THE SEMANTICS OF PROGRAMMING LANGUAGES:
An Elementary Introduction using Structural Operational Semantics

Matthew Hennessy

These notes are very slightly revised versions of selected chapters of the book by the same name, which is now our of print. Comments, suggestions, typos, etc. are very welcome.

Contents

Chapter 1

Preliminaries

In this chapter we review the background material which is necessary before embarking on the study of the main topic of this book: the semantics of programming languages. In the first section we try to make a clear distinction between the syntax and the semantics of languages. In the second section we discuss abstract and concrete syntax definitions and emphasise that to give a semantics to a programming language we require, a priori, a definition of its abstract syntax. Induction is the subject of the remainder of the chapter. In the main body of the book we will use inductive definitions extensively and these final sections give a thorough introduction to them. The third section is devoted to a general but informal introduction to this form of definition. In particular we give a number of examples of inductive definitions for well-known relations and we show how, in general, induction may be used to prove properties of inductively defined objects. In the fourth section we treat a particular case, called *structural induction*, and in the final one we examine the relationship between inductive definitions and proof systems.

1.1 Introduction

The definition of a programming language consists of at least two parts, the *syntax* and the *semantics*. The syntax is concerned with the form of expressions which are allowed in the language or, more precisely, with the sequences of symbols which will be accepted by a compiler/interpreter for the language. On the other hand the semantic definition could, for example, describe the effect of executing or evaluating any syntactically correct expression or program or it could describe how to execute or evaluate them.

In natural languages there is also this distinction between syntactic and semantic issues. The expression *home to went he* is not an English sentence as it violates the rules of English grammar, i.e. the syntactic rules of English. On the other hand, the expression *the fish answered the walk* is syntactically correct. It consists of a noun phrase, *the fish*, followed by a verb, *answered*, followed by an object phrase, *the walk*. Yet it is not English because it is semantically incorrect: it does not make sense.

For programming languages the syntax is usually defined *formally* using BNF notation or some kind of grammar. Most books on a particular programming language will include a section, usually at the end, which will have a formal definition of the allowable expressions in the language. But the semantics of the language, if given at all, is completely informal. At most it might consist of a couple of English sentences beneath the BNF definitions of the various syntactic categories in the language. For example, when introducing as a new kind of command, the expression

While b Do C,

the English phrase giving the semantics might say something like

> to execute this command you execute the command C repeatedly so long as the expression b is true.

In this book we will show that it is possible to give a precise account of the semantics of languages in much the same way as BNF descriptions give precise accounts of their syntax. The book is introductory and therefore we will not see formal semantic definitions for any "real" language, such as *Ada* or *Pascal*, although attempts may be seen in the literature. In fact, the entire subject is not very well developed and there is not even a general consensus on the best method to use. Using any of the proposed methods it is still very difficult to define the semantics of a non-trivial language. You may well ask: why bother?

The first simple answer is to help the users of the language. At the moment it is relatively straightforward to use the definition of the syntax of a language to decide which form of expression can be placed in a given context in a program. When one is relatively unfamiliar with the language, the syntactic definitions are very useful in this way. Without them one would have to submit trial expression to a compiler and wait for the result — a very time-consuming and inefficient solution. With respect to semantic issues, this is exactly what one must do, without a formal semantics. In fact, to discover the general behaviour of a program you might have to submit it to the compiler an infinite number of times! And what if the compiler were implemented incorrectly? Indeed, how does the compiler writer know that the compiler is a correct implementation of the intentions of the language designer? This points to a different but equally important role of a formal semantics: a machine- and compiler-independent standard definition of the language against which implementations might be judged. Anyone who has attempted to transfer a large program from one site to another, with a different compiler, will be aware of the problem caused by the lack of standardised definitions. The program rarely works because the compilers rarely match. A standard formal semantics would provide a machine-independent standard

for compiler writers; any proposed compiler would have to agree with this formal standard.

Returning to the perspective of the user of the language, would it not be nice to be able to turn to the end of the reference manual and use the semantic definition to deduce the behaviour of your proposed program? It could also be used to improve on programs with whose behaviour you are already satisfied. For example, in many programming languages

If true Then C1 Else C2

will have exactly same effect as

C1

and, therefore, we can systematically replace all occurrences of the former with the latter. Without a formal semantics one can only say that the semantic identity of these two statements seems reasonable. With a formal semantics we could actually *prove* that they are equivalent and therefore put our reasoning on a much more secure basis. This is a very simple example but, in many cases, it is far from clear if one statement can be substituted for another and the existence of a formal semantics is essential. Take the following pairs of statements:

C1;(If b Then C2 Else C3)
If b Then (C1;C2) Else (C1;C3)

and

(If b Then C2 Else C3);C1
If b Then (C2;C1) Else (C3;C1)

Are they both pairs of semantically equivalent statements and can each be freely replaced by its partner in any program? The answer depends on a number of factors which would be made apparent by a formal semantics. In order to improve, or in general modify, programs with any degree of assurance, we must be able to justify the semantic equivalence of pairs of statements such as these. Indeed, in one proposed method of program development one starts with a very inefficient but obviously correct program and, by a series of syntactic transformations, transforms it into an efficient program which is still correct. The allowable transformations are given by pairs of statements such as those above, and the consistency of the entire program development method depends on having a formal semantics with which to check these transformations.

1.2 Concrete and Abstract Syntax

A very simplified structure of the use of a typical compiler is as follows:

INPUT : sequence of symbols/program
=== *parser* $\Longrightarrow$
parse tree/internal representation
=== *code generation* $\Longrightarrow$
compiled program
=== *execution* $\Longrightarrow$
OUTPUT

The input is a linear sequence of characters which the user hopes is a syntactically correct program. The parser analyses this string, checks that it is syntactically correct and outputs a parse tree, or possibly some more general internal representation, either of which will describe the structure of the program to be executed. This first section is concerned only with syntactic issues and is governed by the syntactic definition of the language. The remaining phases are the concern of the semantic definition. So, the semantic definition should determine the effect of the execution of a program, given that it is syntactically correct and given a parse tree for the program or, more generally, a structural description of the program. The important point is that a semantic definition does not concern itself with assigning meanings to sequences of symbols; rather it assigns meanings to structural descriptions of programs. These descriptions will be given in terms of what we call *abstract syntax.*

Let us recall BNF syntax definitions. These:

- specify the sequences of symbols accepted by a parser for the language

  and

- determine the structure of the accepted sequences.

They are concerned with allowable sequences of symbols, the so-called *concrete syntax* of the language. The following is an example of a BNF definition for a subset of arithmetic expressions:

$$\begin{aligned} < exp > &::= < num > \mid < exp > < op > < exp > \\ < op > &::= + \mid - \mid * \mid div \\ < num > &::= < digit > \mid < digit > < num > \\ < digit > &::= \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9}. \end{aligned}$$

The terminal symbols are $+, -, *, div, \mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7}, \mathbf{8}, \mathbf{9}$ and the non-terminals $< exp >, < op >, < digit >, < num >$. Any sequence of symbols generated from $< exp >$ is a syntactically correct expression, i.e. a valid expression. However, this is

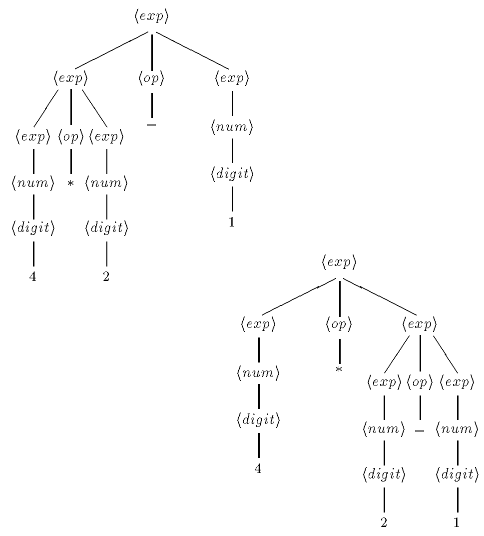


Figure 1.1: Parse trees for $4 * 2 - 1$

not a good BNF description as it specifies two different structures or parse trees for certain sequences of symbols. For example, $\mathbf{4} * \mathbf{2} - \mathbf{1}$ has the two possible parse trees which are given in Figure 1.1. The first interprets it as

$$< exp >< op >< exp >$$

where the first $< exp >$ is $\mathbf{4} * \mathbf{2}$, $< op >$ is $-$ and the second $< exp >$ is the numeral $\mathbf{1}$. The second parse tree also interprets it as

$$< exp >< op >< exp >$$

but in this case the first $< exp >$ is the numeral $\mathbf{4}$, $< op >$ is $*$ and the second $< exp >$ is $\mathbf{2} - \mathbf{1}$. Thus this BNF description could not be used as part of a formal definition of the language of arithmetic expressions as the semantic component would not know which parse tree to use to assign a meaning to $\mathbf{4} * \mathbf{2} - \mathbf{1}$. If it used the first it would obtain the meaning $\mathbf{7}$ whereas the second would lead to $\mathbf{4}$. A correct BNF definition, where each sequence of symbols in the language is assigned a unique parse tree, is :

$$\begin{aligned} < exp > &::= < term > \mid < exp > < lowop > < term > \\ < term > &::= < num > \mid < term > < highop >< num > \\ < num > &::= < digit > \mid < digit > < num > \end{aligned}$$

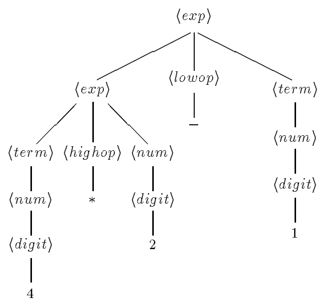


Figure 1.2: The unique parse tree for $\mathbf{4} * \mathbf{2} - \mathbf{1}$

$$
\begin{array}{rcl}
< \mathit{lowop} > & ::= & + \mid - \\
< \mathit{highop} > & ::= & * \mid \mathit{div}.
\end{array}
$$

In this case $\mathbf{4} * \mathbf{2} - \mathbf{1}$ has only one parse tree, which is given in Figure 1.2: it is interpreted as an expression of the form

$$< \mathit{exp} >< \mathit{lowop} >< \mathit{term} >$$

where $< \mathit{lowop} >$ is $-$, $< \mathit{term} >$ is the numeral $\mathbf{1}$ and $< \mathit{exp} >$ is $\mathbf{4} * \mathbf{2}$.

In contrast to BNF definitions, abstract definitions are only designed to specify the structure of expressions allowed in the language. In particular they are *not* concerned with acceptable strings of symbols nor with assigning to each such sequence of symbols a unique structure or parse tree. Instead they describe the set of allowable parse trees for the language.

A very simple definition of the abstract syntax of arithmetic expressions is given in Figure 1.3. It consists of two parts. The first enumerates the syntactic categories and introduces a particular symbol which will be used in the second part to stand for an arbitrary object of that category. In this example there are three different syntactic categories, *Exp*, the principal one for expressions and two auxiliary categories, *Op* for operators and *Num* for numerals. The expression

e in *Exp*

simply means that e will be used as a meta-variable for the syntactic category *Exp*, i.e. will be used as a typical element of *Exp*. Similarly, *op* will designate a typical element from *Op* and n a typical element from *Num*. The second part gives an explanation

1. Syntactic categories

   e in *Exp*

   op in *Op*

   n in *Num*

2. Definitions

$$
\begin{array}{rcl}
op & ::= & + \mid - \mid * \mid div \\
e & ::= & n \mid e' \; op \; e''
\end{array}
$$

Figure 1.3: Abstract for *Exp*

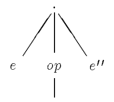
of the *structure* of objects in the different categories, using a BNF-like notation. For example, it says that every expression is of one of the two forms:

1. n, i.e. an element of *Num*

2. a structure which contains three components represented by e', op, e'' respectively. Because of the use of meta-variables it is assumed that both e' and e'' are also elements from *Exp* and *op* is an element from *Op*.

It also says that the category *Op* is very simple. It consists of four elements, $+, -, *, div$, i.e. the elements have essentially no structure. Finally, it says nothing about the structure of elements in the category *Num*. This just means that, whatever use will be made of this abstract syntax, the structure of the elements of *Num* will not play a role. It is best to view these as definitions of allowed parse trees. So the first possibility says that

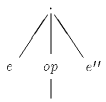
is a possible parse tree for an expression, where n is any element from the syntactic category *Num*, while the second admits



where it is assumed that e' and e'' also represent allowable parse trees for expressions and beneath *op* there is an admissible parse tree for the syntactic category *Op*. Note that the actual definition used to describe the structure of the elements in *Exp* is identical to the erroneous BNF definition for arithmetic expressions. This may be confusing but they are being put to very different uses. One determines the kinds of parse trees which arise when dealing with expressions, i.e. abstract syntax, whereas the other erroneously attempts to assign to each arithmetic expression a parse tree, i.e. attempts to define a concrete syntax. In this book we will never be concerned with concrete syntax. All the BNF-like definitions used will form part of a description of an abstract syntax of a language and therefore will be used to describe the allowable parse trees of that language. Initially the reader might welcome a more graphical representation of these definitions but one quickly gets accustomed to interpreting our linear definitions, such as

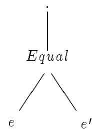
$$e ::= n \mid e' \;\; op \;\; e''$$

as definitions of parse trees. Also, by and large, we will not use a graphical notation to describe actual parse trees. Instead we will represent a parse tree such as



for example, by the linear expression $e \;\; op \;\; e'$. Of course, this will in general lead to ambiguities. However, this will not occur very often and when it does we will explain the intended structure of the expression, often using brackets.

Two further examples of abstract syntax definitions are given in Figure 1.4 and Figure 1.5. The first is an extended version of arithmetic expressions. The new syntactic categories are boolean expressions, *BExp*, variables, *Var*, boolean variables, *BVar*, and boolean operators, *BOp*. The structure of both kinds of variables is ignored and *BOp* simply consists of two constants. An arbitrary expression may now have one of four different structures, the new possibilities being a variable or of the form *If be Then* e' *Else* e'', where it is assumed that *be* is a boolean expression and both e' and e'' are expressions. There are five different possibilities for the structure of a boolean expression, all of which by now should be easy to understand. Note, however that the two syntactic categories *Exp* and *BExp* are not independent of each other. Using what should be a self-expainatory notation for the parse trees of this language one possible parse tree for the syntactic category *BExp* has the form



1. Syntactic categories

   e in *Exp*

   be in *BExp*

   x in *Var*

   bx in *BVar*

   op in *Op*

   bop in *BOp*

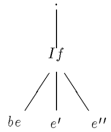
   n in *Num*

2. Definitions

$$\begin{array}{rcl}
e & ::= & x \mid n \mid e' \;\; op \;\; e'' \mid \mathit{If}\ be\ \mathit{Then}\ e'\ \mathit{Else}\ e'' \\
be & ::= & bx \mid T \mid F \mid \mathit{Not}\ be' \mid \mathit{Equal}(e, e') \mid be'\ bop\ be'' \\
op & ::= & + \mid - \mid * \mid div \\
bop & ::= & \mathit{And} \mid \mathit{Or}
\end{array}$$

Figure 1.4: Abstract syntax of extended *Exp*

where e and e' are parse trees from the category *Exp*. Similarly we may form a parse tree in *Exp* from a parse tree from *BExp*, *be*, and two from *Exp*, e and e' by



The second example gives the abstract syntax for a simple imperative programming language, called *While*. The principal syntactic category is *Program*. However, according to the definitions, a program simply consists of a block followed by a period. In turn, a block consists of a declaration followed by ";" which, in turn, is followed by a command. There are five different possible structures for a command, including one of the form

begin B end

where *B* is an arbitrary block. A declaration consists of a sequence of basic declarations, each separated by ";". The basic declarations can have one of two different structures, a variable declaration or a procedure declaration; the latter consists of

1. Syntactic categories

p in *Program*	I in *Id*
B in *Block*	x in *Var*
D in *Dec*	bx in *BVar*
C in *Command*	op in *Op*
e in *Exp*	bop in *BOp*
be in *BExp*	n in *Num*

2. Definitions

$$\begin{array}{rcl}
p & ::= & B. \\
B & ::= & D; C \\
D & ::= & var\ x \mid bvar\, bx \mid procedure\, I : C \mid D'; D'' \\
C & ::= & skip \mid x := e \mid C'; C'' \mid If\ be\ Then\ C'\ Else\ C'' \mid I \\
 & & \mid While\ be\ Do\ C' \mid begin\ B\ end \\
e & ::= & x \mid n \mid e'\ op\ e'' \mid If\ be\ Then\ e'\ Else\ e'' \\
be & ::= & bx \mid T \mid F \mid Not\ be \mid Equal(e, e') \mid be'\ bop\ be'' \\
op & ::= & + \mid - \mid * \mid div \\
bop & ::= & And \mid Or
\end{array}$$

Figure 1.5: Abstract syntax of *While*

the name of the procedure, an identifier, together with the body, which must be a command.

These abstract syntax definitions are a concise and readable manner for conveying a large amount of information concerning the syntax of programming languages. They will be used throughout the text.

1.3 Induction

The main mathematical tool we will use is induction, in a variety of forms. Let us recall the most basic form — mathematical induction. To prove a property, say $P(x)$, of all natural numbers two separate statements must be proved:

1. Prove $P(x)$ is true of 0, i.e. $P(0)$. This is known as the *base case*.

2. On the assumption that $P(k)$ is true, prove $P(k+1)$. This is known as the *inductive case*.

If both can be shown then we can conclude $P(n)$ is true for every natural number n. As a simple example, consider the property $P(x)$ which asserts that the sum of the first n natural numbers is $n(n+1)\ \ div\ 2$,

$$0 + 1 + \ldots\ldots + n = n(n+1)\ \ div\ 2.$$

To show that $P(x)$ is true of all natural numbers we must show:

1. $P(0)$: i.e. $0 = 0(0+1)\ div\, 2$, which is trivially true from the definition of integer division *div*.

2. $P(k+1)$ assuming $P(k)$. $P(k+1)$ is the statement

   $$0 + 1 + \ldots + k + (k+1) = (k+1)(k+2)\ \ div\ 2$$

   which is what we must show. Now

   $$0 + 1 + \ldots + k + (k+1) = (0 + 1 + \ldots + k) + (k+1).$$

   Since we assume $P(k)$ to be true, we have

   $$0 + 1 + \ldots + k = k(k+1)\ \ div\ 2.$$

   Then we have

   $$\begin{array}{rcl}
   0 + 1 + \ldots + k + (k+1) & = & k(k+1)\ \ div\ 2\ +\ (k+1) \\
   & = & k(k+1)\ \ div\ 2\ +\ 2(k+1)\ \ div\ 2 \\
   & = & (k(k+1) + 2(k+1))\ \ div\ 2 \\
   & = & (k+2)(k+1)\ \ div\ 2.
   \end{array}$$

   In other words, we have derived $P(k+1)$.

From 1 and 2 we may now conclude that $0 + \ldots + n = n(n+1)\ \ div\ 2$ for every natural number n. Of course most people know that this is a property of the natural numbers but we have simply used it as a vehicle for explaining mathematical induction.

We will rephrase mathematical induction as a form of induction which is more generally applicable. The set N of natural numbers satisfies the following conditions or rules:

- $0 \in N$
- if $x \in N$ then $x + 1 \in N$.

Many other sets also have these properties. Examples are the set of integers (positive and negative), the set of rational numbers and the set of real numbers. However, these properties have special significance for N; N is the *least* set which has both these properties. This is so obvious that it is difficult to prove. What we mean is that if X is any other set with these properties then $N \subseteq X$. To prove this we use mathematical induction to show that $n \in X$ for every natural number n. More precisely we use mathematical induction to show the property $P(n)$ is true for every natural number n where $P(x)$ is defined by

$$P(x) \text{ if } x \in X.$$

The base case, $0 \in X$, is trivially true because X satisfies the first property . The fact that X satisfies the second property means that the inductive step is also true: $k \in X$ implies $k + 1 \in X$. We can therefore conclude that $n \in X$ for every natural number n, i.e. $N \subseteq X$.

