of environment semantics to the functional language MinML. This extension is non-trivial, and is, in fact, the source of an infamous error in language design!

14.1 Environment Semantics for MinML

The environment semantics for expressions given in Chapter 10 is based on hypothetical judgements of the form

\[ x_1 \Downarrow v_1, \ldots, x_n \Downarrow v_n \vdash e \Downarrow v \]

stating that the expression \( e \) evaluates to the value \( v \), under the assumption that the variables \( x_i \) evaluate to \( v_i \).

Let us naively extend this semantics to MinML using the following rules for functions and applications (the other rules are similar to those for expressions):

\[
\frac{}{\eta \vdash \text{lambda}(\tau, x.e) \Downarrow \text{lambda}(\tau, x.e)}
\]

\[
\frac{\eta \vdash e_1 \Downarrow \text{lambda}(\tau, x.e) \quad \eta \vdash e_2 \Downarrow v_2 \quad \eta, x \Downarrow v_2 \vdash e \Downarrow v}{\eta \vdash \text{app}(e_1, e_2) \Downarrow v}
\]
The idea is that when applying a function to an argument, we bind the parameter of the function to the argument value, and proceed to evaluate the body of the function under the influence of that binding. The resulting value is the value of the application.

Superficially this formulation looks fine, but it is, in fact, incorrect! Consider the following example,

\[
\text{let } f \text{ be } (\lambda x: \text{nat}. \lambda y: \text{nat}. x) 3 \text{ in let } x \text{ be } 5 \text{ in } f x,
\]

which we have written using an informal concrete syntax for the sake of readability.

According to the environment semantics of MinML, evaluation of this expression proceeds by evaluating the binding of \( f \), then binding this value to \( f \) for use within the body. The binding of \( f \) is an application, which is evaluated by binding \( x \) to \( 3 \), and evaluating the body of the function. This is itself a function, \( \lambda y: \text{nat}. x \), which is yielded as result. Then \( f \) is bound to this function, and evaluation proceeds with the inner \texttt{let}. The variable \( x \) is then bound to \( 5 \), and the application \( f x \) is evaluated. After obtaining the bindings for \( f \) and \( x \), we proceed by evaluating \( x \) in the environment in the extension of the current environment with \( y \) bound to \( 5 \). The result, \( 5 \), is the overall result of evaluation.

But now let us evaluate this same expression using the substitution semantics. First, we evaluate the binding of \( f \). This is obtained by evaluating the application of \( \lambda x: \text{nat}. \lambda y: \text{nat}. x \) to the argument \( 3 \), which obtained by substituting \( 3 \) for \( x \) in the body, obtaining \( \lambda y: \text{nat}. 3 \). This function is substituted for \( f \) in the inner \texttt{let}, which is evaluated by substituting \( 7 \) for \( x \) in the application, obtaining \( \lambda y: \text{nat}. 3 \) whose value is \( 3 \).

What went wrong? The problem is that the environment semantics confuses the two distinct occurrences of the variable \( x \) in the program. When evaluating the binding for \( f \), the environment semantics binds \( x \) to \( 3 \), and evaluates \( \lambda y: \text{nat}. x \), which is returned as a value — \textit{with the variable \( x \) occurring freely within it}. Later on, the inner \texttt{let} binds the variable \( x \) to \( 5 \), and this binding is used to govern the evaluation of the body of \( f \), inadvertently confusing the two variables!

Another way to see the problem is to consider the following \( \alpha \)-equivalent expression in which we have renamed the inner \( x \) to \( z \):

\[
\text{let } f \text{ be } (\lambda x: \text{nat}. \lambda y: \text{nat}. x) 3 \text{ in let } z \text{ be } 5 \text{ in } f z,
\]

This should not change the meaning of the expression, yet when evaluated using the environment semantics, the evaluation process “gets stuck”
because it lacks a binding for $x$ at the point where the application $f\ z$ is evaluated! The substitution semantics encounters no such difficulties, and properly assigns this expression the value 3, as it should.

The confusion of bindings incurred by the evaluation semantics in this example is sometimes called \textit{dynamic binding}. The idea is that the binding for $x$ used during evaluation of the innermost application is the \textit{dynamically most recent} binding for $x$ in the environment. However, as we have just seen, this policy fails to respect $\alpha$-equivalence (renaming of bound variables), and does not agree with the substitution semantics. It is therefore \textit{wrong}, and must be corrected.\footnote{Historically, this error was introduced in the very first implementation of Lisp, and was later diagnosed as a mistake by McCarthy. The phrase \textit{dynamic binding} is, to this author’s mind, simply a ruse intended to turn the mistake into a feature by giving it a name.}

The \textit{correct} treatment of variables is, by contrast, called \textit{static binding}. Static binding is simply the discipline of assigning binding sites to variables based on their textual occurrences, independently of any dynamic execution model, as we have detailed in Chapter 6. The problem, therefore, lies not with the concepts of binding and scope, but with the evaluation semantics itself.

\section{14.2 Closures}

To avoid these difficulties we must ensure that the free variables of a function are not detached from their environment. The main idea is to regard the environment as an \textit{explicit substitution}, a data structure that records what is to be substituted for a variable without actually doing it. Only when the variable is encountered do we replace it by its binding in the environment, effectively \textit{delaying} substitution as long as possible. To avoid the confusions described in the preceding section, we attach the environment to a $\lambda$-abstraction at the point where the abstraction is evaluated, resulting in a configuration of the form

$$\text{clos}(\eta,\lambda(x.e)),$$

which is called a \textit{closure}. The idea is that the environment “closes” the free variables of the $\lambda$-abstraction by providing bindings for them. These are the bindings that are used when the function body is evaluated, not those in the ambient environment at the point of application.
To give a proper environment semantics for MinML we introduce two new syntactic categories, *values* and *environments*.

Values \( v \) :: =  \( \text{num}[n] \mid \text{clos}(\eta, \lambda(x.e)) \)

Envs \( \eta \) :: =  \( \bullet \mid \eta, x=v \)

In this setting values are no longer forms of expression, but are instead a special syntactic category of their own.

The environment semantics for MinML is re-formulated to determine the value, in the sense of the preceding grammar, for each expression. The key changes are exemplified by the following two rules:

\[
\begin{align*}
\eta \vdash \lambda(\tau, x.e) & \Downarrow \text{clos}(\eta, \lambda(\tau, x.e)) \\
\eta \vdash e_1 & \Downarrow \text{clos}(\eta', \lambda(\tau, x.e)) \\
\eta \vdash e_2 & \Downarrow v_2 \\
\eta', x=v_2 \vdash & e \Downarrow v \\
\eta \vdash \text{app}(e_1, e_2) & \Downarrow v
\end{align*}
\]

Notice that we *switch environments* from \( \eta \), the ambient environment, to \( \eta' \), the environment of the closure, when evaluating the function body. This ensures that the free variables of the body are governed by the environment in effect at the point where the function is created, not at the point where the function is applied.

To characterize the well-formed values we enrich the static semantics with rules for (closed) values. The typing rule for closures is as follows:

\[
\frac{\eta : \Gamma' \quad \Gamma', x : \tau_1 \vdash e : \tau_2}{\text{clos}(\eta, \lambda(\tau_1, x.e)) : \text{arrow}(\tau_1, \tau_2)}
\]

Notice that the body of the function may have any number of free variables, which are governed by the hypotheses \( \Gamma' \).

The typing rule for closures makes use of the following typing rules for environments:

\[
\frac{v : \tau \quad \eta : \Gamma'}{\eta, x=v : \Gamma', x : \tau}
\]

The judgement \( \eta : \Gamma' \) means that the environment \( \eta \) provides bindings for the variables in \( \Gamma' \). The bindings are all closed values of appropriate type, as determined by \( \Gamma' \).

The environment semantics may be proved equivalent to the substitution semantics using a technical device that “expands out” the delayed
substitutions occurring in closures. This operation is inductively defined as follows:

\[
\begin{align*}
\text{num}[n]^* &= \text{num}[n] \\
[\eta] e = e' &\Rightarrow \text{clos}(\eta, \lambda(x.e))^* = \lambda(x.e')
\end{align*}
\]

The \([\eta] e\) stands for the result of the simultaneous substitution of \(\eta\) into \(e\), as defined in Chapter 6.

The (corrected) environment and substitution semantics are equivalent.

**Theorem 14.1 (Equivalence)**

\(\eta \vdash e \Downarrow v\) iff \([\eta] e \Downarrow v^*\).

### 14.3 Exercises

1. Complete the definition of the environment semantics for MinML.

2. Prove that if \(\Gamma \vdash e : \tau, \emptyset \vdash \eta : \Gamma\), and \(\eta \vdash e \Downarrow v\), then \(\emptyset \vdash v : \tau\).

3. Prove the equivalence theorem.

4. Re-formulate the transition semantics for MinML in terms of environments. What difficulties do you encounter? How might they be overcome?
Part V

Data Structures: Aggregates and Variants
Chapter 15

Product Types

The binary product of two types consists of ordered pairs of values, one from each type in the order specified. The associated eliminatory forms are projections, which select the first and second component of a pair. The nullary product, or unit, type consists solely of the unique “null tuple” of no values, and has no associated eliminatory form.

More generally, the general, or n-ary, product of \( n \geq 0 \) types consists of the ordered n-tuples of values, with the eliminatory forms being the \( i \)th projection, where \( 0 \leq i < n \).

The labelled product, or record, type consists of labelled n-tuples in which the components are labelled by names. The eliminatory forms access the field of a specified name.

15.1 Nullary and Binary Products

Let us extend the abstract syntax of MinML with the following constructs:

\[
\begin{align*}
\text{Types} & \quad \tau : := \text{unit} \mid \text{prod}(\tau_1, \tau_2) \\
\text{Expr's} & \quad e : := \text{un} \mid \text{pair}(e_1, e_2) \mid \text{fst}(e) \mid \text{snd}(e)
\end{align*}
\]

In examples we write \( \tau_1 * \tau_2 \) for \( \text{prod}(\tau_1, \tau_2) \), \( \langle e_1, e_2 \rangle \) for \( \text{pair}(e_1, e_2) \) and \( \langle \rangle \) for \( \text{un} \).

The type \( \text{prod}(\tau_1, \tau_2) \) is sometimes called the binary product of the types \( \tau_1 \) and \( \tau_2 \), and the type unit is correspondingly called the nullary product (of no types). We sometimes speak loosely of product types in such a way as to cover both the binary and nullary cases.

The introductory form for the product type is called pairing, and its eliminatory forms are called projections. For the unit type the introductory
form is called the unit object, or null tuple. There is no eliminatory form, there being nothing to extract from a null tuple!

The static semantics of product types is given by the following rules:

\[
\begin{align*}
\Gamma \vdash \text{un} : \text{unit} & \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash \text{pair}(e_1, e_2) : \text{prod}(\tau_1, \tau_2) & \quad \Gamma \vdash e : \text{prod}(\tau_1, \tau_2) \\
\Gamma \vdash \text{fst}(e) : \tau_1 & \quad \Gamma \vdash \text{snd}(e) : \tau_2
\end{align*}
\]

There are two different dynamic semantics for product types, the eager semantics, and the lazy semantics. The eager semantics is specified by the following rules:

\[
\begin{align*}
\text{un value} & \quad e_1 \text{ value} \quad e_2 \text{ value} \\
\text{pair}(e_1, e_2) & \quad \text{pair}(e'_1, e'_2) \\
\text{fst}(e) & \quad \text{fst}(e') \\
\text{snd}(e) & \quad \text{snd}(e') \\
\text{fst}(\text{pair}(e_1, e_2)) & \quad e_1 \\
\text{snd}(\text{pair}(e_1, e_2)) & \quad e_2
\end{align*}
\]

According to these rules a pair \(\text{pair}(e_1, e_2)\) is a value only if both \(e_1\) and \(e_2\) are values. Evaluation of a projection necessarily implies evaluation of its argument to determine what pair to project from.

The lazy semantics for tuples is specified by the following rules:

\[
\begin{align*}
\text{un value} & \quad \text{pair}(e_1, e_2) \text{ value} \\
\text{fst}(e) & \quad \text{fst}(e') \\
\text{snd}(e) & \quad \text{snd}(e') \\
\text{fst}(\text{pair}(e_1, e_2)) & \quad e_1 \\
\text{snd}(\text{pair}(e_1, e_2)) & \quad e_2
\end{align*}
\]

According to these rules any ordered pair is a value, regardless of whether its components are values. Therefore there are no “search” rules for evaluating the components of a projection.
Theorem 15.1 (Safety)
Under either the lazy or the eager semantics of pairing,

1. If \( e : \tau \) and \( e \mapsto e' \), then \( e' : \tau \).

2. If \( e : \tau \) then either \( e \) value or there exists \( e' \) such that \( e \mapsto e' \).

Proof: The proof is left as an exercise to the reader. ■

15.2 General Products

The syntax of general product types is given by the following grammar:

\[
\begin{align*}
\text{Types} & \quad \tau : = t\text{pl}(\tau_0, \ldots, \tau_{n-1}) \\
\text{Expr's} & \quad e : = t\text{pl}(e_0, \ldots, e_{n-1}) \mid \text{prj}(i, e)
\end{align*}
\]

Formally, this grammar is indexed by the size, \( n \), of the general product type under consideration. In addition the projections are indexed by a natural number constant, \( 0 \leq i < n \), indicating the position to select from the \( n \)-tuple. The re-use of the operator \( t\text{pl} \) for both a type constructor and a term constructor should cause no confusion, but formally there are two operators of arity \( n \), one for forming types, the other for forming expressions.

We may either take these constructs as primitives, treating products as special cases, or define these constructs in terms of products, as follows:

\[
\begin{align*}
\text{tpl}(\tau_0, \ldots, \tau_{n-1}) & = \begin{cases} 
\text{unit} & \text{if } n = 0 \\
\text{prod}(\tau_0, \text{tpl}(\tau_1, \ldots, \tau_{n-1})) & \text{if } n > 0
\end{cases} \\
\text{tpl}(e_0, \ldots, e_{n-1}) & = \begin{cases} 
\text{un} & \text{if } n = 0 \\
\text{pair}(e_0, \text{tpl}(e_1, \ldots, e_{n-1})) & \text{if } n > 1
\end{cases} \\
\text{prj}(j, e) & = \begin{cases} 
\text{fst}(e) & \text{if } j = 0 \\
\text{prj}(j - 1, \text{snd}(e)) & \text{if } j > 0
\end{cases}
\end{align*}
\]

These definitions are a bit tricky. The definitions of the \( n \)-ary product type and the \( n \)-tuple expression are defined for \( n > 0 \) in terms of their definition for \( n - 1 \). Moreover, the projections are further parameterized by a constant \( 0 \leq i < n \) indicating the position to project; these are defined for \( i > 0 \) in terms of their definitions for \( i - 1 \).

We leave it to the reader to derive the static and dynamic semantics for general product types implied by these definitions.
15.3 Labelled Products

Labelled product, or record, types are a useful generalization of product types in which the components are accessed by name, rather than by position. The benefits of this should be clear: one cannot be expected to remember the intended meaning of the 7th component of a 13-tuple!

The syntax for records is quite similar to that for n-tuples:

\[
\begin{align*}
\text{Types} & : \quad \tau \ := \ \text{rcd}(l_0, \tau_0, \ldots, l_{n-1}, \tau_{n-1}) \\
\text{Expr's} & \quad e \ := \ \text{rcd}(l_0, e_0, \ldots, l_{n-1}, e_{n-1}) | \text{prj}(l, e)
\end{align*}
\]

We use the meta-variable \( l \) to range over labels, an infinite set of names disjoint from variable names. In concrete syntax one often writes

\[
\begin{align*}
\{l_0 : \tau_0, \ldots, l_{n-1} : \tau_{n-1}\}
\end{align*}
\]

for record types,

\[
\begin{align*}
\{l_0 = e_0, \ldots, l_{n-1} = e_{n-1}\}
\end{align*}
\]

for record expressions, and \( e \cdot l \) for field selection.

The components of a record are called fields. Each field has a label, called the field name, and a type. The components are accessed by field name. We tacitly assume that no two fields are given the same name, and that the order of fields in a record type is irrelevant. That is, we treat as equal any two record types that differ only in the ordering of their fields. That is, the order of fields is considered to be a matter of presentation, and not of substance. This makes sense because the fields are accessed by name, and not by position.

The static semantics of records is given by the following rules:

\[
\begin{align*}
\Gamma \vdash e_0 : \tau_0 & \quad \ldots \quad \Gamma \vdash e_{n-1} : \tau_{n-1} \\
\Gamma \vdash \text{rcd}(l_0, e_0, \ldots, l_{n-1}, e_{n-1}) : \text{rcd}(l_0, \tau_0, \ldots, l_{n-1}, \tau_{n-1}) \\
\Gamma \vdash e : \text{rcd}(l_0, e_0, \ldots, l_{n-1}, \tau_{n-1}) & \Rightarrow \\
\Gamma \vdash \text{prj}(l, e) : \tau_i
\end{align*}
\]

An eager dynamic semantics is specified by these rules:

\[
\begin{align*}
e_0 \text{ value} & \quad \ldots \quad e_{i-1} \text{ value} \quad e_i \longmapsto e_i' \\
\text{rcd}(l_0, e_0, \ldots, l_i, e_i, \ldots, l_{n-1}, e_{n-1}) & \longmapsto \text{rcd}(l_0, e_0, \ldots, l_i, e_i', \ldots, l_{n-1}, e_{n-1})
\end{align*}
\]

\[
\begin{align*}
\text{prj}(l, e) \longmapsto \text{prj}(l, e') \\
e_0 \text{ value} & \quad \ldots \quad e_{n-1} \text{ value} \quad e_0 \longmapsto e_0' \\
\text{prj}(l, \text{rcd}(l_0, e_0, \ldots, l_{n-1}, e_{n-1})) & \longmapsto e_i
\end{align*}
\]

Following the pattern for products, we may also formulate a lazy semantics for records.
Theorem 15.2 (Safety for Records)
1. If \( e : \tau \) and \( e \mapsto e' \), then \( e : \tau' \).
2. If \( e : \tau \), then either \( e \) value or \( e \mapsto e' \) for some \( e' \).

15.4 Exercises

1. State and prove the canonical forms lemma for unit and product types under the eager and under the lazy semantics.

2. Prove the safety theorem for unit and product types under either the eager or the lazy semantics.

3. State the static and dynamic semantics for general products implied by the definitions given in Section 15.1.

4. Functional update, concatenation, restriction, other record operations?
Chapter 16

Sum Types

Most data structures involve alternatives such as the distinction between a leaf and an interior node in a tree, or a choice in the outermost form of a piece of abstract syntax. Importantly, the choice determines the structure of the value. For example, nodes have children, but leaves do not, and so forth. These concepts are expressed by sum types, specifically the binary sum, which offers a choice of two things, and the nullary sum, which offers a choice of no things. These generalize to $n$-ary sums, a choice among $n$ things, and to labelled sums, in which the selection is governed by a label.

16.1 Binary and Nullary Sums

Let us consider the extension of MinML with nullary and binary sums according to the following grammar:

\[
\begin{align*}
\text{Types} & \quad \tau : = \text{void} | \text{sum}(\tau_1, \tau_2) \\
\text{Expr's} & \quad e : = \text{abort}(\tau, e) | \text{inl}(\tau, e) | \text{inr}(\tau, e) | \text{case}(\tau_1, \tau_2, e, x_1.e_1, x_2.e_2)
\end{align*}
\]

The concrete syntax for sum types is $\tau_1 + \tau_2$, and for case expressions is

\[
\text{case } e \{ \text{inl}(x_1: \tau_1) \Rightarrow e_1 | \text{inr}(x_2: \tau_2) \Rightarrow e_2 \}.
\]

The type void is the nullary sum type, whose values are selected from a choice of zero alternatives — there are no values of this type, and so no introductory forms. The eliminatory form, abort($\tau, e$), aborts the computation in the event that $e$ evaluates to a value, which it cannot. The type $\tau = \text{sum}(\tau_1, \tau_2)$ is the binary sum. Its introductory forms have the form inl($\tau, e$) or inl($\tau, e$), indicating which of the two possible choices by tagging a value of the left or right summand as being a value of the sum type.
The eliminatory form performs a case analysis on the tag of a value, decomposing it into its constituent parts.

The static semantics of sum types is given by the following rules:

\[
\begin{align*}
\Gamma \vdash e : \text{void} & \quad \Gamma \vdash \text{abort}(\tau, e) : \tau \\
\Gamma \vdash e_1 : \tau_1 & \quad \tau = \text{sum}(\tau_1, \tau_2) \\
\Gamma \vdash e_2 : \tau_2 & \quad \tau = \text{sum}(\tau_1, \tau_2) \\
\Gamma \vdash \text{inl}(\tau, e) : \tau & \\
\Gamma \vdash \text{inr}(\tau, e) : \tau \\
\Gamma, x_1 : \tau_1 \vdash e_1 : \tau & \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau \\
\Gamma \vdash \text{case}(\tau_1, \tau_2, e, x_1, e_1, x_2, e_2) : \tau
\end{align*}
\]

Just as for the conditional expression considered in Chapter 12, both branches of the case analysis must have the same type. Since the type expresses a static “prediction” on the form of the value of an expression, and since a value of sum type could evaluate to either form at run-time, we must insist that both branches yield the same type.

Just as with products, there are two forms of dynamic semantics, the eager form and the lazy. These differ according to whether the argument to an injection is evaluated at the point the injection is evaluated, or only when, if ever, that underlying value is used. We will give here the eager semantics, and leave the lazy semantics to the reader.

\[
\begin{align*}
e \text{ value} & \quad \text{inl}(\tau, e) \text{ value} \quad \text{inl}(\tau, e) \text{ value} \\
e \mapsto e' & \quad \text{inl}(\tau, e) \mapsto \text{inl}(\tau, e') \quad \text{inl}(\tau, e) \mapsto \text{inl}(\tau, e') \\
e \mapsto e' & \quad \text{case}(\tau_1, \tau_2, e, x_1, e_1, x_2, e_2) \mapsto \text{case}(\tau_1, \tau_2, e', x_1, e_1, x_2, e_2) \\
e \text{ value} & \quad \text{case}(\tau_1, \tau_2, e, x_1, e_1, x_2, e_2) \mapsto [x_1 \leftarrow e] e_1 \\
e \text{ value} & \quad \text{case}(\tau_1, \tau_2, e, x_1, e_1, x_2, e_2) \mapsto [x_2 \leftarrow e] e_2
\end{align*}
\]

The coherence of the static and dynamic semantics is stated and proved as usual.

**Theorem 16.1 (Safety)**

1. If \( e : \tau \) and \( e \mapsto e' \), then \( e' : \tau \).
2. If $e : \tau$, then either $e \text{ value}$ or $e \longmapsto e'$ for some $e'$.

One use of sum types is to define the *Boolean* type, which has the following syntax:

\[
\begin{align*}
\text{Types} & : = \tau \mid \text{bool} \\
\text{Expr's} & : = e \mid \text{tt} \mid \text{ff} \mid \text{if}(e,e_1,e_2)
\end{align*}
\]

This type is definable in the presence of sums and nullary products according to the following equations:

\[
\begin{align*}
\text{bool} & = \text{sum}(\text{unit},\text{unit}) \\
\text{tt} & = \text{inl}(\text{bool},\text{un}) \\
\text{ff} & = \text{inr}(\text{bool},\text{un}) \\
\text{if}(e,e_1,e_2) & = \text{case}(\text{unit},\text{unit},e,x_1.e_1,x_2.e_2)
\end{align*}
\]

The variables $x_1$ and $x_2$ are dummies, since their type, unit, determines their value, un, and, moreover, they do not occur freely in $e_1$ or $e_2$.

Another use of sums is to define the *option* types, which have the following syntax:

\[
\begin{align*}
\text{Types} & : = \tau \mid \text{option}(\tau) \\
\text{Expr's} & : = \text{nothing} \mid \text{just}(e) \mid \text{optcase}(\tau,e,e_1,x.e_2)
\end{align*}
\]

The type $\text{option}(\tau)$ represents the type of "optional" values of type $\tau$. The introductory forms are $\text{nothing}$, corresponding to "no value", and $\text{just}(e)$, corresponding to a specified value of type $\tau$. The elimination form discriminates between the two possibilities.

The option type is definable from sums and nullary products according to the following equations:

\[
\begin{align*}
\text{option}(\tau) & = \text{sum}(\text{unit},\tau) \\
\text{nothing} & = \text{inl}(\text{option}(\tau),\text{un}) \\
\text{just}(e) & = \text{inr}(\text{option}(\tau),e) \\
\text{optcase}(\tau,e,e_1,x.e_2) & = \text{case}(\text{unit},\tau,e,x_1.e_1,x_2.e_2)
\end{align*}
\]

We leave it to the reader to examine the static and dynamic semantics implied by these definitions.

It is important to understand the difference between the types unit and void, which are often confused. The type unit has exactly one element, un, whereas the type void has no elements at all. Consequently, if $e : \text{unit}$, then if $e$ evaluates to a value, it must be unit — in other words, $e$ has no
interesting value (but it could diverge). On the other hand, if $e : \text{void}$, then $e$ must diverge, because if it were to have a value, it would have to be a value of type $\text{void}$, of which there are none. This shows that the $\text{void}$ type in Java and related languages is really the type $\text{unit}$, because it indicates that an expression of that type has no interesting result, not that it must diverge!

### 16.2 Labelled Sums

Binary and nullary sums are sufficient to define generalized $n$-ary sums, in a manner analogous to the definition of $n$-ary products from nullary and binary products in Chapter 15. We leave the details of this derivation to the reader, and concentrate instead on labelled sums, or labelled variants. Labelled sums are a form of $n$-ary sum in which the alternatives are labelled by names, rather than by positions.

The syntax of labelled sums is given by the following grammar:

- **Types**
  \[ \tau : = \text{sum}(l_0, \tau_0, \ldots, l_{n-1}, \tau_{n-1}) \]

- **Expr’s**
  \[ e : = \text{inj}(\tau, l_i, e) \mid \text{case}(e, l_0, \tau_0, x_0, e_0, \ldots, l_{n-1}, \tau_{n-1}, x_{n-1}, e_{n-1}) \]

The syntax is a bit heavy compared to products, so it may help to see the concrete syntax as well. The concrete syntax of labelled sum types has the form

\[ [l_0 : \tau_0, \ldots, l_{n-1} : \tau_{n-1}] \]

while that of injections has the form $[l = e]_{\tau}$, and that of case analyses have the form

\[ \text{case } e \{ [l_0 = x_0 : \tau_0] \Rightarrow e_0, \ldots, [l_{n-1} = x_{n-1} : \tau_{n-1}] \Rightarrow e_{n-1} \} \]

It is an awkwardness of the syntax that injections must be marked with the sum type into which the injection is being made. This is to ensure that every expression has a unique type, since we cannot recover the entire sum type from the type of one of its variants. In Chapter 33 we will consider ways to relax this requirement by introducing subtyping.

The static semantics is given by the following rules:

\[
\frac{e : \tau_i \quad \tau = \text{sum}(l_0, \tau_0, \ldots, l_{n-1}, \tau_{n-1}) \quad 0 \leq i < n}{\Gamma \vdash \text{inj}(\tau, l_i, e) : \tau}
\]

\[
\frac{\Gamma \vdash e : \text{sum}(l_0, \tau_0, \ldots, l_{n-1}, \tau_{n-1}) \\ \Gamma, x_0 : \tau_0 \vdash e_0 : \tau \quad \cdots \quad \Gamma, x_{n-1} : \tau_{n-1} \vdash e_{n-1} : \tau}{\Gamma \vdash \text{case}(e, l_0, \tau_0, x_0, e_0, \ldots, l_{n-1}, \tau_{n-1}, x_{n-1}, e_{n-1}) : \tau}
\]
These rules are a straightforward generalization of those for binary sums to permit an arbitrary number of labelled variants.

We leave as an exercise to formulate the (eager or lazy) dynamic semantics of labelled sums and to prove this extension sound.

16.3 Exercises

1. Formulate general $n$-ary sums in terms of nullary and binary sums.
Chapter 17

Recursive Types

Product and sum types are used to model aggregates whose size and shape is known statically — the number and type of components of a tuple and the number and type of the variants in a sum are determined by the type of the value. Recursive types are used to model aggregates whose size and shape known only dynamically. For example, the length of a list is, in general, determined only at run-time; the type does not reveal the length, or even whether the list is empty or non-empty.

The key to achieving this is to permit self-referential, or recursive, types. Much as a recursive function requires a name for “itself” to permit functions whose iteration is determined dynamically, so must a recursive type have a name for itself to represent data whose size and shape is determined dynamically, rather than statically. And just as recursive functions must be “unrolled” when they are called to effect iteration, so must values of recursive type be “rolled” and “unrolled” to effect unbounded aggregation.

17.1 Recursive Types

The abstract syntax of recursive types is given by the following grammar:

\[
\begin{align*}
\text{Types} & \quad \tau \ ::= \ t \mid \text{rec}(t.\tau) \\
\text{Expr's} & \quad e \ ::= \ \text{roll}(\tau,e) \mid \text{unroll}(e)
\end{align*}
\]

The meta-variable \( t \) ranges over a class of type names, which serve as names for types. In the recursive type \( \text{rec}(t.\tau) \), the type name, \( t \), refers to the recursive type itself, in a sense to be made clear in the static semantics given below. The one-step unrolling of \( \text{rec}(t.\tau) \) is the type \( [t \leftarrow \text{rec}(t.\tau)] \tau \) obtained by substituting the recursive type for \( t \) in \( \tau \). When this is the case, the
recursive type is sometimes said to be the one-step rolling of the substituted type. Note, however, that the unrolling does not determine the rolling! That is, given \([t \leftarrow \text{rec}(t, \tau)]\), one cannot recover \(\text{rec}(t, \tau)\), because \(t\) may not occur in \(\tau\), or may occur multiply.

The introductory form, \(\text{roll}(\tau, e)\), introduces a value of recursive type in terms of a value of its one-step unrolling, and the eliminatory form, \(\text{unroll}(e)\), extracts from a value of recursive type a value of its unrolling. In implementation terms the operation \(\text{roll}(\tau, e)\) may be thought of as an abstract “pointer” to a value of the unrolled type, and the operation \(\text{unroll}(e)\) “chases” the pointer to obtain that value from a value of the corresponding rolled type.

The static semantics of this extension of MinML consists of two forms of judgement, \(\tau\) type, and \(e : \tau\). Whereas the latter is the familiar membership judgement stating that expression \(e\) is of type \(\tau\), the former is a formation judgement stating that \(\tau\) is a well-formed type expression. This is required to rule out types that involve type names that do not refer to any recursive type. We may define this judgement by a set of rules that involve hypothetical judgements of the form

\[
t_1\text{ type}, \ldots, t_n\text{ type} \vdash t_1, \ldots, t_n\text{ }\tau\text{ type},
\]

where the assumptions govern the type names that may appear in \(\tau\). We write \(\Delta\) for a finite set of assumptions of the above form, and drop the index from the turnstile as usual. The rules for recursive type formation is as follows:

\[
\Delta, t\text{ type} \vdash \tau\text{ type} \\
\Delta \vdash \text{rec}(t, \tau)\text{ type}
\]

In the presence of other types, such as product and sum types, we would add formation rules such as these:

\[
\Delta \vdash \text{unit type} \\
\Delta \vdash \tau_1\text{ type} \quad \Delta \vdash \tau_2\text{ type} \\
\Delta \vdash \text{prod}(\tau_1, \tau_2)\text{ type}
\]

\[
\Delta \vdash \text{void type} \\
\Delta \vdash \tau_1\text{ type} \quad \Delta \vdash \tau_2\text{ type} \\
\Delta \vdash \text{sum}(\tau_1, \tau_2)\text{ type}
\]

The static semantics of recursive types is defined by the following rules:

\[
\tau\text{ type} \\
\tau = \text{rec}(t, \tau') \\
\Gamma \vdash e : [t \leftarrow \tau]\tau' \\
\Gamma \vdash \text{roll}(\tau, e) : \tau
\]
17.2 Unbounded Data Structures

\[ \Gamma \vdash e : \text{rec}(t, \tau') \]
\[ \Gamma \vdash \text{unroll}(e) : [t \leftarrow \tau] \tau' \]

These rules express an inverse relationship stating that a recursive type is isomorphic to its unrolling, with the operations roll and unroll being the witnesses to the isomorphism.

Operationally, this is expressed by the following dynamic semantics rules:

\[
\frac{e \text{ value}}{\text{unroll}(\text{roll}(\tau, e)) \rightarrow e}
\]
\[
\frac{e \rightarrow e'}{\text{unroll}(e) \rightarrow \text{unroll}(e')}
\]
\[
\frac{e \rightarrow e'}{\text{roll}(\tau, e) \rightarrow \text{roll}(\tau, e')}
\]

These rules specify an eager semantics for rolling, but it would also be possible to consider a lazy semantics in which \(\text{roll}(\tau, e)\) is a value, regardless of whether \(e\) is a value or not, and in which the last transition rule is correspondingly suppressed.

It is quite easy to establish the safety of this extension to MinML:

**Theorem 17.1 (Safety)**
1. If \(e : \tau\) and \(e \rightarrow e'\), then \(e' : \tau\).
2. If \(e : \tau\), then either \(e\) value, or there exists \(e'\) such that \(e \rightarrow e'\).