So, from mathematical induction we can show that N is the least set satisfying the above two properties. But the converse is also true: If we *define* N to be the least set which satisfies them, then we can derive mathematical induction. Suppose we know that for a particular property $P(x)$, $P(0)$ is true, and under the assumption that $P(k)$ is true, we can prove $P(k+1)$. Then let X be the set of numbers which satisfies $P(x)$, i.e. $X = \{ k \mid P(k) \ \textit{is true} \}$. X satisfies both the properties above and, since N is the least such set, we have $N \subseteq X$, i.e. $P(n)$ is true for every natural number n. In other words, by defining N as the least set satisfying a number of conditions, we automatically get a form of induction for proving properties of elements of N. In this case the form of induction coincides with mathematical induction. In general we will define many sets, relations, etc. as the least ones satisfying a set of conditions or rules. Each such definition will give us a form of induction, called *Rule induction*, for proving properties of the sets, relations, etc. For this reason these are called *inductive definitions*.

As an example of an inductive definition, consider the set of even natural numbers, EV. It satisfies the conditions or rules:

1. $0 \in EV$
2. $k \in EV$ *implies* $k + 2 \in EV$

and, although many other sets also satisfy them, one can prove (again, using mathematical induction) that EV is the least set satisfying them. However, let us just *define* EV to be the least set which satisfies these two conditions. What form of Rule induction do we get for EV from this definition? Suppose we wish to show that $P(x)$ is true of every element in EV. Let $X = \{ x \mid P(x) \ \textit{is true} \}$. So we wish to show that $EV \subseteq X$. Now EV is the least set satisfying the conditions 1 and 2 above and therefore to show $EV \subseteq X$ it is sufficient to show that X also satisfies them. The two conditions translate to:

- $0 \in X$, i.e. $P(0)$ is true

- $k \in X$ implies $k + 2 \in X$, i.e. $P(k)$ is true implies $P(k + 2)$ is true.

So, if we can show both of these statements we can conclude $EV \subseteq X$, i.e. $P(k)$ is true for every element of EV. This form of induction is called Rule induction for EV since it uses the defining rules or conditions of EV.

As an example of the use of this definition of EV let us show that $2n \in EV$ for every natural number n. Let $P(k)$ be the property

$$2k \in EV.$$

So we want to show that $P(k)$ is true for every natural number k. For this reason we use the instance of Rule induction associated with the inductive definition of N. There are two clauses in the definition of N and therefore it is necessary to prove the corresponding two statements of the property P. We must show:

- $P(0)$ is true, i.e. $2 \times 0 \in EV$
- assuming $P(k)$, i.e. assuming $2k \in EV$, prove $P(k + 1)$, i.e. $2(k + 1) \in EV$.

The first statement is trivial since 2×0 is 0 and, by the definition of EV, $0 \in EV$. The second requirement is also straightforward. We can assume $2k \in EV$. By the defining property of EV we therefore have that $2k + 2 \in EV$, i.e. $2(k + 1) \in EV$. We have now shown that P satisfies the two defining properties of N and therefore by Rule induction it follows that $P(n)$ is true for every natural number n.

This proof only used the fact that the set EV satisfied the two properties, namely $0 \in EV$ and if $x \in EV$ then $x+2 \in EV$, and so the proof would actually work for any set which has these properties. Although EV has the characteristic of being the least set satisfying this property, this fact was not necessary in the above proof. Let us now see an example where it is necessary. We will prove that if $n \in EV$ and $m \in EV$, then $n + m \in EV$. This is a statement about all elements of the set EV as opposed to all natural numbers and, therefore, it is appropriate to use the induction principle for EV, i.e. Rule induction based on the definition of EV, rather than mathematical induction or the Rule induction based on the definition of the natural numbers. Let $P(x)$ be the property

$$\text{for every } m \in EV,\ x + m \in EV.$$

We prove that $P(x)$ is true for every $x \in EV$, thereby showing that $n \in EV$ and $m \in EV$ implies $n + m \in EV$.

ing to the inductive definition of EV, in order to prove $P(x)$ for every $x \in EV$ it is sufficient to prove:

1. $P(0)$, i.e. for every $m \in EV$, $0 + m \in EV$
2. $P(k+2)$, assuming $P(k)$, i.e. under the assumption that $k+m \in EV$ for every m, prove $k + 2 + m \in EV$ for every m.

The first requirement is immediate since $0 + m = m$. The second requirement is also immediate from the defining property of EV. We know from the inductive assumption that $k+m \in EV$. From the definition of EV we therefore have that $k+m+2 \in EV$, i.e. $k+2+m \in EV$.

Note that both these proofs of the induction requirements are essentially trivial. The power comes from the induction principle associated with EV, Rule induction. This is frequently the case. Having set up a statement which one wants to prove as an instance of Rule induction, the proof of the induction requirements are often straightforward. However, it is not always easy to recognise the required instance of induction.

We now introduce a more descriptive notation for inductive definitions. Each rule in such a definition has the form

if **premises** then **conclusions**.

We will write this in the form

$$\frac{\textbf{premises}}{\textbf{conclusions}}$$

Thus the inductive definition of the natural numbers consists of the two rules

$$\frac{}{0 \in N} \qquad \frac{k \in N}{k+1 \in N}$$

Note that in the first rule there is nothing above the line as there is no premis in the condition. The inductive definition for EV is very similar:

$$\frac{}{0 \in EV} \qquad \frac{k \in EV}{k+2 \in EV}$$

We hope that this notation will make the definitions more readable and that the corresponding form of Rule induction will be more apparent.

As another example, we define a relation DIV over the natural numbers, but instead of writing $< x, y > \in DIV$ we simply write $x\ DIV\ y$. The defining rules for DIV are:

$$\frac{}{n\ DIV\ 0} \qquad \frac{n\ DIV\ k}{n\ DIV\ (n+k)}$$

Here it is assumed that n and k range over arbitrary natural numbers. What is the set defined inductively by these conditions, i.e. the least set satisfying these rules? One can check that it is essentially a definition of the predicate "x divides y", i.e.

$$DIV\ = \{ < n, m > \mid \text{for some } k, n * k = m \}. \quad \text{(How?)}.$$

From this inductive definition we get another instance of Rule induction, an inductive proof method for proving properties $P(x, y)$ for every pair (x, y) such that $x\ DIV\ y$. The first rule gives the requirement

$P(n, 0)$

and the second gives the requirement

$P(n, k)$ implies $P(n, n+k)$.

Let us use this inductive method to prove the property

if $x\ DIV\ y$ and $x \in\ EV$ then $y \in EV$.

Here $P(x, y)$ is "$x \in EV$ implies $y \in EV$", and we must derive two statements:

1. $P(n, 0)$, i.e. if $n \in EV$ then $0 \in EV$

2. $P(n, k)$ implies $P(n, n+k)$.

The first is trivially true since $0 \in EV$ by the definition of EV. For the second let us try to show that $P(n, n+k)$ is true, i.e. $n \in EV$ implies $n+k \in EV$. Suppose $n \in EV$. We are working under the assumption that $P(n, k)$ is true and therefore $k \in EV$. We have already shown that $x+y \in EV$ whenever $x \in EV$ and $y \in EV$ and therefore we can conclude $n+k \in EV$.

Therefore $P(x, y)$ is true for every $< x, y >$ in DIV, i.e. $x\ DIV\ y$ implies that if $x \in EV$ then $y \in EV$.

One must be a little careful in giving inductive definitions. One can easily write down definitions which make little or no sense. For example, take the single rule

$$\frac{n \in X}{n+3 \in X}$$

The set inductively defined by this rule is the empty set ϕ ! The set ϕ satisfies this condition for if $n \in \phi$ then also $n+3 \in \phi$. This is vacuously true because no number is in ϕ. Since ϕ satisfies the condition it also clearly is the least set to do so, since it is contained in every other set.

To avoid having the empty set as the set defined inductively by a set of conditions, you must ensure that there is at least one condition which has no premis. This provides the base case.

One can get into worse trouble. Consider the following three rules:

$$\frac{}{0 \in X} \qquad \frac{n \in X}{n+2 \in X} \qquad \frac{n+2 \in X}{n \notin X}$$

This set of rules cannot define *any* set inductively. For example, suppose S is this set. Then by the first rule $0 \in S$. By the second $2 \in S$ and by the third $0 \notin S$. We cannot have a set that both contains 0 and does not contain 0. So S cannot exist. The problem arises because we used negative statements in the defining rules. To ensure that we don't run into inconsistences like this we will always use only positive statements in these inductive definitions. Apart from this restriction we will be fairly informal about their format.

A somewhat more detailed exposition of Rule induction may be found in Chapter 4 of [Win93] and a thorough exposition of the mathematical foundations are given in [P.A83].

1.4 Structural Induction

Many of our inductive definitions will be for sets of objects which are either completely syntactic in nature or have sufficient amount of structure to them that they can be considered to be at least partially syntactic. The resulting Rule induction in these cases is called *structural induction*. We explain this by using as an example lists of natural numbers. To emphasise their structure we use the notation for lists from the programming language *ML*, : [] is the empty list and the infix operator :: represents prefixing by a natural number. So, if n is a natural number and l is a list $n :: l$ is a new list whose first element is n and remainder is the original list l. The set of all natural number lists may be defined inductively using the two rules

$$\frac{}{[] \in \mathit{Lists}(N)} \qquad \frac{l \in \mathit{Lists}(N)}{n :: l \in \mathit{Lists}(N)}$$

The associated inductive proof method is :

to prove the property $P(x)$ true of all natural number lists it is sufficient to:

- prove $P([])$ is true
- assuming $P(l)$ is true, prove $P(n :: l)$ is also true.

This form of induction is called *structural induction* as it uses induction on the list structure of the objects. Intuitively the defining rules for *Lists(N)* stipulate that every list has a certain structure: either it is [] or it has the more complicated structure $n :: l$, where l is a list and n a natural number. [] is a simple constructor for lists which has no component. The list $n :: l$ is formed by applying the constructor $n :: \_$ to the component list l. The inductive proof method proceeds by examining the possible structure of an arbitrary list. If it is [], then $P([])$ must be established; if it is $n :: l$, then $P(n :: l)$ must be established for every n, on the assumption that P is true of the component in the structure, i.e. $P(l)$.

Let us look at an example of the use of this structural induction. For an arbitrary l in *Lists(N)* let $max(l)$ be the largest element in l (or 0 if l is empty), $sum(l)$ the sum of its elements and $len(l)$ its length. We show that for every l in *List(N)*

$$sum\ l \leq max\ l * len\ l.$$

Let $P(x)$ be this statement, i.e. $sum(x) \leq max(x) * len(x)$. We must show both of

1. $P([])$, i.e. $sum([]) \leq max([]) * len([])$,
   which is trivial since $sum\ ([]) = 0$.

2. $P(n :: l)$ assuming $P(l)$ is true.
   Now $sum(n :: l) = n + sum(l)$. Because we may assume $P(l)$ to be true $sum(l) \leq max(l) * len(l)$. So $sum(n :: l) \leq n + max(l) * len(l)$. Now $n \leq max(n :: l)$ and $max(l) \leq max(n :: l)$ and therefore this may be rewritten as

$$\begin{aligned} sum(n :: l) &\leq max(n :: l) + max(n :: l) * len(l) \\ &= max(n :: l) * (1 + len(l)) \\ &= max(n :: l) * len(n :: l) \end{aligned}$$

   since $len(n :: l) = 1 + len(l)$.

In general, structural induction applies to sets defined using a set of constructors. The general form of the inductive method states that to prove $P(x)$ for all elements in such a structurally defined set S, it is sufficient to prove the following:

for each constructor $C(\_, \ldots, \_)$, assuming $P(s_1), \ldots, P(s_k)$ is true, show that $P(C(s_1, ..., s_k))$ is also true.

In the case of *Lists(N)* there are two constructors [], with no components and $n :: \_$ with one component. To be more accurate there is a constructor of the second form for each natural number n. Each constructor gives rise to one clause in the statement of the inductive proof method.

Abstract syntax definitions may be looked upon as inductive definitions and, more particularly structurally defined sets, because they define objects which are purely syntactic in nature. We choose as our last example the set of arithmetic expressions defined by the abstract syntax definition given in Figure 1.3. Another way of stating this is to say that *Exp* is the set defined inductively by the two rules

$$\frac{}{n \in \mathit{Exp}} \qquad \frac{e \in \mathit{Exp}, e' \in \mathit{Exp}}{e\ \ op\ \ e' \in \mathit{Exp}}$$

where it is assumed that n ranges over the numerals *Num* and *op* over *Op*. Here there are two constructors or, more strictly, two classes of constructors. We have the trivial constructor **n** with no components, for each numeral **n** in *Num*, and for each *op* in *Op* the constructor $\_\ \ op\ \ \_$ with two components. The associated inductive proof method, *structural induction* for *Exp*, is the following: to show $P(x)$ for every element of *Exp* it is necessary to:

1. show $P(\mathbf{n})$ is true for every **n** in *Num*

2. assuming $P(e)$ and $P(e')$ is true, show $P(e\ \ op\ \ e')$ is also true for each *op* in *Op*.

To give an example of the use of this structural induction, let *Ocount(e)* be the number of operator symbols in the expression e and *Ncount(e)* the number of numeral symbols. We show that for every e in *Exp*, $Ocount(e) < Ncount(e)$. Using structural induction there are two statements to prove:

1. $OCount(\mathbf{n}) < Ncount(\mathbf{n})$ for every numeral $\mathbf{n}$.
   This is trivial since $OCount(\mathbf{n}) = 0$ and $Ncount(\mathbf{n}) = 1$.

2. $OCount(e \;\; op \;\; e') < Ncount(e \;\; op \;\; e')$ assuming that $OCount(e) < Ncount(e)$ and $OCount(e') < Ncount(e')$. Now

$$\begin{array}{rcl} OCount(e \;\; op \;\; e') & = & 1 + OCount(e) + OCount(e') \\ & < & Ncount(e) + Ncount(e') \\ & & \text{using the inductive hypothesis.} \end{array}$$

1.5 Inductive Relations and Proof Systems

Let us consider the definition of the set or relation *DIV* given in Section 1.3. It consists of two conditions, which we now name:

$$\text{RuleB} \quad \frac{}{n \;\; DIV \;\; 0}$$

$$\text{RuleI} \quad \frac{n \;\; DIV \;\; k}{n \;\; DIV \;\; (n+k)}$$

This defines a relation *DIV*, namely the least relation satisfying these rules. They may also be used as rules to *prove* that a pair of natural numbers are related via *DIV*. For example, suppose we want to know if $3 \;\; DIV \;\; 9$ is true or false. This cannot be because of RuleB since all conclusions to this rule are of the form $k \; DIV 0$ and 9 is different from 0. It might be a consequence of RuleI if it were the case that $3 \;\; DIV \;\; 6$, for in this case we could apply the rule to obtain $3 \;\; DIV \;\; 3+6$, i.e. $3 \;\; DIV \;\; 9$.

This line of reasoning can be summed up by

if $3 \;\; DIV \;\; 6$ then $3 \;\; DIV \;\; 9$ by RuleI.

Now we can examine the statement $3 \;\; DIV \;\; 6$ and see if we can deduce *it* from the rules. Once more it cannot be because of RuleB since $6 \neq 0$. But it could be a result of an application of RuleI to $3 \;\; DIV \;\; 3$. If we were assured of $3 \;\; DIV \;\; 3$, then RuleI would imply $3 \;\; DIV \;\; 6$. So, combining this piece of reasoning with that above, we obtain

1. if $3 \;\; DIV \;\; 3$ then

2. $3 \;\; DIV \;\; 6$ by applying RuleI to 1
and so
3. $3 \;\; DIV \;\; 9$ by applying RuleI to 2.

Once more we can examine the hypothesis $3 \;\; DIV \;\; 3$ and conclude that if we were assured of $3 \;\; DIV \;\; 0$ then another application of RuleI would give the required $3 \;\; DIV \;\; 3$. This would then give the sequence of deductions:

1. if $3 \;\; DIV \;\; 0$ then
2. $3 \;\; DIV \;\; 3$ by applying RuleI to 1, and so
3. $3 \;\; DIV \;\; 6$ by applying RuleI to 2, and so
4. $3 \;\; DIV \;\; 9$ by applying RuleI to 3.

The final hypothesis, $3 \;\; DIV \;\; 0$, is an instance of RuleB which has no premis. So the entire line of reasoning, now based on no hypothesis, can be written as

1. 3 *DIV 0* by RuleB
2. 3 *DIV 3* by applying RuleI to 1
3. 3 *DIV 6* by applying RuleI to 2
4. 3 *DIV 9* by applying RuleI to 3.

This is a *proof* of the statement $3 \;\; DIV \;\; 9$ from the two rules RuleB and RuleI. In general the rules of an inductive definition give rise to a *Proof system*, i.e. a method for generating proofs. A proof in a proof system is a sequence of statements

$$\begin{array}{l} S_1 \\ S_2 \\ \ldots \\ \ldots \\ S_k \end{array}$$

such that each statement is a result of applying one of the rules of the proof system to statements earlier in the sequence. This means that S_1 must be the result of applying a rule which has no premis. It also implies that if the set of rules contains none which has no premis then there are no proofs from these rules. The proof above is said to be a proof of the final statement, S_k which, in turn, is called a *theorem* of the proof system generated by the rules, or is a *consequence* of the corresponding set of rules. Returning to our previous example, we can now say that $3 \;\; DIV \;\; 9$ is a theorem of the proof system determined by the inductive definition of *DIV* or is a consequence of the rules RuleB and RuleI.

Can we prove $4 \;\; DIV \;\; 7$? Well, we cannot apply RuleB to derive it since $0 \neq 7$. We may be able to apply RuleI to derive it if we were assured of $4 \;\; DIV \;\; 3$. So the potential proof would look like

1. if $4\ \ DIV\ \ 3$ then
2. $4\ \ DIV\ \ 7$ by applying RuleI to 1.

How can we go about deriving a proof of $4\ \ DIV\ \ 3$ from these rules? Once more we cannot apply RuleB because $0 \neq 3$. But, also, we cannot apply RuleI since $4\ \ DIV\ \ 3$ cannot be made to match any instance of this rule. Any potential match would have to match n to 4 and there is no natural number k such that $4 + k$ is 3. The result is that we cannot prove $4\ \ DIV\ \ 7$ using these two rules. Because of this we can conclude that $4\ \ DIV\ \ 7$ is not actually true. We can come to this conclusion because there is a close link between the relation determined by an inductive definition and the theorems which can be proved in the corresponding proof system.

Consider an arbitrary inductive definition, i.e. set of rules (which use only positive statements) and let L be the least relation satisfying them. Let P_L be the relation generated by the proof system; that is, assuming L is a binary relation, $< n, m > \in P_L$ if there is a proof of $n\ L\ m$ in the proof system generated by the inductive definition. First note that this relation P_L satisfies the rules since proofs are constructed by applying them. Since L is the least relation satisfying the rules, it follows by Rule induction that $L \subseteq P_L$. To prove the converse, suppose $< n, m > \in P_L$. Then there is a proof of $n\ P_L\ m$ using the rules. One can prove by mathematical induction on the length of this proof that $< n, m > \in X$ for *any* relation X which satisfies the rules, and in particular that $< n, m > \in L$. So $P_L \subseteq L$, i.e. $P_L = L$.

This means that when we use a set of rules to inductively define a relation this relation is completely determined by the statements which can be derived in the corresponding proof system. This is not of great significance when giving inductive definitions of well-known relations such as DIV and EV. But in future chapters we will give inductive definitions of relations which have no characterisation other than these inductive definitions. In these cases the link with the proof system corresponding to the inductive definition is crucial as it enables us to deduce whether or not two objects are related. In fact the only way of understanding these kinds of relations is by investigating the proof systems associated with their inductive definitions.