### 17.2 Unbounded Data Structures

One example of the use of recursive types is to represent the type of lists of values of a type \(\tau\), which is written \(\text{list}(\tau)\). Informally, a list either either empty, written nil, or is composed of a head, \(v\), and a tail, \(l\), which is written \(\text{cons}(v, l)\). It follows that a general list has the form

\[
\text{cons}(v_1, \text{cons}(v_2, \ldots \text{cons}(v_n, \text{nil})))
\]

for some values \(v_1, \ldots, v_n\) of type \(\tau\). To compute with a list, we use the listcase construct

\[
\text{listcase}(\tau, e, e_0, x, y, e_1),
\]

which discriminates on whether \(e\) is empty or non-empty, resulting in \(e_0\) if empty, and passing the head and tail to \(e_1\) if not.
These constructs are all definable using recursive types, sum types, and product types, according to the following equations:

\[
\begin{align*}
\text{list}(\tau) &= \text{rec}(t.\text{sum(unit,prod}(\tau,t))) \\
\text{nil} &= \text{roll(} \text{list}(\tau), \text{un}) \\
\text{cons}(e_1, e_2) &= \text{roll(} \text{list}(\tau), \text{pair}(e_1, e_2)) \\
\text{listcase}(\tau, e, e_0, x, y, e_1) &= \text{case(} \text{unit, } \tau', \text{unroll(} e), u.e_0, v.e'_1) \\
\end{align*}
\]

where \( \tau' = \text{prod}(\tau, \text{list}(\tau)), u \neq e_0, v \neq e_1, \) and \( e'_1 = [x, y \leftarrow \text{fst}(v), \text{snd}(v)] e_1. \)

Informally, the empty list, nil, is a “pointer” to a null-tuple tagged with inl, and the non-empty list, cons\((e_1, e_2)\), is a “pointer” to a pair tagged with inr, with first component \( e_1 \) and second component \( e_2 \). The listcase construct “chases” the pointer, then cases analyses on the tag, branching to the appropriate case, passing the appropriate data values. Of course more efficient representations are possible, but this representation makes clear the pattern of how unbounded data structures are to be represented using recursive types.

17.3 Un(i)typed Languages

It is customary to distinguish between typed and untyped languages, as if they were alternatives to one another. While it is true that there are ill-defined languages that might be described as untyped, among the well-defined, or safe, languages the supposed distinction is fallacious. A popular form of this misconception is to distinguish between dynamically typed and statically typed languages, often placing them in opposition to one another. Terminology notwithstanding, there is no fundamental distinction or opposition, but rather so-called dynamically typed languages are but a mode of use of static types!

The classical example is the so-called untyped \( \lambda \)-calculus, which is a very elegant language devised by Alonzo Church in the 1930’s. It’s chief characteristic is that the entire language consists of nothing but functions! Functions take functions as arguments and yield functions as results, and all data structures must be represented as functions. Surprisingly, this tiny language is sufficiently powerful to express any computable function\(^1\)

\(^1\)Technically, the notion of a computable function is defined over the natural numbers. So in order for this statement to make sense, the natural numbers must somehow be encoded as functions. This is achieved using a device called the Church numerals, which anticipated by several decades the development of the so-called “object-oriented” view of data.
The abstract syntax of the untyped \(\lambda\)-calculus is given by the following grammar:

\[
\text{\(\lambda\)-terms \( u \) : } \equiv \ x \mid \lambda(x.u) \mid \text{ap}(u_1, u_2)
\]

In concrete syntax these two forms are written \(\lambda x. u\) and \(u_1 u_2\). The former is called a \(\lambda\)-abstraction, and the latter application. The entire language consists of these two constructs, plus variables that range over untyped \(\lambda\)-terms.

The basic form of execution in the untyped \(\lambda\)-calculus is defined by the following transition rules:

\[
\text{ap}(\lambda(x.u_1), u_2) \rightarrow [x \leftarrow u_2]u_1 \quad \text{ap}(u_1, u_2) \rightarrow \text{ap}(u_1', u_2)
\]

In the \(\lambda\)-calculus literature this judgement is called head reduction. The first rule is called \(\beta\)-reduction; it defines the meaning of function application in terms of substitution.

The untyped \(\lambda\)-calculus is faithfully embedded in the typed language MinML, enriched with recursive types. This means that every untyped \(\lambda\)-term has a representation as an expression in MinML in such a way that execution of the representation corresponds to execution according to the rules of the \(\lambda\)-calculus. It is important to understand that this form of embedding is not a matter of writing an interpreter for the \(\lambda\)-calculus in MinML, but rather a direct representation of \(\lambda\)-terms as certain expressions of MinML. That is, the embedding shows that the untyped language is just a particular mode of usage of the typed language! (Consequently, the two forms of language can hardly be seen as in opposition to one another.)

The main idea is the observation that untyped really means uni-typed. The untyped \(\lambda\)-calculus does not have zero types, rather it has exactly one type! This type is the celebrated recursive type

\[
D = \text{rec}(t.\text{arrow}(t, t)).
\]

A value of type \(D\) is of the form \(\text{roll}(D, e)\) where \(e\) is a value of type \(\text{arrow}(D, D)\) — a function whose domain and range are both \(D\). Any such function can be regarded as a value of type \(D\) by “rolling”, and any value of type \(D\) can be turned into a function by “unrolling”. Put in other terms, the recursive type \(D\) satisfies the isomorphism

\[
D \cong \text{arrow}(D, D)
\]

meaning that it is isomorphic to the function space on itself.
This leads to the following embedding, $u^\dagger$, of $u$ into MinML:

\[
x^\dagger = x \\
\lambda(x.u)^\dagger = \text{roll}(D, \lambda(D, x.u^\dagger)) \\
\text{ap}(u_1, u_2)^\dagger = \text{app}(\text{unroll}(u_1^\dagger), u_2^\dagger)
\]

Observe that the embedding of a $\lambda$-abstraction is a value, and that the embedding of an application exposes the function being applied by unrolling the recursive type. Consequently,

\[
\begin{align*}
\text{ap}(\lambda(x.u_1), u_2)^\dagger &= \text{app}(\text{unroll}(\text{roll}(D, \lambda(D, x.u_1^\dagger))), u_2^\dagger) \\
&\mapsto \text{app}(\lambda(D, x.u_1^\dagger), u_2^\dagger) \\
&\mapsto [x\leftarrow u_2^\dagger]u_1^\dagger \\
&= ([x\leftarrow u_2]u_1)^\dagger.
\end{align*}
\]

The last step, stating that the embedding commutes with substitution, is easily proved by induction on the structure of $u_1$. Thus $\beta$-reduction is faithfully implemented by evaluation of the embedded terms. It is also easy to show that if $u_1^\dagger \rightarrow^{*} v_1^\dagger$, then $\text{ap}(u_1, u_2)^\dagger \rightarrow^{*} \text{ap}(v_1, u_2)^\dagger$. Consequently, head reduction in the $\lambda$-calculus is faithfully implemented by evaluation in MinML.

### 17.4 Exercises

1. Derive the static and dynamic semantics of lists induced by the definitions given in Section 17.2.

2. Give a representation of binary trees decorated with values of type $\tau$ at the leaves using recursive types.

3. Can MinML be faithfully embedded in the untyped $\lambda$-calculus?
Chapter 18

Pattern Compilation
Part VI

Control Flow
Chapter 19

Abstract Machines

The technique of specifying the dynamic semantics as a transition system is very useful for theoretical purposes, such as proving type safety, but is too high level to be directly usable in an implementation. One reason is that the use of “search rules” requires the traversal and reconstruction of an expression in order to simplify one small part of it. In an implementation we would prefer to use some mechanism to record “where we are” in the expression so that we may “resume” from that point after a simplification. This can be achieved by introducing an explicit mechanism, called a control stack, that keeps track of the context of an instruction step for just this purpose. By making the control stack explicit the transition rules avoid the need for any premises — every rule is an axiom! This is the formal expression of the informal idea that no traversals or reconstructions are required to implement it.

By making the control stack explicit we move closer to an actual implementation of the language. As we expose more and more of the mechanisms required in an implementation we get closer and closer to the physical machine. At each step along the way, starting with the SOS description and continuing down towards an assembly-level description, we are working with a particular abstract, or virtual, machine. The closer we get to the physical machine, the less “abstract” and the more “concrete” it becomes. But there is no clear dividing line between the levels, rather it is a matter of progressive exposure of implementation details. After all, even machine instructions are implemented using gates, and gates are implemented using transistors, and so on down to the level of fundamental physics.

Nevertheless, some abstract machines are more concrete than others, and recently there has been a resurgence of interest in using them to pro-
vide a hardware-independent computing platform. The idea is to define a
low-enough level abstract machine such that (a) it is easily implementable
on typical hardware platforms, and (b) higher-level languages can be trans-
lated (compiled) to it. In this way it is hoped that most software can be
freed of dependence on specific hardware platforms.\(^1\) It is of paramount
importance that the abstract machine be precisely defined, for otherwise it
is not clear how to translate to it, nor is it clear how to implement it on a
given platform.

In this chapter we introduce the \textit{C machine}, an abstract machine that
makes control flow explicit. Using the tools we have developed in this
book, we give a precise definition of the C machine, and show how to prove
its correctness relative to the semantics of MinML.

\section{The C Machine}

A state, \(s\), of the C machine consists of a \textit{control stack}, \(k\), and a closed expres-
sion, \(e\). States may take one of two forms:

1. An \textit{evaluation} state of the form \(k > e\) corresponds to the evaluation of
   a closed expression, \(e\), relative to a control stack, \(k\).

2. A \textit{return} state of the form \(k < e\), where \(e\) \textit{value}, corresponds to the
   evaluation of a stack, \(k\), relative to a closed value, \(e\).

As an aid to memory, note that the separator “points to” the focal entity
of the state, the expression in an evaluation state and the stack in a return
state.

The control stack represents the context of evaluation. It records the
“current location” of evaluation, the context into which the value of the
current expression is to be returned. Formally, a control stack is a list of
frames:

\begin{equation}
\begin{array}{c}
\varepsilon \text{ stack} \\
\hline
f \text{ frame } k \text{ stack} \\
\hline
f; k \text{ stack}
\end{array}
\end{equation}

\(^1\)This is much easier said than done; it remains an active area of research and develop-
ment.
The definition of frame depends on the language we are evaluating. For MinML the frames are inductively defined by the following rules:\(^2\)

\[
\begin{align*}
\text{plus}(\_, e_2) & \text{ frame} & \text{plus}(v_1, \_) & \text{ frame} \\
\hline
\text{e}_1 \text{ exp} & \text{ e}_2 \text{ exp} & \text{ifz}(-, e_1, e_2) & \text{ frame} \\
\hline
\text{app}(\_, e_2) & \text{ frame} & \text{v}_1 \text{ value} & \text{ app}(v_1, \_) & \text{ frame}
\end{align*}
\]

(19.2)

A frame corresponds to an elimination form in which one argument position is currently under evaluation.

The transition judgement between states of the C is inductively defined by a set of inference rules. We begin with the rules for numbers and arithmetic.

\[
\begin{align*}
k > \text{num}[n] & \rightarrow k < \text{num}[n] \\
k > \text{plus}(e_1, e_2) & \rightarrow \text{plus}(-, e_2); k > e_1 \\
\text{e}_1 \text{ value} & \rightarrow \text{plus}(-, e_2); k < e_1 \rightarrow \text{plus}(e_1, \_); k > e_2 \\
n_1 + n_2 = n \text{ nat} & \rightarrow \text{plus}([\text{num}[n_1], -]; k < \text{num}[n_2]) \rightarrow k < \text{num}[n]
\end{align*}
\]

(19.3)

To evaluate a number, we simply return it to the stack. To evaluate an addition, we push a frame onto the stack recording that we are currently working on its first argument, and continue evaluating that argument. When a value is returned to a stack whose top frame records that we are evaluating the first argument of an addition, we swap that frame with a frame recording that we have completed that evaluation, and continue by evaluating the second argument. Finally, when a value is returned to such a frame, we perform the addition and return the result to the stack.

---

\(^2\)We give only the frames for the primitive operation of addition; those for multiplication and any other primitive operations are defined analogously.
Next, we consider the rules for conditionals.

\[
\frac{k \succ ifz(e, e_1, e_2)}{k \succ ifz(-, e_1, e_2); k \succ e}
\]

\[
\frac{ifz(-, e_1, e_2); k \prec \text{num}[0]}{k \succ e_1}
\quad (n \neq 0)
\]

\[
\frac{ifz(-, e_1, e_2); k \prec \text{num}[n]}{k \succ e_2}
\]

(19.4)

These rules follow a similar 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 functions.

\[
\frac{k \succ \text{fun}(\tau_1, \tau_2, f.x.e)}{k \prec \text{fun}(\tau_1, \tau_2, f.x.e)\prec \text{app}(\tau_1, \tau_2, f.x.e)}
\]

\[
\frac{k \succ \text{app}(e_1, e_2)}{k \prec \text{app}(-, e_2); k \succ e_1}
\]

\[
\frac{e_1 \text{ value}}{\text{app}(-, e_2); k \prec e_1 \preceq \text{app}(e_1, -); k \succ e_2}
\quad (19.5)
\]

\[
\frac{e_2 \text{ value} \quad e_1 = \text{fun}(\tau_1, \tau_2, f.x.e)}{\text{app}(e_1, -); k \prec e_2 \preceq \lbrack f, x := e_1, e_2 \rbrack e}
\]

These rules ensure that the function is evaluated before the argument, applying the function when both have been evaluated.

The initial and final states of the C are defined by the following rules:

\[
\frac{e \succ e_{\text{init}}}{e \prec e_{\text{final}}}
\quad (19.6)
\]

The type safety of the C machine may be proved by defining a judgement \( s \ \text{OK} \) stating that state \( s \) is well-formed, and proving progress and preservation for this judgement. For a state to be well-formed means that its control stack is well-formed and is prepared to accept a value of type \( \tau \), and that its expression is of type \( \tau \).

\[
\frac{k : \text{stack}(\tau) \quad e : \tau}{k \succ e \text{OK}}
\]

\[
\frac{k : \text{stack}(\tau) \quad e : \tau \quad e \text{ value}}{k \prec e \text{OK}}
\]

(19.7)
For a stack to be well-formed means that it is properly composed from well-formed frames. Since each frame has a single “hole” in it, the stack may be seen as accepting a value of type appropriate to that hole. Each frame fills the hole in the preceding frame, until we reach the end of the stack. This raises the question of what is the type of the empty stack? We will fix a type $\tau_{\text{ans}}$ of answers, the ultimate result of the evaluation of a complete program, and use this as the type of the empty stack.

$$
\begin{align*}
\varepsilon & : \text{stack}(\tau_{\text{ans}}) \\
f & : \text{frame}(\tau, \tau') \\
k & : \text{stack}(\tau)
\end{align*}
$$

Finally, the type $\text{frame}(\tau, \tau')$ is the type of frames that accept a value of type $\tau$ (for the hole) and yield a value of type $\tau'$ (once the hole is filled and the frame step is executed).

$$
\begin{align*}
e_1 & : \text{nat} \\
e_2 & : \text{nat} \\
\text{plus}(\varepsilon, e_2) & : \text{frame}(\text{nat}, \text{nat}) \\
e_1 & : \text{nat} \\
\text{plus}(e_1, e_2) & : \text{frame}(\text{nat}, \text{nat}) \\
e_1 & : \text{nat} \\
e_2 & : \text{nat} \\
\text{ifz}(\varepsilon, e_1, e_2) & : \text{frame}(\text{nat}, \text{nat}) \\
e_2 & : \tau_2 \\
\text{app}(\varepsilon, e_2) & : \text{frame}(\text{arrow}(\tau_2, \tau), \tau) \\
e_1 & : \text{arrow}(\tau_2, \tau) \\
\text{app}(e_1, e_2) & : \text{frame}(\tau_2, \tau)
\end{align*}
$$

With these definitions in hand we may state the safety of the C in the usual manner.

**Theorem 19.1 (Safety)**

1. If $s \text{ ok}$ and $s \rightarrow s'$, then $s' \text{ ok}$.
2. If $s \text{ ok}$, then either $s \text{ final}$ or there exists $s'$ such that $s \rightarrow s'$.

### 19.2 Correctness of the C Machine

The structured operational semantics for MinML given in Chapter 12 can be construed as a high-level abstract machine, called the M machine, whose states are closed expressions and whose transitions are as defined there. It is natural to ask whether the C machine correctly implements the semantics.
of MinML: if we evaluate a given expression, \( e \), using the C machine, do we get the same result as would be given by the M machine, and *vice versa*?

Answering this question decomposes into two propositions relating the C and M machines.

**Completeness** If \( e \xrightarrow{*} v \), where \( v \) value, then \( \varepsilon \xrightarrow{*} \varepsilon < v \).

**Soundness** If \( \varepsilon > e \xrightarrow{*} \varepsilon < v \), where \( v \) value, then \( e \xrightarrow{*} v \).

Let us consider, in turn, what is involved in the proof of each part.

For completeness, applying rule induction to the definition of multistep transition from Chapter 3, we must show two things:

1. If \( v \) value, then \( \varepsilon \xrightarrow{*} \varepsilon < v \).
2. If \( e \xrightarrow{*} e' \) and \( \varepsilon > e' \xrightarrow{*} \varepsilon < v \), then \( \varepsilon \xrightarrow{*} \varepsilon < v \).

The first follows immediately from the C machine transition rules. The second, closure under head expansion, requires some work. The obvious strategy is to proceed by induction on the SOS rules defining the M machine transition relation. The chief difficulty in the proof is that we cannot maintain an empty stack during the induction, but we must, instead, consider a general stack, \( k \), in order to complete the argument.

**Lemma 19.2**

If \( e \xrightarrow{*} e' \), then for every \( k \), if \( k > e' \xrightarrow{*} k < v \), then \( k > e \xrightarrow{*} k < v \).

For soundness, observe that it is awkward to reason inductively about the multistep transition from \( \varepsilon > e' \xrightarrow{*} \varepsilon < v \), because the intervening steps may involve alternations of evaluation and return states. Instead we regard each C machine state as encoding an expression, and show that C machine transitions are simulated by M machine transitions under this encoding.

Specifically, we define a judgement, \( s \leadsto e \), stating that state \( s \) “unravels to” expression \( e \). It will turn out that for initial states we have \( \varepsilon > e \leadsto e \) and for final states \( \varepsilon < v \leadsto v \). Then we show that if \( s \xrightarrow{*} s' \), where \( s' \) final, \( s \leadsto e \), and \( s' \leadsto e' \), then \( e \xrightarrow{*} e' \). Applying rule induction to the definition of multistep transition, it is enough to show the following two facts:

1. If \( s \leadsto e \), then \( e \xrightarrow{*} e \).
2. If \( s \xrightarrow{*} s'' \xrightarrow{*} s' \) with \( s' \) final, then if \( s \leadsto e \), \( s'' \leadsto e'' \), \( s' \leadsto e' \), and \( e'' \xrightarrow{*} e' \), then \( e \xrightarrow{*} e' \).
19.2 Correctness of the C Machine

The first is trivial; for the second, it is enough to show the following lemma.

Lemma 19.3
If \( s \rightarrow s', s \Rightarrow e \), and \( s' \Rightarrow e' \), then \( e \Rightarrow^* e' \).

The remainder of this section is devoted to the proofs of these lemmas.

19.2.1 Proof of Completeness

Proof: [of Lemma 19.2] The proof is by induction on the transition rules defining the M machine. We will consider some representative cases here, leaving the rest for the reader.

Suppose that \( e = \text{plus}(e_1, e_2) \), \( e' = \text{plus}(e'_1, e_2) \), and \( e_1 \rightarrow e'_1 \). Suppose further that \( k > e' \Rightarrow^* k < v \) for some \( v \) value. Given the form of \( e' \) the latter transition must have the form

\[
k > e' \Rightarrow \text{plus}(-, e_2); k > e'_1 \Rightarrow^* k < v.
\]

By induction we have \( \text{plus}(-, e_2); k > e_1 \Rightarrow^* k < v \), and hence

\[
k > \text{plus}(e_1, e_2) \Rightarrow \text{plus}(-, e_2); k > e_1 \Rightarrow^* k < v.
\]

Suppose that \( e = \text{plus}([n_1], [n_2]) \) and \( e' = [n] \), where \( n_1 + n_2 = n \) nat. Suppose further that \( k > [n] \Rightarrow^* k < v \) for some \( v \) value. This must have the form \( k > [n] \Rightarrow k < [n] \Rightarrow^* k < v \). Then we have

\[
k > \text{plus}([n_1], [n_2]) \Rightarrow \text{plus}(-, [n_2]); k > [n_1]
\Rightarrow \text{plus}(-, [n_2]); k < [n_1]
\Rightarrow \text{plus}([n_1], -); k > [n_2]
\Rightarrow \text{plus}([n_1], -); k < [n_2]
\Rightarrow k < [n].
\]

This is enough for the result.

The other cases follow a similar pattern.

\[\square\]
19.2.2 Proof of Soundness

We first define the unravelling translation, \( k \circ e \sim e' \), by the following rules:

\[
\begin{align*}
\varepsilon \circ e & \sim e \\
\text{plus}(e_1, e_2) \circ k \circ e_1 & \sim e \quad \text{plus}(e_1, -); k \circ e_2 & \sim e \\
\text{ifz}(e_1, e_2, e_3) \circ k \circ e_1 & \sim e \\
\text{app}(e_1, e_2) \circ k \circ e_1 & \sim e \\
\text{app}(-, e_2) \circ k \circ e_1 & \sim e \\
\text{app}(e_1, -); k \circ e_2 & \sim e
\end{align*}
\]

The notation \( k \circ e \) is stands ambiguously for either form of state, since the distinction does not affect the unravelling translation.

Observe that if \( e \rightarrow e' \), \( k \circ e \sim d \), \( k \circ e' \sim d' \), then \( d \rightarrow d' \). In other words unravelling the stack around a transition does not affect the transition.

We are now in a position to complete the proof of soundness.

**Proof:** [of Lemma 19.3]

The proof is by case analysis on the transitions of the C machine. In each case after unravelling the transition will correspond to zero or one transitions of the M machine.

Suppose that \( s = k > \text{plus}(e_1, e_2) \) and \( s' = \text{plus}(-, e_2) > e_1 \). Note that \( k \circ \text{plus}(e_1, e_2) \sim e \) iff \( \text{plus}(-, e_2); k \circ e_1 \sim e \), from which the result is immediate.

Suppose that \( s = \text{plus}(\text{num}[n_1], -); k < \text{num}[n_2], s' = k < \text{num}[n] \), where \( n_1 + n_2 = n \text{ nat} \). Let \( e \) be such that \( s \sim e \), and note that

\[
k \circ \text{plus}(\text{num}[n_1], \text{num}[n_2]) \sim e
\]

as well. Let \( e' \) be such that \( s' \sim e' \), and so \( e \rightarrow e' \), as required.

\[\blacksquare\]

19.3 The E Machine

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. Substitution is a complicated process, not one that we would ordinarily think of as occurring as a single step of execution of a computer. 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 departure from more familiar models of computation, which maintain a separation between programs and data.

In this section we will present another abstract machine, the E machine, which avoids substitution by maintaining an environment that records the bindings of variables. This introduces complications to do with confusion of variables similar to those discussed in Chapter 14, and we use a similar solution, namely closures, to avoid them.

A significant difference compared to the C machine is that values are no longer forms of expression, but are rather drawn from a new class of E machine values, V. Corresponding, E machine environments, \( \eta \), bind machine values to variables. Finally, E machine stacks, \( K \), and frames, \( F \), may involve environments, and hence are not purely syntactic either.

The states of the E have one of two forms:

1. \( K > e \ [\eta] \), corresponding to evaluating the expression \( e \) on the stack \( K \) relative to the environment \( \eta \).

2. \( K < V \), corresponding to returning the value, \( V \), to the stack, \( K \).

These states are similar to those for the C machine.

The judgements \( \eta \ mvalue \), \( K \ mstack \), \( F \ mframe \), and \( V \ menv \) are simultaneously inductively defined by the following rules. First, an E machine value is either a number or a closure.

\[
\begin{align*}
\frac{\eta \ \text{nat}}{\text{num}[\eta] \ mvalue} & \quad \frac{\text{fun}(\tau_1, \tau_2, f, x.e) \ \text{exp}}{\text{fun}(\tau_1, \tau_2, f, x.e)[\eta] \ mvalue} \\
\end{align*}
\] (19.10)

An E machine stack is a sequence of E machine frames.

\[
\frac{\varepsilon \ \text{mstack}}{F \ mframe \ K \ mstack} \\
\frac{F; K \ mstack}{}
\] (19.11)

An E machine frame is similar to a C frame, except for the attachment of
environments to frames that contain expressions.

\[
\begin{align*}
\frac{e_2 \text{ exp}}{\text{plus}(\neg, e_2)[\eta] \text{ mframe}} & \quad \frac{V_1 \text{ mvalue}}{\text{plus}(V_1, \neg) \text{ mframe}} \\
\frac{e_1 \text{ exp} \quad e_2 \text{ exp}}{\text{ifz}(\neg, e_1, e_2)[\eta] \text{ mframe}} \\
\frac{e_2 \text{ exp}}{\text{app}(\neg, e_2)[\eta] \text{ mframe}} & \quad \frac{V_1 \text{ mvalue}}{\text{app}(V_1, \neg) \text{ mframe}}
\end{align*}
\]

(19.12)

An E environment is a sequence of bindings of variables to E values such that no variable is bound more than once.

\[
\begin{align*}
\frac{\epsilon \text{ env}}{\eta \text{ menv} \quad x \neq \eta \quad V \text{ mvalue}} & \quad \frac{\eta, x = V \text{ menv}}{\epsilon \text{ menv}}
\end{align*}
\]

(19.13)

The transition rules for the E machine are given as follows. To evaluate a variable \(x\), we look up its binding and pass the associated value to the top frame of the control stack.

\[
\eta(x) = V \\
K > x[\eta] \longrightarrow K < V
\]

(19.14)

Arithmetic is handled similarly to the C, except that we must be careful to close expressions that may have free variables in them.

\[
K > \text{num}[n][\eta] \longrightarrow K < \text{num}[n]
\]

(19.15)

\[
K > \text{plus}(e_1, e_2)[\eta] \longrightarrow \text{plus}(\neg, e_2)[\eta]; K > e_1[\eta]
\]

(19.16)

\[
\text{plus}(\neg, e_2)[\eta]; K < V_1 \longrightarrow \text{plus}(V_1, \neg); K > e_2[\eta]
\]

(19.17)

\[
n_1 + n_2 = n \text{ nat} \\
\text{plus}(\text{num}[n_1], \neg); K < \text{num}[n_2] \longrightarrow K < \text{num}[n]
\]

(19.18)
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.

\[
K \triangleright \text{ifz} (e, e_1, e_2) [\eta] \longmapsto \text{ifz}(\neg, e_1, e_2)[\eta]; K \triangleright e [\eta] \tag{19.19}
\]

\[
\text{ifz}(\neg, e_1, e_2)[\eta]; K < \text{num}[0] \longmapsto K \triangleright e_1 [\eta] \tag{19.20}
\]

\[
(n \neq 0) \quad \text{ifz}(\neg, e_1, e_2)[\eta]; K < \text{num}[n] \longmapsto K \triangleright e_2 [\eta] \tag{19.21}
\]

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.

\[
K \triangleright \text{fun}(\tau_1, \tau_2, f.x.e) [\eta] \longmapsto \text{fun}(\tau_1, \tau_2, f.x.e)[\eta] \tag{19.22}
\]

The notation here may be a bit deceptive. On the left-hand side the environment, \(\eta\), is part of the machine state, whereas on the right-hand side it is attached to the function expression to form a closure.

Finally, function applications are evaluated similarly to the C, except that care must be taken with the environment.

\[
K \triangleright \text{app}(e_1, e_2) [\eta] \longmapsto \text{app}(\neg, e_2)[\eta]; K \triangleright e_1 [\eta] \tag{19.23}
\]

\[
\text{app}(\neg, e_2)[\eta]; K < V \longmapsto \text{app}(V, \neg); K \triangleright e_2 [\eta] \tag{19.24}
\]

\[
V_1 = \text{fun}(\tau_1, \tau_2, f.x.e)[\eta] \\
\text{app}(V_1, \neg); K < V_2 \longmapsto K > e [\eta, f=V_1, x=V_2] \tag{19.25}
\]

Notice that the environment of the closure is installed as the environment of execution for the body, augmented with bindings for the function itself and its argument.

Initial states have the form \(\varepsilon > e [\varepsilon]\), with the stack and environment initially empty. The final states of the E machine have the form \(\varepsilon < V\), where \(V\) mvalue, and the stack is again empty.
19.4 Exercises

1. Prove type safety for the C machine.

2. Finish the proofs of the soundness and completeness lemmas.

3. State and prove the correctness of the E relative to the C.
Chapter 20

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 ?? we will show how to build a threads package for concurrent programming using continuations.

### 20.1 Informal Overview

We will extend MinML with the type \(\text{cont}(\tau)\) of continuations accepting values of type \(\tau\). The introductory form for \(\text{cont}(\tau)\) is \(\text{letcc}(\tau, x.e)\), which binds the current continuation (i.e., the current control stack) to the variable \(x\), and evaluates the expression \(e\). The eliminatory form is \(\text{throw}(e_1, e_2)\), which restores the value of \(e_1\) to the control stack that is the value of \(e_2\).

This description makes clear the need for a persistent representation of control stacks so that they may be bound to variables and restored a value is thrown to one.

Here is a simple example, written in an informal concrete syntax. The idea is to multiply the elements of a list, short-circuiting the computation in case zero is encountered. Here’s the code:

\[1\]Close relatives of these primitives are available in SML/NJ in the following forms: for \(\text{letcc}(\tau, x.e)\), write \(\text{SMLofNJ.Cont.callcc}(\text{fn } x \Rightarrow e)\), and for \(\text{throw}(e_1, e_2)\), write \(\text{SMLofNJ.Cont.throw } e_2 e_1\).
fun mult_list (l:int list):int = 
  letcc ret:int cont 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 evaluation proceeds normally. In the unusual case of encountering a 0 in the list, we branch directly to the return point, passing the value 0, effecting an early return from the procedure with result value 0.

Here’s another formulation of the same function:

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

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

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

fun compose(f:\tau'\rightarrow\tau,k:\tau\ cont):\tau'\ cont = ....
The first problem is to obtain the continuation we wish to return. The second problem is how to return it. The continuation we seek is the one in effect at the point of the ellipsis in the expression \( \text{throw } f(\ldots) \) 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 \texttt{letcc}, writing

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

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

\[
\text{fun compose } (f: \tau' \rightarrow \tau, k: \tau \texttt{cont}): \tau' \texttt{cont} = \\
\text{letcc } \texttt{ret}: \tau' \texttt{cont} \texttt{cont in} \\
\text{throw } (f (\texttt{letcc } r \text{ in } \texttt{throw } r \text{ to } \texttt{ret})) \text{ to } k
\]

Notice that the type of \( \texttt{ret} \) is that of a continuation-expecting continuation!

We can do without first-class continuations by creating our own during execution. 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 \textit{continuation-passing style}, or \textit{CPS} for short.

Here’s the code to multiply the elements of a list, without short-circuiting, in continuation-passing style:

\[
\text{fun cps_mult nil } k = k \ 1 \\
| \text{cps_mult } (n::l) \ k = \text{cps_mult } l \ (\text{fn } r \Rightarrow k \ (n * r))
\]

\[
\text{fun mult } l = \text{cps_mult } l \ (\text{fn } r \Rightarrow r)
\]

The short-circuiting version is just as simple:

\[
\text{fun cps_mult_list } l \ k = \\
\text{let fun } \text{cps_mult nil } k0 \ k = k \ 1 \\
| \text{fun } \text{cps_mult } (0::_) \ k0 \ k = k0 \ 0 \\
| \text{fun } \text{cps_mult } (n::l) \ k0 \ k = \text{cps_mult } k0 \ l \ (\text{fn } p \Rightarrow k \ (n*p))
\]

\[
\text{in } \text{cps_mult } l \ k \ k \end
\]
20.2 Semantics of Continuations

We extend the language of MinML expressions with these additional forms:

\[
\begin{align*}
  \text{Types} & \quad \tau & : & : \quad \text{cont}(\tau) \\
  \text{Expr's} & \quad e & : & : \quad \text{letcc}(\tau, x.e) \mid \text{throw}(e_1, e_2) \mid \text{cont}(k)
\end{align*}
\]

The expression \text{cont}(k) is a reified control stack; they arise during evaluation, but are not available as expressions to the programmer.

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

\[
\begin{align*}
  \Gamma, x : \text{cont}(\tau) \vdash e : \tau & \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \text{cont}(\tau_1) \\
  \Gamma \vdash \text{letcc}(\tau, x.e) : \tau & \quad \Gamma \vdash \text{throw}(e_1, e_2) : \tau'
\end{align*}
\]

(20.1)

The result type of a \text{throw} expression is arbitrary because it does not return to the point of the call.

The static semantics of continuation values is given by the following rule:

\[
\Gamma \vdash k : \text{stack}(\tau) \\
\Gamma \vdash \text{cont}(k) : \text{cont}(\tau)
\]

(20.2)

A continuation value \text{cont}(k) has type \text{cont}(\tau) exactly if it is a stack accepting values of type \tau.