Questions

Q1 Use mathematical induction on m to show:

- the sum of the squares of the first m natural numbers is $m * (m + 1)(2m + 1)\ \ div\ \ 6$
- $n\ \ DIV\ \ m * n$ for every $m \in N$
- $n\ \ DIV\ \ l$ implies $n\ \ DIV\ \ m * l$ for every $m \in N$.

Q2 Use the inductive definition of EV in Section 1.3 to prove $n \in EV$ implies $n * k \in EV$ for every $k \in N$.

Q3 Use the inductive definitions of DIV and EV in Section 1.3 to prove

- $n\ \ DIV\ \ m, n\ \ DIV\ \ m'$ implies $n\ \ DIV\ \ m + m'$
- $n\ \ DIV\ \ m, m\ \ DIV\ \ k$ implies $n\ \ DIV\ \ k$.

Q4 Let REM be the ternary relation defined by:

$< x, y >\ \ REM\ \ r$ if r is the remainder when y is divided by x, i.e. if $0 \leq r < x$ and $y = kx + r$ for some $k \geq 0$. (Note that x cannot be 0.)

Give an inductive definition of REM.

Q5 Use the inductive definition of Question 4 to prove:

- $< x, y >\ \ REM\ \ r$ implies $r < x$
- $x\ \ DIV\ \ y$ implies $< x, y >\ \ REM\ \ 0$ whenever $x > 0$
- $< x, y >\ \ REM\ \ r$ implies $x\ \ DIV\ \ (y - r)$
- $< x, y >\ \ REM\ \ r, < x, y >\ \ REM\ \ r'$ implies $r = r'$.

Q6 For any two non-zero natural numbers n, m a *common divisor* of n, m is any number k which divides exactly into both of them, i.e. there exists some z, z' such that $n = z * k$ and $m = z' * k$. The number k is the greatest common divisor of n, m written $k = gcd(n, m)$ if it is a common divisor and, for any other common divisor, $k', k' \leq k$. Give an inductive definition of the relation gcd.

Q7 For any relation R over a set X, i.e. $R \subseteq X \times X$ let R^*, its reflexive transitive closure, be defined by $< x, y > \in R^*$ if there exists $n_0, \ldots, n_k$ in N, for $k \geq 0$, such that $x = n_0, y = n_k$ and $< n_i, n_{i+1} > \in R$ for all $0 \leq i \leq k$. Give an inductive definition of R^*.

Let $A \subseteq X$ be such that if $< x, y > \in R$ and $x \in A$ then $y \in A$. Show that this property is also true of R^*, i.e. if $< x, y > \in R^*$ and $x \in A$ then $y \in A$.

Q8 Complete the proof outlined at the end of Section 1.5 that $P_L \subseteq L$.

Chapter 2

Arithmetic Expressions

In this chapter we examine a very simple language and use it to explain what we mean by a semantic theory. The language is for a subset of arithmetic expressions over numerals which are meant to represent natural numbers. The abstract syntax has already been given in Figure 1.3 but for convenience is repeated in Figure 2.1. Here there are three syntactic categories, *Exp*, the principal one, and the two auxiliary ones, *Op* and *Num*. We are not interested in exactly how elements of *Num* are defined but one could imagine a grammar for sequences of digits. However, from our point of view they may be considered as tokens, objects requiring no further analysis. There are four possible operators in the syntactic category *Op* corresponding to four natural operations on numerals, and expressions are formed in the usual way using numerals and these operators.

Intuitively the semantics of this language is well-understood, even if only informally. The meaning of an expression is a numeral, that is every expression corresponds to a value or numeral and we all know how to calculate it. But how do we formally say which numeral corresponds with each expression? That is, how do we explain in a precise manner the correspondence between expressions and numerals ? This is not much of a problem for this simple language but as the language gets more complicated the correspondence will no longer remain clear. In the following sections we use this language, whose semantics is very clear to everyone, as a vehicle to explain different methods for assigning meanings to languages. In later chapters we will apply some of these methods to more complicated languages.

2.1 Concrete Operational Semantics

One method of describing how to calculate the value of expressions would be to write a compiler or interpreter for the language. Then each time you wanted the value of an expression you simply apply the interpreter or run the code which is returned by the compiler. In this section let us examine how we might do this. A compiler has

1. Syntactic categories

   e in *Exp*

   op in *Op*

   n in *Num*

2. Definitions

$$\begin{array}{rcl} op & ::= & + \mid - \mid * \mid div \\ e & ::= & n \mid e \;\; op \;\; e \end{array}$$

Figure 2.1: Abstract Syntax for *Exp*

to work on a machine and therefore before designing the compiler we must decide on an appropriate machine. For the sake of argument let us choose what I will call a *STACK*-machine. This consists of a stack on which values (i.e. numerals) can be stored and an unspecified arithmetic unit for doing simple calculations on these values. To perform these calculations we assume that the machine also contains the usual kind of locations or variables. There are only three operations on the stack:

$$\begin{array}{rcl} Push & : & Num * STACK \longmapsto STACK \\ Pop & : & STACK \longmapsto STACK \\ Top & : & STACK \longmapsto Num. \end{array}$$

The operation *Push* takes a numeral **n** and a stack s and returns a new stack, the result of putting the numeral **n** on top of the stack s. *Pop* takes a stack and pops the top element from it, if it exists. The result is again a stack containing all the elements of the original stack except the topmost one. Finally *Top* simply reads off the top element of the stack, if it exists, without affecting the stack. Executing *Top* or *Pop* on an empty stack causes a run-time error.

A program for this machine consists of a sequence of instructions for manipulating the *STACK* and doing some arithmetic on the values. We will not be very specific about the exact form they take but we use *SProg* to denote the set of sequences of instructions, i.e. the set of programs. Intuitively a program starts on the machine with the empty stack, performs the sequence of instructions and finally halts. The resulting value is then taken to be the element on top of the stack. A compiler for the language of expressions will be represented as a function

$$COMP\colon Exp \longmapsto SProg.$$

It takes an expression *exp* and returns a program *COMP(exp)* for the *STACK*-

machine. The intended value of the expression, its meaning, is then obtained by running this program on the *STACK*-machine. *COMP* is defined by structural induction on expressions and for convenience we use $Apply_S(op)$ to denote the sequence of instructions which:

1. takes the first two elements from the stack
2. sends them to the arithmetic unit where the operation corresponding to the symbol *op* is applied to them and finally the result is replaced on top of the stack.

Assuming that the arithmetic unit knows about all of the arithmetic operators $Apply_S(op)$ can simply be written as:

$x := Top;$

Pop

$y := Top;$

Pop

$z := Ap(op, x, y);$

$Push\ z$

Since there is only one stack in the machine there is no need to write the stack argument for the operations *Push, Pop* and *Top*. We use $Ap(op, x, y)$ to denote the application of the operation corresponding to the symbol *op* to the values in the locations x and y. We are assuming that the arithmetic unit can carry out these operations. We should emphasise that all of these operations are integer operations and for simplicity we assume that they always return an answer. Moreover we are keeping the language extremely simple and therefore only have numerals corresponding to the non-negative integers or natural numbers. So we take the symbol *div* to represent an approximation to integer division, restricted to our collection of numerals. If m is 0 then $n\ div\ m$ returns 0 for every n, and otherwise it returns the largest numeral k such that $n * k$ is less than or equal to n. Similarly $-$ represents an approximation to subtraction over our numerals. So if m is greater than n, $n - m$ returns the numeral **0**. This is discussed more fully in Section 2.4.

The definition of the compiler may now be given by structural induction on expressions:

$$\begin{aligned} COMP\,(n) &= Push(n) \\ COMP(e\ op\ e') &= COMP(e); COMP(e'); Apply_S(op). \end{aligned}$$

How good is this a method for defining the meaning of expressions? It certainly works, at least if you own a *STACK*-machine. The compiler *COMP* assigns to each expression e a sequence of instructions for the stack machine, *COMP(e)*. To find the meaning of e you simply run the sequence of instructions *COMP(e)* on the *STACK*-machine. However, to be effective you must also be able to implement the compiler on a machine. But, much more importantly, most of the detail of the *STACK*-machine and the compiler and how they work is completely immaterial to our understanding of how to evaluate an expression. The essential point is that the value of a numeral **n** is itself and to evaluate $e\ \ op\ \ e'$ you first evaluate e then evaluate e' and finally apply the calculation corresponding to *op* to the two results. This is what happens when the compiled code is run on the *STACK*-machine but it is also a general recipe which applies to a large number of machines; the simplicity of this recipe is lost in the details of how the *STACK*-machine works and the definition of the compiler. Indeed the compiler is simply an implementation of this recipe on the *STACK*-machine. A compiler for a different machine would reimplement this recipe in a form suitable for the new target machine. However, the essence of our intuitive semantics for this language is captured by this recipe.

2.2 Evaluation Semantics

Instead of giving the meaning of an expression in terms of a specific compiler for a specific machine let us instead formally axiomatise our intuitions about how one would in general go about evaluating an expression. We define a relation

$$evalsto\colon Exp \longmapsto Num.$$

For an expression e $evalsto(e)$ gives its intended meaning, the numeral to which it should evaluate in any correct implementation on any machine, including the *STACK*-machine.

The definition of *evalsto* is an inductive definition. Its defining properties are:

Rule 1 for every numeral $\mathbf{n}, \mathbf{n}\ evalsto\ \mathbf{n}$

Rule 2 if $e\ evalsto\ v$ and $e'\ evalsto\ v'$ then $e\ \ op\ \ e'\ evalsto\ Ap(op, v, v')$.

As with the definition of the compiler we are assuming that we know that each of the symbols $+, -, *, div$ stands for the appropriate operation on numerals, and moreover that we know how to perform these operations. So $Ap(op, v, v')$ is the numeral which results from applying the operator corresponding to the operator symbol *op* to the two numerals v, v'. It should be emphasised that $Ap(op, v, v')$ is a numeral and not some instruction in a language.

These two rules are very simple to understand and capture the essence of how we go about calculating the value of expressions without excessive detail. In fact, these rules constitute an inductive definition of a relation *evalsto* from *Exp* to *Num*. In choosing to focus on this relation we have decided that the value of an expression

Rule CR $$\frac{}{\mathbf{n} \Longrightarrow \mathbf{n}}$$

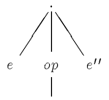
Rule OpR $$\frac{e \Longrightarrow v \qquad e' \Longrightarrow v'}{e \ \ op \ \ e' \Longrightarrow Ap(op, v, v')}$$

Figure 2.2: Evaluation semantics for *Exp*

e is to be a numeral and the relation *evalsto* gives the value we would expect from evaluating an expression. The definition of *evalsto* is quite abstract. It gives some information but very little on how the evaluation takes place or what kind of machine is used; it only gives the expected value.

Instead of using *evalsto* let us introduce the more graphic symbol $\Longrightarrow$ to denote the evaluation relation. Then the two rules constituting the inductive definition of *evalsto* may be rephrased as in Figure 2.2. Here we use the notation for inductive definitions in Chapter 1.

It is our first example of an inductive definition of a relation which has no other formal characterisation. At this point the proof system corresponding to these conditions will come in useful. Let us see how to use it to calculate the value of the expression $\mathbf{3} * \mathbf{4} + \mathbf{8} \ \ div \ \ \mathbf{4} - \mathbf{2}$. The rules are given in terms of the abstract syntax of *Exp* and therefore in order to apply them we need to view this expression not as a sequence of symbols but as a parse tree. As such it is ambiguous so let us use brackets to show its actual structure: $(\mathbf{3}*\mathbf{4})+(\mathbf{8} \ \ div \ \ (\mathbf{4}-\mathbf{2}))$. This indicates that it is parsed as



where e is the expression $\mathbf{3} * \mathbf{4}$ and e' the expression $\mathbf{8} \ \ div \ \ (\mathbf{4} - \mathbf{2})$. In this case the only rule which is applicable is Rule OpR and in order to apply it we need to know

$\mathbf{3} * \mathbf{4} \Longrightarrow ?$

$\mathbf{8} \ \ div \ \ (\mathbf{4} - \mathbf{2}) \Longrightarrow ??.$

Then an application of the Rule OpR would lead to the proof:

1. If $\mathbf{3} * \mathbf{4} \Longrightarrow ?$
2. and $\mathbf{8} \ \ div \ \ (\mathbf{4} - \mathbf{2}) \Longrightarrow ??$
3. then $(\mathbf{3} * \mathbf{4}) + \mathbf{8} \ \ div \ \ (\mathbf{4} - \mathbf{2}) \Longrightarrow Ap(+_{Num}, ?, ??)$ by Rule OpR to 1, 2.

In the last line $+_{Num}$ represents the operation on the numerals corresponding to addition on the natural numbers.
However, for this to be an actual proof, we need to be able to fill in proofs for the two statements

$\mathbf{3} * \mathbf{4} \Longrightarrow ?$

$\mathbf{8} \ \ div \ \ (\mathbf{4} - \mathbf{2}) \Longrightarrow ??.$

Let us consider the first one. This is parsed as $e * e'$ where e is $\mathbf{3}$ and e' is $\mathbf{4}$. So the rule which applies to it is Rule OpR once more. But in order to apply it we need to resolve

$\mathbf{3} \Longrightarrow ?'$

$\mathbf{4} \Longrightarrow ?''.$

Luckily both of these are quite easy as they are direct applications of the first rule Rule CR. So the complete proof for $\mathbf{3} * \mathbf{4} \Longrightarrow ?$ is:

1. $\mathbf{3} \Longrightarrow \mathbf{3}$ by Rule CR
2. $\mathbf{4} \Longrightarrow \mathbf{4}$ by Rule CR
3. $\mathbf{3} * \mathbf{4} \Longrightarrow \mathbf{12}$ by Rule OpR to 1, 2.

When we plug this in to the partial proof for deriving the value of $\mathbf{3}*\mathbf{4}+\mathbf{8} \ \ div \ \ (\mathbf{4}-\mathbf{2})$ we obtain a more complete proof, which is, however, still partial:

1. If $\mathbf{8} \ \ div \ \ (\mathbf{4} - \mathbf{2}) \Longrightarrow ??$
2. $\mathbf{3} \Longrightarrow \mathbf{3}$ by Rule CR
3. $\mathbf{4} \Longrightarrow \mathbf{4}$ by Rule CR
4. $\mathbf{3} * \mathbf{4} \Longrightarrow \mathbf{12}$ by Rule OpR to 2, 3
5. then $\mathbf{3} * \mathbf{4} + \mathbf{8} \ \ div \ \ (\mathbf{4} - \mathbf{2}) \Longrightarrow Ap(+_{Num}, \mathbf{12}, ??)$ by Rule OPR to 1, 4.

To continue trying to deduce the value of $\mathbf{3} * \mathbf{4} + \mathbf{8} \ \ div \ \ (\mathbf{4} - \mathbf{2})$ we must tackle

$\mathbf{8} \ \ div \ \ (\mathbf{4} - \mathbf{2}) \Longrightarrow ??.$

The expression $\mathbf{8} \ \ div \ \ (\mathbf{4} - \mathbf{2})$ is parsed as $e \ \ div \ \ e'$ where e is $\mathbf{8}$ and e' is $\mathbf{4} - \mathbf{2}$ and the only applicable rule is once more Rule OpR. This requires the evaluation of

$\mathbf{8} \Longrightarrow ?'$

$\mathbf{4} - \mathbf{2} \Longrightarrow ??'.$

The first is a straightforward application of the Rule CR whereas the second requires more analysis; in the end, two applications of Rule CR are needed and one of Rule OpR. Putting these together we obtain an evaluation of $\mathbf{8} \ \ div \ \ (\mathbf{4} - \mathbf{2})$:

1. $\mathbf{4} \implies \mathbf{4}$ by Rule CR
2. $\mathbf{2} \implies \mathbf{2}$ by Rule CR
3. $\mathbf{4} - \mathbf{2} \implies \mathbf{2}$ by Rule OpR to 1, 2
4. $\mathbf{8} \implies \mathbf{8}$ by Rule OpR
5. $\mathbf{8}\ \ div\ \ (\mathbf{4} - \mathbf{2}) \implies \mathbf{4}$ by Rule OpR to 3, 4.

Inserting this into the partial derivation for $\mathbf{3} * \mathbf{4} + \mathbf{8}\ \ div\ \ (\mathbf{4} - \mathbf{2})$ we obtain the following complete proof:

1. $\mathbf{4} \implies \mathbf{4}$ by Rule CR
2. $\mathbf{2} \implies \mathbf{2}$ by Rule CR
3. $\mathbf{4} - \mathbf{2} \implies \mathbf{2}$ by Rule OpR to 1, 2
4. $\mathbf{8} \implies \mathbf{8}$ by Rule CR
5. $\mathbf{8}\ div\ (\mathbf{4} - \mathbf{2}) \implies \mathbf{4}$ by Rule OpR to 3, 4
6. $\mathbf{3} \implies \mathbf{3}$ by Rule CR
7. $\mathbf{3} * \mathbf{4} \implies \mathbf{12}$ by Rule OpR to 1, 6
8. $(\mathbf{3} * \mathbf{4}) + \mathbf{8}\ div\ (\mathbf{4} - \mathbf{2}) \implies \mathbf{16}$ by Rule OpR to 5, 7.

So the result is $\mathbf{16}$.

Every expression can be evaluated in the same way and it is easy to see that one could write code in virtually any implementation language for carrying out the evaluation procedure based on these two rules. The important point is that we have captured the essential aspects of the evaluation procedure without any undue bias towards a particular implementation detail; it is at a relatively abstract level. This also helps when we wish to prove properties of the evaluation. For example, let us prove that every expression will evaluate to at most one value. This is not apparent from the definition of $\implies$ but it certainly is a property we would expect of any evaluation mechanism.

Theorem 2.2.1 *For every e in Exp, if $e \implies \mathbf{n}$ and $e \implies \mathbf{n}'$ then $\mathbf{n} = \mathbf{n}'$.*

Proof As pointed out in Section 1.4 *Exp* is defined inductively and therefore one can prove properties of expressions in *Exp* by induction— structural induction. Let $P(x)$ be the property that x has at most a unique value:

If $x \implies \mathbf{m}$ and $x \implies \mathbf{k}$ then $\mathbf{m} = \mathbf{k}$.

We wish to prove $P(e)$ for every $e \in Exp$ and, by structural induction, it is sufficient to prove:

1. $P(\mathbf{n})$ for every numeral $\mathbf{n}$
2. $P(e'\ \ op\ \ e'')$ for every operator symbol op, on the assumption that $P(e')$ and $P(e'')$ are true.

Let us first consider $P(\mathbf{n})$. If $\mathbf{n} \implies \mathbf{m}$, then the only way it can be proved using the rules, Rule CR and Rule OpR, is by an application of Rule CR. This implies that $\mathbf{n} = \mathbf{m}$, since the only possible conclusion to an application of this rule is of the form $\mathbf{n} \implies \mathbf{n}$. By a similar argument $\mathbf{n} = \mathbf{k}$ and, therefore, $\mathbf{m} = \mathbf{k}$.

We now show $P(e'\ \ op\ \ e'')$ on the assumption that $P(e')$ and $P(e'')$ are true. The only way of deriving $e'\ \ op\ \ e'' \implies \mathbf{m}$ is by an application of Rule OpR, and to apply this we need to have already derived $e' \implies \mathbf{m}'$ and $e'' \implies \mathbf{m}''$ for some numerals $\mathbf{m}'$ and $\mathbf{m}''$. In this case $\mathbf{m}$ must be $\mathbf{m}'\ op_{Num}\ \mathbf{m}''$, where op_{Num} is the function on numerals corresponding to the symbol *op*. Applying the same reasoning to $e'\ \ op\ \ e'' \implies \mathbf{k}$, we can see that $\mathbf{k}$ must have the form $\mathbf{k}'\ op_{Num}\ \mathbf{k}''$, where $e' \implies \mathbf{k}'$ and $e'' \implies \mathbf{k}''$. Now we can use the fact that $P(e')$ and $P(e'')$ are true. This means that $\mathbf{m}' = \mathbf{k}'$ and $\mathbf{m}'' = \mathbf{k}''$ from which it follows immediately that $\mathbf{m} = \mathbf{k}$. □

This does not actually state that every expression can be evaluated to a value; it simply says that any resulting value must be unique. However, we would expect every expression to have a value and we can prove that this is indeed a property of our abstract evaluation mechanism $\implies$. We prove another theorem.

Theorem 2.2.2 *For every e in Exp there exists some $\mathbf{n}$ in Num such that $e \implies \mathbf{n}$.*

Proof Once more we use structural induction on *Exp*. This time the required property $P(x)$ is

for some $\mathbf{k} \in Num, x \implies \mathbf{k}$.

To show that $P(e)$ is true for every e in *Exp* we must show:

1. $P(\mathbf{n})$ for every numeral $\mathbf{n}$
2. $P(e'\ \ op\ \ e'')$, on the assumption that both $P(e')$ and $P(e'')$ are true.

The first case is trivial. The required $\mathbf{k}$ is simply $\mathbf{n}$ since we can apply Rule CR to obtain $\mathbf{n} \implies \mathbf{n}$. In the second case we can assume, by induction, that $e' \implies \mathbf{k}'$ and $e'' \implies \mathbf{k}''$ for some numerals $\mathbf{k}', \mathbf{k}''$. Then the required $\mathbf{k}$ is $Ap(op_{Num}, \mathbf{k}', \mathbf{k}'')$ since we can apply Rule OpR to $e' \implies \mathbf{k}'$ and $e'' \implies \mathbf{k}''$ to obtain $e'\ \ op\ \ e'' \implies \mathbf{k}$. □

The inductive definition of $\implies$ is particularly simple because the language *Exp* is simple. Essentially we have one inductive clause for each of the methods for constructing expressions in *Exp*. If we extend the syntactic category *Exp* by adding new methods for constructing terms we must also add corresponding inductive clauses to the definition of $\implies$. We may also wish to extend the language by adding new syntactic categories. In such cases it may be necessary to define how terms in the new syntactic categories are to be evaluated. For example, if we have a syntactic category of boolean expressions it would be natural to introduce a new kind of expression into *Exp* of the form *If be Then e Else e′*, where *be* is a boolean expression. To evaluate this expression we need to be able to evaluate boolean expressions.

In general, to define how terms in a syntactic category, *Cat*, are to be evaluated we must first decide the appropriate set of values, *Val(Cat)*, and then define an evaluation relation from *Cat* to *Val(Cat)*. In our simple example the only syntactic category of interest is *Exp*. Here the set of values, *Val(Exp)*, was taken to be *Num*, the set of numerals. If we extended the language to include boolean expressions, *BExp*, we would have to decide on *Val(BExp)*. We will see more examples of this in later chapters.

In the literature this form of operational semantics, what we call *Evaluation semantics*, is often refered to as *Natural Semantics*. Alternative expositions of it may be found in [NN92, Win93] and non-trivial uses of it may be found on the website of the CROAP project, `http://www.inria.fr/Equipes/CROAP-fra.html`, a major project for developing software tools.

2.3 Computation Semantics

In the previous section we saw how to define a very abstract form of evaluation relation from expression to values, $\Longrightarrow$. For each expression e it defines the resulting value. But it says practically nothing about how the evaluation takes place. For example, the inductive definition tells us that

$$(\mathbf{10} - \mathbf{8}) + (\mathbf{5} \ \ div \ \ \mathbf{2}) * \mathbf{4} \ \Longrightarrow \ \mathbf{10}$$

without telling us anything about how a computation which produces the required value **10** might proceed. In Section 2.1 we have seen one specific and detailed way in which such a computation might be carried out. But this is one of many possible implementations and, from the point of view of an explanation, it is full of extraneous detail. We should be able to explain in more abstract terms what an actual computation which carrries out the evaluation would look like. The key is contained in the proof of this statement above. By examining it we see that, for example, $\mathbf{10} - \mathbf{8}$ must be evaluated, $\mathbf{5} \ \ div \ \ \mathbf{2}$ must be evaluated, etc. We can see that the evaluation could proceed step by step, each step corresponding to part of an overall computation necessary to evaluate the complete expression to its corresponding value. In the case of our particular language each step corresponds to the application of an arithmetic operator.

Instead of inductively defining the overall evaluation relation $\Longrightarrow$ we define a *one-step relation* $\longrightarrow$. A computation consists of a sequence of simple operations each application of which takes one step. The exact nature of these operations will depend on the language in question and sometimes also on the target machine on which the

Rule 1 $$\frac{}{\mathbf{n} \ \ op \ \ \mathbf{n}' \ \longrightarrow \ Ap(op, \mathbf{n}, \mathbf{n}')}$$

Rule 2L $$\frac{e \ \longrightarrow \ e''}{e \ \ op \ \ e' \ \longrightarrow \ e'' \ \ op \ \ e'}$$

Rule 2R $$\frac{e' \ \longrightarrow \ e''}{e \ \ op \ \ e' \ \longrightarrow \ e \ \ op \ \ e''}$$

Figure 2.3: Computation semantics for *Exp*

computation is being performed. For *Exp* a computation proceeds by performing the operations corresponding to the operator symbols occurring in the expression being evaluated, in some particular order, and as the computation proceeds the expression is gradually simplified. So we interpret $e \ \longrightarrow \ e'$ to mean that in one step of the computation e is simplified to e'. This is a very different kind of relation than $\Longrightarrow$. The relation $\longrightarrow$ takes an expression and produces another expression, namely whatever remains to be evaluated. On the other hand, $\Longrightarrow$ takes an expression and produces a numeral, the result of the entire computation. The difference is emphasised by what we call the *type* of a relation. The type of $\longrightarrow$ is given by

$$\longrightarrow \, : Exp \longmapsto Exp$$

whereas that of $\Longrightarrow$ is given by

$$\Longrightarrow \, : Exp \longmapsto Num.$$

In future when we introduce a new inductively defined relation we will always give its type as this is a convenient method of emphasising a simple but important property of relations, namely what kind of arguments they take and return. So

$$R : X \longmapsto Y$$

means that R is a relation between X and Y. Mathematically R is subset of $X \times Y$ and we say that it takes its arguments in X and produces results in Y. An inductive definition of $\longrightarrow$ is given in Figure 2.3. The first rule simply says that a computation may proceed by applying an arithmetic expression to values:

$$\mathbf{n} \ op \ \mathbf{n}' \ \longrightarrow \ Ap(op, \mathbf{n}, \mathbf{n}').$$

The second rule states that we may proceed to compute the value of $e \ \ op \ \ e'$ by trying to compute the value associated with e or with e'. Note that this is much more abstract, or less detailed than the specific computation scheme in Section 2.1;

for example it does not describe which of the arguments should be first evaluated whereas in Section 2.1 the leftmost argument is always evaluated before the second.

Let us consider an example: the expression $(\mathbf{10} - \mathbf{8}) + (\mathbf{5} \ \ div \ \ \mathbf{2}) * \mathbf{4}$. Let us assume that this is parsed as $e + e'$ where e is $(\mathbf{10} - \mathbf{8})$ and e' is $(\mathbf{5} \ \ div \ \ \mathbf{2}) * \mathbf{4}$. So we could apply either Rule 2L or Rule 2R. Suppose we want to apply the latter. Then we would need some expression ? such that

$$(\mathbf{5} \ \ div \ \ \mathbf{2}) * \mathbf{4} \ \longrightarrow ?.$$

We could then apply the rule to obtain

$$(\mathbf{10} - \mathbf{8}) + (\mathbf{5} \ \ div \ \ \mathbf{2}) * \mathbf{4} \ \longrightarrow \ (\mathbf{10} - \mathbf{8}) + ?.$$

So let us look at how to derive

$$(\mathbf{5} \ \ div \ \ \mathbf{2}) * \mathbf{4} \ \longrightarrow ?.$$

In this case there is only one possible rule to apply, Rule 2L, and to apply it we need to know the expression ?? such that

$$\mathbf{5} \ \ div \ \ \mathbf{2} \ \longrightarrow ??.$$

Then we could apply Rule 2L to obtain

$$(\mathbf{5} \ \ div \ \ \mathbf{2}) \ \longrightarrow ?? * \mathbf{4}.$$

The resolution of $\mathbf{5} \ \ div \ \ \mathbf{2} \ \longrightarrow ?$ comes from an application of Rule 1 since both $\mathbf{5}$ and $\mathbf{2}$ are numerals. Therefore an application of the rules to $(\mathbf{5} \ \ div \ \ \mathbf{2}) * \mathbf{4}$ gives the proof:

1. $\mathbf{5} \ \ div \ \ \mathbf{2} \ \longrightarrow \ \mathbf{2}$ by Rule 1
2. $(\mathbf{5} \ \ div \ \ \mathbf{2}) * \mathbf{4} \ \longrightarrow \ \mathbf{2} * \mathbf{4}$ by Rule 2L applied to 1.

This gives us the required subproof with which to finish the derivation from the expression $(\mathbf{10} - \mathbf{8}) + (\mathbf{5} \ \ div \ \ \mathbf{2}) * \mathbf{4}$ and we obtain:

1. $\mathbf{5} \ \ div \ \ \mathbf{2} \ \longrightarrow \ \mathbf{2}$ Rule 1
2. $(\mathbf{5} \ \ div \ \ \mathbf{2}) * \mathbf{4} \ \longrightarrow \ \mathbf{2} * \mathbf{4}$ Rule 2L applied to 1
3. $(\mathbf{10} - \mathbf{8}) + (\mathbf{5} \ \ div \ \ \mathbf{2}) * \mathbf{4} \ \longrightarrow \ (\mathbf{10} - \mathbf{8}) + (\mathbf{2} * \mathbf{4})$ Rule 2R applied to 2.

So using these rules we have shown that

$$(\mathbf{10} - \mathbf{8}) + (\mathbf{5} \ \ div \ \ \mathbf{2}) * \mathbf{4} \ \longrightarrow \ (\mathbf{10} - \mathbf{8}) + (\mathbf{2} * \mathbf{4}).$$

That is, in one step of the computation the term $(\mathbf{10} - \mathbf{8}) + (\mathbf{5} \ \ div \ \ \mathbf{2} * \mathbf{4})$ reduces to the term $(\mathbf{10} - \mathbf{8}) + (\mathbf{2} * \mathbf{4})$. A further step in the computation could be

$$(\mathbf{10} - \mathbf{8}) + (\mathbf{2} * \mathbf{4}) \ \longrightarrow \ (\mathbf{10} - \mathbf{8}) + \mathbf{8}.$$

The only possible further step is

$$(\mathbf{10} - \mathbf{8}) + \mathbf{8} \ \longrightarrow \ \mathbf{2} + \mathbf{8}$$

and the final step is

$$\mathbf{2} + \mathbf{8} \ \longrightarrow \ \mathbf{10}.$$

Each of these individual steps can be derived from the rules in Figure 2.3. So the overall computation is

$$(\mathbf{10} - \mathbf{8}) + (\mathbf{5} \ \ div \ \ \mathbf{2}) * \mathbf{4} \ \longrightarrow \ (\mathbf{10} - \mathbf{8}) + (\mathbf{2} * \mathbf{4}) \ \longrightarrow \ (\mathbf{10} - \mathbf{8}) + \mathbf{8} \ \longrightarrow \ \mathbf{2} + \mathbf{8} \ \longrightarrow \ \mathbf{10}.$$

In four steps the expression $(\mathbf{10} - \mathbf{8}) + (\mathbf{5} \ \ div \ \ \mathbf{2}) * \mathbf{4}$ is reduced to the value $\mathbf{10}$. Of course, there are many other possible computations. For example, we can prove

$$(\mathbf{10} - \mathbf{8}) + (\mathbf{5} \ \ div \ \ \mathbf{2}) * \mathbf{4} \ \longrightarrow \ \mathbf{2} + (\mathbf{5} \ \ div \ \ \mathbf{2}) * \mathbf{4}$$

and

$$\mathbf{2} + (\mathbf{5} \ \ div \ \ \mathbf{2}) * \mathbf{4} \ \longrightarrow \ \mathbf{2} + (\mathbf{2} * \mathbf{4}).$$

Continuing with this term we get the overall computation

$$(\mathbf{10} - \mathbf{8}) + (\mathbf{5} \ div \ \mathbf{2}) * \mathbf{4} \ \longrightarrow \ \mathbf{2} + (\mathbf{5} \ div \ \mathbf{2}) * \mathbf{4} \ \longrightarrow \ \mathbf{2} + (\mathbf{2} * \mathbf{4}) \ \longrightarrow \ \mathbf{2} + \mathbf{8} \ \longrightarrow \ \mathbf{10}.$$

This is a different computation from $(\mathbf{10} - \mathbf{8}) + (\mathbf{5} \ div \ \mathbf{2}) * \mathbf{4}$ which leads to the same result $\mathbf{10}$. Although there are many possible computations, all lead to this same result. Let us introduce some notation for these computations consisting of a sequence of individual steps as allowed by $\longrightarrow$. We use $\longrightarrow^*$ to denote the reflexive transitive closure of $\longrightarrow$, i.e. $e \ \longrightarrow^* \ e'$ if for some $k \geq 0$ there exists expressions $e_0, e_1, \ldots, e_k$ such that

$$e = e_0 \ \longrightarrow \ e_1, \ldots, e_{k-1} \ \longrightarrow \ e_k = e'.$$

Note that for any numeral $\mathbf{n}, \mathbf{n} \ \longrightarrow^* \ \mathbf{n}$ although $\mathbf{n} \ \longrightarrow \ \mathbf{n}$ is not true. If $\longrightarrow$ represents one computation step carried out on some hypothetical machine, $\longrightarrow^*$ represents the result of carrying out an arbitrary number of steps (possibly zero). The normal application would be to run the machine to completion, i.e. until no further computation steps are possible. An expression e is called *canonical* if it gives rise to no computation, i.e. if $e \ \longrightarrow \ e'$ for no expression e'. Then complete computations end in canonical terms. Luckily in our language canonical expressions are exactly the numerals.

Theorem 2.3.1 *The expression e is canonical if and only if e is in Num.*

Proof Here there are two statements to prove:

1. every numeral $\mathbf{n}$ is canonical

2. if the expression e is canonical then it is a numeral.

The first is straightforward. If we examine the rules defining $\longrightarrow$ we see that each of them applies only to expressions of the form $e \;\; op \;\; e'$. In particular, they cannot be applied to expressions of the form $\mathbf{n}$. So for no e' does $\mathbf{n} \longrightarrow e'$, i.e. $\mathbf{n}$ is canonical.

The second statement is only marginally more complicated. We prove the contrapositive statement, namely that if e is not a numeral it is not canonical, by structural induction on e. If it is not a numeral it must have the form $e' \;\; op \;\; e''$ for some expressions e', e''. If either of these, say e' is not a numeral then it is not canonical, i.e. there is an expression f' such that $e' \longrightarrow f'$. Then by Rule 2L we have $e \longrightarrow f' \;\; op \;\; e''$ and so by structural induction e is not canonical. Otherwise both e' and e'' are numerals, say $\mathbf{n}', \mathbf{n}''$ respectively and so we can apply Rule 1 to obtain $e \longrightarrow Ap(op, \mathbf{n}', \mathbf{n}'')$ which again means that e is not canonical. □

This means that if we run the hypothetical machine, of which $\longrightarrow$ represents one computation step, to completion on any expression we obtain a numeral. We can take this result as the value or meaning of the expression. This semantics may be represented by a new relation

$$\rightsquigarrow \; : Exp \longmapsto Num$$

defined by

$$e \rightsquigarrow \mathbf{n} \text{ if } e \longrightarrow^* \mathbf{n}.$$

As we have seen above

$$(\mathbf{10} - \mathbf{8}) + (\mathbf{5} \;\; div \;\; \mathbf{2}) * \mathbf{4} \rightsquigarrow \mathbf{10}.$$

This kind of operational semantics is much more detailed than that of the previous section. It gives some detail of the kinds of computations which one would expect of an implementation although it still gives very little information about how they are to be effected. They could be done on a *STACK*-machine or equally well on a wide variety of other machines.

For our simple language *Exp* we now have two different semantic theories, one represented by $\Longrightarrow$ and the other by $\longrightarrow$ and the derived $\rightsquigarrow$. Luckily they both agree. We can show:

Theorem 2.3.2 *For every $e \in Exp, e \Longrightarrow \mathbf{n}$ if and only if $e \rightsquigarrow \mathbf{n}$.*

Proof Once more there are two statements to prove:

1. $e \Longrightarrow \mathbf{n}$ implies $e \rightsquigarrow \mathbf{n}$

2. the converse $e \rightsquigarrow \mathbf{n}$ implies $e \Longrightarrow \mathbf{n}$.

We prove the first one by structural induction on terms. There are two cases to consider:

- e is $\mathbf{n}$. As we have already noted, for any numeral $\mathbf{n}, \mathbf{n} \rightsquigarrow \mathbf{n}$.

- e is $e' \;\; op \;\; e''$. Then for some $\mathbf{n}', \mathbf{n}''$ we must have $e' \Longrightarrow \mathbf{n}', e'' \Longrightarrow \mathbf{n}''$ and $\mathbf{n} = Ap(op, \mathbf{n}', \mathbf{n}'')$ because the only way to derive $e \;\; op \;\; e' \Longrightarrow \mathbf{n}$ is to use Rule OpR. By induction we may assume $e' \rightsquigarrow \mathbf{n}'$ and $e'' \rightsquigarrow \mathbf{n}''$. Then $e' \;\; op \;\; e'' \Longrightarrow^* \mathbf{n}' \;\; op \;\; e''$ by repeated application of Rule 2L. Also by repeated application of Rule 2R we obtain $\mathbf{n}' \;\; op \;\; e'' \Longrightarrow^* \mathbf{n}' \;\; op \;\; \mathbf{n}''$. Putting these two sequences of derivations together we get $e' \;\; op \;\; e'' \Longrightarrow^* \mathbf{n}' \;\; op \;\; \mathbf{n}''$. We can now apply Rule 1 to obtain $\mathbf{n}' \;\; op \;\; \mathbf{n}'' \longrightarrow Ap(op, \mathbf{n}', \mathbf{n}'')$, i.e. $\mathbf{n}' \;\; op \;\; \mathbf{n}'' \longrightarrow \mathbf{n}$. By applying this reduction to the end of the sequence of reductions in $e' \;\; op \;\; e'' \Longrightarrow^* \mathbf{n}' \;\; op \;\; \mathbf{n}''$ we obtain $e' \;\; op \;\; e'' \Longrightarrow^* \mathbf{n}$, i.e. $e' \;\; op \;\; e'' \rightsquigarrow \mathbf{n}$.

The converse is less straightforward. We show that if $e \longrightarrow e'$ and $e' \Longrightarrow \mathbf{n}$ then $e \Longrightarrow \mathbf{n}$. This is sufficient to establish $e \rightsquigarrow \mathbf{n}$ implies $e \Longrightarrow \mathbf{n}$. (Why ?) We prove this result by structural induction on e. Again there are two possibilities: e is $\mathbf{n}$ or e has the form $f \;\; op \;\; g$. The first cannot be the case since if $e \longrightarrow e'$ then e is not canonical and therefore cannot be a numeral. So e must be $f \;\; op \;\; g$. What form can the derivation $e \longrightarrow e'$ take? There are three possibilities:

1. e' is $f' \;\; op \;\; g$ where Rule 2L was applied to $f \longrightarrow f'$

2. e' is $f \;\; op \;\; g'$ where Rule 2R was applied to $g \longrightarrow g'$

3. f is the numeral $\mathbf{n}'$, g is the numeral $\mathbf{n}''$ and Rule 1 was applied so that e' is $Ap(op, \mathbf{n}', \mathbf{n}'')$.