To define the dynamic semantics, we extend the C machine stacks with two new forms of frame:

\[
\begin{align*}
  e_2 \text{ exp} & \quad e_1 \text{ value} & \quad \text{throw}(-, e_2) \text{ frame} & \quad \text{throw}(e_1, -) \text{ frame}
\end{align*}
\]

(20.3)

Every reified control stack is a value:

\[
\Gamma \vdash \text{cont}(k) \text{ value}
\]

(20.4)

The transition rules for the continuation constructs are as follows:

\[
\Gamma \vdash \text{letcc}(\tau, x.e) \quad \Gamma \vdash k \quad k \quad [x \leftarrow \text{cont}(k)] e
\]

(20.5)

February 4, 2006  Working Draft
Exercises

\[ \text{throw}(v, -); k < \text{cont}(k') \rightarrow k' < v \]  
(20.6)

\[ k > \text{throw}(e_1, e_2) \rightarrow \text{throw}(-, e_2); k > e_1 \]  
(20.7)

\[ e_1 \text{ value} \]
\[ \text{throw}(-, e_2); k < e_1 \rightarrow \text{throw}(e_1, -); k > e_2 \]  
(20.8)

Evaluation of a \texttt{letcc} expression duplicates the control stack; evaluation of a \texttt{throw} expression destroys the current control stack.

The safety of this extension of MinML may be established by a simple extension to the safety proof for the C machine given in Chapter 19.

We need only add typing rules for the two new forms of frame, which are as follows:

\[ e_2 : \text{cont}(\tau) \]
\[ \text{throw}(-, e_2) : \text{frame}(\tau, \tau') \]  
(20.9)

\[ e_1 : \tau \quad e_1 \text{ value} \]
\[ \text{throw}(e_1, -) : \text{frame}(\text{cont}(\tau), \tau') \]

The rest of the definitions remain as in Chapter 19.

**Lemma 20.1 (Canonical Forms)**

If \( e : \text{cont}(\tau) \) and \( e \) value, then \( v = \text{cont}(k) \) for some \( k : \text{stack}(\tau) \).

**Theorem 20.2 (Safety)**

1. If \( s \text{ ok} \) and \( s \rightarrow s' \), then \( s' \text{ ok} \).

2. If \( s \text{ ok} \), then either \( s \text{ final} \) or there exists \( s' \) such that \( s \rightarrow s' \).

**20.3 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 \texttt{compose} \((f, k)\) on the empty stack. Observe that the control stack substituted for \( x \) is

\[ \text{app}(f, -); \text{throw}(-, k); \varepsilon \]  
(20.10)
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 21

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 excep-
tion. 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 ex-

cplicit conditionals to check for and process errors or unusual conditions,
but using exceptions is often more convenient, particularly since the trans-
fer 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.

21.1 Stack Unwinding

Let us consider the extension of MinML with an exception mechanism simi-
lar to that of Standard ML, with the significant simplification that no value
is associated with the exception — the exception is signalled and control is
passed to the nearest dynamically enclosing handler.

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

\[
Expr's \ e :: = \ fail | handle(e_1, e_2)
\]

The expression fail raises an exception. The expression \( handle(e_1, e_2) \)
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:

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

(21.1)

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

(21.2)

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 rules:

\[ e_2 \text{ expr} \]
\[ \frac{}{\text{handle}(−, e_2) \text{ frame}} \]  

(21.3)

The transition rules of the C machine are extended as follows:

\[ k > \text{handle}(e_1, e_2) \longmapsto \text{handle}(−, e_2); k > e_1 \]  

(21.4)

\[ \text{handle}(−, e_2); k < v \longmapsto k < v \]  

(21.5)

\[ \text{handle}(−, e_2); k > \text{fail} \longmapsto k > e_2 \]  

(21.6)

\[ f \neq \text{handle}(−, e_2) \]
\[ \frac{}{f; k > \text{fail} \longmapsto k > \text{fail}} \]  

(21.7)

To evaluate \( \text{handle}(e_1, e_2) \) 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 nearest enclosing handler. We explicitly pop non-handler frames while processing a failure; this is sometimes called \textit{unwinding} the control stack.

The definition of initial state remains the same as for the C machine, but we change the definition of final state to include these two forms:

\[ e \text{ value} \]
\[ \frac{}{\varepsilon < e \text{ final}} \quad \frac{}{\varepsilon > \text{fail final}} \]  

(21.8)
21.2 Handler Stacks

The first of these is as before, corresponding to a normal result with the specified value. The second is new, corresponding to an uncaught exception propagating through the entire program.

It is a straightforward exercise the extend the definition of stack typing given in Chapter 19 to account for the new forms of frame. Using this, safety can be proved by standard means. Note, however, that the meaning of the progress theorem is now significantly different! A well-typed program does not “get stuck” ... but it may well raise an uncaught exception.

**Theorem 21.1 (Safety)**

1. If \( s \text{ ok and } s \xrightarrow{} s' \), then \( s' \text{ ok} \).

2. If \( s \text{ ok} \), then either \( s \text{ final} \) or there exists \( s' \) such that \( s \xrightarrow{} s' \).

21.2 Handler Stacks

Most implementations of exceptions do not unwind the stack frame-by-frame, but rather implement a direct transfer of control to the appropriate handler. But it is not just a matter of an indirect jump; the control stack must also be reset to what it was at the point that handler was established. This can be modelled by augmenting the abstract machine state with a stack of stacks, called the handler stack, which records the information required for a non-local control transfer.

Let us call the augmented machine the \( H \text{ machine} \). The states of the \( H \) machine have one of two forms:

1. \( h \mid k \rightarrow e \) corresponding to evaluation of expression \( e \) on control stack \( k \) and handler stack \( h \);

2. \( h \mid k < e \), corresponding to return of a value \( e \) to control stack \( k \) and handler stack \( h \).

A handler stack is just a stack of control stacks:

\[
\begin{array}{c}
\epsilon \\
\hline
h_{\text{hstack}} \\
\end{array}
\begin{array}{c}
k_{\text{stack}} \\
\hline
h_{\text{hstack}} \\
\end{array}
\]

(21.9)

A crucial invariant of the \( H \) machine execution is that each control stack on the handler stack is an extension of the preceding one.

The key transition rules for the \( H \) machine are as follows. Evaluating a handler installs a new handler by pushing an extension of the current
control stack onto the handler stack:

\[
\begin{align*}
& h \mid k > \text{handle}(e_1, e_2) \longrightarrow (\text{handle}(\varepsilon, e_2); h) \mid \text{handle}(\varepsilon, e_2); k > e_1 \\
& \quad (21.10)
\end{align*}
\]

Notice that the same frame is pushed onto \( k \) to form the top of the handler stack and the current control stack.

When a value is returned to a handler context, both the control stack and the handler stack must be popped:

\[
\begin{align*}
& e_1 \text{ value} \\
& (\text{handle}(\varepsilon, e_2); k) \mid \text{handle}(\varepsilon, e_2); k < e_1 \longrightarrow h \mid k < e_1 \\
& \quad (21.11)
\end{align*}
\]

On normal return from an expression guarded by a handler, the handler is removed and control passes up the control stack.

When a failure occurs, the current control stack is disregarded, and is replaced by the top of the handler stack, which is itself popped:

\[
\begin{align*}
& (\text{handle}(\varepsilon, e_2); k) \mid k' > \text{fail} \longrightarrow h \mid k > e_2 \\
& \quad (21.12)
\end{align*}
\]

Control is passed to the handler, running on the control stack in effect at the time the handler was installed. If the handler stack is empty, the failure is uncaught, and we stop the machine:

\[
\begin{align*}
& \varepsilon \mid k > \text{fail} \longrightarrow \varepsilon \mid \varepsilon > \text{fail} \\
& \quad (21.13)
\end{align*}
\]

Two invariants of the H machine are crucial to its implementation:

1. Each control stack on the handler stack is an extension of its predecessor.

2. Each control stack on the handler stack is a (not necessarily proper) prefix of the current control stack.

This means that we can implement the handler stack as a stack of “pointers” into the control stack recording the spots at which handlers are placed. On failure we simply pop the control stack to the point indicated by the top of the handler stack, pop that handler stack, and resume execution.

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

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

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

(21.14)

\[
\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \texttt{arrow}(\tau_{\text{exn}}, \tau)}{\Gamma \vdash \texttt{handle}(e_1, e_2) : \tau}
\]

(21.15)

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

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

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

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

\[
| \text{div (m, n) = ... raise "Arithmetic overflow occurred."} ... 
\]

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 \(\tau_{\text{exn}}\) to be, say, \texttt{int}, since that would require coding up exceptions as numbers, much like “error numbers” in Unix. Again, completely unworkable in practice, and completely non-modular (different modules are bound to conflict over their numbering scheme).

A more reasonable choice would be to define \(\tau_{\text{exn}}\) to be a distinguished datatype, say \texttt{exc}. For example, we might have the declaration...
data type exc = Div | Overflow | Match | Bind

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

\[
\text{fun div (m, 0) = raise Div} \\
| \text{div (m, n) = ... raise Overflow ...}
\]

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:

\[
\text{exception Div} \\
\text{exception Overflow}
\]

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 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. It just so happens that the exception mechanism uses this type as the type of values carried with exceptions. But the type is meaningful and useful entirely independently of the exception control mechanism! In particular, the exception declaration merely introduces a “new” value constructor that has no connection whatsoever with any active or inactive exception handler.

## 21.4 Exercises

1. Hand-simulate the evaluation of a few simple expressions with exceptions and handlers to get a feeling for how it works.
2. State and prove the safety of the formulation of exceptions using a handler stack.

3. Prove that the H machine indeed maintains the invariants stated above.
Chapter 22

Coroutines and Cooperative Threads

A coroutine consists of two routines that pass control back and forth between them in a completely symmetric manner — in a sense each is a subroutine of the other. A simple model of cooperative threads arises by regarding each thread to be a coroutine of a scheduler.

22.1 Coroutines

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

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

147
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 con-
tents. 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 an-
other 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 fol-
lowing 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 corre-
sponding 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
Cooperative Threads

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 cooperative 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 imple-
mented by creating a new thread. Yielding is achieved by enqueueing the
current thread and dispatching; exiting is a simple dispatch, abandoning
the current thread entirely. This implementation is suggestive of a slogan
suggested by Olin Shivers: “A thread is a trajectory through continuation
space”. During its lifetime a thread of control is represented by a succes-
sion of continuations that are enqueued onto and dequeued from the ready
queue.

Here is a simple implementation of threads:

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

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

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

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

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

There is much more to be said about threads! 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 code.
22.3 Exercises
Part VII

Polymorphism and Data Abstraction
Chapter 23
Polymorphism

MinML is an explicitly typed language. The abstract syntax is defined to have sufficient type information to ensure that all expressions have a unique type. In particular the types of the parameters of a function must be chosen when the function is defined.

While this is not itself a serious problem, it does expose a significant weakness in the MinML type system. For example, there is no way to define a generic procedure for composing two functions whose domain and range match up appropriately. Instead we must define a separate composition operation for each choice of types for the functions being composed. Here is one composition function

\[
\text{fun } (f:\text{string} \to \text{int}: (\text{char} \to \text{string}) \to (\text{string} \to \text{int}) \text{ is}
\text{fun } (g:\text{char} \to \text{string}):\text{string} \to \text{int} \text{ is}
\text{fun } (x:\text{string}):\text{int} \text{ is apply}(f, \text{apply}(g, x)),
\]

and here is another

\[
\text{fun } (f:\text{float} \to \text{double}: (\text{int} \to \text{float}) \to (\text{int} \to \text{double}) \text{ is}
\text{fun } (g:\text{int} \to \text{float}):\text{int} \to \text{double} \text{ is}
\text{fun } (x:\text{int}):\text{double} \text{ is apply}(f, \text{apply}(g, x)).
\]

The annoying thing is that both versions of function composition execute the same way; they differ only in the choice of types of the functions being composed. This is rather irksome, and very quickly gets out of hand in practice. Statically typed languages have long been criticized for precisely this reason. Fortunately this inflexibility is not an inherent limitation of statically typed languages, but rather a limitation of the particular type system we have given to MinML. A rather straightforward extension is
sufficient to provide the kind of flexibility that is essential for a practical language. This extension is called *polymorphism*.

While ML has had such a type system from its inception (circa 1978), few other languages have followed suit. Notably the Java language suffers from this limitation (but the difficulty is mitigated somewhat in the presence of subtyping). Plans are in the works, however, for adding polymorphism (called *generics*) to the Java language. A compiler for this extension, called Generic Java, is already available.

## 23.1 A Polymorphic Language

*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.
23.1 A Polymorphic Language

Abstract Syntax

The abstract syntax of PolyMinML is defined by the following extension to the MinML grammar:

- **Polytypes**  \( \sigma : : = \tau | \forall t(\sigma) \)
- **Monotypes**  \( \tau : : = \ldots | t \)
- **Expressions**  \( e : : = \ldots | \text{Fun } t \text{ in } e | \text{inst}(e, \tau) \)
- **Values**  \( v : : = \ldots | \text{Fun } t \text{ in } e \)

The variable \( t \) ranges over a set of type variables, which are written ML-style ‘\( a \)', ‘\( b \)', and so on in examples. In the polytype \( \forall t(\sigma) \) the type variable \( t \) is bound in \( \sigma \); we do not distinguish between polytypes that differ only in the names of bound variables. Since the quantifier can occur only at the outermost level, in ML it is left implicit. An expression of the form \( \text{Fun } t \text{ in } e \) is a polymorphic function with parameter \( t \) and body \( e \). The variable \( t \) is bound within \( e \). An expression of the form \( \text{inst}(e, \tau) \) is a polymorphic instantiation of the polymorphic function \( e \) at monotype \( \tau \). Notice that we may only instantiate polymorphic functions with monotypes. In examples we write \( f [\tau] \) for polymorphic instantiation, rather than the more verbose \( \text{inst}(f, \tau) \).

We write \( \text{FTV}(\tau) \) (respectively, \( \text{FTV}(\sigma), \text{FTV}(e) \)) for the set of free type variables occurring in \( \tau \) (respectively, \( \sigma, e \)). Capture-avoiding substitution of a monotype \( \tau \) for free occurrences of a type variable \( t \) in a polytype \( \sigma \) (resp., monotype \( \tau' \), expression \( e \)) is written \( [t\leftarrow\tau]\sigma \) (resp., \( [t\leftarrow\tau]\tau' \), \( [t\leftarrow\tau]e \)).

Static Semantics

The static semantics of PolyMinML is a straightforward extension to that of MinML. One significant change, however, is that we must now keep track of the scopes of type variables, as well as ordinary variables. In the static semantics of MinML a typing judgement had the form \( \Gamma \vdash e : \tau \), where \( \Gamma \) is a context assigning types to ordinary variables. Only those variables in \( \text{dom}\Gamma \) may legally occur in \( e \). For PolyMinML we must introduce an additional context, \( \Delta \), which is a set of type variables, those that may legally occur in the types and expression of the judgement.

The static semantics consists of rules for deriving the following two judgements:

\[ \Delta \vdash \sigma \text{ ok} \quad \sigma \text{ is a well-formed type in } \Delta \]
\[ \Gamma \vdash_\Delta e : \sigma \quad e \text{ is a well-formed expression of type } \sigma \text{ in } \Gamma \text{ and } \Delta \]
The rules for validity of types are as follows:

\[
\begin{align*}
\frac{t \in \Delta}{\Gamma \vdash t \text{ ok}} & \quad (23.1) \\
\Delta \vdash \text{int} \text{ ok} & \quad (23.2) \\
\Delta \vdash \text{bool} \text{ ok} & \quad (23.3) \\
\Delta \vdash \tau_1 \text{ ok} & \Delta \vdash \tau_2 \text{ ok} \\
\Delta \vdash \text{arrow}(\tau_1, \tau_2) \text{ ok} & \quad (23.4) \\
\Delta \cup \{t\} \vdash \sigma \text{ ok} & t \not\in \Delta \\
\Delta \vdash \forall \ell(\sigma) \text{ ok} & \quad (23.5)
\end{align*}
\]

The auxiliary judgement $\Delta \vdash \Gamma$ is defined by the following rule:

\[
\frac{\Gamma(\ell) \text{ ok } (\forall \ell \in \text{dom}(\Gamma))}{\Delta \vdash \Gamma \text{ ok}} \quad (23.6)
\]

The rules for deriving typing judgements $\Gamma \vdash \Delta e : \sigma$ are as follows. We give only the rules specific to PolyMinML; the remaining rules are those of MinML, augmented with a set $\Delta$ of type variables.

\[
\begin{align*}
\frac{\Gamma \vdash \Delta \cup \{t\} \ell : \sigma & \quad t \not\in \Delta}{\Gamma \vdash \Delta \text{ Fun } t \text{ in } \ell : \forall \ell(\sigma)} & \quad (23.7) \\
\frac{\Gamma \vdash \ell \Delta e : \forall \ell(\sigma) & \Delta \vdash \tau \text{ ok}}{\Gamma \vdash \Delta \text{ inst}(e, \tau) : \{t \leftarrow \tau\} \sigma} & \quad (23.8)
\end{align*}
\]

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 _(_:t):v is apply(f, apply(g, x))
```
23.1 A Polymorphic Language

It is easy to check that it has type
\[ \forall t(\forall u(\forall v(\text{arrow}(\text{arrow}(u,v)),\text{arrow}(\text{arrow}(t,u),(\text{arrow}(t,v))))))). \]

We will need the following technical lemma stating that typing is preserved under instantiation:

**Lemma 23.1 (Instantiation)**
If \( \Gamma \vdash \Delta \cup \{t\} e : \sigma \), where \( t \notin \Delta \), and \( \Delta \vdash \tau \) ok, then \( [t \leftarrow \tau] \Gamma \vdash [t \leftarrow \tau] e : [t \leftarrow \tau] \sigma \).

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 23.2 (Canonical Forms)**
If \( v : \forall t(\sigma) \), then \( v = \text{Fun} \ t \ \text{in} \ e \) 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:

\[
\text{inst}(\text{Fun} \ t \ \text{in} \ e, \tau) \mapsto [t \leftarrow \tau] e \tag{23.9}
\]

\[
\text{inst}(e, \tau) \mapsto \text{inst}(e', \tau) \tag{23.10}
\]

It is then a simple matter to prove safety for this language.

**Theorem 23.3 (Preservation)**
If \( e : \sigma \) and \( e \mapsto e' \), then \( e' : \sigma \).

The proof is by induction on evaluation.

**Theorem 23.4 (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, 

\textit{albeit} one in which polymorphic abstraction and instantiation is explicit, rather than implicit, as it is in ML. To obtain the first-class variant of PolyMinML, we simply ignore the distinction between poly- and mono-types, regarding them all as simply types. Everything else remains unchanged, including the proofs of progress and preservation.

With first-class polymorphism we may consider types such as

\[
\text{arrow}(\forall t (\text{arrow}(t)), \forall t (\text{arrow}(t))),
\]

which cannot be expressed in the ML-like fragment. This is the type of functions that accept a polymorphic function as argument and yield a polymorphic function (of the same type) as result. If \( f \) has the above type, then \( f(\text{Fun} \ t \ x : t : \text{is} \ x) \) is well-formed. However, the application \( f(\text{fun} \ (x : \text{int}) : \text{int} \is \+(x, 1)) \) is ill-formed, because the successor function does not have type \( \forall t (\text{arrow}(t, t)) \). 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):

\[
\forall t (\text{arrow}((\text{arrow}(t, t), (\text{arrow}(t, t)))),
\]

This is the type of polymorphic functions that, for each type \( t \), accept a function on \( t \) and yield another function on \( t \). If \( g \) has this type, the expression \( \text{inst}(g, \text{int})(\text{succ}) \) is well-formed, since we first instantiate \( g \) at \( \text{int} \), then apply it to the successor function.

The situation gets more interesting in the presence of data structures such as lists and reference cells. It is a worthwhile exercise to consider the difference between the types \( \forall t \sigma\text{list} \) and \( \forall t (\sigma\text{list}) \) for various choices of \( \sigma \). Note once again that the former type cannot be expressed in ML, whereas the latter can.

Recall the following counterexample to type soundness for the early version of ML without the so-called value restriction:

\[
\begin{align*}
\text{let} & \quad \text{val } r : ('a -> 'a) \text{ ref} = \text{ref (fn } x : '\text{a} \Rightarrow x) \\
& \quad \text{in} \\
& \quad r := (\text{fn } x : \text{int} \Rightarrow x+1) ; (！r) (\text{true}) \\
\end{align*}
\]
23.1 A Polymorphic Language

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 \( \forall t (\text{arrow}(t, t) \text{ ref}) \) of polymorphic functions yielding reference cells containing a function from a type to itself.

2. The type \( \forall t (\text{arrow}(t, t)) \text{ ref} \) of reference cells containing polymorphic functions yielding a function from a type to itself.

(Notice the similarity to the distinctions discussed above.) For this example to be well-formed, we rely on an inconsistent reading of the example. At the point of the \texttt{val} binding we are treating \( r \) as a value of the latter type, namely a reference cell containing a polymorphic function. But in the body of the \texttt{let} we are treating it as a value of the former type, a polymorphic function yielding a reference cell. We cannot have it both ways at once!

To sort out the error let us make the polymorphic instantiation and abstraction explicit. Here’s one rendering:

\[
\text{let}
\begin{align*}
\quad \text{val } r : \text{All } 'a (('a \rightarrow 'a) \text{ ref}) &= \\
\quad \text{Fun } 'a \in \text{ref } (\text{fn } x:'a \Rightarrow x) \text{ end}
\end{align*}
\text{in}
\begin{align*}
\quad r[\text{int}] &= (\text{fn } x:\text{int} \Rightarrow x+1) ; ( !(r[\text{bool}]) )(\text{true})
\end{align*}
\text{end}
\]

Notice that we have made the polymorphic abstraction explicit, and inserted corresponding polymorphic instantiations. This example is type correct, and hence (by the proof of safety above) sound. But notice that it allocates \textit{two} reference cells, not \textit{one}! Recall that polymorphic functions are values, and the binding of \( r \) is just such a value. Each of the two instances of \( r \) executes the body of this function separately, each time allocating a new reference cell. Hence the unsoundness goes away!

Here’s another rendering that is, in fact, ill-typed (and should be, since it “gets stuck”!).

February 4, 2006   Working Draft
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.

### 23.2 ML-style Type Inference

ML-style type inference may be viewed as a translation from the implicitly typed syntax of ML to the explicitly-typed syntax of PolyMinML. Specifically, the type inference mechanism performs the following tasks:

- Attaching type labels to function arguments and results.
- Inserting polymorphic abstractions for declarations of polymorphic type.
- Inserting polymorphic instantiations whenever a polymorphic declared variable is used.

Thus in ML we may write

\[
\text{val I : 'a -> 'a = fn x => x}
\]
\[
\text{val n : int = I(I)(3)}
\]

This stands for the PolyMinML declarations\(^1\)

\(^1\)We’ve not equipped PolyMinML with a declaration construct, but you can see from the example how this might be done.
val I : \forall t . \text{arrow}(t,t) = \text{Fun} \text{in} \text{fun}_t(x:t):t = x
val n : \text{int} = \text{inst}(\text{I,arrow(int,int)})(\text{inst}(:,:,\text{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 \text{arrow(int,int)}, the second at type \text{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 . ((\text{arrow}(t,t))\text{ref}).

As we’ve seen, when viewed in this way, the example is not problematic, provided that polymorphic abstractions are seen as values. For in this case the two instances of r generate two distinct reference cells, and no difficulties arise. Unfortunately, ML does not treat polymorphic abstractions as values! Only one reference cell is allocated, which, in the absence of the value restriction, would lead to unsoundness.

Why does the value restriction save the day? In the case that the polymorphic expression is not a value (in the ML sense) the polymorphic abstraction that is inserted by the type inference mechanism changes a non-value into a value! This changes the semantics of the expression (as we’ve seen, from allocating one cell, to allocating two different cells), which violates the semantics of ML itself.\(^2\) However, if we limit ourselves to values in the first place, then the polymorphic abstraction is only ever wrapped around a value, and no change of semantics occurs. Therefore\(^3\), 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 \text{ref} (\text{fn} x \Rightarrow x) is not a value, but such is the nature of the value restriction.

\(^2\)One could argue that the ML semantics is incorrect, which leads to a different language.
\(^3\)This would need to be proved, of course.
Chapter 24

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.
24.1 Existential Types

The syntax of PolyMinML is extended with the following constructs:

\[
\begin{align*}
\text{Polytypes} & : \quad \sigma ::= \ldots | \exists t \sigma \\
\text{Expressions} & : \quad e ::= \ldots | \text{pack } \tau \text{ with } e \text{ as } \sigma \\
\text{Values} & : \quad v ::= \ldots | \text{pack } \tau \text{ with } v \text{ as } \exists t \sigma
\end{align*}
\]

The polytype \(\exists t(\sigma)\) is called an existential type. An existential type is the interface of an abstract type. An implementation of the existential type \(\exists t(\sigma)\) is a package value of the form \(\text{pack } \tau \text{ with } v \text{ as } \exists t(\sigma)\) consisting of a monotype \(\tau\) together with a value \(v\) of type \([t \leftarrow \tau]\sigma\). The monotype \(\tau\) is the representation type of the implementation; the value \(v\) is the implementation of the operations of the abstract type. A client makes use of an implementation by opening it within a scope, written \(\text{open } e_1 \text{ as } t \text{ with } x : \sigma \text{ in } e_2\), where \(e_1\) is an implementation of the interface \(\exists t(\sigma)\), and \(e_2\) is the client code defined in terms of an unknown type \(t\) (standing for the representation type) and an unknown value \(x\) of type \(\sigma\) (standing for the unknown operations).

In an existential type \(\exists t(\sigma)\) the type variable \(t\) is bound in \(\sigma\), and may be renamed at will to satisfy uniqueness requirements. In an expression of the form \(\text{open } e_1 \text{ as } t \text{ with } x : \sigma \text{ in } e_2\), the type variable \(t\) and the ordinary variable \(x\) are bound in \(e_2\), and may also be renamed at will to satisfy non-occurrence requirements. As we will see below, renaming of bound variables is crucial for ensuring that an abstract type is "new" in the sense of being distinct from any other type whenever it is opened for use in a scope. This is sometimes called generativity of abstract types, since each occurrence of 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.

24.1.1 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:

\[
\begin{align*}
\Delta \cup \{ t \} & \vdash \sigma \text{ ok } \quad t \notin \Delta \\
\hline
\Delta & \vdash \exists t(\sigma) \text{ ok }
\end{align*}
\]
24.1 Existential Types

The requirement \( t \not\in \Delta \) may always be met by renaming the bound variable.

The typing rule for packages is as follows:

\[
\frac{\Delta \vdash \tau \text{ ok} \quad \Delta \vdash \exists \tau \sigma \text{ ok} \quad \Gamma \vdash \epsilon : [t \leftarrow \tau] \sigma}{\Gamma \vdash \text{pack } \tau \text{ with } \epsilon \text{ as } \exists \tau \sigma}
\]  
(24.2)

The implementation, \( \epsilon \), of the operations “knows” the representation type, \( \tau \), of the ADT.

The typing rule for opening a package is as follows:

\[
\frac{\Delta \vdash \tau_c \text{ ok} \quad \Gamma, x: \sigma \vdash_{\Delta \cup \{t\}} \epsilon_c : \tau_c \quad \Gamma \vdash \epsilon_i : \exists \tau \sigma \quad t \not\in \Delta}{\Gamma \vdash \text{open } \epsilon_i \text{ as } t \text{ with } x : \sigma \text{ in } \epsilon_c : \tau_c}
\]  
(24.3)

This is a complex rule, so study it carefully! Two things to note:

1. The type of the client, \( \tau_c \), must not involve the abstract type \( t \). This prevents the client from attempting to export a value of the abstract type outside of the scope of its definition.

2. The body of the client, \( \epsilon_c \), is type checked without knowledge of the representation type, \( t \). The client is, in effect, polymorphic in \( t \).

As usual, the condition \( t \not\in \Delta \) can always be met by renaming the bound variable \( t \) of the \textit{open} expression to ensure that it is distinct from all other active types \( \Delta \). It is in this sense that abstract types are “new”! Whenever a client opens a package, it introduces a local name for the representation type, which is bound within the body of the client. By our general conventions on bound variables, this local name may be chosen to ensure that it is distinct from any other such local name that may be in scope, which ensures that the “new” type is different from any other type currently in scope. At an informal level this ensures that the representation type is “held abstract”; we will make this intuition more precise in Section ?? below.

24.1.2 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:

\[
\frac{\epsilon \mapsto \epsilon'}{\text{pack } \tau \text{ with } \epsilon \text{ as } \sigma \mapsto \text{pack } \tau \text{ with } \epsilon' \text{ as } \sigma}
\]  
(24.4)
Opening a package begins by evaluating the package expressions:

\[ e_i \mapsto e'_i \]

\[ \text{open } e_i \text{ as } t \text{ with } x : \sigma \text{ in } e_c \mapsto \text{open } e'_i \text{ as } t \text{ with } x : \sigma \text{ in } e_c \] (24.5)

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:

\[ \sigma = \exists t'(\sigma') \]

\[ \text{open } (\text{pack } \tau \text{ with } v \text{ as } \sigma') \text{ as } t \text{ with } x : \sigma' \text{ in } e_c \mapsto [t, x \leftarrow \tau, v] e_c \] (24.6)

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.

### 24.1.3 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 24.1 (Preservation)**

If \( e : \tau \) and \( e \mapsto e' \), then \( e' : \tau \).

**Lemma 24.2 (Canonical Forms)**

If \( v : \exists t(\sigma) \) is a value, then \( v = \text{pack } \tau \text{ with } v' \text{ as } \exists t(\sigma) \) for some monotype \( \tau \) and some value \( v' : [t \leftarrow \tau]_{\sigma} \).

**Theorem 24.3 (Progress)**

If \( e : \tau \) then either \( e \text{ value} \) or there exists \( e' \) such that \( e \mapsto e' \).

### 24.2 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:

<table>
<thead>
<tr>
<th>PolyMinML + Existentials</th>
<th>Standard ML</th>
</tr>
</thead>
<tbody>
<tr>
<td>Existential type</td>
<td>Signature</td>
</tr>
<tr>
<td>Package</td>
<td>Structure, with opaque ascription</td>
</tr>
<tr>
<td>Opening a package</td>
<td>open declaration</td>
</tr>
</tbody>
</table>
Here is an example of these correspondences in action. In the sequel we will use ML-like notation with the understanding that it is to be interpreted in PolyMinML in the following fashion.

Here is an ML signature for a persistent representation of queues:

```ml
signature QUEUE =
sig
  type queue
  val empty : queue
  val insert : int * queue -> queue
  val remove : queue -> int * queue
end
```

This signature is deliberately stripped down to simplify the development. In particular we leave undefined the meaning of `remove` on an empty queue.

The corresponding existential type is $\sigma_q = \exists q(\tau_q)$, where

$$\tau_q : = q*(\text{arrow}((\text{int}*q),q))*(\text{arrow}(q, (\text{int}*q)))$$

That is, the operations of the abstraction consist of a three-tuple of values, one for the empty queue, one for the insert function, and one for the remove function.

Here is a straightforward implementation of the QUEUE interface in ML:

```ml
structure QL :> QUEUE =
struct
  type queue = int list
  val empty = nil
  fun insert (x, xs) = x::xs
  fun remove xs =
    let val (x,xs') = rev xs in (x, rev xs') end
end
```

A queue is a list in reverse enqueue order — the last element to be enqueued is at the head of the list. Notice that we use opaque signature ascription to ensure that the type queue is hidden from the client!

The corresponding package is $e_q : = \text{pack int list with } v_q \text{ as } \sigma_q$, where

$$v_q : = (\text{nil},(v_i,v_r))$$

where $v_i$ and $v_r$ are the obvious function abstractions corresponding to the ML code given above.

Finally, a client of an abstraction in ML might typically open it within a scope:
This corresponds to writing