Consider the first case. Here e' is $f' \;\; op \;\; g$ and assuming $f' \;\; op \;\; g \Longrightarrow \mathbf{n}$ we must show $f \;\; op \;\; g \Longrightarrow \mathbf{n}$. If $f' \;\; op \;\; g \Longrightarrow \mathbf{n}$ then there must be numerals $\mathbf{n}'$ and $\mathbf{n}''$ such that $f' \Longrightarrow \mathbf{n}'$ and $g \Longrightarrow \mathbf{n}''$. We now have that $f \longrightarrow f'$ and $f' \Longrightarrow \mathbf{n}''$ and so by structural induction we may assume $f \Longrightarrow \mathbf{n}'$. By applying Rule OpR we then have $e \Longrightarrow \mathbf{n}$.

The second case is symmetrical with f and g replacing each other. So consider the third and final possibility. We assume $e' \Longrightarrow \mathbf{n}$, which means that $\mathbf{n}$ is $Ap(op, \mathbf{n}', \mathbf{n}'')$ and we must show that $e \Longrightarrow \mathbf{n}$. But by Rule CR, $f \Longrightarrow \mathbf{n}'$ and $g \Longrightarrow \mathbf{n}''$ and so applying Rule OpR we obtain the required $e \; (= f \;\; op \;\; g) \Longrightarrow Ap(\;\; op \;\; , n', n'') \; (= \mathbf{n})$. □

This theorem states that for the language *Exp* both the Evaluation semantics and the Computation semantics agree. We could go further and also show that the Concrete semantics given in Section 2.1 is also in agreement. This would amount to showing that $e \implies \mathbf{n}$ if and only if whenever we compile the expression e to obtain the program *Comp(e)* for the *STACK*-machine when *Comp(e)* is executed it will always terminate with the value **n** on top of the stack. However, this is not particularly straightforward and therefore the topic will not be pursued.

In the literature this form of operational semantics is often called *structural operational semantics* because the inference rules are usually given relative to the stucture of the terms or programs of the language. The primary reference is [Plo81] although again alternative, and more detailed, expositions may be found in [NN92, Win93].

2.4 Denotational Semantics

When doing calculations, either by hand or on a computer, we usually have in our minds some mental picture or model of the objects we are manipulating. For example, when calculating $2 + 4$ we are certainly not interested in the actual symbols 2 and 4, which are nothing more than arrangements of very small black dots on white paper, or electrical charges stored in some device. The symbol 2 is simply a physical representation of an abstract entity, the natural number whose representation in English is "two", in the language *Exp* is the numeral **2** and which we have been representing in ordinary text as 2. Where does this entity exist, can we ever see it, manipulate it directly or squeeze it? These are all philosophical problems which we would like to avoid. People have argued for quite a long time about whether numbers even exist. But for our purposes we can simply say that most cultures, including ours, have built up over centuries a conceptual model of the natural numbers. This consists of a set of objects referred to in English by "one", "two", "three", ..., in French by "un", "deux", "trois", ... and in mathematical notation by 1, 2, 3, ... Whether or not they exist in any real sense is immaterial. We are all very familiar with the natural numbers and how to reason about them; we know how to apply functions such as addition, subtraction, multiplication, etc., to natural numbers to obtain new natural numbers.

This discussion brings us to our final view of the semantics of our simple language *Exp*: expressions in this language are simply representing these abstract objects called natural numbers. This is certainly true of the subset of *Exp* consisting of the numerals, *Num*. *Num* is the infinite set of symbols **0**, **1**, **2**, etc., each of which represents a natural number. Let us use N to represent this set of abstract objects, the natural numbers. Then there is a natural mapping or interpretation from the set of symbols, *Num*, to the set of corresponding abstract objects N. It maps the symbol **0** to the number 0, the symbol **1** to the number 1 and so on. This mapping is so natural that in everyday language we tend to ignore it or we may not even realise it is there. But for our purposes it is essential to bring it out into the cold light of day and examine it. We will use I to refer to this mapping:

$$I\colon Num \longmapsto N.$$

This function maps the numeral **n** to the natural number n, i.e. n is the natural meaning of the symbol **n**. We will sometimes use the notation $\mathbf{n}_I$ to denote the result of applying the mapping I to the numeral **n**. Of course $\mathbf{n}_I$ is simply n but the notation serves to emphasise the role of the mapping I.

Just as there is a natural meaning of the symbols $\mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots$, there is also a natural meaning for the symbols $+, *, -,$ *div*. In themselves these, again, are nothing more than black strokes on white paper, but you and I know that when we use the symbol $+$ we are using it to represent an abstract object, namely the mathematical function called "addition on the natural numbers". It might also be used to represent other kinds of addition, such as "addition on the real numbers" but in the present context it should be obvious that it represents "addition on the natural numbers".

What is a function? There are many possible answers but we can take a function f from a set A to a set B to be a collection of ordered pairs, i.e. a subset of $A \times B$, which satisfies the conditions:

- **totality:** for every $a \in A$ there is some $b \in B$ such that $(a, b) \in f$
- **uniqueness:** for every $a \in A$ there is at most one $b \in B$ such that $(a, b) \in f$.

For example, addition is a function from $\langle N, N\rangle$ to N and consists of the ordered pairs

$$((0,0),0), ((0,1),1), ((1,1),2), ((1,0),1), ((0,2),2), \ldots.$$

Again, let us use I to refer to this natural association between the familiar symbols $+, -, *,$ *div*, and the abstract functions over the natural numbers they represent. We use the symbol op_I to represent the function over the natural numbers which I associates to the symbol *op*. So $+_I$ is the actual function of addition over the natural numbers, $*_I$ is the function multiplication and $-_I$ is the function subtraction, again over the natural numbers. It is an approximation to the usual subtraction function in that $n -_I m$ is the natural number 0 whenever m is greater than or equal to n. Finally, div_I is the somewhat less well-known function approximating integer division discussed before; $n\ div_I\ m$ is the largest natural number k such that $m *_I k$ is less than or equal to n; if m is 0 by convention this is taken to be 0.

What have we done so far? It is tempting to say nothing because all we have done is to elucidate the standard, intuitive and natural associations of all of the symbols used in the definition of the language *Exp*. More formally, we have provided a mathematical interpretation for the language. This consists of:

1. a space of meanings; in this case the set of abstract entities, N, called natural numbers

2. a meaning in this space for every constant in the language; in this case the constants are the numerals *Num* and we associate with the numeral $\mathbf{n}$ the corresponding natural number n; that is for every constant $\mathbf{n}$, $\mathbf{n}_I$ is the natural number n

3. an association between the operator symbols op and actual functions over the space of meanings, op_I: here op is a symbol whereas op_I is an abstract object, namely a function. In this case we interpret the function symbols $+, *, -, div$, as the natural corresponding functions addition, multiplication, subtraction and division over N.

This interpretation of the symbols used in *Exp* may be extended to all terms in the language. For example, to interpret the term $\mathbf{1}+\mathbf{2}$ we first interpret the two argument symbols $\mathbf{1}, \mathbf{2}$ to the natural numbers 1 and 2, we then interpret the function symbol $+$ to the addition function $+_I$ over the natural numbers, apply it to the numbers $1, 2$ and obtain the number 3. More generally, to interpret the expression $e \;\; op \;\; e'$:

1. first interpret the expressions e, e' to natural numbers n, n', say,

2. then interpret the function symbol op to some function f over the natural numbers,

3. the meaning of the expression $e \;\; op \;\; e'$ is then the natural number $f(n, n')$.

Let us introduce some notation for these concepts. Given the interpretation I we obtain a meaning function from *Exp* to N. We use $[\![\ldots]\!]_I$ to denote this function; so for every expression e, $[\![e]\!]_I$ is a natural number, the meaning of e. The function $[\![\ldots]\!]_I$ is defined by structural induction:

$$
\begin{array}{rcl}
[\![\mathbf{n}]\!]_I & = & n_I \\
[\![e \; op \; e']\!]_I & = & op_I([\![e]\!]_I, [\![e']\!]_I).
\end{array}
$$

This assigns to every expression a meaning:

Theorem 2.4.1 *For every expression e there exists some natural number k such that $[\![e]\!]_I = k$.*

Proof The proof is by structural induction on terms. We must prove two statements:

1. for every numeral $\mathbf{n}$ there exists a natural number k such that $[\![\mathbf{n}]\!]_I = k$

2. assuming that there are natural numbers k' and k'' such that $[\![e']\!]_I = k'$ and $[\![e'']\!]_I = k''$ then there is a natural number k such that $[\![e' \;\; op \;\; e'']\!]_I = k$.

If we show these two statements then we will have proved the theorem. The first is immediate as the required k is simply n, i.e. $\mathbf{n}_I$. The second is also straightforward. The required k is simply $k' \;\; op_I \;\; k''$. Remember op_I is a total function which takes two natural numbers and returns a natural number. □

We have now completed our description of the denotational semantics of *Exp*. It is called a *denotational* semantics because it views the language as a set of expressions for denoting abstract objects, in this case the natural numbers. In general, to give a denotational semantics to a language we must design an interpretation for it. This consists of:

1. for each syntactic category Cat a corresponding set of abstract objects Cat_I

2. in order to interpret expressions we must also associate with each method of constructing an expression in the syntactic category Cat an appropriate abstract function over the set of corresponding abstract objects Cat_I.

Using these two associations we can then define a meaning function $[\![\ldots]\!]_I : Cat \longmapsto Cat_I$ for each syntactic category in the language. The function $[\![\ldots]\!]_I$ is defined, as above, by structural induction over terms.

How do we decide on an appropriate set of abstract meanings? In the case of *Exp* it is quite straightforward as we have a natural understanding of these expressions as denoting natural numbers. With more general languages it is often quite difficult. Programs will often denote functions of some kind, from input values to output values, but as the language gets more complicated it is often far from clear what set of functions should be taken as the space of meanings. Often in these cases we have a better intuition about the operational semantics than about a "correct" denotational semantics.

We can also try to check that both semantic views of the language, the operational and denotational, are in harmony with each other. Let us examine this question for the language *Exp*. The operational semantics is entirely concerned with symbols. It associates with each expression e a numeral $\mathbf{n}$, namely the unique numeral $\mathbf{n}$ such that $e \Longrightarrow \mathbf{n}$. However, this indirectly also associates with each expression e a natural number, namely the unique natural number $\mathbf{n}_I$ such that $e \Longrightarrow \mathbf{n}$. That is, to find the natural number associated with an expression e we first evaluate it to a numeral by finding the numeral $\mathbf{n}$ such that $e \Longrightarrow \mathbf{n}$ and then interpret this numeral $\mathbf{n}$ as a natural number n, i.e. $\mathbf{n}_I$. This natural number coincides with that obtained from the denotational semantics:

Theorem 2.4.2 *For every e in Exp, $e \Longrightarrow \mathbf{n}$ if and only if $[\![e]\!]_I = n$.*

Proof Here there are two statements to prove: if $e \Longrightarrow \mathbf{n}$ then $[\![e]\!]_I = n$ and conversely if $[\![e]\!]_I = n$ then $e \Longrightarrow \mathbf{n}$. Let us consider the second. We suppose that

$[\![e]\!]_I = n$ and prove by structural induction on terms that $e \Longrightarrow \mathbf{n}$. There are two cases to consider:

1. When e is some numeral $\mathbf{k}$.
   In this case $[\![e]\!]_I$ is k and by the first defining condition of $\Longrightarrow$, Rule CR, we can conclude $\mathbf{k} \Longrightarrow \mathbf{k}$.

2. When e has the form $e'\ \ op\ \ e''$.
   In this case we may assume by structural induction that $e' \Longrightarrow \mathbf{k}'$ where $[\![e']\!]_I = k'$ and $e'' \Longrightarrow \mathbf{k}''$ where $[\![e'']\!]_I = k''$. By the definition of $[\![\ldots]\!]_I$, n is $k'\ \ op_I\ \ k''$. Also by applying Rule OpR we obtain $e \Longrightarrow Ap(op, \mathbf{k}', \mathbf{k}'')$. Now we are assuming that the *Apply* mechanism works correctly; when it is applied to a symbol and two numerals it returns the proper numeral, i.e. when applied to op it acts on numerals in the same way as the corresponding function op_I acts on natural numbers. This means that $k'\ \ op_I\ \ k'' = Ap(op, \mathbf{k}', \mathbf{k}'')_I$.

The converse statement, $e \Longrightarrow \mathbf{n}$ implies $[\![e]\!]_I = n$ has a similar proof by structural induction on terms and it also relies on the fact that the *Apply* mechanism correctly models the natural functions associated with the operator symbols. The proof is left to the reader. □

So both approaches to the semantics of *Exp* result in essentially the same meaning for expressions. If this turned out to be false what conclusions could we draw? In the case of *Exp* we would have to conclude that the operational semantics is incorrect. This is because we have a firm understanding of the language *Exp* in terms of the natural numbers and mathematical functions over them. We would have to change the operational rules to reflect that understanding. With more complicated languages the answer is no longer clear-cut. As mentioned above, for programming languages we often have a clearer intuition of the operational nature of programs and the problem is to find an abstract space of meanings for programs which reflects this understanding.

This completes our exposition of the different approaches to the semantics of programming languages. The language we used, *Exp*, is extremely simple and its meaning is well known to everybody. We have used such a simple language so as to concentrate on explaining the methods used to give semantics to languages in general. At one extreme we have the first operational approach, called Concrete operational semantics, where the language is compiled or interpreted into some simpler machine language. Often the machine in question is a hypothetical or abstract machine with a very simple architecture and a limited number of simple instructions. This kind of semantic description is often of great value to an implementer as it contains lots of detail and suggestions about implementation issues. The second approach, Evaluation semantics, is of less value to an implementer as it takes a higher-level view of the evaluation of expressions. This level of description in general leads to a better

understanding of the constructs of the language without undue reference to external considerations. The approach of Computation semantics lies somewhere between the first two approaches. It avoids implementation details and discussion of machine instructions but it describes the individual computation steps necessary to evaluate an expression and the correct temporal ordering between them. The more abstract approach to semantics is taken to the extreme in denotational semantics. Here we view the language simply as a method for expressing objects in an abstract model. In particular, all reference to ideas of computation is omitted. This level of description is most appropriate for analysing programs or more generally investigating a range of properties of programs. For example, we can easily argue that the two expressions $(\mathbf{7}+\mathbf{10})*\mathbf{2}$ and $(\mathbf{9}*\mathbf{4})-\mathbf{2}$ evaluate to the same value. Moreover, the argument does not involve any reasoning about possible computations: $[\![(\mathbf{9}*\mathbf{4})-\mathbf{2}]\!]_I = [\![(\mathbf{7}+\mathbf{10})*\mathbf{2}]\!]_I$ follows by well-known properties of the mathematical functions addition and subtraction. In general when using denotational semantics to reason about programs the arguments are usually concerned with abstract mathematical objects. On the other hand, if an operational semantics is used, these arguments involve reasoning about properties of computations. These are usually much more difficult, particularly as the language concerned gets more complex. However, the disadvantage of the denotational approach is that it gives very little help to anybody who wishes to implement the language.

There are a resonable number of textbooks which deal with denotational semantics. A good place to start is with [NN92, Win93] and more advanced material may be found in [Sch88, Gun92].

The remainder of this book will concentrate on operational semantics. We give a series of examples of semantic descriptions of simple programming languages mainly using the approach of Evaluation semantics and Computation semantics. The principal aim is to explain these approaches more fully through examples and hopefully to convince the reader that they are not only useful but also widely applicable. The text is mainly descriptive. For each example language we first give its abstract syntax and outline informally, in English, the intended semantics. This is then followed by a formal semantics using either Evaluation or Computation semantics. Indeed we will see that for some languages the distinction between these two levels of descriptions is not very clear-cut.

We do not put these formal descriptions to any great use; they serve simply as examples of formal and precise descriptions of the semantics of programming languages which could be used by designers, implementers or programmers. However, to indicate how these formal descriptions might be used we will usually use them to prove one or two simple properties of the programming language in question. These properties include *well-definedness*, i.e. every program or expression returns a result or *determinacy*, i.e. every program returns at most one result. Since in each case the definition of the formal semantics is in terms of inductive definitions, all these proofs use induction of some form or another.

Questions

Q1 What is the program for the *STACK*-machine which results from compiling the expression $(\mathbf{7}+\mathbf{9})*(\mathbf{3}-\mathbf{5})$?

Q2 Design an interpreter for the language *Exp* to run on a *STACK*-machine. Here the machine has two components S,C where S is a stack for holding values and C is the control part for holding the expression to be evaluated, or, more generally, a list of expressions. To evaluate an expression you start the machine in the configuration $< \epsilon, e >$ where ϵ is the empty stack. It should finish in the configuration $< \mathbf{n}, \epsilon >$, where $\mathbf{n}$ is the value of the expression e. (Here we are using ϵ to represent both the empty stack and the empty list) One rule for the functioning of the machine is

$$< s, e \;\; op \;\; e' > \Longrightarrow < s, e.e'.op >$$

where . denotes concatenation. What are the other two rules?

Q3 Write $e \leadsto_I \mathbf{n}$ if whenever the machine is started in the configuration $< \epsilon, e >$ it terminates in the configuration $< \mathbf{n}, \epsilon >$. Prove that $e \implies \mathbf{n}$ implies $e \leadsto_I \mathbf{n}$.

Q4 Prove the converse to Question 3, $e \leadsto_I \mathbf{n}$ implies $e \implies \mathbf{n}$.

Q5 In the Computation semantics for *Exp* the evaluation of $e \;\, op \;\, e'$ can proceed by evaluating e or e' or even interleaving their evaluations. In most compilers expressions are evaluated in a left-to-right manner only. Define a one-step relation $\longrightarrow_{LR}$ for *Exp* which implements this strategy. As an example, the following should not be allowed:

$$(\mathbf{1}+\mathbf{2})*(\mathbf{3}+\mathbf{4}) \longrightarrow_{LR} (\mathbf{1}+\mathbf{2})*\mathbf{7}$$

Q6 Show that $\longrightarrow_{LR}$ is deterministic, i.e. if $e \longrightarrow_{LR} e'$ and $e \longrightarrow_{LR} e''$, then e' and e'' coincide.

Q7 Prove $e \implies \mathbf{n}$ if and only if $e \leadsto_{LR} \mathbf{n}$ where $\leadsto_{LR}$ is defined in the same way as $\leadsto$ but using $\longrightarrow_{LR}$ in place of $\longrightarrow$.

Q8 Suppose that the application of some of the operators can give rise to errors. In other words, $Ap(op, v, v')$ can either return a numeral or a new constant *error*. This might happen, for example, if division by zero is undefined or subtraction of a number by a larger number is not allowed. Redesign both the Evaluation semantics and the Computation semantics to take this into account.

Q9 Prove Theorem 2.3.2 for the Evaluation and Computation semantics of Question 8.

Chapter 3

A Simple Functional Language

In this chapter we develop an Evaluation semantics for a simple functional language. We take as a starting point the language for arithmetic expressions from the previous chapter. We gradually add new features until, at the end of this chapter, we will have defined the entire language. Each time we introduce a new feature we will extend the abstract syntax and add appropriate rules to the inductive definition of the evaluation relation.

3.1 Variables

We first add variables to the language. The new abstract syntax is given in Figure 3.1. It is identical to that for *Exp* except for one new syntactic category of variables, *Var*. As with numerals we are not interested in their structure; they are considered as tokens and consequently we do not include a definition of their structure. We call the resulting language *Exp2*.

To evaluate an expression with variables we must know what values the variables stand for. For example $(x*y)-x$ evaluates to **0** when x is **0** and y is **6**, but to **1** when x is **1** and y is **2**. In an implementation (or a Concrete operational semantics) we would have to say precisely how this association of values to variables is implemented. At our more abstract level it is sufficient to say that such an association or *environment* is simply a function from the set of variables *Var* to the set of values *Num*. We use Greek letters such as ρ to denote these functions and write ρ: $Var \longmapsto Num$ to emphasise that it is a function from *Var* to *Num*; also $\rho(x)$ denotes the numeral associated with the variable x. Finally, we let *ENV* denote the set of all these associations or environments. An actual implementation of the language would have to decide how environments are to be stored, accessed and updated. The details of these mechanisms do not shed much light on the meanings of these expressions and therefore we are justified in abstracting away from them.

Now the problem is to specify the value to which an expression evaluates given

1. Syntactic categories

   e in *Exp2*

   op in *Op*

   n in *Num*

   x in *Var*

2. Definitions

$$\begin{aligned} op &::= + \mid - \mid * \mid div \\ e &::= x \mid n \mid e' \ op \ e'' \end{aligned}$$

Figure 3.1: Abstract Syntax for *Exp2*

Rule CR $$\frac{}{\rho \vdash \mathbf{n} \implies \mathbf{n}}$$

Rule VarR $$\frac{}{\rho \vdash x \implies \rho(x)}$$

Rule OpR $$\frac{\rho \vdash e \implies v \qquad \rho \vdash e' \implies v'}{\rho \vdash e \ op \ e' \implies Ap(op, v, v')}$$

Figure 3.2: Evaluation semantics for *Exp2*

a particular environment. As we have already seen the set of values appropriate for expressions is the set of numerals, i.e. $Val(Exp2) = Num$. Technically the type of the evaluation relation is given by

$$\implies : ENV \longmapsto Exp2 \longmapsto Num.$$

In a given environment an expression will evaluate to a numeral. We will write this as $\rho \vdash e \implies v$ and it may be read "given the environment ρ the expression e evaluates to the value v." Since in this case values are always numerals we could also write $\rho \vdash e \implies \mathbf{n}$ but we will tend to use instead $\rho \vdash e \implies v$ in order to emphasise the role of values. The definition of the semantics of the new language is very simple; it is obtained by adding one extra clause to that of the previous chapter. The entire inductive definition is given in Figure 3.2.

The first rule, Rule CR, simply says that a numeral evaluates to itself in any

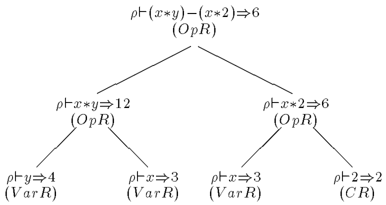


Figure 3.3: A proof tree

environment. The second says that to evaluate a variable you look up its value in the environment. The third is as before: to evaluate e *op* e' you first evaluate e and e' in the same environment and then apply the function associated with the symbol *op*. Notice that the environment never changes. When calculating the value of e in ρ all subexpressions are evaluated in the same environment ρ .

Let us consider an example. We will evaluate $(x * y) - (x * \mathbf{2})$ in an environment ρ , where $\rho(x)$ is $\mathbf{3}$ and $\rho(y)$ is $\mathbf{4}$. Rather than going through the long process of discovering the application of the Rules CR, VarR and OpR which will lead to the particular v such that $\rho \vdash (x*y) - (x*\mathbf{2}) \implies v$ we simply give the resulting proof:

1. $\rho \vdash \mathbf{2} \implies \mathbf{2}$	by Rule CR
2. $\rho \vdash x \implies \mathbf{3}$	by Rule VarR
3. $\rho \vdash x * \mathbf{2} \implies \mathbf{6}$	by Rule OpR to 1, 2
4. $\rho \vdash y \implies \mathbf{4}$	by Rule VarR
5. $\rho \vdash x * y \implies \mathbf{12}$	by Rule OpR to 2, 4
6. $\rho \vdash (x * y) - (x * \mathbf{2}) \implies \mathbf{6}$	by Rule OpR to 3, 5.

This is a "bottom-up" presentation of the proof of $\rho \vdash (x*y) - (x*\mathbf{2}) \implies \mathbf{6}$ in that it shows how the proof is gradually built up. However it is usually much more understandable and instructive to read these proofs in a "top-down" or goal-oriented fashion, namely starting with the conclusion and working through the layers of assumptions necessary in order to arrive at the conclusion. We urge readers to read these proofs in this "top-down" fashion; they should certainly be constructed in this manner. In fact the most understandable way of presenting these proofs is using *proof trees*. The proof tree corresponding to this proof is given in Figure 3.3. At the top of the tree is the theorem to be proved and the structure of the tree reflects the structure of the proof with the sub-trees reflecting the history of the derivation. However, as proofs get long it becomes difficult to present them in this style on one page but the reader is encouraged to think in terms of proof trees rather than our linear presentations of them.

We can show, as in the previous chapter, that, in a given environment, every expression has a unique value.

Theorem 3.1.1 *For every environment ρ:*

1. *for every expression e in Exp2 if $\rho \vdash e \implies \mathbf{n}$ and $\rho \vdash e \implies \mathbf{n}'$ then $\mathbf{n} = \mathbf{n}'$*
2. *for every expression e in Exp2 there exists a numeral $\mathbf{n}$ such that $\rho \vdash e \implies \mathbf{n}$.*

Proof The proofs are very similar to those of the corresponding results for the language *Exp* in the previous chapter. We examine the second one only. As with the corresponding Theorem 2.2.1 for *Exp* the proof is by structural induction but this time there are three cases in place of two. The required property, $P(e)$, is essentially the same:

there exists a numeral $\mathbf{n} \in Num$, such that $\rho \vdash e \implies \mathbf{n}$.

We must show:

1. $P(\mathbf{k})$ for every numeral $\mathbf{k}$
2. $P(x)$ for every variable x
3. $P(e' \;\; op \;\; e'')$, on the assumption that both $P(e')$ and $P(e'')$ are true.

The first case is trivial as the required numeral is $\mathbf{k}$ itself. The second case is also trivial: the required numeral $\mathbf{n}$ is $\rho(x)$ since we can apply Rule VarR to deduce $\rho \vdash x \implies \rho(x)$. The third case is identical to the corresponding case of the proof of Theorem 2.2.1 for *Exp*. □

We can also show that the value of an expression e in an environment ρ depends only on the value assigned by ρ to those variables which actually occur in e; if we change the value of ρ at other variables the value of e remains the same. Intuitively this can be seen to be true because in any proof of $\rho \vdash e \implies v$ any application of Rule VarR to variables which do not occur in e are superfluous and may be eliminated. The shortened proof will still be a proof of $\rho \vdash e \implies v$. Indeed in a natural construction of the proof these superfluous rules would never even be introduced. We will now prove this fact formally. For any two environments ρ, ρ' and any set of variables X we say $\rho =_X \rho'$ if for every variable x in X $\rho(x) = \rho'(x)$, i.e. the values corresponding to all the variables in X must be the same in both environments. They may of course be different for variables which are not in X. Note that if X is the empty set then $=_X$ equates all environments while if it is the set of all variables *Var* then $\rho =_X \rho'$ only if ρ and ρ' are exactly the same environments.

Let $Var(e)$ denote the set of all variables which occur in e. (How would you formally define $Var\,(e)$?). Then the property we prove of our operational semantics is:

Theorem 3.1.2 *If* $\rho =_{Var(e)} \rho'$ *then* $\rho \vdash e \implies \mathbf{n}$ *implies* $\rho' \vdash e \implies \mathbf{n}$.

Now $\rho =_{Var(e)} \rho'$ implies $\rho' =_{Var(e)} \rho$ and so if we can prove this theorem we will have shown that

If $\rho =_{Var(e)} \rho'$ then $\rho \vdash e \implies \mathbf{n}$ if and only if $\rho' \vdash e \implies \mathbf{n}$.

Proof As usual the proof is by structural induction. We wish to establish the property $P(e)$:

$\rho \vdash e \implies \mathbf{n}$ implies $\rho' \vdash e \implies \mathbf{n}$

for every expression e in *Exp2* and therefore we must establish:

1. $P(\mathbf{k})$ for every numeral $\mathbf{k}$.
   Suppose $\rho \vdash \mathbf{k} \implies \mathbf{n}$. The only rule which can be used to derive this statement is Rule CR and therefore $\mathbf{k} = \mathbf{n}$. Using Rule CR again we get $\rho' \vdash \mathbf{k} \implies \mathbf{n}$.

2. $P(x)$ for every variable x
   Suppose $\rho \vdash x \implies \mathbf{n}$. Then $\mathbf{n}$ must be $\rho(x)$ since in this case Rule VarR must have been applied. We can also apply it to obtain $\rho' \vdash x \implies \rho'(x)$. Now since $x \in Var(e)$ we have $\rho(x) = \rho'(x)$ and so $\rho' \vdash x \implies \mathbf{n}$.

3. $P(e \ \ op \ \ e')$ for every operator op assuming $P(e)$ and $P(e')$ are true.
   Suppose $\rho \vdash e \ \ op \ \ e' \implies \mathbf{n}$. Then there exists two values v and v' such that $\rho \vdash e \implies v$, $\rho \vdash e' \implies v'$ and $\mathbf{n}$ is $Ap(op, v, v')$. By induction we have that $\rho' \vdash e \implies v$ and $\rho' \vdash e' \implies v'$ and by applying Rule OpR we obtain $\rho' \vdash e \ \ op \ \ e' \implies \mathbf{n}$. □

At this stage the reader should be aware that our approach to defining the semantics of expressions does not depend very much on the specific set of operators chosen. We have used the four operators $+, -, *, div$ as examples and more operators could easily be added. We simply have to assume that the *Apply* mechanism, modelled by the relation *Ap* is capable of interpreting the new symbols correctly. For the moment we will continue to use our very specific language for expressions but in later chapters we will be less concerned with the precise collection of allowed operators.

3.2 Local Variables

In this section we add the concept of local variables or local declarations to the language. This is a very common feature of many programming languages. In the new expression

$$let \ \ x = \mathbf{7} \ \ in \ \ (y + x) * (z + x)$$

1. Syntactic categories

   e in *Exp3*

   op in *Op*

   n in *Num*

   x in *Var*

2. Definitions

$$\begin{array}{rcl} op & ::= & + \ | \ - \ | \ * \ | \ div \\ e & ::= & n \ | \ x \ | \ e' \ \ op \ \ e'' \ | \ \ let \ \ x = e' \ \ in \ \ e'' \end{array}$$

Figure 3.4: Abstract Syntax for *Exp3*

intuitively we mean that the expression $(y + x) * (z + x)$ is to be evaluated on the understanding that the value of x is the numeral $\mathbf{7}$. Of course we also need values for y and z and they will be obtained from the environment in which the overall expression is to be evaluated.

The new abstract syntax is just a slight extension of the previous one: it is obtained by adding a new kind of expression for local declarations of variables. The entire syntax of the extended language *Exp3* is given in Figure 3.4.

To formalise the operational semantics we introduce some notation for modifying environments. If ρ is an environment, x a variable and v a value, then we use $\rho[v/x]$ to denote a new environment. It is almost the same as the environment ρ. At the variable x, ρ evaluates to $\rho(x)$ whereas $\rho[v/x]$ evaluates to v; at every other variable y different from x, ρ and $\rho[v/x]$ evaluate to the same value, namely $\rho(y)$. Formally given ρ: $Var \longmapsto Num$, the new function $\rho[v/x]$: $Var \longmapsto Num$ is defined by

$$\rho[v/x](y) = \begin{cases} v & \text{if } y \text{ is } x \\ \rho(y) & \text{if } y \text{ is not } x. \end{cases}$$

The operational semantics of the new language can now be defined by adding one extra rule:

$$\text{Rule LocR} \quad \frac{\begin{array}{c}\rho \vdash e \implies v \\ \rho[v/x] \vdash e' \implies v'\end{array}}{\rho \vdash \ let \ \ x = e \ \ in \ \ e' \implies v'}$$

The new rule says that to evaluate the expression $let \ \ x = e \ \ in \ \ e'$ in an environment ρ you first evaluate e in the environment ρ to v and then evaluate e' in the environment $\rho[v/x]$, i.e. the environment which associates v with x and is the same as ρ at every other variable. This corresponds with our intuition. In $let\ x = e\ in\ e'$ we evaluate e' in the normal environment, except that we associate with the variable x the value of e.

As an example let us evaluate $let\ x = \mathbf{7}\ in\ x * y + (x\ div\ y)$ in an environment ρ where $\rho(x) = \mathbf{3}$ and $\rho(y) = \mathbf{2}$. The following is the resulting derivation:

1. $\rho \vdash \mathbf{7} \implies \mathbf{7}$	by Rule CR
2. $\rho[\mathbf{7}/x] \vdash x \implies \mathbf{7}$	by Rule VarR
3. $\rho[\mathbf{7}/x] \vdash y \implies \mathbf{2}$	by Rule VarR
4. $\rho[\mathbf{7}/x] \vdash x\ div\ y \implies \mathbf{3}$	applying Rule OpR to 2, 3
5. $\rho[\mathbf{7}/x] \vdash x * y \implies \mathbf{14}$	applying Rule OpR to 2, 3
6. $\rho[\mathbf{7}/x] \vdash x * y + x\ div\ y \implies \mathbf{17}$	Rule OpR to 4, 5
7. $\rho \vdash\ let\ x = \mathbf{7}\ in\ x * y + (x\ div\ y) \implies \mathbf{17}$	Rule LocR to 6, 1.

As another example, let us evaluate the expression $let\ y = x + \mathbf{3}\ in\ y * y + x$ in an environment ρ where $\rho(x) = \mathbf{2}$ and $\rho(y) = \mathbf{3}$ as follows:

1. $\rho \vdash \mathbf{3} \implies \mathbf{3}$	by Rule CR
2. $\rho \vdash x \implies \mathbf{2}$	by Rule VarR
3. $\rho \vdash x + \mathbf{3} \implies \mathbf{5}$	applying Rule OpR to 1, 2
4. $\rho[\mathbf{5}/y] \vdash x \implies \mathbf{2}$	by Rule VarR
5. $\rho[\mathbf{5}/y] \vdash y \implies \mathbf{5}$	by Rule VarR
6. $\rho[\mathbf{5}/y] \vdash y * y \implies \mathbf{25}$	applying Rule OpR to 5, 5
7. $\rho[\mathbf{5}/y] \vdash y * y + x \implies \mathbf{27}$	applying Rule OpR to 4, 6
8. $\rho \vdash\ let\ y = x + \mathbf{3}\ in\ y * y + x \implies \mathbf{27}$	applying Rule LocR to 3, 7.

Notice that in both examples the value of the local variable (x in the first example and y in the second example) in the original environment ρ had no effect on the evaluation. The introduction of the $let\ \ldots$ construction introduces a number of subtleties into the role of variables which we now investigate. One question is to determine the variables on which the value of an expression depends.

In the previous section we saw that the value of an expression e in the language *Exp2* only depended on $Var\,(e)$, the variables occurring in e; we proved $\rho =_{Var(e)} \rho'$ implies that $\rho \vdash e \implies \mathbf{n}$ if and only if $\rho' \vdash e \implies \mathbf{n}$. We now prove a similar result for the language *Exp3*. However, this time the set of variables on which the value depends is smaller than $Var(e)$. For example, in the expression considered above, $let\ y = x + \mathbf{3}\ in\ y * y + x$, the value does not depend on the value of y although y occurs in the expression. Instead it depends on the variables which occur in the expression but which are not "bound" by any local declaration such as y in $let\ y = \ldots\ in\ \ldots$. These are called the *free variables* of an expression, $FVar\,(e)$, and are defined as follows:

1. $FVar\,(\mathbf{n}) = \phi$

2. $FVar\,(x) = \{x\}$
3. $FVar\,(e\ op\ e') = FVar\,(e) \cup FVar\,(e')$
4. $FVar\,(\ let\ x = e\ in\ e') = (FVar\,(e') - \{x\}) \cup FVar\,(e)$.

These four clauses define $FVar$ for every expression e since e must be of one of the four forms considered. Notice that y is not a free variable of the expression $let\ y = x + \mathbf{3}\ in\ y * y + x$; the only free variable is x. The bound or local variables of e, $BVar\,(e)$, on the other hand are defined by:

1. $BVar\,(\mathbf{n}) = \phi$
2. $BVar\,(x) = \phi$
3. $BVar\,(e\ op\ e') = BVar\,(e) \cup BVar\,(e')$
4. $BVar\,(\ let\ x = e\ in\ e') = BVar\,(e) \cup BVar\,(e') \cup \{x\}$.

Although it may seem strange, variables may be both free and bound in an expression. For example, this is the case with y in $let\ y = y + \mathbf{4}\ in\ e$. This represents the fact that two occurrences of y play different roles in the expression. The first occurrence of y is a bound occurrence where it plays the role of a dummy variable; it indicates that all (free) occurrences of y in e should in fact refer to $y + \mathbf{4}$. This occurrence of y could be replaced by another variable without affecting the value of the overall expression provided that the same change of variable was carried out in e. The second occurrence of y, in $y + \mathbf{4}$, is a genuine free occurrence. This means that as the value of y varies the value of the overall expression $let\ y = y + \mathbf{4}\ in\ e$ will also vary.

We can now prove that the value of an expression depends only on the values associated with its free variables. More specifically we prove that we may change the value of any variable not in $FVar\,(e)$ without changing the value of e.

Theorem 3.2.1 *For every expression e in Exp3 if $\rho =_{FVar(e)} \rho'$ then $\rho \vdash e \implies \mathbf{n}$ implies $\rho' \vdash e \implies \mathbf{n}$.*

Proof The proof is by structural induction on e and the property we wish to establish for every expression e in *Exp3* is $P(e)$:

> for any two environments ρ and ρ', if $\rho =_{FVar(e)} \rho'$ then $\rho \vdash e \implies \mathbf{n}$ implies $\rho' \vdash e \implies \mathbf{n}$.

Since expressions can now be any one of four forms there are four cases to consider:

1. $P(\mathbf{k})$ for every numeral $\mathbf{k}$
2. $P(x)$ for every variable x

3. $P(e' \ \ op \ \ e'')$ for every operator op assuming both $P(e')$ and $P(e'')$ are true

4. $P($ *let* $x = e'$ *in* $e'')$ assuming both $P(e')$ and $P(e'')$ are true.

The first three cases are identical to the corresponding cases of the corresponding proof of Theorem 3.1.2. So we concentrate on the final one, when e is of the form *let* $x = e'$ *in* e''. Suppose $\rho \vdash$ *let* $x = e'$ *in* $e'' \implies \mathbf{n}$. This must have been established with Rule LocR. So there must be a value v' such that $\rho \vdash e' \implies v'$ and $\rho[v'/x] \vdash e'' \implies \mathbf{n}$. Now $FVar\,(e') \subseteq FVar\,(e)$ and so we may assume that $\rho =_{FVar(e')} \rho'$. By induction we now have that $\rho' \vdash e' \implies v'$. It may not be that $FVar\,(e'') \subseteq FVar\,(e)$ because x might be in $FVar\,(e'')$ and not in $FVar\,(e)$. However $\rho[v'/x] =_{\{x\}} \rho'[v'/x]$ since both environments associate the value v' with x. So $\rho[v'/x] =_{\{x\} \cup FVar(e)} \rho'[v'/x]$ and since $FVar\,(e'') \subseteq FVar\,(e) \cup \{x\}$ we have that $\rho[v'/x] =_{FVar(e'')} \rho'[v'/x]$. So from $\rho[v'/x] \vdash e'' \implies \mathbf{n}$ we deduce $\rho'[v'/x] \vdash e'' \implies \mathbf{n}$ by induction and therefore we may apply Rule LocR to obtain $\rho' \vdash$ *let* $x = e'$ *in* $e'' \implies \mathbf{n}$. □

This theorem states that the value of an expression depends only on the values associated by the environment to its free variables. In the remainder of this section we will explore further the roles of free and bound variables and in particular their effect on substitution. It may be skipped by the uninterested reader.

The value associated with variables which only appear bound play no role in the evaluation. In fact if we are careful we may change local variables without affecting the meaning of expressions. For example, the expressions *let* $y = \mathbf{3}$ *in* $y * y + x$ and *let* $z = \mathbf{3}$ *in* $z * z + x$ have exactly the same value. The exact value depends on the value of x, i.e. the value associated with x by the environment. For example, if x is **10** the value of the expression is **19** while if it is **5** its value is **14**. In fact, the y in *let* $y = \mathbf{3}$ *in* $y * y + x$ can be changed to any variable other than x without affecting the meaning of the expression. However, if we change it to x we get a different value: $\rho \vdash$ *let* $x = \mathbf{3}$ *in* $x * x + x \implies \mathbf{12}$ regardless of the value associated by ρ to x. This is because x is a free variable of the original expression *let* $y = \mathbf{3}$ *in* $y*y+x$ and substituting x for the bound variable y turns a free variable into a bound variable. In general changing a free variable into a bound variable will change the value of an expression. So one may only replace the local variables in an expression if the replacement does not interfere with the free variables. This clash in the roles of variables is usually avoided by defining a notion of substitution which takes this problem into account.