```
local
  open QL
in
  ...
end
```

in the existential type formalism, using pattern matching syntax for tuples.
Part VIII

Cost Semantics and Parallelism
Chapter 25

Cost Semantics

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

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

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

25.1 Evaluation Semantics

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

\[
\frac{}{n \Downarrow n} \quad (25.1)
\]

\[
\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{+ (e_1, e_2) \Downarrow n_1 + n_2} \quad (25.2)
\]
174 25.2 Relating Evaluation Semantics to Transition Semantics

(and similarly for the other primitive operations).

\begin{align*}
\text{true} & \Downarrow \text{true} \quad \text{false} \Downarrow \text{false} \\
\text{if } e \text{ then } e_1 \text{ else } e_2 \Downarrow v & \quad (25.4) \\
\text{if } e \text{ then } e_1 \text{ else } e_2 \Downarrow v & \quad (25.5)
\end{align*}

\begin{align*}
\text{fun } f (x: \tau_1) : \tau_2 \text{ is } e & \Downarrow \text{fun } f (x: \tau_1) : \tau_2 \text{ is } e' \quad (25.6) \\
 e_1 \Downarrow v_1 & \quad e_2 \Downarrow v_2 \quad [f, x \leftarrow v_1, v_2]e \Downarrow v \\
\text{apply}(e_1, e_2) \Downarrow v & \quad (25.7)
\end{align*}

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

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

25.2 Relating Evaluation Semantics to Transition Semantics

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

**Theorem 25.1**

1. If \(e \Downarrow v\), then \(e \rightarrow^* v\).

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

**Proof:**

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

Since \( v_1 \Downarrow v_1 \) and \( v_2 \Downarrow v_2 \), the result follows immediately from the assumption that \( e' \Downarrow v \). Now suppose that \( e = \text{apply}(e_1, e_2) \) and \( e' = \text{apply}(e'_1, e_2) \), where \( e_1 \rightarrow e'_1 \). Assume that \( e' \Downarrow v \); we are to show that \( e \Downarrow v \). It follows that \( e'_1 \Downarrow v_1, e_2 \Downarrow v_2, \) and \( [f, x \leftarrow v_1, v_2]e \Downarrow v \). By induction \( e_1 \Downarrow v_1 \), and hence \( e \Downarrow v \). The remaining cases are handled similarly. It follows by induction on the rules defining multi-step evaluation that if \( e \rightarrow^* v \), then \( e \Downarrow v \). The base case, \( v \rightarrow^* v \), follows from the fact that \( v \Downarrow v \). Now suppose that \( e \rightarrow e' \rightarrow^* v \). By induction \( e' \Downarrow v \), and hence \( e \Downarrow v \) by what we have just proved.

\[ \square \]

25.3 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 12.

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.

\[
\begin{align*}
\frac{}{n \Downarrow^0 n} & \quad \quad (25.8) \\
\frac{e_1 \Downarrow^{k_1} n_1 \quad e_2 \Downarrow^{k_2} n_2}{+ (e_1, e_2) \Downarrow^{k_1+k_2+1} n_1 + n_2} & \quad (25.9)
\end{align*}
\]

(and similarly for the other primitive operations).

\[
\begin{align*}
\text{true} \Downarrow^0 \text{true} \quad & \quad \text{false} \Downarrow^0 \text{false} \quad (25.10) \\
\frac{e \Downarrow^k \text{true} \quad e_1 \Downarrow^{k_1} v}{\text{if } e \text{ then } e_1 \text{ else } e_2 \Downarrow^{k+k_1+1} v} & \quad (25.11) \\
\frac{e \Downarrow^k \text{false} \quad e_2 \Downarrow^{k_2} v}{\text{if } e \text{ then } e_1 \text{ else } e_2 \Downarrow^{k+k_2+1} v} & \quad (25.12)
\end{align*}
\]
25.4 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 \( e \stackrel{k}{\rightarrow} e' \) by induction on \( k \) as follows. For the basis, we define \( e \stackrel{0}{\rightarrow} e' \) iff \( e = e' \); if \( k = k' + 1 \), we define \( e \stackrel{k}{\rightarrow} e' \) to hold iff \( e \stackrel{k'}{\rightarrow} e'' \) to hold.

**Theorem 25.2**

For any closed expression \( e \) and closed value \( v \) of the same type, \( e \Downarrow^k v \) iff \( e \stackrel{k}{\rightarrow} v \).

**Proof:** From left to right we proceed by induction on the definition of the cost semantics. For example, consider the rule for function application. We have \( e = \text{apply}(e_1, e_2) \) and \( k = k_1 + k_2 + k + 1 \), where

1. \( e_1 \Downarrow^{k_1} v_1 \),
2. \( e_2 \Downarrow^{k_2} v_2 \),
3. \( v_1 = \text{fun } f (x : \tau_1) : \tau_2 \text{ is } e \),
4. \( [f, x \leftarrow v_1, v_2]e \Downarrow^k v \).

By induction we have

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

and hence

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

which is enough for the result.

From right to left we proceed by induction on \(k\). For \(k = 0\), we must have \(e = v\). By inspection of the cost evaluation rules we may check that \(v \downarrow^0 v\) for every value \(v\). For \(k = k' + 1\), we must show that if \(e \mapsto e'\) and \(e' \downarrow^{k'} v\), then \(e \downarrow^k v\). This is proved by a subsidiary induction on the transition rules. For example, suppose that \(e = e_1(e_2) \mapsto e'_1(e_2) = e'\), with \(e_1 \mapsto e'_1\). By hypothesis \(e'_1(e_2) \downarrow^k v\), so \(k = k_1 + k_2 + k_3 + 1\), where

1. \(e'_1 \downarrow^{k_1} v_1\),
2. \(e_2 \downarrow^{k_2} v_2\),
3. \(v_1 = \text{fun } f (x: \tau_1): \tau_2 \text{ is } e\),
4. \([f, x \leftarrow v_1, v_2]e \downarrow^{k_3} v\).

By induction \(e_1 \downarrow^{k_1 + 1} v_1\), hence \(e \downarrow^{k + 1} v\), as required. ■
Chapter 26

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.

26.1 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 26.3 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 ?? but now write $e \rightarrow^{seq} e'$ for the transition relation to stress that this is the sequential semantics. The sequential evaluation rules for pairs are as follows:

\[
\frac{e_1 \rightarrow^{seq} e_1'}{(e_1, e_2) \rightarrow^{seq} (e_1', e_2)}
\] (26.1)

\[
\frac{v_1 \text{ value } e_2 \rightarrow^{seq} e_2'}{(v_1, e_2) \rightarrow^{seq} (v_1, e_2')}
\] (26.2)

\[
\frac{v_1 \text{ value } v_2 \text{ value}}{\text{split } (v_1, v_2) \text{ as } (x, y) \text{ in } e \rightarrow^{seq} [x, y \leftarrow v_1, v_2] e}
\] (26.3)

\[
\frac{e_1 \rightarrow^{seq} e_1'}{\text{split } e_1 \text{ as } (x, y) \text{ in } e_2 \rightarrow^{seq} \text{split } e_1' \text{ as } (x, y) \text{ in } e_2}
\] (26.4)

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:

\[
\frac{e_1 \rightarrow^{par} e_1' \quad e_2 \rightarrow^{par} e_2'}{(e_1, e_2) \rightarrow^{par} (e_1', e_2')}
\] (26.5)

\[
\frac{e_1 \rightarrow^{par} e_1' \quad v_2 \text{ value}}{(e_1, v_2) \rightarrow^{par} (e_1', v_2)}
\] (26.6)

\[
\frac{v_1 \text{ value } e_2 \rightarrow^{par} e_2'}{(v_1, e_2) \rightarrow^{par} (v_1, e_2')}
\] (26.7)

Three rules are required to account for the possibility that evaluation of one component may complete before the other.

When presented two semantics for the same language, it is natural to ask whether they are equivalent. They are, in the sense that both semantics deliver the same value for any expression. This is the precise statement of what we mean by “implicit parallelism”.

---

1It might be preferable to admit progress on either $e_1$ or $e_2$ alone, without requiring the other to be a value.
Theorem 26.1
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 $e \mapsto^{*}_{seq} e'$, $e' \mapsto^{*}_{par} v$, then $e \mapsto^{*}_{par} v$. This is proved by induction on the sequential evaluation relation. For example, suppose that 

$$(e_1, e_2) \mapsto^{seq}_{seq} (e'_1, e_2) \mapsto^{*}_{par} (v_1, v_2),$$

where $e_1 \mapsto^{seq}_{seq} e'_1$. 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 

$$(e_1, e_2) \mapsto^{par}_{par} (e'_1, e'_2) \mapsto^{*}_{seq} (v_1, v_2)$$

with $e_1 \mapsto^{par}_{par} e'_1$ and $e_2 \mapsto^{par}_{par} e'_2$. 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.

26.2 Work and Depth

An operational semantics for a language induces a measure of time complexity for expressions, namely the number of steps required to evaluate that expression to a value. The sequential complexity of an expression is its time complexity relative to the sequential semantics; the parallel complexity is its time complexity relative to the parallel semantics. These can, in general, be quite different. Consider, for example, the following naïve implementation of the Fibonacci sequence in MinML with products:
fun fib (n:int):int is
  if n=0 then 1
  else if n=1 then 1
  else plus(fib(n-1),fib(n-2)) fi fi

where plus is the following function on ordered pairs:

fun plus (p:int*int):int is
  split p as (m:int,n:int) in m+n

The sequential complexity of fib \( n \) is \( O(2^n) \), whereas the parallel complexity of the same expression is \( O(n) \). The reason is that each recursive call spawns two further recursive calls which, if evaluated sequentially, lead to an exponential number of steps to complete. However, if the two recursive calls are evaluated in parallel, then the number of parallel steps to completion is bounded by \( n \), since \( n \) is decreased by 1 or 2 on each call. Note that the same number of arithmetic operations is performed in each case! The difference is only in whether they are performed simultaneously.

This leads naturally to the concepts of work and depth. The work of an expression is the total number of primitive instruction steps required to complete evaluation. Since the sequential semantics has the property that each rule has at most one premise, each step of the sequential semantics amounts to the execution of exactly one instruction. Therefore the sequential complexity coincides with the work required. (Indeed, work and sequential complexity are often taken to be synonymous.) The work required to evaluate fib \( n \) is \( O(2^n) \).

On the other hand the depth of an expression is the length of the longest chain of sequential dependencies in a complete evaluation of that expression. A sequential dependency is induced whenever the value of one expression depends on the value of another, forcing a sequential evaluation ordering between them. In the Fibonacci example the two recursive calls have no sequential dependency among them, but the function itself sequentially depends on both recursive calls — it cannot return until both calls have returned. Since the parallel semantics evaluates both components of an ordered pair simultaneously, it exactly captures the independence of the two calls from each, but the dependence of the result on both. Thus the parallel complexity coincides with the depth of the computation. (Indeed, they are often taken to be synonymous.) The depth of the expression fib \( n \) is \( O(n) \).

With this in mind, the cost semantics introduced in Chapter 25 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:

$$
\frac{\begin{array}{c}
e_1 \downarrow^{w_1,d_1} v_1 \\
\ne_2 \downarrow^{w_2,d_2} v_2 
\end{array}}{\begin{array}{c}
(e_1,e_2) \downarrow^{w_1+w_2,\max(d_1,d_2)} (v_1,v_2)
\end{array}} \quad (26.8)
$$

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 26.2**

For any closed, well-typed expression $e$, $e \downarrow^{w,d} v$ iff $e \rightarrow^{w}_{\text{seq}} v$ and $e \rightarrow^{d}_{\text{par}} v$.

**Proof:** From left to right, we proceed by induction on the cost semantics. For example, we must show that if $e_1 \rightarrow^{d_1}_{\text{par}} v_1$ and $e_2 \rightarrow^{d_2}_{\text{par}} v_2$, then

$$
(e_1,e_2) \rightarrow^{d}_{\text{par}} (v_1,v_2),
$$

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 $e_1 \rightarrow^{d_1}_{\text{par}} v_1$ and $e_2 \rightarrow^{d_2}_{\text{par}} v_2$. It follows that

$$
(e_1,e_2) \rightarrow^{d_1}_{\text{par}} (v_1,e_2')
\rightarrow^{d_2}_{\text{par}} (v_1,v_2).
$$

For the converse, we proceed by considering work and depth costs separately. For work, we proceed as in Chapter 25. For depth, it suffices to show that if $e \rightarrow^{d}_{\text{par}} e'$ and $e' \downarrow^{d} v$, then $e \downarrow^{d+1} v$.\(^3\) For example, suppose that $(e_1,e_2) \rightarrow_{\text{par}} (e_1',e_2')$, with $e_1 \rightarrow^{d_1}_{\text{par}} v_1$ and $e_2 \rightarrow^{d_2}_{\text{par}} v_2$. 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.  

\(^2\)If we choose, we might evaluate arguments of primop’s in parallel, in which case the depth complexity would be calculated as one more than the maximum of the depths of its arguments. We will not do this here since it would only complicate the development.

\(^3\)The work component of the cost is suppressed here for the sake of clarity.
26.3 Vector Parallelism

To support vector parallelism we will extend MinML with a type of vectors, which are finite sequences of values of a given type whose length is not determined until execution time. The primitive operations on vectors are chosen so that they may be executed in parallel on a shared memory multiprocessor, or SMP, in constant depth for an arbitrary vector.

The following primitives are added to MinML to support vectors:

\[
\begin{align*}
\text{Types} & \quad \tau & \vdash \tau \text{ vector} \\
\text{Expr's} & \quad e & \vdash [e_0, \ldots, e_{n-1}] | \text{elt}(e_1, e_2) | \text{size}(e) | \text{index}(e) | \text{map}(e_1, e_2) | \text{update}(e_1, e_2) \\
\text{Values} & \quad v & \vdash [v_0, \ldots, v_{n-1}]
\end{align*}
\]

These expressions may be informally described as follows. The expression \([e_0, \ldots, e_{n-1}]\) evaluates to an \(n\)-vector whose elements are given by the expressions \(e_i, 0 \leq i < n\). The operation \(\text{elt}(e_1, e_2)\) retrieves the element of the vector given by \(e_1\) at the index given by \(e_2\). The operation \(\text{size}(e)\) returns the number of elements in the vector given by \(e\). The operation \(\text{index}(e)\) creates a vector of length \(n\) (given by \(e\)) whose elements are \(0, \ldots, n-1\). The operation \(\text{map}(e_1, e_2)\) applies the function given by \(e_1\) to every element of \(e_2\) in parallel. Finally, the operation \(\text{update}(e_1, e_2)\) yields a new vector of the same size, \(n\), as the vector \(v\) given by \(e_1\), but whose elements are updated according to the vector \(v'\) given by \(e_2\). The elements of \(e_2\) are triples of the form \((b, i, x)\), where \(b\) is a boolean flag, \(i\) is a non-negative integer less than or equal to \(n\), and \(x\) is a value, specifying that the \(i\)th element of \(v\) should be replaced by \(x\), provided that \(b = \text{true}\).

The static semantics of these primitives is given by the following typing rules:

\[
\Gamma \vdash e_1 : \tau \quad \cdots \quad \Gamma \vdash e_n : \tau \\
\Gamma \vdash [e_0, \ldots, e_{n-1}] : \tau \text{ vector} \tag{26.9}
\]

\[
\Gamma \vdash e_1 : \tau \text{ vector} \quad \Gamma \vdash e_2 : \text{int} \\
\Gamma \vdash \text{elt}(e_1, e_2) : \tau \tag{26.10}
\]

\[
\Gamma \vdash e : \tau \text{ vector} \\
\Gamma \vdash \text{size}(e) : \text{int} \tag{26.11}
\]

\[
\Gamma \vdash e : \text{int} \\
\Gamma \vdash \text{index}(e) : \text{int vector} \tag{26.12}
\]
The parallel dynamic semantics is given by the following rules. The most important is the parallel evaluation rule for vector expressions, since this is the sole source of parallelism:

\[
\forall i \in I \ (e_i \mapsto_{\text{par}} e'_i) \quad \forall i \notin I \ (e'_i = e_i \land e_i \text{ value})
\]

(26.15)

where \( \emptyset \neq I \subseteq \{0, \ldots, n-1\} \). This allows for the parallel evaluation of all components of the vector that have not yet been evaluated.

For each of the primitive operations of the language there is a rule specifying that its arguments are evaluated in left-to-right order. We omit these rules here for the sake of brevity. The primitive instructions are as follows:

\[
\text{elt}([v_0, \ldots, v_{n-1}], i) \mapsto_{\text{par}} v_i
\]

(26.16)

\[
\text{size}([v_0, \ldots, v_{n-1}]) \mapsto_{\text{par}} n
\]

(26.17)

\[
\text{index}(n) \mapsto_{\text{par}} [0, \ldots, n-1]
\]

(26.18)

\[
\text{map}(v, [v_0, \ldots, v_{n-1}]) \mapsto_{\text{par}} [\text{apply}(v, v_0), \ldots, \text{apply}(v, v_{n-1})]
\]

(26.19)

\[
\text{update}([v_0, \ldots, v_{n-1}], [(b_0, i_0, x_0), \ldots, (b_{k-1}, i_{k-1}, x_{k-1})])
\]

\[
\mapsto_{\text{par}} [v'_0, \ldots, v'_{n-1}]
\]

(26.20)

where for each \( i \in \{i_0, \ldots, i_{k-1}\} \), if \( b_i \) is \text{true}, then \( v'_i = x_i \), and otherwise \( v'_i = v_i \). If an index \( i \) appears more than once, the rightmost occurrence takes precedence over the others.
The sequential dynamic semantics of vectors is defined similarly to the parallel semantics. The only difference is that vector expressions are evaluated in left-to-right order, rather than in parallel. This is expressed by the following rule:

\[ e_i \mapsto_{\text{seq}} e'_i \]

\[ [v_0, \ldots, v_{i-1}, e_i, e_{i+1}, \ldots, e_{n-1}] \mapsto [v_0, \ldots, v_{i-1}, e'_i, e_{i+1}, \ldots, e_{n-1}] \]

(26.21)

We write \( e \mapsto_{\text{seq}} e' \) to indicate that \( e \) steps to \( e' \) under the sequential semantics.

With these two basic semantics in mind, we may also derive a cost semantics for MinML with vectors, where the work corresponds to the number of steps required in the sequential semantics, and the depth corresponds to the number of steps required in the parallel semantics. The rules are as follows.

Vector expressions are evaluated in parallel.

\[ \forall 0 \leq i < n \ (e_i \Downarrow_{\text{seq}}^{w_i, d_i} v_i) \]

\[ [e_0, \ldots, e_{n-1}] \Downarrow_{\text{seq}}^{w, d} [v_0, \ldots, v_{n-1}] \]

(26.22)

where \( w = \sum_{i=0}^{n-1} w_i \) and \( d = \max_{i=0}^{n-1} d_i \).

Retrieving an element of a vector takes constant work and depth.

\[ e_1 \Downarrow_{\text{seq}}^{w_1, d_1} [v_0, \ldots, v_{n-1}] \quad e_2 \Downarrow_{\text{seq}}^{w_2, d_2} i \quad (0 \leq i < n) \]

\[ \text{elt}(e_1, e_2) \Downarrow_{\text{seq}}^{w_1 + w_2 + 1, d_1 + d_2 + 1} v_i \]

(26.23)

Retrieving the size of a vector takes constant work and depth.

\[ e \Downarrow_{\text{seq}}^{w, d} [v_0, \ldots, v_{n-1}] \]

\[ \text{size}(e) \Downarrow_{\text{seq}}^{w+1, d+1} n \]

(26.24)

Creating an index vector takes linear work and constant depth.

\[ e \Downarrow_{\text{seq}}^{w, d} n \]

\[ \text{index}(e) \Downarrow_{\text{seq}}^{w+n, d+1} [0, \ldots, n - 1] \]

(26.25)
26.3 Vector Parallelism

Mapping a function across a vector takes constant work and depth beyond the cost of the function applications.

\[
e_1 \Downarrow^{w_1,d_1} \nu \quad e_2 \Downarrow^{w_2,d_2} [v_0, \ldots, v_{n-1}]
\]

\[
[\text{apply}(\nu, v_0), \ldots, \text{apply}(\nu, v_{n-1})] \Downarrow^{w,d} [v'_0, \ldots, v'_{n-1}]
\]

\[
\text{map}(e_1, e_2) \Downarrow^{w_1+w_2+w+1,d_1+d_2+d+1} [v'_0, \ldots, v'_{n-1}]
\]

(26.26)

Updating a vector takes linear work and constant depth.

\[
e_1 \Downarrow^{w_1,d_1} [v_0, \ldots, v_{n-1}] \quad e_2 \Downarrow^{w_2,d_2} [(b_1, i_1, x_1), \ldots, (b_k, i_k, x_k)]
\]

\[
\text{update}(e_1, e_2) \Downarrow^{w_1+w_2+k+n,d_1+d_2+1} [v'_0, \ldots, v'_{n-1}]
\]

(26.27)

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 26.3**

*For the extension of MinML with vectors, \(e \Downarrow^{w,d} \nu\) iff \(e \mapsto_{\text{par}}^d \nu\) and \(e \mapsto_{\text{seq}}^w \nu\).*
Chapter 27

A Parallel Abstract Machine

The parallel operational semantics described in Chapter 26 abstracts away some important aspects of the implementation of parallelism. For example, the parallel evaluation rule for ordered pairs

\[
e_1 \overset{\text{par}}{\rightarrow} e_1' \quad e_2 \overset{\text{par}}{\rightarrow} e_2'
\]

\[
\langle e_1, e_2 \rangle \overset{\text{par}}{\rightarrow} \langle e_1', e_2' \rangle
\]

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.

27.1 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.
Types \( \tau : = \) int | bool | unit | \( \tau_1 \times \tau_2 \) | arrow(\( \tau_1, \tau_2 \))

Expressions \( e : = \) \( \nu \) | let \( x_1 : \tau_1 \) be \( e_1 \) and \( x_2 : \tau_2 \) be \( e_2 \) in \( e \) end |
\( o(v_1, ..., v_n) \) | if \( \tau \) then \( e_1 \) else \( e_2 \) |
apply(\( v_1, v_2 \)) | split \( v \) as \( (x_1, x_2) \) in \( e \)

Values \( v : = \) \( \nu \) | \( n \) | true | false | () | (\( v_1, v_2 \)) | fun \( x (y : \tau_1) : \tau_2 \) is \( e \)

The binding conventions are as for MinML with product types, with the additional specification that the variables \( x_1 \) and \( x_2 \) are bound within the body of a let expression. Note that variables are regarded as values only for the purpose of defining the syntax of the language; evaluation is, as ever, defined only on closed terms.

As will become apparent when we specify the dynamic semantics, the “sequential let” is definable from the “parallel let”:

\[
\text{let} \; \tau_1 : x_1 \; \text{be} \; e_1 \; \text{in} \; e_2 \; = \; \text{let} \; x_1 : \tau_1 \; \text{be} \; e_1 \; \text{and} \; x : \text{unit} \; \text{be} \; () \; \text{in} \; e_2 \end{equation}

where \( x \) does not occur free in \( e_2 \). Using these, the “parallel pair” is definable by the equation

\[
(e_1, e_2)_{\text{par}} : = \text{let} \; x_1 : \tau_1 \; \text{be} \; e_1 \; \text{and} \; x_2 : \tau_2 \; \text{be} \; e_2 \; \text{in} \; (x_1, x_2) \; \text{end}
\]

whereas the “(left-to-right) sequential pair” is definable by the equation

\[
(e_1, e_2)_{\text{seq}} : = \text{let} \; \tau_1 : x_1 \; \text{be} \; e_1 \; \text{in} \; \text{let} \; \tau_2 : x_2 \; \text{be} \; e_2 \; \text{in} \; (x_1, x_2).
\]

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:

\[
\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e : \tau \\
\Gamma \vdash \text{let} \; x_1 : \tau_1 \; \text{be} \; e_1 \; \text{and} \; x_2 : \tau_2 \; \text{be} \; e_2 \; \text{in} \; e \; \text{end} : \tau
\]

(27.1)

It is a simple exercise to give a parallel structured operational semantics to this language in the style of Chapter 26. In particular, it would employ the following rules for the parallel let construct.

\[
\begin{align*}
e_1 \mapsto_{\text{par}} e'_1 & \quad e_2 \mapsto_{\text{par}} e'_2 \\
\text{let} \; x_1 : \tau_1 \; \text{be} \; e_1 \; \text{and} \; x_2 : \tau_2 \; \text{be} \; e_2 \; \text{in} \; e \; \text{end} & \mapsto_{\text{par}} e'_1 \text{ and } e'_2 \text{ in } e \end{align*}
\]

(27.2)
27.2 A Parallel Abstract Machine

The essence of parallelism is the simultaneous execution of several programs. Each execution is called a thread of control, or thread, for short. The problem of devising a parallel abstract machine is how to represent multiple threads of control, in particular how to represent the creation of new threads and synchronization between threads. The $P$-machine is designed to represent a parallel computer with an unbounded number of processors in a simple and elegant manner.

The main idea of the $P$-machine is to represent the state of a parallel computer by a nested composition of parallel let statements representing the active threads in a program. Each step of the machine consists of executing all of the active instructions in the program, resulting in a new $P$-state.

In order to account for the activation of threads and the synchronization of their results we make explicit the process of activating an expression, which corresponds to assigning it to a processor for execution. Execution of a parallel let instruction whose constituent expressions have not yet been activated consists of the activation of these expressions. Execution of a parallel let whose constituents are completely evaluated consists of substituting the values of these expressions into the body of the let, which is itself then activated. Execution of all other instructions is exactly as before, with the result being made active in each case.

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 27.1
Prove preservation and progress for the static and dynamic semantics just given.
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:

\[
\begin{align*}
\mathcal{L} & := l : - | l : \text{let } x_1 : \tau_1 \text{ be } L_1 \text{ and } x_2 : \tau_2 \text{ be } L_2 \text{ in } e \end{align*}
\]

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:

\[
\begin{align*}
e_1 & \rightarrow e'_1 \quad \cdots \quad e_n \rightarrow e'_n \quad \mathcal{L}[l = e]_{i=1}^{n} & \mapsto \mathcal{L}[l = e']_{i=1}^{n}
\end{align*}
\]

The relation \( e \rightarrow e' \) 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:

\[
\begin{align*}
\text{let } x_1 : \tau_1 \text{ be } e_1 \text{ and } x_2 : \tau_2 \text{ be } e_2 \text{ in } e \quad & \rightarrow \quad \text{let } x_1 : \tau_1 \text{ be } l_1 : e_1 \text{ and } x_2 : \tau_2 \text{ be } l_2 : e_2 \text{ in } e \tag{27.5}
\end{align*}
\]

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.
27.3 Cost Semantics, Revisited

The second instruction is the join axiom, which completes execution of a parallel let:

\[
\begin{align*}
\text{let } x_1 : \tau_1 \text{ be } l_1 : v_1 \text{ and } x_2 : \tau_2 \text{ be } l_2 : v_2 \text{ in } e \text{ end} \longrightarrow [x_1, x_2 \leftarrow v_1, v_2] e
\end{align*}
\]

(27.6)

The other instructions are inherited from the M-machine. For example, function application is defined by the following instruction:

\[
\begin{align*}
\text{apply}(v_1, v_2) \longrightarrow [f, x \leftarrow v_1, v_2] e
\end{align*}
\]

(27.7)

This completes the definition of the P-machine.

**Exercise 27.2**

State and prove preservation and progress relative to the P-machine.

27.3 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 26 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:

\[
\begin{align*}
e_1 \Downarrow_{w_1, d_1} v_1 \quad e_2 \Downarrow_{w_2, d_2} v_2 \quad [x_1, x_2 \leftarrow v_1, v_2] e \Downarrow_{w, d} v \\
\text{let } x_1 : \tau_1 \text{ be } e_1 \text{ and } x_2 : \tau_2 \text{ be } e_2 \text{ in } e \text{ end} \Downarrow_{w', d'} v
\end{align*}
\]

(27.8)

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 \( e \rightarrow^* P v \) is the total number of primitive instruction steps performed in the sequence; it is the sequential cost of executing the expression \( e \).

**Theorem 27.3**

If \( e \Downarrow_{w, d} v \), then \( l : e \rightarrow^d_P l : v \) with work \( w \).
Proof: The proof from left to right proceeds by induction on the cost semantics. For example, consider the cost semantics of the parallel let construct. By induction we have

1. \( l_1 : e_1 \rightarrow_p^{d_1} l_1 : v_1 \) with work \( w_1 \);
2. \( l_2 : e_2 \rightarrow_p^{d_2} l_2 : v_2 \) with work \( w_2 \);
3. \( l : [x_1, x_2 \leftarrow v_1, v_2] e \rightarrow_p^d l : v \) with work \( w \).

We therefore have the following P-machine evaluation sequence:

\[
\begin{align*}
&l : \text{let } x_1 : \tau_1 \text{ be } e_1 \text{ and } x_2 : \tau_2 \text{ be } e_2 \text{ in } e \end{align*}
\]

\( \rightarrow_p \)

\[
\begin{align*}
&l : \text{let } x_1 : \tau_1 \text{ be } l_1 : e_1 \text{ and } x_2 : \tau_2 \text{ be } l_2 : e_2 \text{ in } e \end{align*}
\]

\( \rightarrow_p^{\max(d_1, d_2)} \)

\[
\begin{align*}
&l : \text{let } x_1 : \tau_1 \text{ be } l_1 : v_1 \text{ and } x_2 : \tau_2 \text{ be } l_2 : v_2 \text{ in } e \end{align*}
\]

\( \rightarrow_p \)

\[
\begin{align*}
&l : [x_1, x_2 \leftarrow v_1, v_2] e \end{align*}
\]

\( \rightarrow_p^d \)

\[
\begin{align*}
&l : v
\end{align*}
\]

The total length of the evaluation sequence is \( \max(d_1, d_2) + d + 2 \), as required by the depth cost, and the total work is \( w_1 + w_2 + w + 2 \), as required by the work cost.

27.4 Provable Implementations (Summary)

The semantics of parallelism given above is based on an idealized parallel computer with an unlimited number of processors. In practice this idealization must be simulated using some fixed number, \( p \), of physical processors. In practice \( p \) is on the order of 10’s of processors, but may even rise (at the time of this writing) into the 100’s. In any case \( p \) does not vary with input size, but is rather a fixed parameter of the implementation platform. The important question is how efficiently can one simulate unbounded parallelism using only \( p \) processors? That is, how realistic are the costs assigned to the language by our semantics? Can we make accurate predictions about the running time of a program on a real parallel computer based on the idealized cost assigned to it by our semantics?

The answer is yes, through the notion of a provably efficient implementation. While a full treatment of these ideas is beyond the scope of this book, it is worthwhile to summarize the main ideas.
Theorem 27.4 (Blelloch and Greiner)

If $v \downarrow^{w,d} v$, then $v$ can be evaluated on an SMP with $p$-processors in time $O(w/p + d \lg p)$.

For our purposes, an SMP is any of a wide range of parallel computers, including a CRCW PRAM, a hypercube, or a butterfly network. Observe that for $p = 1$, the stated bound simplifies to $O(w)$, as would be expected.

To understand the significance of this theorem, observe that the definition of work and depth yields a lower bound of $\Omega(\max(w/p, d))$ on the execution time on $p$ processors. We can never complete execution in fewer than $d$ steps, and can, at best, divide the total work evenly among the $p$ processors. The theorem tells us that we can come within a constant factor of this lower bound. The constant factor, $\lg p$, represents the overhead of scheduling parallel computations on $p$ processors.

The goal of parallel programming is to maximize the use of parallelism so as to minimize the execution time. By the theorem this will occur if the term $w/p$ dominates, which occurs if the ratio $w/d$ of work to depth is at least $p \lg p$. This ratio is sometimes called the parallelizability of the program. For highly sequential programs, $d$ is directly proportional to $w$, yielding a low parallelizability — increasing the number of processors will not speed up the computation. For highly parallel programs, $d$ might be constant or proportional to $\lg w$, resulting in a large parallelizability, and good utilization of the available computing resources. It is important to keep in mind that it is not known whether there are inherently sequential problems (for which no parallelizable solution is possible), or whether, instead, all problems can benefit from parallelism. The best that we can say at the time of this writing is that there are problems for which no parallelizable solution is known.

To get a sense of what is involved in the proof of Blelloch and Greiner’s theorem, let us consider the assumption that the index operation on vectors (given in Chapter 26) has constant depth. The theorem implies that index is implementable on an SMP in time $O(n/p + \lg p)$. We will briefly sketch a proof for this one case. The main idea is that we may assume that every processor is assigned a unique number from 0 to $p - 1$. To implement index, we simply allocate, but do not initialize, a region of memory of the appropriate size, and ask each processor to simultaneously store its identifying number $i$ into the $i$th element of the allocated array. This works directly if the size of the vector is no more than the number of processors. Otherwise, we may divide the problem in half, and recursively build two index vectors of half the size, one starting with zero, the other with $n/2$. 

February 4, 2006 Working Draft
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 IX

Imperative Programming
Chapter 28

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 20 and 21. Store effects are dynamic modifications to mutable storage. This chapter is concerned with store effects.

28.1 References

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

$$e ::= l | \text{ref}(e) | !e | e_1 := e_2$$

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

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

\[
\frac{\Lambda(l) = \tau} {\Lambda; \Gamma \vdash l : \tau \text{ref}} \quad (28.1)
\]

\[
\frac{\Lambda; \Gamma \vdash e : \tau} {\Lambda; \Gamma \vdash \text{ref}(e) : \tau \text{ref}} \quad (28.2)
\]

\[
\frac{\Lambda; \Gamma \vdash e : \tau \text{ref}} {\Lambda; \Gamma \vdash !e : \tau} \quad (28.3)
\]

\[
\frac{\Lambda; \Gamma \vdash e_1 : \tau_2 \text{ref} \quad \Lambda; \Gamma \vdash e_2 : \tau_2} {\Lambda; \Gamma \vdash e_1 := e_2 : \tau_2} \quad (28.4)
\]

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

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

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

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

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

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

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

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

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

Working Draft February 4, 2006
A state \((M, e)\) is \textit{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:

\[
\begin{align*}
\text{dom}(M) &= \text{dom}(\Lambda) \quad \forall l \in \text{dom}(\Lambda) \quad \Lambda(l) &\vdash M(l) : \Lambda(l) \\
\end{align*}
\]

\(\vdash M : \Lambda \) (28.12)

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

The typing rule for memories is reminiscent of the typing rule for recursive functions — we are allowed to assume the typing that we are trying to prove while trying to prove it. This similarity is no accident, as the following example shows. Here we use ML notation, but the example can be readily translated into MinML extended with references:
28.1 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 28.1

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 $\Lambda$ such that $\vdash M : \Lambda$.

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

$$
\frac{\vdash M : \Lambda \quad \Lambda; \vdash e : \tau}{(M,e) \text{ ok}}
$$

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

Theorem 28.2 (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 $\Lambda; \vdash e : \tau$, then there exists $\Lambda' \supseteq \Lambda$ such that $\vdash M' : \Lambda'$ and $\Lambda'; \vdash e' : \tau$.

Exercise 28.3

Prove Theorem 28.2. 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 28.4 (Progress)**

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

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

**Exercise 28.5**

Prove Theorem 28.4 by induction on typing of machine states.
Chapter 29

Monads

As we saw in Chapter 28 one way to combine functional and imperative programming is to add a type of reference cells to MinML. This approach works well for call-by-value languages, because we can easily predict where expressions are evaluated, and hence where references are allocated and assigned. For call-by-name languages this approach is problematic, because in such languages it is much harder to predict when (and how often) expressions are evaluated.

Enriching ML with a type of references has an additional consequence that one can no longer determine from the type alone whether an expression mutates storage. For example, a function of type \( \text{arrow}(\text{int}, \text{int}) \) 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 \( \text{arrow}(\text{int}, \text{int}) \) is a pure function (has no effects when evaluated), whereas a function of type \( \text{int} \rightarrow \text{int} \) may have an effect when applied. The monadic approach is more popular for call-by-name languages, but is equally sensible for call-by-value languages.

---

1We need to introduce cbv and cbn earlier, say in Chapter 12.
29.1 A Monadic Language

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:

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

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

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

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

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

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

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

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

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

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

Their typing rules are as follows:

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

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

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

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

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

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

The dynamic semantics of monadic MinML is an extension to that of MinML. Evaluation of pure expressions does not change, but we must
add rules governing evaluation of impure expressions. For the purposes of describing mutable storage, we must consider transitions of the form 

\[(M, m) \xrightarrow{\tau} (M', m')\], where M and M' are memories, as in Chapter 28.

\[
e \mapsto e'
\]

\[
(M, \text{return } e) \mapsto (M, \text{return } e')
\]

\[
(M, m_1) \mapsto (M', m'_1)
\]

\[
(M, \text{bind } x: \tau \leftarrow m_1 \text{ in } m_2) \mapsto (M', \text{bind } x: \tau \leftarrow m'_1 \text{ in } m_2)
\]

\[
(M, \text{bind } x: \tau \leftarrow \text{return } v \text{ in } m_2) \mapsto (M, [x \leftarrow v]m_2)
\]

The evaluation rules for the reference primitives are as in Chapter 28.

### 29.2 Reifying Effects

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

The syntax required for this extension is as follows:

**Types**

\[
\tau : : = !\tau
\]

**Pure**

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

**Impure**

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

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

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

\[
\Gamma \vdash m : \tau \\
\Gamma \vdash \text{box}(m) : !\tau
\]

\[
\Gamma \vdash e : !\tau \\
\Gamma \vdash \text{unbox}(e) : \tau
\]
The dynamic semantics is given by the following transition rules:

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

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

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

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

is replaced by the pure function expression

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

The impure application,

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

is replaced by

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

which unboxes, hence executes, the suspended computation.

29.3 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 X

Lazy Evaluation
Chapter 30

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.

### 30.1 Lazy Types

We will first explore the addition of lazy data types to a strict functional language. We will focus on a specific example, the type of lazy lists. For the sake of simplicity we’ll consider only lazy lists of integers, but nothing hinges on this assumption.\(^1\) For the rest of this section we’ll drop the modifier “lazy”, and just write “list”, instead of “lazy list”.

The key idea is to treat a computation of a list element as a value of list type, where a computation is simply a memoized, delayed evaluation of an expression. By admitting computations as values we can support lazy lists in a strict language. In particular the call-by-value evaluation strategy is not disrupted. Passing a lazy list to a function does not cause the delayed computation to be evaluated; rather, it is passed in delayed form to the function as a computation of that type. Pattern matching on a value of list type requires that the computation be forced to expose the underlying

---

\(^1\)It simply allows us to avoid forward-referencing the concept of polymorphism.
list element, which is then analyzed and deconstructed. It is very important to keep in mind the distinction between evaluation of an expression of list type, and forcing a value of list type. The former simply yields a computation as value, whereas the latter evaluates and memoizes the delayed computation.

One consequence of laziness is that the tail of a (non-empty) lazy list, need not “exist” at the time the non-empty list is created. Being itself a lazy list, the tail need only be produced “on demand”, by forcing a computation. This is the key to using lazy lists to model interactive input and to represent infinite data structures. For example, we might define the infinite list of natural numbers by the equation

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

where the function \text{iterate} is defined (informally) by the equation

\[ \text{iterate \ } f \ x = \text{lcons \ } (x, \text{iterate \ } f \ (f \ x)), \]

where \text{lcons} creates a non-empty lazy list with the specified head and tail. We must think of \text{nats} as being created on demand. Successive elements of \text{nats} are created by successive recursive calls to \text{iterate}, which are only made as we explore the list.

Another approach to defining the infinite list of natural numbers is to make use of self-reference, as illustrated by the following example. The infinite sequence of natural numbers may be thought as a solution to the recursion equation

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

where \text{successor} and \text{lmap} are the evident functions. Here again we must think of \text{nats} as being created on demand. Successive elements of \text{nats} are created as follows. When we inspect the first element of \text{nats}, it is immediately revealed to be 0, as specified. When we inspect the second element, we apply \text{lmap \ successor} to \text{nats}, then inspect the head element of the result. This is \text{successor}(0), or 1; it’s tail is the result of mapping \text{successor} over that list — that is, the result of adding 2 to every element of the original list, and so on.
30.1 Lazy Types

30.1.1 Lazy Lists in an Eager Language

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

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

In the expression \text{lazy}\ x\ \text{is}\ e the variable \(x\) is bound within \(e\); in the expression \text{lcase}\ e\ \text{of}\ \text{lnil}\Rightarrow e_0 \mid \text{lcons}(x, y)\Rightarrow 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 \text{lnil} and \text{lcons} — or by a recursion equation — using \text{lazy}\ x\ \text{is}\ e, where \(e\) is a lazy list expression. The idea is that the variable \(x\) stands for the list constructed by \(e\), and may be used within \(e\) to refer to the list itself. For example, the infinite list of 1’s is given by the expression

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

More interesting examples can be expressed using recursive definitions such as the following definition of the list of all natural numbers:

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

To complete this definition we must define \text{lmap}. This raises a subtle issue that is very easy to overlook. A natural choice is as follows:

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

Unfortunately this definition doesn’t work as expected! Suppose that \(f\) is a function of type \text{int}\rightarrow\text{int} and that \(l\) is a non-empty lazy list. Consider what happens when we evaluate the expression \text{map}\ f\ l. The \text{lcase} forces evaluation of \(l\), which leads to a recursive call to the internal function \text{lmapf}, which forces the evaluation of the tail of \(l\), and so on. If \(l\) is an infinite list, the application diverges.

The problem is that the result of a call to \text{map}\ f\ l should be represented by a \text{computation} of a list, in which subsequent calls to \text{map} on the tail(s) of that list are delayed until they are needed. This is achieved by the following coding trick:
fun map(f:int->int):llist->llist is
    fun lmapf(l:llist) is
        lazy 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 30.1
Give a formal definition of nats in terms of iterate according to the informal equation given earlier. You will need to make use of lazy function definitions.

The static semantics of these lazy list expressions is given by the following typing rules:

\[ \Gamma \vdash \text{lnil} : \text{llist} \]  \hspace{1cm} (30.1)

\[ \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{llist} \]
\[ \Gamma \vdash \text{lcons}(e_1, e_2) : \text{llist} \]  \hspace{1cm} (30.2)

\[ \Gamma, x : \text{llist} \vdash e : \text{llist} \]
\[ \Gamma \vdash \text{lazy} x \text{ is} e : \text{llist} \]  \hspace{1cm} (30.3)

\[ \Gamma \vdash e : \text{llist} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \text{int}, y : \text{llist} \vdash e_1 : \tau \]
\[ \Gamma \vdash \text{lcase} e \text{ of lnil} \Rightarrow e_0 | \text{lcons}(x, y) \Rightarrow e_1 : \tau \]  \hspace{1cm} (30.4)
In Rule 30.2 the body, $e$, of the lazy list expression $\text{lazy } x \text{ is } e$ is type checked under the assumption that $x$ is a lazy list.

We will consider two forms of dynamic semantics for lazy lists. The first, which exposes the “evaluate on demand” character of lazy evaluation, but neglects the “evaluate at most once” aspect, is given as follows. First, we regard $\text{lnil}$, $\text{lcons}(e_1, e_2)$, and $\text{lazy } x \text{ is } e$ to be values, independently of whether their constituent expressions are values. Second, we evaluate case analyses according to the following transition rules:

\[
\begin{align*}
\text{lcase lnil of lnil} & \Rightarrow e_0 \mid \text{lcons}(x, y) \Rightarrow e_1 \Rightarrow e_0 \quad (30.5) \\
\text{lcase lcons}(e_h, e_t) \text{ of lnil} & \Rightarrow e_0 \mid \text{lcons}(x, y) \Rightarrow e_1 \\
& \Rightarrow \quad \text{let } x: \text{int} \text{ be } e_h \text{ in } \text{let } y: \text{llist} \text{ be } e_t \text{ in } e_1 \quad (30.6) \\
\text{lcase (lazy } z \text{ is } e) \text{ of lnil} & \Rightarrow e_0 \mid \text{lcons}(x, y) \Rightarrow e_1 \\
\text{lcase } [z \leftarrow \text{lazy } z \text{ is } e] \text{ of lnil} & \Rightarrow e_0 \mid \text{lcons}(x, y) \Rightarrow e_1 \\
\text{let } x: \text{int} \text{ be } e_h \text{ in } y: \text{llist} \text{ be } e_t \text{ in } e_1 \\
\end{align*}
\]

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 30.2**

Define the functions $\text{lhd: llist} \rightarrow \text{int}$ and $\text{ltl: llist} \rightarrow \text{llist}$. Trace the evaluation of $\text{lhd}(\text{ltl}(\ldots(\text{ltl}(\text{nats}))\ldots))$, with $n$ iterations of $\text{ltl}$, and verify that it evaluates to the number $n$.

**Exercise 30.3**

State and prove the soundness of the non-memoizing dynamic semantics with respect to the static semantics given above.
Consider the lazy list value \( v = \text{lazy } x \text{ is } x \). It is easy to verify that \( e \) is well-typed, with type \( \text{list} \). It is also easy to see that performing a case analysis on \( v \) leads to an infinite regress, since \([x \leftarrow v] x = v\). The value \( v \) is an example of a “black hole”, a value that, when forced, will lead back to the value itself, and, moreover, is easily seen to lead to divergence. Another example of a black hole is the value

\[
\text{lazy } x \text{ is } (\text{lmap succ } x)
\]

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

What is it that makes the recursive list

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

well-defined? This expression is \textit{not} a black hole because the occurrence of \text{nats} in the body of the recursive list expression is “guarded” by the call to \text{lcons}.

**Exercise 30.4**

Develop a type discipline that rules out black holes as ill-formed. Hint: Define a judgement \( \Gamma \vdash e \downarrow x \), which means that \( x \) is guarded within \( e \). Ensure that \( \text{lazy } x \text{ is } e \) is well-typed only if \( x \) is guarded within \( e \).

**Exercise 30.5**

It is often convenient to define several lists simultaneously by mutual recursion. Generalize \( \text{lazy } x \text{ is } e \) to admit simultaneous recursive definition of several lists at once.

The foregoing dynamic semantics neglects the “evaluate at most once” aspect of laziness — if a lazy list expression is ever evaluated, its value should be stored so that re-evaluation is avoided should it ever be analyzed again. This can be modeled by introducing a memory that holds delayed computations whenever they are created. The memory is updated if (and only if) the value of that computation is ever required. Thus no evaluation is ever repeated, and some pending evaluations may never occur at all. This is called \textit{memoization}.

The memoizing dynamic semantics is specified by an abstract machine with states of the form \((M, e)\), where \( M \) is a memory, a finite mapping of variables to values, and \( e \) is an expression whose free variables are all in the domain of \( M \). Free variables are used to stand for the values of list expressions; they are essentially pointers into the memory, which stores the value of the expression. We therefore regard free variables as values; these are in fact the only values of list type in this semantics.
The transition rules for the memoizing dynamic semantics are as follows:

\[(x \not\in \text{dom}(M)) \quad (M, \text{lazy } z \text{ is } e) \quad \mapsto \quad (M[x=\text{lazy } z \text{ is } e], x) \quad (30.9)\]

\[(x \not\in \text{dom}(M)) \quad (M, \text{lnil}) \quad \mapsto \quad (M[x=\text{lnil}], x) \quad (30.10)\]

\[(x \not\in \text{dom}(M)) \quad (M, \text{lcons}(e_1, e_2)) \quad \mapsto \quad (M[x=\text{lcons}(e_1, e_2)], x) \quad (30.11)\]

\[(M(z) = \text{lnil}) \quad (M, \text{lcase } z \text{ of } \text{lnil} \Rightarrow e_0 \mid \text{lcons}(x, y) \Rightarrow e_1) \quad \mapsto \quad (M, e_0) \quad (30.12)\]

\[(M(z) = \text{lcons}(v_h, v_t)) \quad (M, \text{lcase } z \text{ of } \text{lnil} \Rightarrow e_0 \mid \text{lcons}(x, y) \Rightarrow e_1) \quad \mapsto \quad (M, x, y \leftarrow v_h, v_t), e_1) \quad (30.13)\]

\[(M(z) = \text{lcons}(e_h, e_t)) \quad (M[z=\bullet], e_h) \quad \mapsto^* (M', v_h) \quad (M'[z=\bullet], e_t) \quad \mapsto^* (M'', v_t) \quad (M, \text{lcase } z \text{ of } \text{lnil} \Rightarrow e_0 \mid \text{lcons}(x, y) \Rightarrow e_1) \quad \mapsto \quad (M', v_h) \quad (M''[z=\text{lcons}(v_h, v_t)], [x, y \leftarrow v_h, v_t], e_1) \quad (30.14)\]

\[(M(z) = \text{lcase } z \text{ is } e) \quad (M[z=\bullet], e) \quad \mapsto^* (M', v) \quad (M, \text{lcase } z \text{ of } \text{lnil} \Rightarrow e_0 \mid \text{lcons}(x, y) \Rightarrow e_1) \quad \mapsto \quad (M', e) \quad (M'[z=v], \text{lcase } v \text{ of } \text{lnil} \Rightarrow e_0 \mid \text{lcons}(x, y) \Rightarrow e_1) \quad (30.15)\]

\[\]

\[(M, e) \quad \mapsto \quad (M', e') \quad (M, \text{lcase } e \text{ of } \text{lnil} \Rightarrow e_0 \mid \text{lcons}(x, y) \Rightarrow e_1) \quad \mapsto \quad (M', e') \quad (30.16)\]

\[
\]

**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 `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 30.6**
Convince yourself that the replacement of $z$ by $\bullet$ in the second premise of Rule 30.14 is redundant — the location $z$ is already guaranteed to be bound to $\bullet$.

**Exercise 30.7**
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 30.8**
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 30.9**
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 30.10
Consider again the definition of the natural numbers as the lazy list

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

Prove that, for the non-memoized semantics, that accessing the \( n \)th element requires \( O(n^2) \) time, whereas in the memoized semantics the same computation requires \( O(n) \) time. This shows that memoization can improve the asymptotic complexity of an algorithm (not merely lower the constant factors).

30.1.2 Delayed Evaluation and Lazy Data Structures

Another approach to lazy evaluation in the context of a strict language is to isolate the notion of a delayed computation as a separate concept. The crucial idea is that a delayed computation is a value that can, for example, appear in a component of a data structure. Evaluation of a delayed computation occurs as a result of an explicit force operation. Computations are implicitly memoized in the sense that the first time it is forced, its value is stored and returned immediately should it ever be forced again. Lazy data structures can then be built up using standard means, but with judicious use of delayed computations to ensure laziness.

Since the technical details of delayed computation are very similar to those just outlined for lazy lists, we will go through them only very briefly. Here is a syntactic extension to MinML that supports delayed evaluation:

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

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

The static semantics is given by the following rules:

\[
\Gamma \vdash e : \tau \\
\Gamma \vdash \text{delay} \, x \, \text{is} \, e : \tau \text{ computation} \quad (30.17)
\]

\[
\Gamma \vdash e_1 : \tau_1 \text{ computation} \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \\
\Gamma \vdash \text{eval} \, e_1 \, \text{as} \, x \, \text{in} \, e_2 : \tau_2 \quad (30.18)
\]
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:

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

\[(M, \text{delay } x \text{ is } e) \mapsto (M[x=\text{delay } x \text{ is } e], x)\] (30.19)

\[(M(z) = \text{delay } z \text{ is } e) \mapsto (M'[z=v], [x\leftarrow v]e)\] (30.20)

\[(M(z) = v) \mapsto (M', [x\leftarrow v]e)\] (30.21)

\[(M, e_1) \mapsto (M', e'_1)\]

\[(M, \text{eval } e_1 \text{ as } x \text{ in } e_2) \mapsto (M', \text{eval } e'_1 \text{ as } x \text{ in } e_2)\] (30.22)

**Exercise 30.11**

*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:

\[
\Gamma \vdash \text{cnil} : \text{lcell}
\] (30.23)

\[
\Gamma \vdash e_h : \text{int} \quad \Gamma \vdash e_i : \text{llist} \\
\Gamma \vdash \text{ccons}(e_h, e_i) : \text{lcell}
\] (30.24)

\[
\Gamma \vdash e : \text{lcell} \quad \Gamma \vdash e_n : \tau \quad \Gamma, x:\text{int}, y:\text{llist} \vdash e_c : \tau \\
\Gamma \vdash \text{ccase } e \text{ of cnil } \Rightarrow e_n | \text{ccons}(x,y) \Rightarrow e_c : \tau
\] (30.25)

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:
30.1 Lazy Types

\begin{verbatim}
lnil = lazy _ is cnil
lcons(e_h,e_l) = lazy _ is ccons(e_h, e_l)
\end{verbatim}

\begin{verbatim}
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
\end{verbatim}

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 \texttt{llist} with \texttt{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 \texttt{llist} of lazy lists (delayed computations of cells) and the type \texttt{lcell} of lazy cells (which specify emptiness and define the first element of non-empty lists).
Chapter 31

Lazy Languages

So far we’ve been considering the addition of lazy types to eager languages. Now we’ll consider the alternative, the notion of a lazy language and, briefly, the addition of eager types to a lazy language.

As we said in the introduction the main features of a lazy language are the call-by-need argument-passing discipline together with lazy value constructors that construct values of a type from delayed computations. Under call-by-value the arguments to functions and constructors are evaluated before the function is called or the constructor is applied. Variables are only ever bound to fully-evaluated expressions, or values, and constructors build values out of other values. Under call-by-need arguments are passed to functions in delayed, memoized form, without evaluating them until they are needed. Moreover, value constructors build delayed, memoized computations out of other delayed, memoized computations, without evaluation. Variables are, in general, bound to pending computations that are only forced when (and if) that value is required. Once forced, the binding is updated to record the computed value, should it ever be required again.

The interesting thing is that the static typing rules for the lazy variant of MinML are exactly the same as those for the eager version. What is different is how those types are interpreted. In an eager language values of type int are integer values (i.e., numbers); in a lazy language they are integer computations, some of which might not even terminate when evaluated. Similarly, in an eager language values of list type are finite sequences of values of the element type; in a lazy language values of list type are computations of such sequences, which need not be finite. And so on. The important point is that the types have different meanings in lazy languages
than they do in strict languages.

One symptom of this difference is that lazy languages are very liberal in admitting recursive definitions compared to eager languages. In an eager language it makes no sense to admit recursive definitions such as

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

or

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

Roughly speaking, neither of these recursion equations has a solution. There is no integer value \(x\) satisfying the equation \(x = 1 + x\), nor is there any finite list satisfying the equation \(x = \text{cons}(1, x)\).

However, as we’ve already seen, equations such as

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

and

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

do make sense, precisely because they define recursive computations, rather than values. The first example defines a computation of an integer that, when forced, diverges; the second defines a computation of a list that, when forced, computes a non-empty list with 1 as first element and the list itself as tail.

In a lazy language every expression stands for a computation, so it is always sensible to make a recursive definition such as

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

Syntactically this looks like the inadmissible definition discussed above, but, when taken in the context of a lazy interpretation, it makes perfect sense as a definition of a recursive computation — the value of \(x\) is the divergent computation of an integer.

The downside of admitting such a liberal treatment of computations is that it leaves no room in the language for ordinary values! Everything’s a computation, with values emerging as those computations that happen to have a trivial evaluation (e.g., numerals are trivial computations in the sense that no work is required to evaluate them). This is often touted as an advantage of lazy languages — the “freedom” to ignore whether something is a value or not. But this appearance of freedom is really bondage. By admitting only computations, you are deprived of the ability to work...
with plain values. For example, lazy languages do not have a type of natural numbers, but rather only a type of computations of natural numbers. Consequently, elementary programming techniques such as definition by mathematical induction are precluded. The baby's been thrown out with the bathwater.

In recognition of this most lazy languages now admit eager types as well as lazy types, moving them closer in spirit to eager languages that admit lazy types, but biased in the opposite direction. This is achieved in a somewhat unsatisfactory manner, by relying on data abstraction mechanisms to ensure that the only values of a type are those that are generated by specified strict functions (those that evaluate their arguments). The reason it is unsatisfactory is that this approach merely limits the possible set of computations of a given type, but still admits, for example, the undefined computation as an element of every type.

### 31.0.3 Call-by-Name and Call-by-Need

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

\[
\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \text{rec} \ x : \tau \ is \ e : \tau}
\] (31.1)

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

\[
\text{rec} \ f : \text{arrow}(\tau_1, \tau_2) \ is \ \text{fn} \ \tau_1 : x \ in \ e.
\]

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:

\[
\frac{}{\text{fn} \ \tau : x \ in \ e(e') \mapsto [x \leftarrow \text{fn} \ \tau : x \ in \ e, e']e}
\] (31.2)

This is known as the \textit{call-by-name}\(^1\) rule, according to which arguments are

---

\(^1\)The terminology is well-established, but not especially descriptive. As near as I can tell the idea is that we pass the "name" of the computation (i.e., the expression that engenders it), rather than its value.
passed to functions in unevaluated form, deferring their evaluation until
the point at which they are actually used.

The only additional rule required is the one for recursive computations. 
But this is entirely straightforward:

\[
\text{rec } x : \tau \mapsto \left[ x \leftarrow \text{rec } x : \tau \mapsto e \right] e
\] (31.3)

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

**Exercise 31.1**

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

\[\text{rec } f : \text{arrow}(\tau_1, \tau_2) \mapsto \text{fn } \tau_1 : x \mapsto e\]
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 un-
evaluated, and later, if ever, in their evaluated forms. The difference here
is that all expressions define computations that must be stored. Since the
main ideas are similar to those used to define lazy lists, we simply give the
evaluation rules here.

The state of computation is a pair \((M, e)\) where \(M\) is a finite memory
mapping variables to values, and \(e\) is an expression whose free variables
lie within the domain of \(M\). Final states have the form \((M, v)\), where \(v\) is a
closed value. In particular, \(v\) is not a variable.

Nearly all of the rules of MinML carry over to the present case nearly un-
changed, apart from propagating the memory appropriately. For example,
the rules for evaluating addition expressions is as follows:

\[
(M, e_1) \mapsto (M', e'_1)
\]

\[
(M, + (e_1, e_2)) \mapsto (M', + (e'_1, e'_2))
\] (31.4)

\[
(M, e_2) \mapsto (M', e'_2)
\]

\[
(M, + (v_1, e_2)) \mapsto (M', + (v_1, e'_2))
\] (31.5)

\[
(M, + (n_1, n_2)) \mapsto (M, n_1 + n_2)
\] (31.6)
The main differences are in the rule for function application and the need for additional rules for variables and recursive computations.

\[
\frac{(x \notin \text{dom}(M))}{(M, \text{fn}\, \tau : x \in e(e')) \mapsto (M[x = e'], e)} \quad (31.7)
\]

\[
\frac{(M(x) = v)}{(M, x) \mapsto (M, v)} \quad (31.8)
\]

\[
\frac{(M(x) = e) \quad (M[x = \bullet], e) \mapsto^* (M', v)}{(M, x) \mapsto (M'[x = v], v)} \quad (31.9)
\]

\[
\frac{(x \notin \text{dom}(M))}{(M, \text{rec}\, x : \tau \text{ is } e) \mapsto (M[x = e], e)} \quad (31.10)
\]

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

### 31.0.4 Strict Types in a Lazy Language

As discussed above, lazy languages are committed to the fundamental principle that the elements of a type are computations, which include values, and not just values themselves. This means, in particular, that every type contains a “divergent” element, the computation that, when evaluated, goes into an infinite loop.\(^2\)

One consequence, alluded to above, is that recursive type equations have overly rich solutions. For example, in this setting the recursive type equation

\[
\text{data llist = lnil | lcons of int * list}
\]

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

\(^2\)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.
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

\[
\text{rec ones is lcons (1, ones)}
\]

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 ex-
ample) by insisting that cons evaluate its tail whenever it is applied. This
is called a \textit{strictness} annotation. If cons is strict in its second argument, then
the equation

\[
\text{rec ones is cons (1, ones)}
\]

denotes the divergent computation, rather than the infinite list of ones.

These informal ideas correspond to different rules for evaluating con-
structors. We will illustrate this by giving a non-memoizing semantics for
lazy MinML extended with eager lists. It is straightforward to adapt this to
the memoizing case.

In the fully lazy case the rules for evaluation are these. First, we regard
\texttt{lnil} as a value, and regard \texttt{lcons(e1,e2)} as a value, regardless of whether
\(e_1\) or \(e_2\) are values. Then we define the transition rules for case analysis as
follows:

\[
\begin{align*}
\text{lcase lnil of lnil} & \Rightarrow e_n \\
\text{lcase lcons(e1,e2) of lnil} & \Rightarrow e_n \\
\text{lcase lcons(e1,e2) of lcons(x,y)} & \Rightarrow e_c \mapsto \text{[x,y←e1,e2]} e_c
\end{align*}
\]

If instead we wish to rule out infinite lists, then we may choose to regard
\texttt{lcons(e1,e2)} 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(e1,e2)} 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 XI

Subtyping
Chapter 32

Subtyping

A subtype relation is a pre-order\(^1\) on types that validates the subsumption principle: if \(\sigma\) is a subtype of \(\tau\), then a value of type \(\sigma\) may be provided whenever a value of type \(\tau\) is required. This means that a value of the subtype should “act like” a value of the supertype when used in supertype contexts.

32.1 Subsumption

We will consider two extensions of MinML with subtyping. The first, MinML with implicit subtyping, is obtained by adding the following rule of implicit subsumption to the typing rules of MinML:

\[
\Gamma \vdash e : \sigma, \sigma <: \tau \quad \quad \Gamma \vdash e : \tau
\]

With implicit subtyping the typing relation is no longer syntax-directed, since the subsumption rule may be applied to any expression \(e\), without regard to its form.

The second, called MinML with explicit subtyping, is obtained by adding to the syntax by adding an explicit cast expression, \((\tau) e\), with the following typing rule:

\[
\Gamma \vdash e : \sigma, \sigma <: \tau \quad \quad \Gamma \vdash (\tau) e : \tau
\]

The typing rules remain syntax-directed, but all uses of subtyping must be explicitly indicated.

\(^1\)A pre-order is a reflexive and transitive binary relation.
We will refer to either variation as MinML<: when the distinction does not matter. When it does, the implicit version is designated MinML<: i, the implicit MinML<: e.

To obtain a complete instance of MinML<: we must specify the subtype relation. This is achieved by giving a set of subtyping axioms, which determine the primitive subtype relationships, and a set of variance rules, which determine how type constructors interact with subtyping. To ensure that the subtype relation is a pre-order, we tacitly include the following rules of reflexivity and transitivity:

\[
\tau <: \tau \\
\rho <: \sigma \quad \sigma <: \tau \\
\rho <: \tau
\]

Note that pure MinML is obtained as an instance of MinML<: e by giving no subtyping rules beyond these two, so that \( \sigma <: \tau \) iff \( \sigma = \tau \).

The dynamic semantics of an instance of MinML<: must be careful to take account of subtyping. In the case of implicit subsumption the dynamic semantics must be defined so that the primitive operations of a supertype apply equally well to a value of any subtype. In the case of explicit subsumption we need only ensure that there be a means of casting a value of the subtype into a corresponding value of the supertype.

The type safety of MinML<:, in either formulation, is assured, provided that the following subtyping safety conditions are met:

- For MinML<: e, if \( \sigma <: \tau \), then casting a value of the subtype \( \sigma \) to the supertype \( \tau \) must yield a value of type \( \tau \).
- For MinML<: i, the dynamic semantics must ensure that the value of each primitive operation is defined for closed values of any subtype of the expected type of its arguments.

Under these conditions we may prove the Progress and Preservation Theorems for either variant of MinML<:.

**Theorem 32.1 (Preservation)**

For either variant of MinML<:, under the assumption that the subtyping safety conditions hold, if \( e : \tau \) and \( e \rightarrow e' \), then \( e' : \tau \).

**Proof:** By induction on the dynamic semantics, appealing to the casting condition in the case of the explicit subsumption rule of MinML<: e. 

\[\blacksquare\]
32.2 Varieties of Subtyping

Theorem 32.2 (Progress)
For either variant of MinML, under the assumption that the subtyping safety conditions hold, if \( e : \tau \), then either \( e \) is a value or there exists \( e' \) such that \( e \mapsto e' \).

Proof: By induction on typing, appealing to the subtyping condition on primitive operations in the case of primitive instruction steps.

32.2 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.

32.2.1 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, \texttt{float}, of floating point numbers. It is not necessary to be very specific about this extension, except to say that we enrich the language with floating point constants and arithmetic operations. We will designate the floating point operations using a decimal point, writing \( +. \) for floating point addition, and so forth.\(^2\)

By analogy with mathematical practice, we will consider taking the type \texttt{int} to be a subtype of \texttt{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 postu-\(^2\)
late that this is possible, writing \( n.0 \) for the floating point representation of the integer \( n \), and noting that \( n.0 \) has type \texttt{float}.\(^3\)

To ensure the safety of implicit subsumption we must ensure that the floating point arithmetic operations are well-defined for integer arguments. For example, we must ensure that an expression such as \( +.\ (3, 4) \) has a well-defined value as a floating point number. To achieve this, we simply require that floating point operations implicitly convert any integer arguments to floating point before performing the operation. In the foregoing example evaluation proceeds as follows:

\[
+.(3, 4) \mapsto +.(3.0, 4.0) \mapsto 7.0.
\]

This strategy requires that the floating point operations detect the presence of integer arguments, and that it convert any such arguments to floating point before carrying out the operation. We will have more to say about this inefficiency in Section 33.2 below.

### 32.2.2 Function Subtyping

Suppose that \texttt{int <: float}. What subtyping relationships, if any, should hold among the following four types?

1. \texttt{arrow(int,int)}
2. \texttt{arrow(int,float)}
3. \texttt{arrow(float,int)}
4. \texttt{arrow(float,float)}

To determine the answer, keep in mind the subsumption principle, which says that a value of the subtype should be usable in a supertype context.

Suppose \( f : \texttt{arrow(int,int)} \). If we apply \( f \) to \( x : \texttt{int} \), the result has type \texttt{int}, and hence, by the arithmetic subtyping axiom, has type \texttt{float}. This suggests that

\[
\texttt{arrow(int,int)} \ll< \texttt{arrow(int,float)}
\]

is a valid subtype relationship. By similar reasoning, we may derive that

\[
\texttt{arrow(float,int)} \ll< \texttt{arrow(float,float)}
\]

\(^3\)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.
32.2 Varieties of Subtyping

is also valid.

Now suppose that \( f : \text{arrow}(\text{float}, \text{int}) \). If \( x : \text{int} \), then \( x : \text{float} \) by subsumption, and hence we may apply \( f \) to \( x \) to obtain a result of type \( \text{int} \). This suggests that

\[
\text{arrow}(\text{float}, \text{int}) <: \text{arrow}(\text{int}, \text{int})
\]

is a valid subtype relationship. Since \( \text{arrow}(\text{int}, \text{int}) <: \text{arrow}(\text{int}, \text{float}) \), it follows that

\[
\text{arrow}(\text{float}, \text{int}) <: \text{arrow}(\text{int}, \text{float})
\]

is also valid.

Subtyping rules that specify how a type constructor interacts with subtyping are called \textit{variance} principles. If a type constructor \textit{preserves} subtyping in a given argument position, it is said to be \textit{covariant} in that position. If, instead, it \textit{inverts} subtyping in a given position it is said to be \textit{contravariant} in that position. The discussion above suggests that the function space constructor is covariant in the range position and contravariant in the domain position. This is expressed by the following rule:

\[
\frac{\tau_1 <: \sigma_1 \quad \sigma_2 <: \tau_2}{\text{arrow}(\sigma_1, \sigma_2) <: \text{arrow}(\tau_1, \tau_2)}
\]

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:

\[
(\text{arrow}(\tau_1, \tau_2)) \nu \leftarrow \text{fn } \in : \tau_1 \text{ in } (\tau_2) \nu((\sigma_1) x)
\]

Here \( \nu \) has type \( \text{arrow}(\sigma_1, \sigma_2) \), \( \tau_1 <: \sigma_1 \), 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.
32.2.3 Product and Record Subtyping

In Chapter ?? 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 \(i\)th component is accessed by projection, \(e_i\).

Similarly, record types have as elements records of the form \(\{l_1: e_1, \ldots, l_n: e_n\}\) whose \(i\)th component is accessed by field selection, \(e_i\).

Using the subsumption principle as a guide, it is natural to consider a tuple type to be a subtype of any of its prefixes:

\[
\frac{m > n}{\tau_1 \cdots \tau_m <: \tau_1 \cdots \tau_n}
\]

Given a value of type \(\tau_1 \cdots \tau_n\), we can access its \(i\)th component, for any \(1 \leq i \leq n\). If \(m > n\), then we can equally well access the \(i\)th component of an \(m\)-tuple of type \(\tau_1 \cdots \tau_m\), obtaining the same result. This is called \textit{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 \textit{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 \textit{depth subtyping} rules for tuples and records, respectively.
32.2 Varieties of Subtyping

To ensure safety for explicit subsumption we must define the meaning of casting from a sub- to a super-type. The two forms of casting corresponding to width and depth subtyping may be consolidated into one, as follows:

\[ m \geq n \]
\[(\tau_1 \cdots \tau_n) < v_1, \ldots, v_m > \rightarrow <(\tau_1) v_1, \ldots, (\tau_n) v_n > .\]

An analogous rule defines the semantics of casting for record types.

To ensure safety for implicit subsumption we must ensure that projection is well-defined on a subtype value. In the case of tuples this means that the operation of accessing the \( i \)th component from a tuple must be insensitive to the size of the tuple, beyond the basic requirement that it have size at least \( i \). This can be expressed schematically as follows:

\[ <v_1, \ldots, v_i, \ldots> . i \rightarrow v_i. \]

The ellision indicates that fields beyond the \( i \)th are not relevant to the operation. Similarly, for records we postulate that selection of the \( l \)th field is insensitive to the presence of any other fields:

\[ \{l:v,\ldots\} . l \rightarrow v. \]

The ellision expresses the independence of field selection from any “extra” fields.

32.2.4 Reference Subtyping

Finally, let us consider the reference types of Chapter 28. What should be the variance rule for reference types? Suppose that \( r \) has type \( \sigma \text{ ref} \). We can do one of two things with \( r \):

1. Retrieve its contents as a value of type \( \sigma \).
2. Replace its contents with a value of type \( \sigma \).

If \( \sigma <: \tau \), then retrieving the contents of \( r \) yields a value of type \( \tau \), by subsumption. This suggests that references are covariant:

\[ \sigma <: \tau \]
\[ \Rightarrow \]
\[ \sigma \text{ ref} <: \tau \text{ ref}. \]
On the other hand, if $\tau <: \sigma$, then we may store a value of type $\tau$ into $r$. This suggests that references are contravariant:

\[
\begin{array}{c}
\tau <: \sigma \\
\sigma \text{ref} <: \tau \text{ref}.
\end{array}
\]

Given that we may perform either operation on a reference cell, we must insist that reference types are \textit{invariant}:

\[
\begin{array}{c}
\sigma <: \tau \quad \tau <: \sigma \\
\sigma \text{ref} <: \tau \text{ref}.
\end{array}
\]

The premise of the rule is often strengthened to the requirement that $\sigma$ and $\tau$ be equal:

\[
\begin{array}{c}
\sigma = \tau \\
\sigma \text{ref} <: \tau \text{ref}
\end{array}
\]

since there are seldom situations where distinct types are mutual subtypes.

A similar analysis may be applied to any mutable data structure. For example, \textit{immutable} sequences may be safely taken to be covariant, but \textit{mutable} sequences (arrays) must be taken to be invariant, lest safety be compromised.
Chapter 33

Implementing Subtyping

33.1 Type Checking With Subtyping

Type checking for MinML<: , in either variant, clearly requires an algorithm for deciding subtyping: given $\sigma$ and $\tau$, determine whether or not $\sigma <: \tau$. The difficulty of deciding type checking is dependent on the specific rules under consideration. In this section we will discuss type checking for MinML<: , under the assumption that we can check the subtype relation.

Consider first the explicit variant of MinML<: . Since the typing rules are syntax directed, we can proceed as for MinML, with one additional case to consider. To check whether $(\sigma)e$ has type $\tau$, we must check two things:

1. Whether $e$ has type $\sigma$.
2. Whether $\sigma <: \tau$.

The former is handled by a recursive call to the type checker, the latter by a call to the subtype checker, which we assume given.

This discussion glosses over an important point. Even in pure MinML it is not possible to determine directly whether or not $\Gamma \vdash e : \tau$. For suppose that $e$ is an application $e_1(e_2)$. To check whether $\Gamma \vdash e : \tau$, we must find the domain type of the function, $e_1$, against which we must check the type of the argument, $e_2$. To do this we define a *type synthesis* function that determines the unique (if it exists) type $\tau$ of an expression $e$ in a context $\Gamma$, written $\Gamma \vdash e \Rightarrow \tau$. To check whether $e$ has type $\tau$, we synthesize the unique type for $e$ and check that it is $\tau$. 

243
This methodology applies directly to MinML<sub>{\leq}</sub> by using the following rule to synthesize a type for a cast:

\[ \Gamma \vdash e \Rightarrow \sigma \quad \sigma < : \tau \]

\[ \Gamma \vdash (\tau) e \Rightarrow \tau \]

Extending this method to MinML<sub>{i}</sub> 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 MinML<sub>{i}</sub> are as follows:

\[
(\Gamma(x) = \tau) \\
\Gamma \vdash x \Rightarrow \tau \\
\Gamma \vdash n \Rightarrow \text{int} \\
\Gamma \vdash \text{true} \Rightarrow \text{bool} \\
\Gamma \vdash \text{false} \Rightarrow \text{bool} \\
\Gamma \vdash e_1 \Rightarrow \sigma_1 \quad \sigma_1 < : \tau_1 \\
\quad \vdots \\
\Gamma \vdash e_n \Rightarrow \sigma_n \quad \sigma_n < : \tau_n \\
\Gamma \vdash o(e_1, \ldots, e_n) \Rightarrow \tau
\]

where o is an \( n \)-ary primitive operation with arguments of type \( \tau_1, \ldots, \tau_n \), and result type \( \tau \). We use subsumption to ensure that the argument types are subtypes of the required types.

\[
\Gamma \vdash e \Rightarrow \sigma \quad \sigma < : \text{bool} \\
\Gamma \vdash e_1 \Rightarrow \tau_1 \\
\Gamma \vdash e_2 \Rightarrow \tau_2 \\
\Gamma \vdash \text{if e then e_1 else e_2} \Rightarrow \tau
\]

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.\(^1\)

\[
\Gamma[f:\text{arrow}(\tau_1, \tau_2)][x:\tau_1] \vdash e \Rightarrow \tau_2 \\
\Gamma \vdash \text{fun f (x:}\tau_1:\tau_2\text{ is e} \Rightarrow \text{arrow}(\tau_1, \tau_2))
\]

\[
\Gamma \vdash e_1 \Rightarrow \text{arrow}(\tau_2, \tau) \\
\Gamma \vdash e_2 \Rightarrow \sigma_2 \quad \sigma_2 < : \tau_2 \\
\Gamma \vdash e_1(e_2) \Rightarrow \tau
\]

We use subsumption to check that the argument type is a subtype of the domain type of the function.

\(^1\)This may be avoided by requiring that the subtype relation have least upper bounds "whenever necessary"; we will not pursue this topic here.
Theorem 33.1

1. If $\Gamma \vdash e \Rightarrow \sigma$, then $\Gamma \vdash e : \sigma$.

2. If $\Gamma \vdash e : \tau$, then there exists $\sigma$ such that $\Gamma \vdash e \Rightarrow \sigma$ and $\sigma <: \tau$.

Proof:

1. By a straightforward induction on the definition of the type synthesis relation.

2. By induction on the typing relation.

33.2 Implementation of Subtyping

33.2.1 Coercions

The dynamic semantics of subtyping sketched above suffices to ensure type safety, but is in most cases rather impractical. Specifically,

- Arithmetic subtyping relies on run-time type recognition and conversion.
- Tuple projection depends on the insensitivity of projection to the existence of components after the point of projection.
- Record field selection depends on being able to identify the $l$th field in a record with numerous fields.
- Function subtyping may require run-time checks and conversions to match up calling conventions.

These costs are significant. Fortunately they can be avoided by taking a slightly different approach to the implementation of subtyping. Consider, for example, arithmetic subtyping. In order for a mixed-mode expression such as $+.\ (3,4)$ to be well-formed, we must use subsumption to weaken the types of 3 and 4 from $\text{int}$ to $\text{float}$. This means that type conversions are required exactly insofar as subsumption is used during type checking — a use of subsumption corresponds to a type conversion.

Since the subsumption rule is part of the static semantics, we can insert the appropriate conversions during type checking, and omit entirely
the need to check for mixed-mode expressions during execution. This is called a **coercion interpretation** of subsumption. It is expressed formally by augmenting each subtype relation \( \sigma <: \tau \) with a function value \( v \) of type \( \text{arrow}(\sigma, \tau) \) (in pure MinML) that coerces values of type \( \sigma \) to values of type \( \tau \). The augmented subtype relation is written \( \sigma <: \tau \sim v \).

Here are the rules for arithmetic subtyping augmented with coercions:

\[
\begin{align*}
\tau <: \tau \sim \text{id}_\tau \\
\rho <: \sigma \sim v \\
\sigma <: \tau \sim \nu' \\
\rho <: \tau \sim \nu; \nu'
\end{align*}
\]

\[
\text{int} <: \text{float} \sim \text{to_float} \\
\text{arrow}(\sigma_1, \sigma_2) <: \text{arrow}(\tau_1, \tau_2) \sim \text{arrow}(\nu_1, \nu_2)
\]

These rules make use of the following auxiliary functions:

1. Primitive conversion: \( \text{to_float} \).
2. Identity: \( \text{id}_\tau = \text{fn} x : \tau \text{ in } x \).
3. Composition: \( v; \nu' = \text{fn} x : \tau \text{ in } \nu'(v(x)) \).
4. Functions: \( \text{arrow}(\nu_1, \nu_2) = \text{fn} f : \text{arrow}(\sigma_1, \sigma_2) \text{ in } \text{fn} x : \tau_1 \text{ in } \nu_2(f(v_1(x))) \).

The coercion interpretation is type correct. Moreover, there is at most one coercion between any two types:

**Theorem 33.2**

1. If \( \sigma <: \tau \sim v \), then \( \vdash \nu : \text{arrow}(\sigma, \tau) \).
2. If \( \sigma <: \tau \sim \nu_1 \) and \( \sigma <: \tau \sim \nu_2 \), then \( \vdash \nu_1 \equiv \nu_2 : \text{arrow}(\sigma, \tau) \).

**Proof:**

1. By a simple induction on the rules defining the augmented subtyping relation.
2. Follows from these equations:

   (a) Composition is associative with \( \text{id} \) as left- and right-unit element.

   (b) \( \text{arrow}(\text{id}, \text{id}) \equiv \text{id} \).

   (c) \( \text{arrow}(\nu_1, \nu_2) ; (\text{arrow}(\nu'_1, \nu'_2)) \equiv \text{arrow}((\nu'_1; \nu_1), (\nu_2; \nu'_2)) \).
33.2 Implementation of Subtyping

The type checking relation is augmented with a translation from MinML\textsubscript{i} to pure MinML that eliminates uses of subsumption by introducing coercions:

\[
\begin{align*}
\Gamma &\vdash e : \sigma \rightsquigarrow e' \quad \sigma <: \tau \rightsquigarrow v \\
\Gamma &\vdash e : \tau \rightsquigarrow v(e')
\end{align*}
\]

The remaining rules simply commute with the translation. For example, the rule for function application becomes

\[
\begin{align*}
\Gamma &\vdash e_1 : \text{arrow}(\tau_2, \tau) \rightsquigarrow e'_1 \quad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow e'_2 \\
\Gamma &\vdash e_1(e_2) : \tau \rightsquigarrow e'_1(e'_2)
\end{align*}
\]

**Theorem 33.3**

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 \(\Gamma \vdash e_1 \equiv e_2 : \tau\) in pure MinML.

3. If \(e : \text{int} \rightsquigarrow e'\) 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:

\[
\begin{align*}
m > n &\quad \{l_1 : \tau_1, \ldots, l_m : \tau_m\} <: \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \rightsquigarrow \text{drop}_{m, n, l, \tau}
\end{align*}
\]
33.2 Implementation of Subtyping

\[
\sigma_1 <: \tau_1 \leadsto v_1 \quad \ldots \quad \sigma_n <: \tau_n \leadsto v_n
\]
\[\{l_1: \sigma_1, \ldots, l_n: \sigma_n\} <: \{l_1: \tau_1, \ldots, l_n: \tau_n\} \leadsto \text{copy}_{n,l,\sigma,v}\]

These rules make use of the following coercion functions:

\[
\text{drop}_{m,n,l,\sigma} = \text{fn} \; x: \{l_1: \sigma_1, \ldots, l_m: \sigma_m\} \; \text{in} \; \{l_1:x.l_1, \ldots, l_n:x.l_n\}
\]

\[
\text{copy}_{n,l,\sigma,v} = \text{fn} \; x: \{l_1: \sigma_1, \ldots, l_n: \sigma_n\} \; \text{in} \; \{l_1:v_1(x.l_1), \ldots, l_n:v_n(x.l_n)\}
\]

In essence this approach represents a trade-off between the cost of subsumption and the cost of field selection. By creating a new record whenever subsumption is used, we make field selection cheap. On the other hand, we can make subsumption free, provided that we are willing to pay the cost of a search whenever a field is selected from a record.

But what if record fields are mutable? This approach to coercion is out of the question, because of aliasing. Suppose that a mutable record value \(v\) is bound to two variables, \(x\) and \(y\). If coercion is applied to the binding of \(x\), creating a new record, then future changes to \(y\) will not affect the new record, nor vice versa. In other words, uses of coercion changes the semantics of a program, which is unreasonable.

One widely-used approach is to increase slightly the cost of field selection (by a constant factor) by separating the "view" of a record from its "contents". The view determines the fields and their types that are present for each use of a record, whereas the contents is shared among all uses. In essence we represent the record type \(\{l_1: \tau_1, \ldots, l_n: \tau_n\}\) by the product type \(\{l_1: \text{int}, \ldots, l_n: \text{int}\} \ast (\tau \text{array})\).

The field selection \(l.e\) becomes a two-stage process:

\[
\text{snd}(e) \left[\text{fst}(e).l\right]
\]

Finally, coercions copy the view, without modifying the contents. If \(\sigma = \{l_1: \sigma_1, \ldots, l_n: \sigma_n\}\) and \(\tau = \{l_1: \text{int}, \ldots, l_n: \text{int}\}\), then

\[
\text{drop}_{m,n,l,\sigma} = \text{fn} \; x \; \text{in} \; \text{drop}_{m,n,l,\tau}(\text{fst}(x)), \text{snd}(x))
\]
Part XII

Inheritance
Chapter 34

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.

34.1 Abstract Syntax

The abstract syntax of FJ is given by the following grammar:

\[
\begin{align*}
\text{Classes} & : = \text{class } c \text{ extends } c \{ f ; k d \} \\
\text{Constructors} & : = c(x) \{ \text{super}(x) ; \text{this}.f=x \} \\
\text{Methods} & : = c m(x) \{ \text{return } e \} \\
\text{Types} & : = c \\
\text{Expressions} & : = x \mid e.f \mid e.m(e) \mid \text{new } c(e) \mid (c) e
\end{align*}
\]

The variable \( f \) ranges over a set of field names, \( c \) over a set of class names, \( m \) over a set of method names, and \( x \) over a set of variable names. We assume that these sets are countably infinite and pairwise disjoint. We assume that there is a distinguished class name, \texttt{Object}, standing for the root of the class hierarchy. It’s role will become clear below. We assume that there is a distinguished variable \texttt{this} that cannot otherwise be declared in a program.

As a notational convenience we use “underbarring” to stand for sequences of phrases. For example, \( d \) stands for a sequence of \( d \)’s, whose
individual elements we designate \(d_1, \ldots, d_k\), where \(k\) is the length of the sequence. We write \(c f\) for the sequence \(c_1 f_1, \ldots, c_k f_k\), where \(k\) is the length of the sequences \(c\) and \(f\). Similar conventions govern the other uses of sequence notation.

The class expression

\[
\text{class } c \text{ extends } c' \{ c f; k d \}
\]

declares the class \(c\) to be a subclass of the class \(c'\). The subclass has additional fields \(c f\), single constructor \(k\), and method suite \(d\). The methods of the subclass may override those of the superclass, or may be new methods specific to the subclass.

The constructor expression

\[
c(c' x', c y) \{ \text{super}(x'); \text{this}.f=x; \}
\]

declares the constructor for class \(c\) with arguments \(c' x', c y\), corresponding to the fields of the superclass followed by those of the subclass. The variables \(x'\) and \(y\) are bound in the body of the constructor. The body of the constructor indicates the initialization of the superclass with the arguments \(x'\) and of the subclass with arguments \(y\).

The method expression

\[
c m(c y) \{ \text{return } e; \}
\]

declares a method \(m\) yielding a value of class \(c\), with arguments \(y\) of class \(c\) and body returning the value of the expression \(e\). The variables \(y\) and \(this\) are bound in \(e\).

The set of types is, for the time being, limited to the set of class names. That is, the only types are those declared by a class. In Java there are more types than just these, including the primitive types integer and boolean and the array types.

The set of expressions is the minimal "interesting" set sufficient to illustrate subtyping and inheritance. The expression \(e.f\) selects the contents of field \(f\) from instance \(e\). The expression \(e.m(e)\) invokes the method \(m\) of instance \(e\) with arguments \(e\). The expression \(\text{new } c(e)\) creates a new instance of class \(c\), passing arguments \(e\) to the constructor for \(c\). The expression \((c) e\) casts the value of \(e\) to class \(c\).

The methods of a class may invoke one another by sending messages to \(this\), standing for the instance itself. We may think of \(this\) as a bound variable of the instance, but we will arrange things so that renaming of \(this\) is never necessary to avoid conflicts.
34.1 Abstract Syntax