We use $e[e'/x]$ to denote the result of substituting the expression e' for x in the expression e. In fact this is a very poor description as the substitution should take into account the peculiarities of free and bound variables. In particular, as we have just discussed, we do not want free variables in e' to be bound or captured in the resulting expression. Also we do not want to substitute e' for every occurrence of x in

e, only for the *free* occurrences. The value of e depends only on these occurrences and not the bound occurrences and, therefore, the latter should be ignored. For example, substituting $y + \mathbf{4}$ for x in

$$(\textit{let } x = \mathbf{3} \textit{ in } x + y) + x$$

should result in

$$(\textit{let } x = \mathbf{3} \textit{ in } x + y) + y + \mathbf{4}$$

rather than

$$(\textit{let } x = \mathbf{3} \textit{ in } y + \mathbf{4} + y) + y + \mathbf{4}.$$

The formal definition of $e[e'/x]$ may be given by structural induction on e:

1. $x[e'/x] = e'$
2. $y[e'/x] = y$ if x and y are different
3. $\mathbf{n}[e'/x] = \mathbf{n}$
4. $(e_1 \ \ op \ \ e_2)[e'/x] = e_1[e'/x] \ \ op \ \ e_2[e'/x]$
5. $(\textit{let } x = e_1 \textit{ in } e_2)[e'/x] = \textit{let } x = (e_1[e'/x]) \textit{ in } e_2$
6. $(\textit{let } y = e_1 \textit{ in } e_2)[e'/x] = \textit{let } y = (e_1[e'/x]) \textit{ in } (e_2[e'/x])$ if $y \notin FVar\,(e')$
7. $(\textit{let } y = e_1 \textit{ in } e_2)[e'/x] = \textit{let } z = (e_1[e'/x]) \textit{ in } ((e_2[z/y])[e'/x])$ if $y \in FVar\,(e')$, where z is any variable not in $FVar\,(e') \cup FVar\,(e_2)$.

Notice that in the last case the resulting expression will depend on the particular variable z chosen. It is often best to choose a z which does not occur at all in e_2, as in the following examples:

1. $(\textit{let } x = x + \mathbf{4} \textit{ in } x * y)[y * \mathbf{3}/x] = \textit{let } x = (y * \mathbf{3}) + \mathbf{4} \textit{ in } x * y$
2. $(\textit{let } x = y + \mathbf{4} \textit{ in } y + x)[x + \mathbf{2}/y] = \textit{let } w = x + \mathbf{2} + \mathbf{4} \textit{ in } x + \mathbf{2} + w$
3. $(\textit{let } y = (\textit{let } y = x + \mathbf{3} \textit{ in } y * x) \textit{ in } x + y)[x + y/x] = \textit{let } w$ $= (\textit{let } z = x + y + \mathbf{3} \textit{ in } z * (x + y)) \textit{ in } x + y + w.$

This definition of substitution is not particularly straightforward and is much more difficult than normal textual substitution. However, it is the correct form of substitution for the language with local declarations as it respects the roles of local and free variables. That it is the natural form of substitution may be seen from the following property:

$\rho \vdash e[e'/x] \implies v$ if and only if there is a value v' such that $\rho \vdash e' \implies v'$ and $\rho[v'/x] \vdash e \implies v$.

We may also restate in a natural fashion the fact that we may change the local variables in an expression without changing its value:

$\rho \vdash$ *let* $x = e$ *in* $e' \implies v$ if and only if $\rho \vdash$ *let* $y = e$ *in* $(e'[y/x]) \implies v$ for every $y \notin FVar(e')$.

These two results are set as questions at the end of the chapter. Unfortunately they cannot be proved by structural induction. This is because substitution is not a purely structural operation on expressions. Sometimes it is necessary to change the structure a little; bound variables may have to be changed and consequently changes are made to components to compensate. In the seventh clause of the definition the result of substituting e' for x changes the bound variable of e and also the component e_2. So in general results have to be proved by induction on the size of expressions rather than their structure.

The reader should also be able to prove the standard results about this semantics, determinacy and well-definedness : relative to a given environment ρ every expression has a unique value.

3.3 Boolean Values

We now extend the syntax by introducing a new syntactic category for boolean expressions, *BExp*. Two new constants are introduced, T and F, and intuitively we expect boolean expressions to evaluate to one of these constants in the same way as expressions from *Exp* evaluate to numerals. In order to differentiate between the two different kinds of expressions in future we will refer to the latter as *arithmetic* expressions and the former as *boolean* expressions. To complete the language we also introduce boolean variables. The abstract syntax for this new extension is given in Figure 3.5.

The syntax for expressions is the same as before except that we have one new construct *If* be *Then* e' *Else* e''. Intuitively to evaluate such an expression you first evaluate the boolean expression be; if it is true evaluate e', otherwise evaluate e''. Boolean expressions are formed in the usual way with the boolean connectives *And, Or* and *Not*. In addition we have the expression $Equal(e, e')$ which intuitively evaluates to T if e and e' evaluate to the same value and F if they evaluate to different values. Note that e and e' must be arithmetic expressions. This is reflected in the fact that we use the meta-variables e, e' rather than be, be' in $Equal(e, e')$.

As explained in Section 2.2 each new significant syntactic category requires a corresponding set of values in order to extend the Evaluation semantics to that category. For arithmetic expressions the appropriate set of values is the numerals,

1. Syntactic categories

   e in *Exp4*

   be in *BExp*

   op in *Op*

   bop in *BOp*

   n in *Num*

   x in *Var*

   bx in *BVar*

2. Definitions

$$
\begin{array}{rcl}
op & ::= & + \mid - \mid * \mid div \\
bop & ::= & And \mid Or \\
e & ::= & n \mid x \mid e' \; op \; e'' \mid let \; x = e' \; in \; e'' \mid If \; be \; Then \; e' \; Else \; e'' \\
be & ::= & bx \mid T \mid F \mid be' \; bop \; be'' \mid Not \; be' \mid Equal(e, e')
\end{array}
$$

Figure 3.5: Abstract syntax for *Exp4*

i.e. $Val(Exp) = Num$. For boolean expressions it is natural to take the corresponding set of values to consist of simply the two boolean constants T and F, i.e. $Val(BExp) = \{T, F\}$. So every boolean expression will either evaluate to T or to F. We will often refer to $\{T, F\}$ as the *boolean values* and *Num* as the *arithmetic values*. Of course the evaluation will now depend not only on associating arithmetic values with arithmetic variables but also boolean values with the boolean variables. So we generalise *ENV* to be the set of functions

$$\rho\colon Var \cup BVar \longmapsto Num \cup \{T, F\}$$

which are "type - respecting", i.e. $\rho(x)$ is in *Num* whenever x is in *Var* and $\rho(bx)$ is in $\{T, F\}$ whenever bx is in *BVar*. The evaluation relation now takes one of the forms

$$\rho \vdash e \implies_A v$$

reducing an arithmetic expression e to an arithmetic value v in *Num* or

$$\rho \vdash be \implies_B bv$$

$$\text{Rule CR}\qquad \frac{}{\rho \vdash \mathbf{n} \Longrightarrow_A \mathbf{n}}$$

$$\text{Rule VarR}\qquad \frac{}{\rho \vdash x \Longrightarrow_A \rho(x)}$$

$$\text{Rule IfR}\qquad \frac{\rho \vdash be \Longrightarrow_B T \quad \rho \vdash e \Longrightarrow_A v}{\rho \vdash \mathit{If}\ be\ \mathit{Then}\ e\ \mathit{Else}\ e' \Longrightarrow_A v}$$

$$\frac{\rho \vdash be \Longrightarrow_B F \quad \rho \vdash e' \Longrightarrow_A v'}{\rho \vdash \mathit{If}\ be\ \mathit{Then}\ e\ \mathit{Else}\ e' \Longrightarrow_A v'}$$

$$\text{Rule OpR}\qquad \frac{\rho \vdash e \Longrightarrow_A v \quad \rho \vdash e' \Longrightarrow_A v'}{\rho \vdash e\ op\ e' \Longrightarrow_A Ap(op, v, v')}$$

$$\text{Rule LocR}\qquad \frac{\rho \vdash e \Longrightarrow_A v \quad \rho[v/x] \vdash e' \Longrightarrow_A v'}{\rho \vdash\ let\ x = e\ in\ e' \Longrightarrow_A v'}$$

Figure 3.6: Evaluation semantics for *Exp4*: $\Longrightarrow_A$

reducing a boolean expression be to a boolean value bv in $\{T, F\}$. The types of these relations are given by

$$\Longrightarrow_A : ENV \longmapsto Exp4 \longmapsto Num \quad and \quad \Longrightarrow_B : ENV \longmapsto BExp \longmapsto \{T, F\}.$$

They are axiomatised in Figure 3.6 and Figure 3.7.

Most of the rules are the same as before. In Rule CR there are two new boolean constants T and F. Rule OpR is also unchanged except that in addition to the arithmetic operators we have the two boolean operators And and Or and we assume that the *Apply* mechanism interprets these as the usual boolean functions. That is

$$Ap(And, x, y) = \begin{cases} T & \text{if } x = T \text{ and } y = T \\ F & \text{otherwise} \end{cases}$$

and

$$Ap(Or, x, y) = \begin{cases} T & \text{if } x = T \text{ or } y = T \\ F & \text{otherwise.} \end{cases}$$

$$\text{Rule CR}\qquad \frac{}{\rho \vdash T \Longrightarrow_B T} \qquad\qquad \frac{}{\rho \vdash F \Longrightarrow_B F}$$

$$\text{Rule VarR}\qquad \frac{}{\rho \vdash bx \Longrightarrow_B \rho(bx)}$$

$$\text{Rule EqR}\qquad \frac{\rho \vdash e \Longrightarrow_A v \quad \rho \vdash e' \Longrightarrow_A v}{\rho \vdash Equal(e, e') \Longrightarrow_B T} \qquad\qquad \frac{\rho \vdash e \Longrightarrow_A v \quad \rho \vdash e' \Longrightarrow_A v'}{\rho \vdash Equal(e, e') \Longrightarrow_B F}$$

if v is different from v'

$$\text{Rule BOpR}\qquad \frac{\rho \vdash be \Longrightarrow_B bv \quad \rho \vdash be' \Longrightarrow_B bv'}{\rho \vdash be\ bop\ be' \Longrightarrow_B Ap(bop, bv, bv')}$$

$$\text{Rule NotR}\qquad \frac{\rho \vdash be \Longrightarrow_B T}{\rho \vdash Not\ be \Longrightarrow_B F} \qquad\qquad \frac{\rho \vdash be \Longrightarrow_B F}{\rho \vdash Not\ be \Longrightarrow_B T}$$

Figure 3.7: Evaluation semantics for *Exp4*: $\Longrightarrow_B$

The rule for local declarations in arithmetic expressions remains unchanged. Rule IfR is for evaluating the conditional expressions, $\mathit{If}\ be\ \mathit{Then}\ e\ \mathit{Else}\ e'$. It says that the result of evaluating this expression is the value of e if be evaluates to T and the value of e' if it evaluates to F. The rule for evaluating $Equal(e, e')$ is also straightforward. The result is always either T or F, being T if e and e' evaluate to the same arithmetic value and F if they give different values. Finally we have the obvious rule for evaluating $Not\ \ be$. This could also have been handled as part of Rule OpR using the boolean function of one argument

$$Not\ : Val(BExp) \longmapsto Val(BExp)$$

defined by $Ap(Not, T) = F$, $Ap(Not, F) = T$.

Note that the definitions of $\Longrightarrow_A$ and $\Longrightarrow_B$ are intrinsically intertwined. In Rule EqR a conclusion involving $\Longrightarrow_B$ depends on premises involving $\Longrightarrow_A$ while in Rule IfR a conclusion involving $\Longrightarrow_A$ depends on a premis involving $\Longrightarrow_B$. This means that neither of these relations can be defined independently, in isolation from the other. So both sets of rules should be considered as one inductive definition which *simultaneously* defines two relations $\Longrightarrow_A$ and $\Longrightarrow_B$.

As an example of the use of these rules let us evaluate the expression $\mathit{If}\ \ Equal(x, y)\ \ \mathit{Then}\ \ z\ \ \mathit{Else}\ \ x + y$ in an environment ρ where $\rho(x) = \mathbf{0}, \rho(y) = \mathbf{1}$ and $\rho(z) = \mathbf{2}$.

1. $\rho \vdash x \implies_A \mathbf{0}$	Rule VarR
2. $\rho \vdash y \implies_A \mathbf{1}$	Rule VarR
3. $\rho \vdash Equal(x,y) \implies_B F$	RuleEq to 1,2
4. $\rho \vdash x+y \implies_A \mathbf{1}$	Rule OpR to 1, 2
5. $\rho \vdash If\ Equal(x,y)\ Then\ z\ Else\ x+y \implies_A \mathbf{1}$	Rule IfR to 3, 4.

To show that this new abstract evaluation mechanism makes sense, we must prove two results:

1. for every ρ and e in *Exp4* there exists some value v such that $\rho \vdash e \implies_A v$
2. $\rho \vdash e \implies_A v$ and $\rho \vdash e \implies_A v'$ imply $v = v'$.

The first statement says that every arithmetic expression can be evaluated to some value and the second says that this value is unique. As usual we would prove these statements by structural induction on terms. But a problem arises due to the interdependence of $\implies_A$ and $\implies_B$. For example, in the proof of the first statement when we are considering the case when e has the form $If\ b\ Then\ e\ Else\ e'$, we will need to know if b evaluates to a boolean value. We will not be able to assume this by induction as the inductive hypothesis is only about arithmetic expressions. So we need to prove statements not only about $\implies_A$ but also $\implies_B$. The correct statement of the theorems are:

Theorem 3.3.1

1. For every arithmetic expression e in Exp4 and every environment ρ there exists some numeral $\mathbf{n}$ *such that* $\rho \vdash e \implies_A \mathbf{n}$

2. for every boolean expression be in BExp and every environment ρ there exists some boolean value bv such that $\rho \vdash be \implies_B bv$.

Theorem 3.3.2

1. For every arithmetic expression e in Exp and every environment ρ if $\rho \vdash e \implies_A v$ and $\rho \vdash e \implies_A v'$ then $v = v'$

2. for every boolean expression be in BExp and every environment ρ if $\rho \vdash be \implies_B bv$ and $\rho \vdash be \implies_B bv'$ then $bv = bv'$.

Both these theorems are now proved by structural induction and in each case the statements about $\implies_A$ and $\implies_B$ are proved simultaneously. So, for example, in the first theorem when we are examining the case $If\ be\ Then\ e\ Else\ e'$ we will be able to assume by induction that either $be \implies_B T$ or $be \implies_B F$. Similarly when examining the case of the boolean expression $Equal(e,e')$ we will be able to assume

that for some arithmetic values v, v', $e \implies_A v$ and $e' \implies_A v'$. The details of the proofs are left to the reader. They are very similar to the corresponding proofs in the previous sections.

3.4 Function Definitions

As a final extension to the language we add user-defined function definitions. The idea is to introduce a new syntactic category of function or procedure names and to associate with the name a body or definition. For example, take the squaring function. We use *Square* as the name of the function and its definition is given by:

$$Square(x) \Longleftarrow x * x.$$

On the left-hand side we have $Square(x)$, which consists of the name of the function *Square* and a list of formal parameters. In this case the list is of length 1 namely (x). The definition is on the right-hand side and consists of an expression constructed in the usual way.

We call such a definition a *declaration* as it declares the meaning to be associated with the name *Square*. Now, in the course of evaluation, if we come across an occurrence of *Square*, we can use this declaration to make sense of it. Consider, for example, the evaluation of the expression $Square(\mathbf{3})$ in the context of this declaration. We want the value which results from applying the function whose name is *Square* to the value $\mathbf{3}$. The declaration gives us the current interpretation of *Square* and it is therefore natural to say that the value of $Square(\mathbf{3})$ is exactly the value of

$$x * x$$

in an environment where the formal variable x is bound to the numeral $\mathbf{3}$.

If we apply our existing rules for evaluating expressions to this expression in this new environment we see that its value is $\mathbf{9}$.

This is the basis for a very simple rule for handling user-defined functions: whenever we have a declaration of the form

$$F(x) \Longleftarrow e$$

then the evaluation rule

$$\frac{\rho \vdash e' \implies v' \qquad \rho[v'/x] \vdash e \implies v}{\rho \vdash F(e') \implies v}$$

captures the intended meaning of the definition. This is exactly the rule which we applied informally above to evaluate the expression $Square(\mathbf{3})$. It is a very powerful rule as it applies not only to simple function definitions such as *Square* but also definitions of functions which use their own name in their definitions, so-called *recursive* *definitions*. For example, take the factorial function. We use *Fac* as the name of the function and its definition is given by:

$$Fac(x) \Longleftarrow If\ \ Equal(x, \mathbf{0})\ \ Then\ \ \mathbf{1}\ \ Else\ \ x * Fac(x - \mathbf{1}).$$

This consists of an expression constructed in the usual way except that in addition the name of the function may also be used. Let us now see, at least informally, how a rule such as the above may be used to evaluate an expression involving *Fac*. Consider the expression $Fac(\mathbf{2})$ in the context of this declaration. We want the value which results from applying the function whose name is *Fac* to the value $\mathbf{2}$. The declaration gives us the current interpretation of *Fac* and, as with the user-defined function *Square* above, it is natural to say that the value of $Fac(\mathbf{2})$ is exactly the value of

$If\ \ Equal(x, \mathbf{0})\ \ Then\ \ \mathbf{1}\ \ Else\ \ x * Fac(x - \mathbf{1})$
in an environment where the formal variable x is bound to the numeral $\mathbf{2}$.

On applying our existing rules to this expression in this new environment we see that its value is exactly the value of

$$\mathbf{2} * Fac(\mathbf{1})$$

because $Equal(x, \mathbf{0})$ evaluates to the boolean value F. Now, to find the value of this, we must first have the value of $Fac(\mathbf{1})$. This once more coincides with the value of

$$If\ \ Equal(x, \mathbf{0})\ \ Then\ \ \mathbf{1}\ \ Else\ \ x * Fac(x - \mathbf{1})$$

evaluated this time in an environment where the formal parameter x is bound to the numeral $\mathbf{1}$. This in turn coincides with the value of

$$\mathbf{1} * Fac(\mathbf{0})$$

because once more $Equal(x, \mathbf{0})$ evaluates to F in this environment. With one more application of our new rule for function definitions, $Fac(\mathbf{0})$ evaluates to $\mathbf{1}$ because the definition of *Fac* is interpreted in an environment where x is $\mathbf{0}$. So $\mathbf{1} * Fac(\mathbf{0})$ evaluates to $\mathbf{1} * \mathbf{1}$, i.e. $\mathbf{1}$. Substituting back into the above expressions we see that $Fac(\mathbf{2})$ evaluates to $\mathbf{2}$.