```java
class Pt extends Object {
    int x;
    int y;
    Pt (int x, int y) {
        super(); this.x = x; this.y = y;
    }
    int getx () { return this.x; }
    int gety () { return this.y; }
}
class CPt extends Pt {
    color c;
    CPt (int x, int y, color c) {
        super(x,y);
        this.c = c;
    }
    color getc () { return this.c; }
}
```

Figure 34.1: A Sample FJ Program

A class table $T$ is a finite function assigning classes to class names. The classes declared in the class table are bound within the table so that all classes may refer to one another via the class table.

A program is a pair $(T, e)$ consisting of a class table $T$ and an expression $e$. We generally suppress explicit mention of the class table, and consider programs to be expressions.

A small example of FJ code is given in Figure 34.1. 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.
34.2 Static Semantics

The static semantics of FJ is defined by a collection of judgments of the following forms:

\[
\begin{align*}
\tau <: \tau' & \quad \text{subtyping} \\
\Gamma \vdash e : \tau & \quad \text{expression typing} \\
d \text{ok in } c & \quad \text{well-formed method} \\
C \text{ ok} & \quad \text{well-formed class} \\
T \text{ ok} & \quad \text{well-formed class table} \\
\text{fields}(c) = \xi f & \quad \text{field lookup} \\
\text{type}(m, c) = \xi \rightarrow c & \quad \text{method type}
\end{align*}
\]

The rules defining the static semantics follow.

Every variable must be declared:

\[
\begin{align*}
\Gamma(x) &= \tau \\
\Gamma \vdash x : \tau
\end{align*}
\] (34.1)

The types of fields are defined in the class table.

\[
\begin{align*}
\Gamma \vdash e_0 : c_0 & \quad \text{fields}(c_0) = \xi f \\
\Gamma \vdash e_{0}.f_i : c_i
\end{align*}
\] (34.2)

The argument and result types of methods are defined in the class table.

\[
\begin{align*}
\Gamma \vdash e_0 : c_0 & \quad \Gamma \vdash \xi : \xi' \\
\text{type}(m, c_0) &= \xi' \rightarrow c & \xi < : \xi' \\
\Gamma \vdash e_0.m(e) : c
\end{align*}
\] (34.3)

Instantiation must provide values for all instance variables as arguments to the constructor.

\[
\begin{align*}
\Gamma \vdash \xi : \xi & \quad \xi < : \xi' \quad \text{fields}(c) = \xi' f \\
\Gamma \vdash \text{new } c(e) : c
\end{align*}
\] (34.4)

All casts are statically valid, but must be checked at run-time.

\[
\begin{align*}
\Gamma \vdash e_0 : d \\
\Gamma \vdash (c) e_0 : c
\end{align*}
\] (34.5)
The subtyping relation is read directly from the class table. Subtyping is the smallest reflexive, transitive relation containing the subclass relation:

\[ \tau <: \tau \]  

(34.6)

\[ \begin{array}{c}
\tau <: \tau' \\
\tau' <: \tau'' \\
\hline
\tau <: \tau''
\end{array} \]  

(34.7)

\[ T(c) = \text{class } c \text{ extends } c' \{ \ldots; \ldots \} \]  

(34.8)

A well-formed class has zero or more fields, a constructor that initializes the superclass and the subclass fields, and zero or more methods. To account for method override, the typing rules for each method are relative to the class in which it is defined.

\[ k = c(c', x', \xi x) \{ \text{super}(x'); \text{this.f} = \xi; \} \]  

\[ \text{fields}(c') = c' f' \]  

\[ d \text{ ok in } c \]  

\[ \text{class } c \text{ extends } c' \{ \xi f; k d \} \text{ ok} \]  

(34.9)

Method overriding takes account of the type of the method in the superclass. The subclass method must have the same argument types and result type as in the superclass.

\[ T(c) = \text{class } c \text{ extends } c' \{ \ldots; \ldots \} \]  

\[ \text{type}(m, c') = c \rightarrow c_0 \]  

\[ \xi ; c \text{ this.c } \vdash e_0 : c_0 \]  

\[ c_0 m(\xi x) \{ \text{return } e_0 ; \} \text{ ok in } c \]  

(34.10)

A class table is well-formed iff all of its classes are well-formed:

\[ \forall c \in \text{dom}(T) \; T(c) \text{ ok} \]  

\[ T \text{ ok} \]  

(34.11)

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:

\[ T \text{ ok} \; \emptyset \vdash e : \tau \]  

\[ (T, e) \text{ ok} \]  

(34.12)
The auxiliary lookup judgments determine the types of fields and methods of an object. The types of the fields of an object are determined by the following rules:

\[
\text{fields}(\text{Object}) = \bullet
\]  

34.13

\[
T(c) = \text{class } c \text{ extends } c' \{ c_f; \ldots \} \quad \text{fields}(c') = c'_f'
\]

\[
\text{fields}(c) = c_f' e_f'
\]  

34.14

The type of a method is determined by the following rules:

\[
T(c) = \text{class } c \text{ extends } c' \{ \ldots; \ldots d \}
\]

\[
d_i = c_i m(c_i x) \{ \text{return } e; \}
\]

\[
\text{type}(m_i, c) = c_i \rightarrow c_i
\]  

34.15

\[
T(c) = \text{class } c \text{ extends } c' \{ \ldots; \ldots d \}
\]

\[
m \notin d \quad \text{type}(m, c') = c_i \rightarrow c_i
\]

\[
\text{type}(m, c) = c_i \rightarrow c_i
\]  

34.16

34.3 Dynamic Semantics

The dynamic semantics of FJ may be specified using SOS rules similar to those for MinML. The transition relation is indexed by a class table \( T \), which governs the semantics of casting and dynamic dispatch (which see below). In the rules below we omit explicit mention of the class table for the sake of brevity.

An instance of a class has the form \( \text{new } c(e) \), where each \( e_i \) is a value.

\[
\text{new } c(e) \quad \text{value}
\]

34.17

Since we arrange that there be a one-to-one correspondence between instance variables and constructor arguments, an instance expression of this form carries all of the information required to determine the values of the fields of the instance. This makes clear that an instance is essentially just a labelled collection of fields. Each instance is labelled with its class, which is used to guide method dispatch.
Field selection retrieves the value of the named field from either the subclass or its superclass, as appropriate.

\[
\text{fields}(c) = \begin{cases} 
\text{new } c(c').f' \mapsto e'_i & \text{if } f \in c'
\end{cases}
\]

(34.18)

\[
\text{fields}(c) = \begin{cases} 
\text{new } c(c').f \mapsto e_i & \text{if } f \notin c'
\end{cases}
\]

(34.19)

Message send replaces this by the instance itself, and replaces the method parameters by their values.

\[
\text{body}(m, c) = x \rightarrow e_0 \rightarrow e' = \text{new } c'(c).e_0 \mapsto \text{new } c(c).
\]

(34.20)

Casting checks that the instance is of a sub-class of the target class, and yields the instance.

\[
\text{c <: c' \rightarrow e = \text{new } c(c').e \mapsto \text{new } c(c)}
\]

(34.21)

These rules determine the order of evaluation:

\[
e_0 \mapsto e'_0
\]

(34.22)

\[
e_0 \cdot f \mapsto e'_0 \cdot f
\]

(34.23)

\[
e_0 \mapsto e'_0
\]

(34.24)

\[
e_0 \mapsto e'_0
\]

(34.25)

\[
e_0 \mapsto (c) e_0 \mapsto (c) e'_0
\]

(34.26)
Dynamic dispatch makes use of the following auxiliary relation to find the correct method body.

\[
T(c) = \text{class } c \text{ extends } c' \{ \ldots ; \ldots \} \\
\quad d_i = c_i m_i(c_i x) \{ \text{return } e; \} \\
\quad \text{body}(m_i, c) = x \rightarrow e
\]  

(34.27)

\[
T(c) = \text{class } c \text{ extends } c' \{ \ldots ; \ldots \} \\
\quad m \notin d \quad \text{body}(m, c') = x \rightarrow e \\
\quad \text{body}(m, c) = x \rightarrow e
\]  

(34.28)

Finally, we require rules for evaluating sequences of expressions from left to right, and correspondingly defining when a sequence is a value (i.e., consists only of values).

\[
e_1 \text{ value } \ldots \quad e_i \text{ value } \quad e_i \mapsto e'_i \\
e_1, \ldots, e_{i-1}, e_i, e_{i+1}, \ldots, e_n \mapsto e_1, \ldots, e_{i-1}, e'_i, e_{i+1}, \ldots, e_n
\]  

(34.29)

\[
e_1 \text{ value } \ldots \quad e_n \text{ value} \\
\quad \mapsto \quad \text{value}
\]  

(34.30)

This completes the dynamic semantics of FJ.

## 34.4 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 34.1 (Preservation)**

Assume that \( T \) is a well-formed class table. If \( e : \tau \) and \( e \mapsto e' \), then \( e' : \tau' \) for some \( \tau' \) such that \( \tau' <: \tau \).

The statement of Progress must take account of the possibility that a cast may fail at execution time. Note, however, that field selection or message send can never fail — the required field or method will always be present.
Theorem 34.2 (Progress)
Assume that $T$ is a well-formed class table. If $e : \tau$ then either

1. $v$ value, or

2. $e$ contains an instruction of the form $(c) \text{new } c'(e_0)$ with $e_0$ value and $c' \not<: c$, or

3. there exists $e'$ such that $e \rightarrow e'$.

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 34.3 (Canonical Forms)
If $e : c$ and $e$ value, then $e$ has the form $\text{new } c'(e_0)$ with $e_0$ value and $c' <: c$.

34.5 Acknowledgement

This chapter is based on “Featherweight Java: A Minimal Core Calculus for Java and GJ” by Atsushi Igarashi, Benjamin Pierce, and Philip Wadler.
34.5 Acknowledgement
Chapter 35

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.

35.1 Inheritance Mechanisms in Java

35.1.1 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). 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 to the $f$ field of the instance $c$. Since distinct instances have distinct fields, there is no essential connection between $c.f$ and $c'.f$ when $c$ and $c'$ are

---

\(^2\)Classes can have multiple constructors that are distinguished by overloading. We will not discuss overloading here.
35.1 Inheritance Mechanisms in Java

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 them-
selves) 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.

35.1.2 Subclasses

A class may be defined by inheriting the visible fields and methods of an-
other class. The new class is said to be a subclass of the old class, the super-
class. 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 addi-
tional 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 “spe-
cialize” 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 over-
ride. The “old” version can still be accessed explicitly from the subclass by
referring to \texttt{super.m}. The keyword \texttt{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 \texttt{public} attribute in the subclass, unless overridden. A \texttt{protected} component is "semi-private" — accessible to the subclass, but not otherwise publically visible.\footnote{Actually, Java assigns \texttt{protected} components “package scope”, but since we are not discussing packages here, we will ignore this issue.}

Inheritance can also be limited using finality constraints. If a method is declared \texttt{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 \texttt{this}, a non-final method, then the behavior of the final method can still be changed by the sub-class by overriding the non-final method. By declaring an entire class to be final, no class can inherit from it. This serves to ensure that any instance of this class invokes the code from this class, and not from any subclass of it.

Instantiation of a subclass of a class proceeds in three phases:

1. The instance variables of the subclass, which include those of the superclass, are allocated.

2. The constructor of the superclass is invoked to initialize the superclass’s instance variables.

3. The constructor of the subclass is invoked to initialize the subclass’s instance variables.

The superclass constructor can be explicitly invoked by a subclass constructor by writing \texttt{super(e1,...,en)}, but \textit{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.

\subsection*{35.1.3 Abstract Classes and Interfaces}

An \textit{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 \texttt{this}, but since their implementation is not provided, abstract classes do not themselves have
Inheritance Mechanisms in Java

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. A class may also be declared to implement one or more interfaces, meaning that the class provides the public methods of the interface, with their specified types. Since interfaces are special kinds of classes, Java is sometimes said to provide multiple inheritance of interfaces, but only single inheritance of implementation. For similar reasons an interface may be declared to extend multiple interfaces, provided that the result types of their common methods coincide.

The purpose of declaring an interface for a class is to support writing generic code that works with any instance providing the methods specified in the interface, without requiring that instance to arise from any particular position in the inheritance hierarchy. For example, we may have two unrelated classes in the class hierarchy providing a method \( m \). If both classes are declared to implement an interface that mentions \( m \), then code programmed against this interface will work for an instance of either class.

The literature on Java emphasizes that interfaces are descriptive of behavior (to the extend that types alone allow), whereas classes are prescriptive of implementation. While this is surely a noble purpose, it is curious that interfaces are classes in Java, rather than types. In particular interfaces are unable to specify the public fields of an instance by simply stating their types, which would be natural were interfaces a form of type. Instead fields in interfaces are forced to be constants (public, static, final), precluding their use for describing the public instance variables of an object.

---

4Classes that do not specify a superclass implicitly extend the class Object of all objects.
35.2 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.

35.2.1 Subtyping

To define the Java subtype relation we need two auxiliary relations. The *subclass* relation, `C ⊲ 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 ⊲ 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 ▼ I`, is defined to hold exactly when a class `C` is declared to implement an interface that inherits from `I`. 
35.2 Subtyping in Java

The Java subtype relation is inductively defined by the following rules. Subtyping is reflexive and transitive:

\[
\frac{}{\tau <: \tau} \quad (35.1)
\]

\[
\frac{\tau <: \tau' \quad \tau' <: \tau''}{\tau <: \tau''} \quad (35.2)
\]

Arrays are covariant type constructors, in the sense of this rule:

\[
\frac{}{\tau <: \tau' \quad \tau' [] <: \tau'' []} \quad (35.3)
\]

Inheritance implies subtyping:

\[
\frac{}{C <: C'} \quad (35.4)
\]

Implementation implies subtyping:

\[
\frac{}{C \triangleright I \quad C <: I} \quad (35.5)
\]

Every class is a subclass of the distinguished “root” class Object:

\[
\frac{}{\tau <: \text{Object}} \quad (35.6)
\]

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).

35.2.2 Subsumption

The subsumption principle tells us that if \(e\) is an expression of type \(\tau\) and \(\tau <: \tau'\), then \(e\) is also an expression of type \(\tau'\). In particular, if a method is declared with a parameter of type \(\tau\), then it makes sense to provide an argument of any type \(\tau'\) such that \(\tau' <: \tau\). Similarly, if a constructor takes a parameter of a type, then it is legitimate to provide an argument of a subtype of that type. Finally, if a method is declared to return a value of type \(\tau\), then it is legitimate to return a value of any subtype of \(\tau\).
This brings up an awkward issue in the Java type system. What should be the type of a conditional expression \( e ? e_1 : e_2 \)? Clearly \( e \) should have type `boolean`, and \( e_1 \) and \( e_2 \) should have the same type, since we cannot in general predict the outcome of the condition \( e \). In the presence of subtyping, this amounts to the requirement that the types of \( e_1 \) and \( e_2 \) have an upper bound in the subtype ordering. To avoid assigning an excessively weak type, and to ensure that there is a unique choice of type for the conditional, it would make sense to assign the conditional the least upper bound of the types of \( e_1 \) and \( e_2 \). Unfortunately, two types need not have a least upper bound! For example, if an interface \( I \) extends incomparable interfaces \( K \) and \( L \), and \( J \) extends both \( K \) and \( L \), then \( I \) and \( J \) do not have a least upper bound — both \( K \) and \( L \) are upper bounds of both, but neither is smaller than the other. To deal with this Java imposes the rather ad hoc requirement that either the type of \( e_1 \) be a subtype of the type of \( e_2 \), or vice versa, to avoid the difficulty.

A more serious difficulty with the Java type system is that the array subtyping rule, which states that the array type constructor is covariant in the type of the array elements, violates the subsumption principle. To understand why, recall that we can do one of two things with an array: retrieve an element, or assign to an element. If \( \tau <: \tau' \) and \( A \) is an array of type \( \tau \[ \] \), then retrieving an element of \( A \) yields a value of type \( \tau \), which is by hypothesis an element of type \( \tau' \). So we are OK with respect to retrieval. Now consider array assignment. Suppose once again that \( \tau <: \tau' \) and that \( A \) is an array of type \( \tau \[ \] \). Then \( A \) is also an array of type \( \tau' \[ \] \), according to the Java rule for array subtyping. This means we can assign a value \( x \) of type \( \tau' \) to an element of \( A \). But this violates the assumption that \( A \) is an array of type \( \tau \[ \] \) — one of its elements is of type \( \tau' \).

With no further provisions the language would not be type safe. It is a simple matter to contrive an example involving arrays that incurs a 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.

### 35.2.3 Dynamic Dispatch

According to Java typing rules, if \( C \) is a sub-class of \( D \), then \( C \) is a sub-type of \( D \). Since the instances of a class \( C \) have type \( C \), they also, by subsumption, have type \( D \), as do the instances of class \( D \) itself. In other words, if the static type of an instance is \( D \), it might be an instance of class \( C \) or an
instance of class $D$. In this sense the static type of an instance is at best an approximation of its dynamic type, the class of which it is an instance.

The distinction between the static and the dynamic type of an object is fundamental to object-oriented programming. In particular method specialization is based on the dynamic type of an object, not its static type. Specifically, if $C$ is a sub-class of $D$ that overrides a method $m$, then invoking the method $m$ of a $C$ instance $o$ will always refer to the overriding code in $C$, even if the static type of $o$ is $D$. That is, method dispatch is based on the dynamic type of the instance, not on its static type. For this reason method specialization is sometimes called *dynamic dispatch*, or, less perspicuously, *late binding*.

How is this achieved? Essentially, every object is tagged with the class that created it, and this tag is used to determine which method to invoke when a message is sent to that object. The constructors of a class $C$ “label” the objects they create with $C$. The method dispatch mechanism consults this label when determining which method to invoke.\(^5\)

The same mechanism is used to ensure that array assignments do not lead to type insecurities. Suppose that the static type of $A$ is $C[ ]$, and that the static type of instance $o$ is $C$. By covariance of array types the dynamic type of $A$ might be $D[ ]$ for some sub-class $D$ of $C$. But unless the dynamic type of $o$ is also $D$, the assignment of $o$ to an element of $A$ should be prohibited. This is ensured by an explicit run-time check. In Java every single array assignment incurs a run-time check whenever the array contains objects.\(^6\)

### 35.2.4 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 el-

---

\(^5\)In practice the label is a pointer to the vector of methods of the class, and the method is accessed by indexing into this vector. But we can just as easily imagine this to be achieved by a case analysis on the class name to determine the appropriate method vector.

\(^6\)Arrays of integers and floats do not incur this overhead, because numbers are not objects.
element types. However, when retrieving an element from a container its static type is `Object`; we lost track of its dynamic type during type checking. If we wish to use such an object in any meaningful way, we must recover its dynamic type so that message sends are not rejected at compile time.

Java supports a safe form of casting, or change of type. A cast is written \((\tau) e\). The expression \(e\) is called the subject of the cast, and the type \(\tau\) is the target type of the cast. The type of the cast is \(\tau\), provided that the cast makes sense, and its value is that of \(e\). In general we cannot determine whether the cast makes sense until execution time, when the dynamic type of the expression is available for comparison with the target type. For example, every instance in Java has type `Object`, but its true type will usually be some class further down the type hierarchy. Therefore a cast applied to an expression of type `Object` cannot be validated until execution time.

Since the static type is an attenuated version of the dynamic type of an object, we can classify casts into three varieties:

1. **Up casts**, in which the static type of the expression is a subtype of the target type of the cast. The type checker accepts the cast, and no run-time check is required.

2. **Down casts**, in which the static type of the expression is a supertype of the target type. The true type may or may not be a subtype of the target, so a run-time check is required.

3. **Stupid casts**, in which the static type of the expression rules out the possibility of its dynamic type matching the target of the cast. The cast is rejected.

Similar checks are performed to ensure that array assignments are safe.

Note that it is up to the programmer to maintain a sufficiently strong invariant to ensure that down casts do not fail. For example, if a container is intended to contain objects of a class \(C\), then retrieved elements of that class will typically be down cast to a sub-class of \(C\). It is entirely up to the programmer to ensure that these casts do not fail at execution time. That is, the programmer must maintain the invariant that the retrieved element really contains an instance of the target class of the cast.
35.3 Methodology

With this in hand we can (briefly) discuss the methodology of inheritance in object-oriented languages. As we just noted, in Java subclassing entails subtyping — the instance type of a subclass is a subtype of the instance type of the superclass. It is important to recognize that this is a methodological commitment to certain uses of inheritance.

Recall that a subtype relationship is intended to express a form of behavioral equivalence. This is expressed by the subsumption principle, which states that subtype values may be provided whenever a supertype value is required. In terms of a class hierarchy this means that a value of the subclass can be provided whenever a value of the superclass is required. For this to make good sense the values of the subclass should “behave properly” in superclass contexts — they should not be distinguishable from them.

But this isn’t necessarily so! Since inheritance admits overriding of methods, we can make almost arbitrary changes to the behavior of the superclass when defining the subclass. For example, we can turn a stack-like object into a queue-like object (replacing a LIFO discipline by a FIFO discipline) by inheritance, thereby changing the behavior drastically. If we are to pass off a subclass instance as a superclass instance using subtyping, then we should refrain from making such drastic behavioral changes.

The Java type system provides only weak tools for ensuring a behavioral subtyping relationship between a subclass and its superclass. Fundamentally, the type system is not strong enough to express the desired constraints. To compensate for this Java provides the finality mechanism to limit inheritance. Final classes cannot be inherited from at all, ensuring that values of its instance type are indeed instances of that class (rather than an arbitrary subclass). Final methods cannot be overridden, ensuring that certain aspects of behavior are “frozen” by the class definition.

Nominal subtyping may also be seen as a tool for enforcing behavioral subtyping relationships. For unless a class extends a given class or is declared to implement a given interface, no subtyping relationship holds. This helps to ensure that the programmer explicitly considers the behavioral subtyping obligations that are implied by such declarations, and is therefore an aid to controlling inheritance.

---

7 Limited only by finality declarations in the superclass.
8 Nor is the type system of any other language that I am aware of, including ML.
35.3 Methodology
Part XIII

Program Equivalence
Chapter 36

Functional Equivalence

One of the beauties of functional programming is the ease with which we may reason about equivalence of expressions. Informally, we say that two expressions $e_1$ and $e_2$ of the same type are equivalent iff replacing $e_1$ by $e_2$ in a complete program doesn’t change its final result. By a “complete program” we mean a closed expression of type `int` or `bool`; the final result of a complete program is therefore a number or a Boolean constant. What is important here is that the final outcome be finitely observable — we can see immediately that the answer is `false` or `17`. Since functions are essentially “infinite” objects (in the sense that the graph of a function on the integers is infinite), we would not regard functions as observable outcomes of a complete program.

We can think of a usage of an expression in a complete program as an “experiment” or “observation” performed on that expression. The idea is that the program “uses” the expression to compute an observable quantity that we can regard as a kind of test performed on that expression. For this reason the notion of equivalence just described is sometimes called observational equivalence — two expressions are observationally equivalent iff any experiment performed on one yields the same observable outcome as the same experiment performed on the other. This relation is also called contextual equivalence to emphasize that equivalence is determined by considering all contexts in which the two expressions might be used to form a complete program. In philosophical logic this relation is known as Leibniz’s Principle of Identity of Indiscernibles — two things are equal iff we cannot tell them apart.

Observational equivalence is very difficult to handle. To determine whether or not $e_1$ and $e_2$ are observationally equivalent requires us to con-
sider all possible programs that use them to compute an integer! This quickly gets out of hand. What we need are alternative criteria for establishing observational equivalences that avoid the need to explicitly consider all possible usages of the expressions in question. Unfortunately, a rigorous development of such alternatives would take us far beyond the scope of the course. We will content ourselves with stating, without proof, a collection of laws of equivalence that are useful for deriving equations between expressions such as these:

1. $x + (y + z)$ is equivalent to $(x + y) + z$.

2. $\text{rev}(\text{rev}(x))$ is equivalent to $x$ (where $\text{rev}$ is the list reversal function).

3. If $v = \text{fun} \ f \ (x: \tau_1): \tau_2 \ is \ e$ and $v.\tau_1$ is a value, then $v(v_1)$ is equivalent to $[f, x←v, v_1]e$.

In the next section we make precise the definition of observational equivalence, and state an alternative characterization of it that is often easier to handle. In the subsequent section we enumerate a collection of valid principles of equivalence for variants of MinML.

### 36.1 Expression Equivalence

We begin with the notion of Kleene equality between complete programs. Kleene equivalence captures what we mean by “same outcome” for complete programs. Two complete programs $p_1$ and $p_2$ are Kleene equivalent, written $p_1 \simeq p_2$, exactly when $p_1 \Downarrow v$ iff $p_2 \Downarrow v$. That is, either both $p_1$ and $p_2$ diverge (fail to halt), or both converge to the same number or Boolean constant (perhaps in very different ways, using very different amounts of time and space).

A context $C$ is a complete program with a single “hole” into which we may insert an expression. That is, $C$ has the form $\ldots \bullet \ldots$, where the $\bullet$ indicates the “hole” in the program. We write $C\{e\}$ for the result of filling the hole in $C$ with the expression $e$ to obtain $\ldots e \ldots$. The expression $e$ might have free variables that are captured when inserted into the hole. For example, if $C$ is the program context $(\text{fun} \ f \ (x: \text{int}): \text{int is } \bullet)\ (3)$ and $e$ is the expression $x + 5$, then $C\{x + 5\}$ is the program

$$(\text{fun} \ f \ (x: \text{int}): \text{int is } x + 5)\ (3).$$
Suppose that $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$. We define the observational equivalence relation $\Gamma \vdash e_1 \equiv_{\text{obs}} e_2 : \tau$ to hold iff for every program context $C$ such that $C\{e_1\}$ and $C\{e_2\}$ are programs, $C\{e_1\} \simeq C\{e_2\}$. That is, every use of $e_1$ has the same observable outcome as the corresponding use of $e_2$, and vice-versa.

As we remarked earlier, it is rather difficult to establish that two expressions are observationally equivalent. A direct application of the definition leaves us no recourse but to consider all possible program contexts $C$, which quickly gets out of hand.

A more usable characterization of observational equivalence, called applicative equivalence, is defined as follows. For closed expressions $e_1$ and $e_2$ of type $\tau$, we define $e_1 \equiv_{\text{app}} e_2 : \tau$ by induction on the structure of $\tau$ as follows:

- If $\tau = \text{int}$, or $\tau = \text{bool}$, then $e_1 \equiv_{\text{app}} e_2 : \tau$ iff $e_1 \simeq e_2$.

- If $\tau = \text{arrow}(\tau_1, \tau_2)$, then $e_1 \equiv_{\text{app}} e_2 : \tau$ iff for every $v : \tau_1$, $e_1(v) \equiv_{\text{app}} e_2(v) : \tau_2$.

This relation is extended to open expressions by substitution of closed values of appropriate type for the free variables. Let $\Gamma$ be the context $x_1 : \tau_1, \ldots, x_n : \tau_n$. We define $\Gamma \vdash e_1 \equiv_{\text{app}} e_2 : \tau$ to hold iff

$$[x_1, \ldots, x_n \gets v_1, \ldots, v_n]e_1 \equiv_{\text{app}} [x_1, \ldots, x_n \gets v_1, \ldots, v_n]e_2$$

for every substitution of closed values $v_1, \ldots, v_n$ of type $\tau_1, \ldots, \tau_n$ for $x_1, \ldots, x_n$.

An important result of Milner’s states that applicative and observational equivalence coincide. The proof is non-trivial, and is omitted from this brief exposition.

**Theorem 36.1 (Milner’s Context Lemma)**

$\Gamma \vdash e_1 \equiv_{\text{obs}} e_2 : \tau$ iff $\Gamma \vdash e_1 \equiv_{\text{app}} e_2 : \tau$.

The point of the context lemma is that we may interpret the relation $\Gamma \vdash e_1 \equiv_{\text{obs}} e_2 : \tau$ as expressing the universally-quantified formula

$$\forall v_1 : \tau_1, \ldots, \forall v_n : \tau_n [x_1, \ldots, x_n \gets v_1, \ldots, v_n]e_1 \equiv_{\text{app}} [x_1, \ldots, x_n \gets v_1, \ldots, v_n]e_2 : \tau$$

where the quantifiers range over closed values of the appropriate type. For example,

$$x : \text{int}, y : \text{int} \vdash x + y \equiv_{\text{obs}} y + x : \text{int}$$

means that for every $m$ and $n$, $m + n \simeq n + m$. 
36.2 Some Laws of Equivalence

In this section we summarize some useful principles of equivalence. These are all valid observational equivalences, but we will not prove this. What is important is to get a feeling for what are some valid principles of equivalence, and how to use them in practice. Since observational and applicative equivalence coincide, we will write \( \Gamma \vdash e_1 \cong e_2 : \tau \) for equality of expressions of type \( \tau \) relative to a context \( \Gamma \), where we tacitly assume that \( \Gamma \vdash e_i : \tau \) for \( i = 1, 2 \).

In the presentation of the rules, we use \( v \) to stand for an open value, either a variable, a constant, or an function expression (perhaps with free variables occurring within it). We admit variables as values because, in a call-by-value language, variables are only ever bound to values, and hence may be taken as standing for a fixed, but unknown, value.

It will be convenient to make use of a designated non-terminating expression of each type \( \tau \), written \( \Omega_\tau \), which is defined to be the expression

\[
(f \text{ fun } f(x:\text{int}):\tau \text{ is } f(x) \text{ end})(0).
\]

It is easy to check that \( \Omega_\tau \) diverges (loops forever) when evaluated.

36.2.1 General Laws

First, equivalence is indeed an equivalence relation — it is reflexive, symmetric, and transitive.

\[
\Gamma \vdash e \cong e : \tau \tag{36.1}
\]

\[
\begin{align*}
\Gamma \vdash e_2 \cong e_1 : \tau \\
\Gamma \vdash e_1 \cong e_2 : \tau
\end{align*}
\]

\[
\Gamma \vdash e_1 \cong e_2 : \tau \\
\Gamma \vdash e_2 \cong e_3 : \tau
\]

\[
\Gamma \vdash e_1 \cong e_3 : \tau
\]  

Second, equivalence is a congruence — we may replace a sub-expression of any expression by an equivalent one to obtain an equivalent expression. This is most easily stated by a collection of rules that ensure that we may replace equivalent sub-expressions to obtain equivalent expressions. We will give just a few of these here; the rest follow a similar pattern.
36.2 Some Laws of Equivalence

\[\Gamma \vdash e_1 \equiv e_1' : \tau_1 \quad \cdots \quad \Gamma \vdash e_n \equiv e_n' : \tau_n\]
\[\Gamma \vdash o(e_1, \ldots, e_n) \equiv o(e_1', \ldots, e_n') : \tau\] (36.4)

\[\Gamma \vdash e_1 \equiv e_1' : \text{arrow}(\tau_2, \tau) \quad \Gamma \vdash e_2 \equiv e_2' : \tau_2\]
\[\Gamma \vdash e_1(e_2) \equiv e_1'(e_2') : \tau\] (36.5)

\[\Gamma[f:\text{arrow}(\tau_1, \tau_2)][x:\tau_1] \vdash e \equiv e' : \tau_2\]
\[\Gamma \vdash \text{fun } f \,(x : \tau_1) : \tau_2 \text{ is } e \equiv \text{fun } f \,(x : \tau_1) : \tau_2 \text{ is } e' : \text{arrow}(\tau_1, \tau_2)\] (36.6)

Finally, equivalence is stable under substitution of values for free variables.
\[\Gamma[x:\tau] \vdash e \equiv e' : \tau'\]
\[\Gamma \vdash [x \leftarrow v]e \equiv [x \leftarrow v]' : \tau'\] (36.7)

The restriction to values is essential; this rule is not true for general expression substitution! A counterexample is given in the next subsection.

36.2.2 Symbolic Evaluation

Evaluation of an expression in accordance with the rules of the operational semantics results in an equivalent expression. This is called “symbolic evaluation” because the transformations may involve expressions with free variables, which are regarded as values for the purposes of these rules.

An application of a primitive operation may be simplified if we know the values of its arguments:
\[\Gamma \vdash o(v_1, \ldots, v_n) \equiv v : \tau,\] (36.8)
where \(v\) is the result of applying \(o\) to \(v_1, \ldots, v_n\).

Similarly, if we know the result of the boolean test, then a conditional may be simplified:
\[\Gamma \vdash \text{if true then } e_1 \text{ else } e_2 \equiv e_1 : \tau\] (36.9)

\[\Gamma \vdash \text{if false then } e_1 \text{ else } e_2 \equiv e_2 : \tau\] (36.10)
An application may be simplified if we know the function and the argument is a value. Note that either the function or argument may be open expressions (containing free variables!)

\[ \Gamma \vdash v(v_1) \equiv [f, x \leftarrow v, v_1]e : \tau_2 \]  

(36.11)

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

**Exercise 36.2**

Using these rules, check that \( (fn \ x \ in \ 3 \ end)(z) \) is equivalent to 3. Show that \( (fn \ x \ in \ 3 \ end)(\Omega_{\text{int}}) \) is not equivalent to 3. Conclude that substitution of non-values for free variables does not preserve equivalence.

### 36.2.3 Extensionality

Two functions are equivalent if they are equivalent on all arguments.

\[ \Gamma \vdash e(x) \equiv e'(x) : \tau_2 \]

\[ \Gamma \vdash e \equiv e' : \text{arrow}(\tau_1, \tau_2) \]  

(36.12)

In other words, if two functions are equal for all closed argument values, then they are equal.

### 36.2.4 Strictness Properties

The evaluation rules of MinML impose a *call-by-value* evaluation order on function applications and primitive operations. This can be captured equationally by a set of *strictness* equations that are defined in terms of the divergent expressions \( \Omega_{\tau} \). We may state that an expression \( e \) of type \( \tau \) diverges as an equation by stating that \( e \equiv \Omega_{\tau} \). The following rules give some conditions under which expressions are divergent.

If any argument of a primitive operation is divergent, so is the whole expression:

\[ \Gamma \vdash o(e_1, \ldots, e_{i-1}, \Omega_{\tau_i}, e_{i+1}, \ldots, e_n) \equiv \Omega_{\tau} : \tau \]  

(36.13)

If the test expression of a conditiona is divergent, so is the conditional.

\[ \Gamma \vdash \text{if } \Omega_{\text{bool}} \text{ then } e_1 \text{ else } e_2 \equiv \Omega_{\tau} : \tau \]  

(36.14)
36.2 Some Laws of Equivalence

If the function or argument of an application is divergent, so is the entire expression:

\[ \Gamma \vdash \Omega_{\text{arrow}(\tau_1, \tau)}(e_2) \cong \Omega_\tau : \tau \] (36.15)

\[ \Gamma \vdash e_1(\Omega_\tau) \cong \Omega_\tau : \tau \] (36.16)

36.2.5 Arithmetic Laws

Arithmetic and comparison operations behave as expected. For example, addition is associative and commutative, and equality test on integers is an equivalence relation. In general appropriate laws governing the primitive operations on integers hold, provided that they hold mathematically. The same could not be said for floating point (for which addition is not even associative!). Observe that these laws fail, in general, in the presence of effects such as writing to the screen or destructively updating a reference cell! In that case we must restrict attention to values, not general expressions.

\[ \Gamma \vdash e_1 + e_2 \cong e_2 + e_1 : \text{int} \] (36.17)

\[ \Gamma \vdash e_1 + (e_2 + e_3) \cong (e_1 + e_2) + e_3 : \text{int} \] (36.18)

\[ \Gamma \vdash e_1 = e_2 \cong e_2 = e_1 : \text{bool} \] (36.19)

36.2.6 Products

For the extension of MinML to product types,\(^1\) we have the following symbolic evaluation rule:

\[ \Gamma \vdash \text{split}(v_1, v_2) \text{ as } (x_1 : \tau_1, x_2 : \tau_2) \text{ in } e \cong [x_1, x_2 \leftarrow v_1, v_2]e \] (36.20)

\(^1\)We do not consider nested or wildcard patterns, for the sake of simplicity. It is a simple matter to extend these rules to the more general case.
We may, in general, replace equals by equals:

\[
\begin{align*}
\Gamma &\vdash e_1 \cong e'_1 : \tau_1 \quad \Gamma \vdash e_2 \cong e'_2 : \tau_2 \\
\Gamma &\vdash (e_1, e_2) \cong (e'_1, e'_2) : \tau_1 \star \tau_2
\end{align*}
\]

(36.21)

\[
\Gamma \vdash e_1 \cong e'_1 : \tau_1 \star \tau_2 \quad \Gamma[x_1:\tau_1][x_2:\tau_2] \vdash e_2 \cong e'_2 : \tau'
\]

\[
\Gamma \vdash \text{split } e_1 \text{ as } (x'_1:\tau_1, x'_2:\tau_2) \text{ in } e_2 \cong \text{split } e'_1 \text{ as } (x'_1:\tau_1, x'_2:\tau_2) \text{ in } e'_2
\]

(36.22)

Finally, the expected strictness properties hold:

\[
\Gamma \vdash (\Omega \tau_1, e_2) \cong \Omega (\tau_1, \tau_2)
\]

(36.23)

\[
\Gamma \vdash (e_1, \Omega \tau_2) \cong \Omega (\tau_1, \tau_2)
\]

(36.24)

\[
\Gamma \vdash \text{split } \Omega \tau_1 \star \tau_2 \text{ as } (x'_1:\tau_1, x'_2:\tau_2) \text{ in } e \cong \Omega \tau'
\]

(36.25)

### 36.2.7 Lists

We may extend MinML with list types \( \tau \text{list} \) by adding the expressions \( \text{nil} \), \( \text{cons}(e_1, e_2) \), and \( \text{listcase } e \) of \( \text{nil} \Rightarrow e' \mid \text{cons}(x, y) \Rightarrow e'' \), with the following typing rules:

\[
\Gamma \vdash \text{nil} : \tau \text{list}
\]

(36.26)

\[
\Gamma \vdash \text{cons}(e_1, e_2) : \tau \text{list}
\]

(36.27)

\[
\begin{align*}
\Gamma &\vdash e : \tau \text{list} \quad \Gamma \vdash e' : \tau' \\
\Gamma[x:\tau][y:\tau \text{list}] &\vdash e'' : \tau'
\end{align*}
\]

\[
\Gamma \vdash \text{listcase } e \text{ of } \text{nil} \Rightarrow e' \mid \text{cons}(x, y) \Rightarrow e'' : \tau'
\]

(36.28)

The following symbolic evaluation and strictness rules express the evaluation of these constructs:

\[
\Gamma \vdash \text{listcase } \text{nil} \Rightarrow e' \mid \text{cons}(x, y) \Rightarrow e'' \cong e' : \tau'
\]

(36.29)
36.2 Some Laws of Equivalence

\[ \Gamma \vdash \text{listcase cons}(v_1, v_2) \text{ of nil} \Rightarrow e' \mid \text{cons}(x_1, x_2) \Rightarrow e'' \cong [x_1, x_2 \leftarrow v_1, v_2] e'' : \tau' \]  
(36.30)

\[ \Gamma \vdash \text{cons}(\Omega, e) \cong \Omega_{\text{list}} : \tau_{\text{list}} \]  
(36.31)

\[ \Gamma \vdash \text{cons}(e, \Omega_{\text{list}}) \cong \Omega_{\text{list}} : \tau_{\text{list}} \]  
(36.32)

\[ \Gamma \vdash \text{listcase} \Omega_{\text{list}} \text{ of nil} \Rightarrow e' \mid \text{cons}(x_1, x_2) \Rightarrow e'' \cong \Omega_{\tau'} : \tau' \]  
(36.33)

Most importantly, we may prove equivalences by induction on the structure of a list. Suppose that \( \Gamma [x : \tau_{\text{list}}] \vdash e_i : \tau' \) \((i = 1, 2)\). To prove that \( \Gamma [x : \tau_{\text{list}}] \vdash e_1 \cong e_2 : \tau' \), it suffices to show the following two facts:

1. \( \Gamma \vdash [x \leftarrow \text{nil}] e_1 \cong [x \leftarrow \text{nil}] e_2 : \tau' \)

2. For every \( \Gamma \vdash v_h : \tau \) and \( \Gamma \vdash v_l : \tau_{\text{list}} \), if \( \Gamma \vdash [x \leftarrow v_l] e_1 \cong [x \leftarrow v_l] e_2 : \tau' \), then \( \Gamma \vdash [x \leftarrow \text{cons}(v_h, v_l)] e_1 \cong [x \leftarrow \text{cons}(v_h, v_l)] e_2 : \tau' \).

**Exercise 36.3**

The list append and reversal functions are defined as follows.\(^2\)

\[
\begin{align*}
\text{fun app} \ (l : \tau_{\text{list}}, m : \tau_{\text{list}}) : \tau_{\text{list}} \ &\text{is} \\
\ &\text{listcase} \ l \ &\text{of nil} \ &\Rightarrow \ &m \ &\mid \ h : : t \ &\Rightarrow \ h : : \ \text{app} \ (t, m) \ &\text{end end} \\
\text{fun} \ \text{rev} \ (l : \tau_{\text{list}}) \ &\text{is} \\
\ &\text{listcase} \ l \ &\text{of nil} \ &\Rightarrow \ &\text{nil} \ &\mid \ h : : t \ &\Rightarrow \ &\text{app} \ (\text{rev}(t), [h]) \ &\text{end end}
\end{align*}
\]

Use list induction and the laws of expression equivalence to prove the following two facts:

1. \( x : \tau_{\text{list}}, y : \tau_{\text{list}} \vdash \text{rev}(\text{app}(x, y)) \cong \text{app}(\text{rev}(y), \text{rev}(x)) : \tau_{\text{list}} \)

2. \( x : \tau_{\text{list}} \vdash \text{rev}(\text{rev}(x)) \cong x : \tau_{\text{list}} \)

\(^2\)Officially, the two-argument append function is written using bind as follows:

\[
\begin{align*}
\text{fun} \ \text{app} \ (lm : \tau_{\text{list}} * \tau_{\text{list}}) : \tau_{\text{list}} \ &\text{is} \\
\ &\text{bind} \ (l : \tau_{\text{list}}, m : \tau_{\text{list}}) \ &\text{to} \ lm \ &\text{in} \ \ldots \ &\text{end end.}
\end{align*}
\]

We use infix notation for the append function for the sake of clarity.
36.2 Some Laws of Equivalence
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 given 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 of these (by the hypothesis that its type is “right”), you’re much more likely to have written it if the set of possibilities is smaller.

The technical foundation for these remarks is called parametricity. The goal of this chapter 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 lan-
guages with mutable references!

37.1 Informal Overview

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 \( \forall t (\text{arrow}(t, t)) \). What function could it be?

1. It could diverge when instantiated — \( f [\tau] \) goes into an infinite loop. Since \( f \) is polymorphic, its behavior cannot depend on the choice of \( \tau \), so in fact \( f [\tau'] \) diverges for all \( \tau' \) if it diverges for \( \tau \).

2. It could converge when instantiated at \( \tau \) to a function \( g \) of type \( \text{arrow}(\tau, \tau) \) that loops when applied to an argument \( v \) of type \( \tau \) — i.e., \( g(v) \) runs forever. Since \( f \) is polymorphic, \( g \) must diverge on every argument \( v \) of type \( \tau \) if it diverges on some argument of type \( \tau \).

3. It could converge when instantiated at \( \tau \) to a function \( g \) of type \( \text{arrow}(\tau, \tau) \) that, when applied to a value \( v \) of type \( \tau \) returns a value \( v' \) of type \( \tau \). Since \( f \) is polymorphic, \( g \) cannot depend on the choice of \( v \), so \( v' \) must in fact be \( v \).

Let us call cases (1) and (2) uninteresting. The foregoing discussion suggests that the only interesting function \( f \) of type \( \forall t (\text{arrow}(t, t)) \) 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 \( \forall t (\text{arrow}(\text{list}, \text{list})) \). What function could it be?

1. The identity function that simply returns its argument.

2. The constantly-nil function that always returns the empty list.

3. A function that drops some elements from the list according to a predetermined (data-independent) algorithm — e.g., always drops the first three elements of its argument.
4. A permutation function that reorganizes the elements of its argument.

The characteristic that these functions have in common is that their behavior is entirely determined by the spine of the list, and is independent of the elements of the list. For example, \( f \) cannot be the function that drops all “even” elements of the list — the elements might not be numbers! The point is that the type of \( f \) is polymorphic in the element type, but reveals that the argument is a list of unspecified elements. Therefore it can only depend on the “list-ness” of its argument, and never on its contents.

In general if a polymorphic function behaves the same at every type instance, we say that it is parametric in that type. In PolyMinML all polymorphic functions are parametric. In Standard ML most functions are, except those that involve equality types. The equality function is not parametric because the equality test depends on the type instance — testing equality of integers is different than testing equality of floating point numbers, and we cannot test equality of functions. Such “pseudo-polymorphic” operations are said to be ad hoc, to contrast them from parametric.

How can parametricity be exploited? As we will see later, parametricity is the foundation for data abstraction in a programming language. To get a sense of the relationship, let us consider a classical example of exploiting parametricity, the polymorphic Church numerals. Let \( N \) be the type \( \forall t \langle \text{arrow}(t, \text{arrow}((\text{arrow}(t, t)), t)) \rangle \). What are the interesting functions of the type \( N \)? Given any type \( \tau \), and values \( z : \tau \) and \( s : \text{arrow}(\tau, \tau) \), the expression

\[
f[\tau](z)(s)
\]

must yield a value of type \( \tau \). Moreover, it must behave uniformly with respect to the choice of \( \tau \). What values could it yield? The only way to build a value of type \( \tau \) is by using the element \( z \) and the function \( s \) passed to it. A moment’s thought reveals that the application must amount to the \( n \)-fold composition

\[
s(s(\ldots s(z) \ldots)).
\]

That is, the elements of \( N \) are in 1-to-1 correspondence with the natural numbers.

Let us write \( \pi \) for the polymorphic function of type \( N \) representing the natural number \( n \), namely the function
Informal Overview

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 \( n \) at the built-in type \( \texttt{int} \) and apply the result to \( 0 \) and \( \texttt{succ} \), it evaluates to the number \( n \). In general we may think of performing an "experiment" on a value of type \( N \) by instantiating it at a type whose values will constitute the observations, the applying it to operations \( z \) and \( s \) for performing the experiment, and observing the result.

Using this we can calculate with Church numerals. Let us consider how to define the addition function on \( N \). Given \( m \) and \( n \) of type \( N \), we wish to compute their sum \( m + n \), also of type \( N \). That is, the addition function must look as follows:

\[
\text{fn m:N in}
  \text{fn n:N in}
    \text{Fun t in}
      \text{fn z:t in}
        \text{fn s:t->t in}
          \ldots
        \text{end}
      \text{end}
    \text{end}
  \text{end}
\]

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:
37.2 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 \( f : \forall t (\text{arrow}(t,t)) \), then \( f \) must be equal to the identity function, and if \( f : N \), then \( f \) must be equal to some Church numeral \( \bar{n} \). To make the informal idea of parametricity precise, we must clarify what we mean by equality of expressions.

The main idea is to define equality in terms of “experiments” that we carry out on expressions to “test” whether they are equal. The valid experiments on an expression are determined solely by its type. In general we say that two closed expressions of a type \( \tau \) are equal iff either they both diverge, or they both converge to equal values of that type. Equality of closed values is then defined based on their type. For integers and booleans, equality is straightforward: two values are equal iff they are identical. The intuition here is that equality of numbers and booleans is directly observable. Since functions are “infinite” objects (when thought of in terms of their input/output behavior), we define equality in terms of their behavior when applied. Specifically, two functions \( f \) and \( g \) of type \( \text{arrow}(\tau_1,\tau_2) \) are equal iff whenever they are applied to equal arguments of type \( \tau_1 \), they yield equal results of type \( \tau_2 \).

More formally, we make the following definitions. First, we define equality of closed expressions of type \( \tau \) as follows:

\[
e \cong_{\text{exp}} e' : \tau \quad \text{iff} \quad e \mapsto^* v \iff e' \mapsto^* v' \quad \text{and} v \cong_{\text{val}} v' : \tau .
\]
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:

\[
\begin{align*}
  v \equiv_{val} v' : \text{bool} & \iff v = v' = \text{true} \text{ or } v = v' = \text{false} \\
  v \equiv_{val} v' : \text{int} & \iff v = v' = n \text{ for some } n \geq 0 \\
  v \equiv_{val} v' : \text{arrow}(\tau_1, \tau_2) & \iff v_1 \equiv_{val} v'_1 : \tau_1 \text{ implies } v(v_1) \equiv_{exp} v'(v'_1) : \tau_2
\end{align*}
\]

The following lemma states two important properties of this notion of equality.

**Lemma 37.1**

1. Expression and value equivalence are reflexive, symmetric, and transitive.
2. Expression equivalence is a congruence: we may replace any sub-expression of an expression $e$ by an equivalent sub-expression to obtain an equivalent expression.

So far we’ve considered only equality of closed expressions of monomorphic type. The definition is made so that it readily generalizes to the polymorphic case. The idea is that when we quantify over a type, we are not able to say a priori what we mean by equality at that type, precisely because it is “unknown”. Therefore we also quantify over all possible notions of equality to cover all possible interpretations of that type. Let us write $R : \tau \leftrightarrow \tau'$ to indicate that $R$ is a binary relation between values of type $\tau$ and $\tau'$.

Here is the definition of equality of polymorphic values:

\[
v \equiv_{val} v' : \forall t(\sigma) \iff \text{for all } \tau \text{ and } \tau', \text{ and all } R : \tau \leftrightarrow \tau', v [\tau] \equiv_{exp} v' [\tau'] : \sigma
\]

where we take equality at the type variable $t$ to be the relation $R$ (i.e., $v \equiv_{val} v' : t$ 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 $e \rightarrow e_1 \rightarrow e_2 \rightarrow \cdots e_n$. 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 37.2 (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 37.3**

*If \( f : \forall t (\text{arrow}(t,t)) \) is an interesting value, then \( f \cong_{\val} \text{id} : \forall t (\text{arrow}(t,t)), \) where \( \text{id} \) is the polymorphic identity function.*

**Proof:** Suppose that \( \tau \) and \( \tau' \) are monotypes, and that \( R : \tau \leftrightarrow \tau' \). We wish to show that

\[
f[\tau] \cong_{\exp} \text{id}[\tau'] : \text{arrow}(t,t),
\]

where equality at type \( t \) is taken to be the relation \( R \).

Since \( f \) (and \( \text{id} \)) are interesting, there exists values \( f_\tau \) and \( \text{id}_{\tau'} \) such that

\[
f[\tau] \xrightarrow{*} f_\tau
\]

and

\[
\text{id}[\tau'] \xrightarrow{*} \text{id}_{\tau'}.
\]

We wish to show that

\[
f_\tau \cong_{\val} \text{id}_{\tau'} : \text{arrow}(t,t).
\]

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

\[
f_\tau(v_1) \cong_{\exp} \text{id}_{\tau'}(v'_1) : t
\]

By the assumption that \( f \) is interesting (and the fact that \( \text{id} \) is interesting), there exists values \( v_2 \) and \( v'_2 \) such that

\[
f_\tau(v_1) \xrightarrow{*} v_2
\]
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 \( \{ (v, v) \mid v \in R v'_1 \} \). Since \( f : \forall t(\text{arrow}(t, t)) \), we have by the Parametricity Theorem that \( f \cong_{\text{val}} f : \forall t(\text{arrow}(t, t)) \), where equality at type \( t \) is taken to be the relation \( R' \). Since \( v_1 R v'_1 \), we have by definition \( v_1 R' v_1 \). Using the definition of equality of polymorphic type, it follows that
\[
 f_{\tau}(v_1) \cong_{\text{exp}} id_{\tau'}(v_1) : t.
\]

Hence \( v_2 R v'_1 \), as required. ■

You might reasonably wonder, at this point, what the relationship \( f \cong_{\text{val}} id : \forall t(\text{arrow}(t, t)) \) 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 \( f : \forall t(\text{arrow}(t, t)) \), we can show that for every \( t \) and every \( v : t \), \( f [\tau](v) \) evaluates to \( v \). By the preceding theorem \( f \cong_{\text{val}} id : \forall t(\text{arrow}(t, t)) \). Suppose that \( \tau \) is some monotype and \( v : \tau \) is some closed value. Define the relation \( R : \tau \leftrightarrow \tau \) by
\[
v_1 R v_2 \text{ iff } v_1 = v_2 = v.
\]

Then we have by the definition of equality for polymorphic values
\[
f [\tau](v) \cong_{\text{exp}} id [\tau](v) : t,
\]
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 38

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

$$\text{pack } \tau_1 \text{ with } v_1 \text{ as } \exists t(\sigma)$$

and

$$\text{pack } \tau_2 \text{ with } v_2 \text{ as } \exists t(\sigma)$$

of type $\exists t(\sigma)$ iff $v_1 \cong_{\text{val}} v_2 : \sigma$, taking equality at type $t$ to be the relation $R$. The reason for calling such a relation $R$ a bisimulation will become apparent shortly. Two packages are said to be bisimilar whenever there is a bisimulation between them.

Since the client $e_c$ of a data abstraction of type $\exists t(\sigma)$ is essentially a polymorphic function of type $\forall t(\text{arrow}(\sigma, \tau_c))$, where $t \notin \text{FTV}(\tau_c)$, it follows from the Parametricity Theorem that

$$[t, x \leftarrow \tau_1, v_1]e_c \cong_{\exp} [t, x \leftarrow \tau_2, v_2]e_c : \tau_c$$

whenever $R$ is such a bisimulation. Consequently,

$$\text{open } e_1 \text{ as } t \text{ with } x : \sigma \text{ in } e_c \cong_{\exp} \text{open } e_2 \text{ as } t \text{ with } x : \sigma \text{ in } e_c : \tau_c.$$ 

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 \( \exists \alpha(\sigma) \) 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 \textit{candidate} implementation. The Representation Theorem suggests a technique for proving the candidate correct. First, we define a \textit{reference} implementation of the same abstract type that is “obviously correct”. Then we establish that the reference implementation and the candidate implementation are bisimilar. Consequently, they are equivalent, which is to say that the candidate is “equally correct as” the reference implementation.

Returning to the queues example, let us take as a reference implementation the package determined by representing queues as lists. As a candidate implementation we take the package corresponding to the following ML code:

```ml
structure QFB :> QUEUE =
  struct
    type queue = int list * int list
    val empty = (nil, nil)
    fun insert (x, (bs, fs)) = (x::bs, fs)
    fun remove (bs, nil) = remove (nil, rev bs)
    | remove (bs, f::fs) = (f, (bs, fs))
  end
```

We will show that QL and QFB are bisimilar, and therefore indistinguishable by any client.

Define the relation \( R : \text{int list} \leftrightarrow \text{int list}*\text{int list} \) as follows:

\[
R = \{ (l, (b,f)) \mid l \cong b@rev(f) \}
\]

We will show that \( R \) is a bisimulation by showing that implementations of \texttt{empty}, \texttt{insert}, and \texttt{remove} determined by the structures QL and QFB are equivalent relative to \( R \).

To do so, we will establish the following facts:

1. \( \texttt{QL.empty} R \texttt{QFB.empty} \).

2. Assuming that \( m \cong n : \text{int} \) and \( l R (b,f) \), show that

\[
\texttt{QL.insert}((m,l)) R \texttt{QFB.insert}((n,(b,f))).
\]
3. Assuming that \( l \mathrel{R} (b, f) \), show that

\[
QL.\text{remove}(l) \cong_{\text{exp}} \text{QFB.\text{remove}}((b, f)) : \text{int}\ast t,
\]

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 \( \text{QFB} \), up to the relationship \( R \) between their representations.

The proofs of these facts are relatively straightforward, given some relatively obvious lemmas about expression equivalence.

1. To show that \( QL.\text{empty} \mathrel{R} QFB.\text{empty} \), it suffices to show that

\[
\text{nil}\otimes\text{rev}(\text{nil}) \cong_{\text{exp}} \text{exp}\text{nil} : \text{int list},
\]

which is obvious from the definitions of \( \text{append} \) and \( \text{reverse} \).

2. For \( \text{insert} \), we assume that \( m \cong_{\text{val}} n : \text{int} \) and \( l \mathrel{R} (b, f) \), and prove that

\[
QL.\text{insert}(m, l) \mathrel{R} \text{QFB.\text{insert}}(n, (b, f)).
\]

By the definition of \( QL.\text{insert} \), the left-hand side is equivalent to \( m::l \), and by the definition of \( QR.\text{insert} \), the right-hand side is equivalent to \( (n::b, f) \). It suffices to show that

\[
m::l \cong_{\text{exp}} (n::b)\otimes\text{rev}(f) : \text{int list}.
\]

Calculating, we obtain

\[
(n::b)\otimes\text{rev}(f) \cong_{\text{exp}} n::(b\otimes\text{rev}(f)) \cong_{\text{exp}} n::l
\]

since \( l \cong_{\text{exp}} b\otimes\text{rev}(f) \). Since \( m \cong_{\text{val}} n : \text{int} \), it follows that \( m = n \), which completes the proof.

3. For \( \text{remove} \), we assume that \( l \) is related by \( R \) to \( (b, f) \), which is to say that \( l \cong_{\text{exp}} b\otimes\text{rev}(f) \). We are to show

\[
QL.\text{remove}(l) \cong_{\text{exp}} \text{QFB.\text{remove}}((b, f)) : \text{int}\ast t,
\]
taking \( t \) equality to be the relation \( R \). Assuming that the queue is non-empty, so that the \texttt{remove} is defined, we have \( l \equiv_{exp} l'@m \) for some \( l' \) and \( m \). We proceed by cases according to whether or not \( f \) is empty. If \( f \) is non-empty, then \( f \equiv_{exp} n::f' \) for some \( n \) and \( f' \). Then by the definition of \texttt{QFB.remove},

\[
\text{QFB.remove}((b,f)) \equiv_{exp} (n,(b,f')) : \texttt{int*},
\]

relative to \( R \). We must show that

\[
(m,l') \equiv_{exp} (n,(b,f')) : \texttt{int*},
\]

relative to \( R \). This means that we must show that \( m = n \) and \( l' \equiv_{exp} b@\text{rev}(f') : \texttt{int list} \).

Calculating from our assumptions,

\[
\begin{align*}
l &= l'@m \\
    &= b@\text{rev}(f) \\
    &= b@\text{rev}(n::f') \\
    &= b@(\text{rev}(f')@[n]) \\
    &= (b@\text{rev}(f'))@[n]
\end{align*}
\]

From this the result follows. Finally, if \( f \) is empty, then \( b \equiv_{exp} b'@[n] \) for some \( b' \) and \( n \). But then \( \text{rev}(b) \equiv_{exp} n::\text{rev}(b') \), which reduces to the case for \( f \) non-empty.

This completes the proof — by Representation Independence the reference and candidate implementations are equivalent.
Part XIV

Concurrency
Chapter 39

Process Calculus
Chapter 40

Concurrent ML
Part XV

Dynamic Typing
Chapter 41

Dynamic Typing

The formalization of type safety given in Chapter 11 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 11 was how to handle expressions such as 3 div 0, which are well-typed, yet stuck, in apparent violation of the progress theorem. We discussed two possible ways to handle such a situation. One is to enrich the type system so that such an expression is ill-typed. However, this takes us considerably beyond the capabilities of current type systems for practical programming languages. The alternative is to ensure that such ill-defined states are not “stuck”, but rather make a transition to a designated error state. To do so we introduced the notion of a checked error, which is explicitly detected and signalled during execution. Checked errors are constrained with unchecked errors, which are ruled out by the static semantics.

In this chapter we will concern ourselves with question of why there should unchecked errors at all. Why aren’t all errors, including type errors, checked at run-time? Then we can dispense with the static semantics entirely, and, in the process, execute more programs. Such a language is called dynamically typed, in contrast to MinML, which is statically typed.

One advantage of dynamic typing is that it supports a more flexible treatment of conditionals. For example, the expression
(if true then 7 else "7")+1

is statically ill-typed, yet it executes successfully without getting stuck or incurring a checked error. Why rule it out, simply because the type checker is unable to “prove” that the else branch cannot be taken? Instead we may shift the burden to the programmer, who is required to maintain invariants that ensure that no run-time type errors can occur, even though the program may contain conditionals such as this one.

Another advantage of dynamic typing is that it supports heterogeneous data structures, which may contain elements of many different types. For example, we may wish to form the “list”

[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.
41.1 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 12, which we repeat here for convenience:

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

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

\[
\begin{align*}
    v \text{ value} & \quad v_1 \text{ value} & \quad (v \neq \text{fun} \: (x:\tau_1):\tau_2 \: \text{is} \: e) \\
    \text{apply}(v, v_1) & \longmapsto \text{error} \\
    \text{apply}(\text{error}, e_2) & \longmapsto \text{error} \\
    v_1 \text{ value} & \quad \text{apply}(v_1, \text{error}) \longmapsto \text{error} \\
\end{align*}
\]

The first rule states that a run-time error arises from any attempt to apply a non-function to an argument. The other two define the propagation of such errors through other expressions — once an error occurs, it propagates throughout the entire program.

By entirely analogous means we may augment the rest of the semantics of MinML with rules to check for type errors at run time. Once we have done so, it is safe to eliminate the static semantics in its entirety.\(^1\) Having

\(^1\)We may then simplify the language by omitting type declarations on variables and functions, since these are no longer of any use.
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.

### 41.2 Implementing Dynamic Typing

Since both the statically- and the dynamically typed variants of MinML are safe, it is natural to ask which is better. The main difference is in how early errors are detected — at compile time for static languages, at run time for dynamic languages. Is it better to catch errors early, but rule out some useful programs, or catch them late, but admit more programs? Rather than attempt to settle this question, we will sidestep it by showing that the apparent dichotomy between static and dynamic typing is illusory by showing that dynamic typing is a *mode of use* of static typing. From this point of view static and dynamic typing are matters of design for a particular program (which to use in a *given* situation), rather than a doctrinal debate about the design of a programming language (which to use in *all* situations).

To see how this is possible, let us consider what is involved in implementing a dynamically typed language. The dynamically typed variant of MinML sketched above includes rules for run-time type checking. For example, the dynamic semantics includes a rule that explicitly checks for an attempt to apply a non-function to an argument. How might such a check be implemented? The chief problem is that the natural representations of data values on a computer do not support such tests. For example, a function might be represented as a word representing a pointer to a region of memory containing a sequence of machine language instructions. An integer might be represented as a word interpreted as a two’s complement integer. But given a word, you cannot tell, in general, whether it is an integer or a code pointer.

To support run-time type checking, we must adulterate our data representations to ensure that it is possible to implement the required checks. We must be able to tell by looking at the value whether it is an integer, a boolean, or a function. Having done so, we must be able to recover the underlying value (integer, boolean, or function) for direct calculation. When-
41.2 Implementing Dynamic Typing

ever a value of a type is created, it must be marked with appropriate information to identify the sort of value it represents.

There are many schemes for doing this, but at a high level they all amount to attaching a tag to a “raw” value that identifies the value as an integer, boolean, or function. Dynamic typing then amounts to checking and stripping tags from data during computation, transitioning to error whenever data values are tagged inappropriately. From this point of view, we see that dynamic typing should not be described as “run-time type checking”, because we are not checking types at run-time, but rather tags. The difference can be seen in the application rule given above: we check only that the first argument of an application is some function, not whether it is well-typed in the sense of the MinML static semantics.

To clarify these points, we will make explicit the manipulation of tags required to support dynamic typing. To begin with, we revise the grammar of MinML to make a distinction between tagged and untagged values, as follows:

\[
\begin{align*}
\text{Expressions} & \quad e \quad ::= \quad x \mid v \mid o(e_1, \ldots, e_n) \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_2 \mid \text{apply}(e_1, e_2) \\
\text{TaggedValues} & \quad v \quad ::= \quad \text{Int } (n) \mid \text{Bool } \text{(true)} \mid \text{Bool } \text{(false)} \mid \text{Fun } (\text{fun } x(y: \tau_1): \tau_2 \text{ is } e) \\
\text{UntaggedValues} & \quad u \quad ::= \quad \text{true} \mid \text{false} \mid n \mid \text{fun } x(y: \tau_1): \tau_2 \text{ is } e
\end{align*}
\]

Note that only tagged values arise as expressions; untagged values are used strictly for “internal” purposes in the dynamic semantics. Moreover, we do not admit general tagged expressions such as \(\text{Int } (e)\), but only explicitly-tagged values.

Second, we introduce tag checking rules that determine whether or not a tagged value has a given tag, and, if so, extracts its underlying untagged value. In the case of functions these are given as rules for deriving judgements of the form \(v \text{ is } \text{fun } u\), which checks that \(v\) has the form \(\text{Fun } (u)\), and extracts \(u\) from it if so, and for judgements of the form \(v \text{ isnt } \text{fun}\), that checks that \(v\) does not have the form \(\text{Fun } (u)\) for any untagged value \(u\).

\[
\begin{align*}
\text{Fun } (u) & \text{ is } \text{fun } u \\
\text{Int } (\_ ) & \text{ isnt } \text{fun} \quad \text{Bool } (\_ ) & \text{ isnt } \text{fun}
\end{align*}
\]

Similar judgements and rules are used to identify integers and booleans, and to extract their underlying untagged values.
Finally, the dynamic semantics is re-formulated to make use of these judgement forms. For example, the rules for application are as follows:

\[
\begin{align*}
\text{apply}(v, v_1) & \rightarrow [f, x \leftarrow v, v_1]e \\
\text{apply}(v, v_1) & \rightarrow \text{error}
\end{align*}
\]

Similar rules govern the arithmetic primitives and the conditional expression. For example, here are the rules for addition:

\[
\begin{align*}
\text{apply}(v_1, v_2) & \rightarrow \text{Int}(n) \\
\text{apply}(v_1, v_2) & \rightarrow \text{error}
\end{align*}
\]

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.

These rules explicitly check for non-integer arguments to addition.

### 41.3 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:

\[
\text{datatye tagged =}
\begin{align*}
\text{Int of int} & | \\
\text{Bool of bool} & | \\
\text{Fun of tagged -> tagged}
\end{align*}
\]

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.

\[
\text{exception TypeError}
\]
\[
\text{fun checked_add (m:tagged, n:tagged):tagged =}
\begin{align*}
\text{case (m,n) of} \\
(\text{Int a, Int b}) & \Rightarrow \text{Int (a+b)} \\
(\_, \_) & \Rightarrow \text{raise TypeError}
\end{align*}
\]
\[
\text{fun checked_apply (f:tagged, a:tagged):tagged =}
\begin{align*}
\text{case f of} \\
\text{Fun g} & \Rightarrow g \ a \\
\_ & \Rightarrow \text{raise TypeError}
\end{align*}
\]

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.

\[
\text{val het_list : tagged list =}
\begin{align*}
[\text{Int 1, Bool true, Fun (fn x => x)}]
\end{align*}
\]
\[
\text{val f : tagged = hd(tl(tl het_list))}
\]
\[
\text{val x : tagged = checked_apply (f, Int 5)}
\]
The tags on the elements serve to identify what sort of element it is: an integer, a boolean, or a function.

It is enlightening to consider a dynamically typed version of the factorial function:

```haskell
fun dyn_fact (n : tagged) =
  let fun loop (n, a) =
    case n
    of Int m =>
      (case m
        of 0 => a
        | m => loop (Int (m-1),
                     checked_mult (m, a)))
    | _ => raise RuntimeTypeError
  in loop (n, Int 1)
end
```

Notice that tags must be manipulated within the loop, even though we can prove (by static typing) that they are not necessary! Ideally, we would like to hoist these checks out of the loop:

```haskell
fun opt_dyn_fact (n : tagged) =
  let fun loop (0, a) = a
    | loop (n, a) = loop (n-1, n*a)
  in case n
    of Int m => Int (loop (m, 1))
    | _ => raise RuntimeTypeError
  end
```

It is very hard for a compiler to do this hoisting reliably. But if you consider dynamic typing to be a special case of static typing, as we do here, there is no obstacle to doing this optimization yourself, as we have illustrated here.
Part XVI

Storage Management
Chapter 42

Storage Management

The dynamic semantics for MinML given in Chapter 12, and even the C-machine given in Chapter 19, 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.

42.1 The A Machine

The A-machine is defined for an extension of MinML in which we add an additional form of expression, a location, $l$, which will serve as a “reference” or “pointer” into the heap.

Values are classified into two categories, small and large, by the following rules:

\[
\frac{(l \in \text{Loc})}{l \text{ svalue}}
\]  

(42.1)
A state of the A-machine has the form \((H, k, e)\), where \(H\) is a heap, a finite function mapping locations to large values, \(k\) is a control stack, and \(e\) is an expression. A heap \(H\) is said to be self-contained iff \(\text{FL}(H) \subseteq \text{dom}(H)\), where \(\text{FL}(H)\) is the set of locations occurring free in any location in \(H\), and \(\text{dom} H\) is the domain of \(H\).

Stack frames are similar to those of the C-machine, but refined to account for the distinction between small and large values.

\[
\begin{align*}
\frac{e_2 \text{ exp}}{+(\neg, e_2) \text{ frame}} & \quad (42.6) \\
\frac{v_1 \text{ svalue}}{+(v_1, \neg) \text{ frame}} & \quad (42.7)
\end{align*}
\]

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

\[
\begin{align*}
\frac{e_1 \text{ exp} \quad e_2 \text{ exp}}{\text{ if } - \text{ then } e_1 \text{ else } e_2 \text{ frame}} & \quad (42.8) \\
\frac{e_2 \text{ exp}}{\text{ apply}(\neg, e_2) \text{ frame}} & \quad (42.9) \\
\frac{v_1 \text{ svalue}}{\text{ apply}(v_1, \neg) \text{ frame}} & \quad (42.10)
\end{align*}
\]

Notice that \(v_1\) is required to be a small value; a function is represented by a location in the heap, which is small.

As with the C-machine, a stack is a sequence of frames:

\[
\varepsilon \text{ stack} \quad (42.11)
\]
The dynamic semantics of the A-machine is given by a set of rules defining the transition relation \( (H, k, e) \mapsto_A (H', k', e') \). 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:

\[
(H, k, + (e_1, e_2)) \mapsto_A (H, + (-, e_2); k, e_1)
\]

(42.13)

\[
(H, + (-, e_2); k, v_1) \mapsto_A (H, + (v_1, -); k, e_2)
\]

(42.14)

\[
(H, + (n_1, -); k, n_2) \mapsto_A (H, k, n_1 + n_2)
\]

(42.15)

Note that the heap is simply “along for the ride” in these rules.

Booleans are also handled similarly to the C-machine:

\[
(H, k, if \ e \ then \ e_1 \ else \ e_2) \mapsto_A (H, if - \ then \ e_1 \ else \ e_2; k, e)
\]

(42.16)

\[
(H, if - \ then \ e_1 \ else \ e_2; k, true) \mapsto_A (H, k, e_1)
\]

(42.17)

\[
(H, if - \ then \ e_1 \ else \ e_2; k, false) \mapsto_A (H, k, e_2)
\]

(42.18)

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.

\[
(H, k, fun \ x \ (y: \tau_1): \tau_2 \ is \ e) \mapsto_A (H[l \mapsto fun \ x \ (y: \tau_1): \tau_2 \ is \ e], k, l)
\]

(42.19)

where \( l \) is chosen so that \( l \notin \text{dom} \ H \).

Evaluation of the function and argument position of an application is handled similarly to the C-machine.

\[
(H, k, apply(e_1, e_2)) \mapsto_A (H, apply(-, e_2); k, e_1)
\]

(42.20)
Execution of a function call differs from the corresponding C-machine instruction in that the function must be retrieved from the heap in order to determine the appropriate instance of its body. Notice that the location of the function, and not the function itself, is substituted for the function variable!

\[
\begin{align*}
  v_1 \text{ loc } \ H(v_1) &= \text{fun } f \ (x: \tau_1): \tau_2 \ is \ e \\
  (H, \text{apply}(v_1, -); k, v_2) &\mapsto_A (H, k, \ [f, x \leftarrow v_1, v_2] e)
\end{align*}
\] (42.22)

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 42.1**

*If* $H$ *is self-contained and* $(H, k, e) \mapsto_A (H', k', e')$, *then* $H'$ *is also self-contained. Moreover, if* $\text{FL}(k) \cup \text{FL}(e) \subseteq \text{dom } H$, *then* $\text{FL}(k') \cup \text{FL}(e') \subseteq \text{dom } H'$.\)**

It is not too difficult to see that the A-machine and the C-machine have the same “observable behavior” in the sense that both machines determine the same value for closed expressions of integer type. However, it is somewhat technically involved to develop a precise correspondence. The main idea is to define the *heap expansion* of an A-machine state to be the C-machine state obtained by replacing all locations in the stack and expression by their values in the heap. (It is important to take care that the locations occurring in a value stored are themselves replaced by their values in the heap!) We then prove that an A-machine state reaches a final state in accordance with the transition rules of the A-machines iff its expansion does in accordance with the rules of the C-machine. Finally, we observe that the value of a final state of integer type is the same for both machines.

Formally, let $\hat{H}(e)$ stand for the substitution

\[ [l_1, \ldots, l_n \leftarrow H(l_1), \ldots, H(l_n)] e, \]

where $\text{dom } H = \{ l_1, \ldots, l_n \}$. Similarly, let $\hat{H}(k)$ denote the result of performing this substitution on every expression occurring in the stack $k$. 
Theorem 42.2
If \((H, k, e) \rightarrow_A (H', k', e')\), then \(\hat{H}(k) \circ \hat{H}(e) \rightarrow_{C} 0^{1} \hat{H}'(k') \circ \hat{H}'(e')\).

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.

42.2 Garbage Collection

The purpose of the A-machine is to model the memory allocation that would be required in an implementation of MinML. This raises the question of garbage, storage that is no longer necessary for a computation to complete. The purpose of a garbage collector is to reclaim such storage for further use. Of course, in a purely abstract model there is no reason to perform garbage collection, but in practice we must contend with the limitations of finite, physical computers. For this reason we give a formal treatment of garbage collection for the A-machine.

The crucial issue for any garbage collector is to determine which locations are unnecessary for computation to complete. These are deemed garbage, and are reclaimed so as to conserve memory. But when is a location unnecessary for a computation to complete? Consider the A-machine state \((H, k, e)\). A location \(l \in \text{dom}(H)\) is unnecessary, or irrelevant, for this machine state iff execution can be completed without referring to the contents of \(l\). That is, \(l \in \text{dom} H\) is unnecessary iff \((H, k, e) \rightarrow_{A} (H', \epsilon, v)\) iff \((H_l, k, e) \rightarrow_{A} (H''_l, \epsilon, v)\), where \(H_l\) is \(H\) with the binding for \(l\) removed, and \(H''_l\) is some heap.

Unfortunately, a machine cannot decide whether a location is unnecessary!

Theorem 42.3
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 42.4**

If a location $l$ is unreachable in a state $(H, k, e)$, then it is also unnecessary for that state.

Each transition depends only on the locations occurring on the control stack or in the current expression. Some steps move values from the heap onto the stack or current expression. Therefore in a multi-step sequence, execution can depend only on reachable locations in the sense of the definition above.

The set of unreachable locations in a state may be determined by tracing. This is easily achieved by an iterative process that maintains a finite set of locations, called the roots, containing the locations that have been found to be reachable up to that point in the trace. The root set is initialized to the locations occurring in the expression and control stack. The tracing process completes when no more locations can be added. Having found the reachable locations for a given state, we then deem all other heap locations to be unreachable, and hence unnecessary for computation to proceed. For this reason the reachable locations are said to be live, and the unreachable are said to be dead.

Essentially all garbage collectors used in practice work by tracing. But since reachability is only a conservative approximation of necessity, all practical collectors are conservative! So-called conservative collectors are, in fact, incorrect collectors that may deem as garbage storage that is actually necessary for the computation to proceed. Calling such a collector “conservative” is misleading (actually, wrong), but it is nevertheless common practice in the literature.

The job of a garbage collector is to dispose of the unreachable locations in the heap, freeing up memory for later use. In an abstract setting where we allow for heaps of unbounded size, it is never necessary to collect garbage, but of course in practical situations we cannot afford to waste unlimited amounts of storage. We will present an abstract model of a particular form of garbage collection, called copying collection, that is widely used in practice. The goal is to present the main ideas of copying collection, and to prove that garbage collection is semantically “invisible” in the sense that it does not change the outcome of execution.
42.2 Garbage Collection

The main idea of copying collection is to simultaneously determine which locations are reachable, and to arrange that the contents of all reachable locations are preserved. The rest are deemed garbage, and are reclaimed. In a copying collector this is achieved by partitioning storage into two parts, called semi-spaces. During normal execution allocation occurs in one of the two semi-spaces until it is completely filled, at which point the collector is invoked. The collector proceeds by copying all reachable storage from the current, filled semi-space, called the from space, to the other semi-space, called the to space. Once this is accomplished, execution continues using the “to space” as the new heap, and the old “from space” is reclaimed in bulk. This exchange of roles is called a flip.

By copying all and only the reachable locations the collector ensures that unreachable locations are reclaimed, and that no reachable locations are lost. Since reachability is a conservative criterion, the collector may preserve more storage than is strictly necessary, but, in view of the fundamental undecidability of necessity, this is the price we pay for mechanical collection. Another important property of copying collectors is that their execution time is proportion to the size of the live data; no work is expended manipulating reclaimable storage. This is the fundamental motivation for using semi-spaces: once the reachable locations have been copied, the unreachable ones are eliminated by the simple measure of “flipping” the roles of the spaces. Since the amount of work performed is proportional to the live data, we can amortize the cost of collection across the allocation of the live storage, so that garbage collection is (asymptotically) “free”. However, this benefit comes at the cost of using only half of available memory at any time, thereby doubling the overall storage required.

Copying garbage collection may be formalized as an abstract machine with states of the form \((H_f, S, H_t)\), where \(H_f\) is the “from” space, \(H_t\) is the “to” space, and \(S\) is the scan set, the set of reachable locations. The initial state of the collector is \((H, S, \emptyset)\), where \(H\) is the “current” heap and \(\emptyset \neq S \subseteq \text{dom}(H_f)\) is the set of locations occurring in the program or control stack. The final state of the collector is \((H_f, \emptyset, H_t)\), with an empty scan set.

The collector is invoked by adding the following instruction to the A-machine:

\[
\begin{align*}
(H, FL(k) \cup FL(e), \emptyset) & \xrightarrow{G} (H'', \emptyset, H') \\
(H, k, e) & \xrightarrow{A} (H', k, e)
\end{align*}
\]

The scan set is initialized to the set of free locations occurring in either the current stack or the current expression. These are the locations that are
immediately reachable in that state; the collector will determine those that are transitively reachable, and preserve their bindings. Once the collector has finished, the “to” space is installed as the new heap.

Note that a garbage collection can be performed at any time! This correctly models the unpredictability of collection in an implementation, but avoids specifying the exact criteria under which the collector is invoked. As mentioned earlier, this is typically because the current heap is exhausted, but in an abstract setting we impose no fixed limit on heap sizes, preferring instead to simply allow collection to be performed spontaneously according to unspecified criteria.

The collection machine is defined by the following two rules:

\[
(H_f[l = v], S \cup \{l\}, H_t) \xrightarrow{G} (H_f, S \cup \text{FL}(v), H_t[l = v])
\] (42.24)

\[
(H_f, S \cup \{l\}, H_t[l = v]) \xrightarrow{G} (H_f, S, H_t[l = v])
\] (42.25)

The first rule copies a reachable binding in the “from” space to the “to” space, and extends the scan set to include those locations occurring in the copied value. This ensures that we will correctly preserve those locations that occur in a reachable location. The second rule throws away any location in the scan set that has already been copied. This rule is necessary because when the scan set is updated by the free locations of a heap value, we may add locations that have already been copied, and we do not want to copy them twice!

The collector is governed by a number of important invariants.

1. The scan set contains only “valid” locations: \( S \subseteq \text{dom } H_f \cup \text{dom } H_t \);
2. The “from” and “to” space are disjoint: \( \text{dom } H_f \cap \text{dom } H_t = \emptyset \);
3. Every location in “to” space is either in “to” space, or in the scan set:
   \( \text{FL}(H_t) \subseteq S \cup \text{dom } H_t \);
4. Every location in “from” space is either in “from” or “to” space: \( \text{FL}(H_f) \subseteq \text{dom } H_f \cup \text{dom } H_t \).

The first two invariants are minimal “sanity” conditions; the second two are crucial to the operation of the collector. The third states that the “to” space contains only locations that are either already copied into “to” space, or will eventually be copied, because they are in the scan set, and hence in “from” space (by disjointness). The fourth states that locations in “from”
space contain only locations that either have already been copied or are yet to be copied.

These invariants are easily seen to hold of the initial state of the collector, since the “to” space is empty, and the “from” space is assumed to be self-contained. Moreover, if these invariants hold of a final state, then $\text{FL}(H_I) \subseteq \text{dom } H_I$, since $S = \emptyset$ in that case. Thus the heap remains self-contained after collection.

**Theorem 42.5 (Preservation of Invariants)**

If the collector invariants hold of $(H_f, S, H_t)$ and $(H_f, S, H_t) \xrightarrow{G} (H'_f, S', H'_t)$, then the same invariants hold of $(H'_f, S', H'_t)$.

The correctness of the collector follows from the following lemma.

**Lemma 42.6**

If $(H_f, S, H_t) \xrightarrow{G} (H'_f, S', H'_t)$, then $H_f \cup H_t = H'_f \cup H'_t$ and $S \cup \text{dom } H_t \subseteq S' \cup \text{dom } H'_t$.

The first property states that the union of the semi-spaces never changes; bindings are only copied from one to the other. The second property states that the domain of the “to” space together with the scan set does not change.

From this lemma we obtain the following crucial facts about the collector. Let $S = \text{FL}(k) \cup \text{FL}(e)$, and suppose that

$$(H, S, \emptyset) \xrightarrow{G^*} (H'' \cup \emptyset, H').$$

Then we have the following properties:

1. The reachable locations are bound in $H'$: $\text{FL}(k) \cup \text{FL}(e) \subseteq \text{dom } H'$. This follows from the lemma, since the initial “to” space and the final scan set are empty.

2. The reachable data is correctly copied: $H' \subseteq H$. This follows from the lemma, which yields $H = H'' \cup H'$. 
42.2 Garbage Collection
Bibliography