After this informal introduction to function definitions let us now see how to extend the syntax of the the existing language *Exp*4 with declarations. The resulting language is called *Fpl*, shorthand for “a Functional programming language ”. We will allow mutually recursive definitions such as

$$\begin{array}{c} F_1(x_1, \ldots, x_{k_1}) \Longleftarrow e_1 \\ \ldots \\ \ldots \\ F_n(x_1, \ldots, x_{k_n}) \Longleftarrow e_n \end{array}$$

1. Syntactic categories

p in *Prog*	op in *Op*
D in *Dec*	bop in *BOp*
e in *Exp*	n in *Num*
be in *BExp*	x in *Var*
F in *FunVar*	bx in *BVar*

2. Definitions

$$\begin{array}{rcl} p & ::= & < e, D > \\ D & ::= & F(x_1, ..., x_k) \Longleftarrow e \ \mid\ F(x_1, ..., x_k) \Longleftarrow e,\ D' \\ op & ::= & + \mid - \mid * \mid div \\ bop & ::= & And \mid Or \\ e & ::= & n \mid x \mid e'\ op\ e'' \mid let\ x = e'\ in\ e'' \mid If\ be\ Then\ e'\ Else\ e'' \mid \\ & & F(e_1, \ldots, e_k),\ \text{where}\ \ arity(F) = k \\ be & ::= & bx \mid T \mid F \mid be'\ bop\ be'' \mid Not\ be \mid Equal(e, e') \end{array}$$

Figure 3.8: Abstract syntax for *Fpl*

and state a little more formally the required extensions to the definition of the evaluation relation. The abstract syntax is given in Figure 3.8.

We assume a new syntactic category of function names which we call *FunVar*. We are not particularly interested in how they are constructed, in the same way that we are not interested in the set of variables *Var*. We just assume that they exist and use upper case letters such as F, G as typical examples or sometimes strings of letters such as *Fac, Div*, etc. However, each function name we define expects a specific number of arguments. For example, *Fac* only expects one, whereas a function *Rem*, which calculates the remainder of one number when divided by another, takes two. Accordingly, each function name has associated with it a number, greater than or equal to zero, called its arity. If $arity(F)$ is k, then when forming expressions F must be applied to exactly k arguments, i.e. $F(e_1, \ldots, e_n)$ is an expression only when n is k. This restriction cannot be expressed directly in the language for abstract syntax definitions; we simply append a condition to the appropriate clause of the definition of expressions. This is the only new clause in the definition of expressions. The same restriction holds when defining declarations. A declaration is simply a list of definitions each of the form

$F(x_1, \ldots, x_k) \Longleftarrow e.$

This is said to be a definition of the function name F, and e is said to be the *body* of the definition where, of course, k is the arity of the function name F. Finally, a program consists of an expression together with a declaration, $< e, D >$. Intuitively the declaration D supplies the definitions of all the function names in e. However, *a priori* we have no guarantee that this is so as it cannot be enforced using our grammar rules. Instead we must make a further restriction on the syntax, stipulating that for any program $< e, D >$:

1. every function name in e has a corresponding definition in D
2. every function name occurring in D has exactly one definition in D.

We will only consider programs which satisfy these restrictions.

Let us now turn our attention to the evaluation rules. The evaluation of expressions will still require an environment ρ, to give a meaning to the variables which appear in expressions. The evaluation of arithmetic expressions will depend on a declaration to provide a context for the new function symbols which they may contain. For this reason we need to further parametrise the arrow $\Longrightarrow$ with respect to not only environments but also declarations. Now

$$D, \rho \vdash e \Longrightarrow_A v$$

means that, in the environment ρ and assuming that D provides a definition for all the relevant function symbols, the arithmetic expression e evaluates to the arithmetic value v. Technically the evaluation arrow now has the type

$$\Longrightarrow_A : \ Dec \longmapsto ENV \longmapsto Exp \longmapsto Num.$$

This complication, introduced by declarations, applies also to boolean expressions because although the new function names may only appear in arithmetic expressions these arithmetic expressions may be used to form boolean expressions. For example, $Equal(F(e), G(e'))$ is a boolean expression and may only be evaluated if we have declarations associated with the function names F and G. So the new type of $\Longrightarrow_B$ is:

$$\Longrightarrow_B : \ Dec \longmapsto ENV \longmapsto BExp \longmapsto \{T, F\}.$$

These new relations will then provide a means of evaluating programs, all of which are of the form $< e, D >$. We can say that

$$\rho \vdash < e, D > \Longrightarrow v \text{ whenever } D, \rho \vdash e \Longrightarrow_A v.$$

The rules for $\Longrightarrow_A$ are given in Figure 3.9. As one expects, all of the rules from the previous section are inherited and there is only one addition, Rule FunR:

$$\frac{D, \rho \vdash e_i \Longrightarrow_A v_i, 1 \leq i \leq k \qquad D, \rho[v_1/x_1, \ldots, v_k/x_k] \vdash e \Longrightarrow_A v}{D, \rho \vdash F(e_1, \ldots, e_k) \Longrightarrow_A v}$$
$$\text{whenever } F(x_1, \ldots, x_k) \Longleftarrow e \text{ occurs in D}$$

This has already been explained intuitively: to evaluate $F(e_1, \ldots, e_k)$, first evaluate the parameters and then evaluate the body of the definition of F in an environment in which the formal parameters are bound to the values of the actual parameters. As an alternative, one could leave the parameters unevaluated and simply substitute them directly into the body of the definition. Our choice is called the *call-by-value* parameter-passing mechanism whereas the alternative is called *call-by-name*. It is defined formally by the rule

$$\frac{D, \rho \vdash e[e_1/x_1, \ldots, e_k/x_k] \Longrightarrow_A v}{D, \rho \vdash F(e_1, \ldots, e_k) \Longrightarrow_A v}$$
$$\text{whenever } F(x_1, \ldots, x_k) \Longleftarrow e \text{ occurs in D}$$

However, we will only use the call-by-value Rule FunR given in Figure 3.9. The definition of $\Longrightarrow_B$ for *Fpl* is identical to that for *Exp4*, given in Figure 3.7.

As an example consider the program $< Rem(\mathbf{3}, \mathbf{5}), D >$ where D is the declaration

$$
\begin{array}{ll}
Rem(x, y) \Longleftarrow & If\ \ Equal(x, y) \\
 & Then\ \ \mathbf{0} \\
 & Else\ \ \ If\ \ Equal(y - x, \mathbf{0}) \\
 & \qquad\quad Then\ \ y \\
 & \qquad\quad Else\ \ Rem(x, y - x)
\end{array}
$$

and let ρ be any environment. This function calculates the remainder on dividing y by x and uses the fact that $y - x$ evaluates to $\mathbf{0}$ whenever y is less than or equal to x. Recall that we only have numerals corresponding to the natural numbers and $-$ is interpreted as the approximation to subtraction on the natural numbers as explained in Section 2.1. We show how to derive

$$D, \rho \vdash < Rem(\mathbf{3}, \mathbf{5}), D > \Longrightarrow \mathbf{2}.$$

The derivation is quite complicated and the reader is encouraged to discover it directly in a *top-down* or goal-oriented fashion. Otherwise the derivation below will not be very intelligible. For convenience we let e_1 denote the expression

$$If\ \ Equal(y - x, \mathbf{0})\ \ Then\ \ y\ \ Else\ \ Rem(x, y - x).$$

1.	$D,\rho[\mathbf{3}/x,\mathbf{5}/y][\mathbf{3}/x,\mathbf{2}/y] \vdash \mathbf{0} \implies_A \mathbf{0}$	Rule CR
2.	$D,\rho[\mathbf{3}/x,\mathbf{5}/y][\mathbf{3}/x,\mathbf{2}/y] \vdash y \implies_A \mathbf{2}$	Rule VarR
3.	$D,\rho[\mathbf{3}/x,\mathbf{5}/y][\mathbf{3}/x,\mathbf{2}/y] \vdash x \implies_A \mathbf{3}$	Rule VarR
4.	$D,\rho[\mathbf{3}/x,\mathbf{5}/y][\mathbf{3}/x,\mathbf{2}/y] \vdash y - x \implies_A \mathbf{0}$	Rule OpR to 2, 3
5.	$D,\rho[\mathbf{3}/x,\mathbf{5}/y][\mathbf{3}/x,\mathbf{2}/y] \vdash \mathit{Equal}(y-x,\mathbf{0}) \implies_B T$	Rule EqR to 1, 4
6.	$D,\rho[\mathbf{3}/x,\mathbf{5}/y][\mathbf{3}/x,\mathbf{2}/y] \vdash e_1 \implies_A \mathbf{2}$	Rule IfR to 5, 2
7.	$D,\rho[\mathbf{3}/x,\mathbf{5}/y][\mathbf{3}/x,\mathbf{2}/y] \vdash \mathit{Equal}(x,y) \implies_B F$	Rule EqR to 2,3
8.	$D,\rho[\mathbf{3}/x,\mathbf{5}/y][\mathbf{3}/x,\mathbf{2}/y] \vdash e \implies_A \mathbf{2}$	Rule IfR to 6, 7
9.	$D,\rho[\mathbf{3}/x,\mathbf{5}/y] \vdash x \implies_A \mathbf{3}$	Rule VarR
10.	$D,\rho[\mathbf{3}/x,\mathbf{5}/y] \vdash y \implies_A \mathbf{5}$	Rule VarR
11.	$D,\rho[\mathbf{3}/x,\mathbf{5}/y] \vdash y - x \implies_A \mathbf{2}$	Rule OpR to 9, 10
12.	$D,\rho[\mathbf{3}/x,\mathbf{5}/y] \vdash \mathit{Rem}(x,y-x) \implies_A \mathbf{2}$	Rule FunR to 9, 11, 8
13.	$D,\rho[\mathbf{3}/x,\mathbf{5}/y] \vdash \mathbf{0} \implies_A \mathbf{0}$	Rule CR
14.	$D,\rho[\mathbf{3}/x,\mathbf{5}/y] \vdash \mathit{Equal}(y-x,\mathbf{0}) \implies_B F$	Rule EqR to 11, 13
15.	$D,\rho[\mathbf{3}/x,\mathbf{5}/y] \vdash e_1 \implies_A \mathbf{2}$	Rule IfR to 12, 14
16.	$D,\rho[\mathbf{3}/x,\mathbf{5}/y] \vdash \mathit{Equal}(x,y) \implies_B F$	Rule EqR to 9, 10
17.	$D,\rho[\mathbf{3}/x,\mathbf{5}/y] \vdash e \implies_A \mathbf{2}$	Rule IfR to 15, 16
18.	$D,\rho \vdash \mathbf{3} \implies_A \mathbf{3}$	Rule CR
19.	$D,\rho \vdash \mathbf{5} \implies_A \mathbf{5}$	Rule CR
20.	$D,\rho \vdash \mathit{Rem}(\mathbf{3},\mathbf{5}) \implies_A \mathbf{2}$	Rule FunR to 18, 19, 17
21.	$D,\rho \vdash < \mathit{Rem}(\mathbf{3},\mathbf{5}), D > \implies \mathbf{2}$	Rule for programs.

This extension to the language changes drastically its character. In *Exp4* we are assured that in an environment every expression evaluates to some value. This is no longer the case.
For example, consider the program $< F(\mathbf{1}), D >$ where D is the declaration

$$F(x) \Longleftarrow F(x+\mathbf{1}).$$

Does there exist a value v such that $D,\rho \vdash F(\mathbf{1}) \implies_A v$? Intuitively such a v cannot exist because, regardless of the environment, in order to calculate $F(\mathbf{1})$ we need to calculate $F(\mathbf{2})$ which, in turn, needs $F(\mathbf{3})$, which needs $F(\mathbf{4})$, etc. So the calculation cannot start anywhere.

More formally, we can show first that if $D,\rho \vdash F(\mathbf{1}) \implies_A v$ then for every $\mathbf{n}$ in *Num*: $D,\rho \vdash F(\mathbf{n}) \implies_A v$. The only way to derive $D,\rho \vdash F(\mathbf{1}) \implies_A v$ is to use the Rule FunR. This can only be applied if we can derive $D,\rho \vdash F(\mathbf{2}) \implies_A v$. This in turn can only be derived using Rule FunR, which requires a derivation of $D,\rho \vdash F(\mathbf{3}) \implies_A v$. Continuing we see that the existence of a derivation $D,\rho \vdash F(\mathbf{1}) \implies_A v$ necessarily entails a derivation of $D,\rho \vdash F(\mathbf{n}) \implies_A v$ for every n in *Num*. Now consider the length of the alleged derivation of $D,\rho \vdash F(\mathbf{1}) \implies_A v$. Suppose it involves $k \geq 0$ steps. The last step to be applied is Rule FunR to $D,\rho \vdash F(\mathbf{2}) \implies_A v$. So the derivation of $D,\rho \vdash F(\mathbf{2}) \implies_A v$ requires $k-1$ steps. Similarly the derivation of $D,\rho \vdash F(\mathbf{3}) \implies_A v$ requires $k-2$ steps. Continuing we see that $D,\rho \vdash F(k+\mathbf{1}) \implies_A v$ requires no steps, i.e. it must be an instance of a rule with no premises. This is patently false, as demonstrated simply by examining all the possible rules. It follows that the supposed derivation of $D,\rho \vdash F(\mathbf{1}) \implies_A v$ does not exist.

This example shows that some expressions, or more accurately some programs, may not yield an actual value. However, we can still show determinacy, i.e. that at most one value will ever be produced by a program in a given environment.

Theorem 3.4.1 *For every environment ρ and every program p, $\rho \vdash p \implies v$ and $\rho \vdash p \implies v'$ imply $v = v'$.*

With the previous more simple languages this property was proved using structural induction on expressions. In our extended language this is no longer possible. Note that in order to prove the theorem it is sufficient to prove the corresponding result for expressions:

$$D,\rho \vdash e \implies_A v \text{ and } D,\rho \vdash e \implies_A v' \text{ implies } v = v'.$$

Suppose we tried to use structural induction on terms to prove this statement. We would eventually have to consider the case when e has the structure $F(e_1,\ldots,e_k)$ for some function symbol F and expressions $e_1,\ldots,e_k$. By induction we could assume that each e_i evaluates to a unique value. But in order to conclude that the application of Rule FunR must consequently lead to a unique value, we must also be able to assume that e evaluates, in the modified environment, to a unique value, where the function symbol F is defined in the declaration D by $F(x_1,\ldots,x_k) \Longleftarrow e$. But this assumption cannot be made since the expression "e" is not a sub-expression of the expression we are considering, "$F(e_1,\ldots,e_k)$". In general it bears no relation to it and may in fact be much more complicated than the sub-expressions $e_1,\ldots,e_k$. So, in order to prove the theorem, we need a new form of induction. Luckily the relations $\implies_B$ and $\implies_A$ are defined inductively in exactly the same way as EV and DIV in Chapter 1 and therefore they have associated with them an instance of Rule induction. Let us briefly recall this proof method. Suppose $P(x,y)$ is a property and we wish to establish

$$e \implies_A v \text{ implies } P(e,v).$$

Then it is sufficient to consider each condition defining $\implies_A$,

$$\frac{\mathit{premis}}{\mathit{conclusion}}$$

and assuming P is true of the premis, prove that P is also true of the conclusion. That is, it is sufficient to show that P is preserved by each of the defining conditions of $\Longrightarrow_A$. We use this proof method to establish the uniqueness result.

This proof method often looks very mysterious and, when used, it seems as if we are "getting something for free". The trick is to think of a suitable property P which, on the one hand, implies the required result and, on the other, is preserved by each of the rules. Most of the effort goes into finding the property P; once it is found checking that it does indeed satisfy the conditions is often virtually automatic. The next theorem is a typical case in point.

Theorem 3.4.2

1. *For every ρ, if $D, \rho \vdash be \Longrightarrow_B bv$ and $D, \rho \vdash be \Longrightarrow_B bv'$, then $bv = bv'$*
2. *for every ρ, if $D, \rho \vdash e \Longrightarrow_A v$ and $D, \rho \vdash e \Longrightarrow_A v'$ then $v = v'$.*

Proof

Both results are proved simultaneously by Rule induction. Let $P_B(\rho, be, bv)$ be the condition for boolean expressions:

$$\text{for every } bv' \; D, \rho \vdash be \Longrightarrow_B bv' \text{ implies } bv' = bv$$

and $P_A(\rho, e, v)$ be the same condition for arithmetic expressions:

$$\text{for every } v' \; D, \rho \vdash e \Longrightarrow_A v' \text{ implies } v' = v.$$

We show that these predicates are preserved by the defining conditions of $\Longrightarrow_B$ and $\Longrightarrow_A$ and therefore we can conclude

$$D, \rho \vdash be \Longrightarrow_B bv \text{ implies } P_B(\rho, be, bv)$$

and

$$D, \rho \vdash e \Longrightarrow_A v \text{ implies } P_A(\rho, e, v).$$

If we spell out these results they mean

$$D, \rho \vdash be \Longrightarrow_B bv \text{ and } D, \rho \vdash be \Longrightarrow_B bv' \text{ imply } bv = bv'$$

and

$$D, \rho \vdash e \Longrightarrow_A v \text{ and } D, \rho \vdash e \Longrightarrow_A v' \text{ imply } v = v'$$

which is what we are required to show.

We must show that P_B is preserved by the eight conditions in the definition of $\Longrightarrow_B$ and that P_A is preserved in the seven conditions in that of $\Longrightarrow_A$. Most of these are trivial as soon as we decipher what exactly is required and we examine only a representative sample.

1. Rule EqR. Let us consider one of this pair of rules, say when T is produced:

$$\frac{\begin{array}{c} D, \rho \vdash e \Longrightarrow_A v \\ D, \rho \vdash e' \Longrightarrow_A v \end{array}}{D, \rho \vdash Equal(e, e') \Longrightarrow_B T}$$

There are two premises, $D, \rho \vdash e \Longrightarrow_A v$, $D, \rho \vdash e' \Longrightarrow_A v$ and one conclusion, $D, \rho \vdash Equal(e, e') \Longrightarrow_B T$. So we assume $P_A(\rho, e, v)$ and $P_A(\rho, e', v)$ and we must prove $P_B(\rho, Equal(e, e'), T)$, i.e. $D, \rho \vdash Equal(e, e') \Longrightarrow_B bv'$ implies $bv' = T$. So suppose $D, \rho \vdash Equal(e, e') \Longrightarrow_B bv'$. Rule EqR must have been used to derive this statement. So there must be two arithmetic values w, w' with $D, \rho \vdash e \Longrightarrow_A w$ and $D, \rho \vdash e' \Longrightarrow_A w'$. But $P_A(\rho, e, v)$ implies that $w = v$ while $P_A(\rho, e', v)$ implies $w' = v$ also. This in turn implies that $bv' = T$, by Rule EqR.

The remaining cases for $\Longrightarrow_B$ are equally straightforward. Now let us consider two cases for $\Longrightarrow_A$.

2. Rule LocR:

$$\frac{\begin{array}{c} D, \rho \vdash e \Longrightarrow_A v \\ D, \rho[v/x] \vdash e' \Longrightarrow_A w' \end{array}}{D, \rho \vdash \; let \;\; x = e \;\; in \;\; e' \Longrightarrow_A w'}$$

Here there are two premises, $D, \rho \vdash e \Longrightarrow_A v$ and $D, \rho[v/x] \vdash e' \Longrightarrow_A w'$, and one conclusion, $D, \rho \vdash \; let \;\; x = e \;\; in \;\; e' \Longrightarrow_A w'$. Therefore to show that the rule preserves the property p we assume $P_A(\rho, e, v)$ and $P_A(\rho[v/x], e', w')$ and prove that $P_A(\rho, \; let \;\; x = e \;\; in \;\; e', w')$ is a consequence, i.e $D, \rho \vdash \; let \;\; x = e \;\; in \;\; e' \Longrightarrow_A v'$ implies $v' = w'$. So suppose $D, \rho \vdash \; let \;\; x = e \;\; in \;\; e' \Longrightarrow_A v'$. To derive this Rule LocR must have been used. So there must be some value, w say, such that $D, \rho \vdash e \Longrightarrow_A w$ and $\rho[w/x] \vdash e' \Longrightarrow_A v'$. By $P_A(\rho, e, v)$ we have that $w = v$ and therefore $\rho[v/x] \vdash e' \Longrightarrow_A v'$. Now $P_A(\rho[v/x], e', w')$ in turn implies $v' = w'$.

3. Rule FunR:

$$\frac{\begin{array}{c} D, \rho \vdash e_i \Longrightarrow_A v_i, \; 1 \leq i \leq k \\ D, \rho[v_1/x_1, \ldots, v_k/x_k] \vdash e \Longrightarrow_A v \end{array}}{D, \rho \vdash F(e_1, \ldots, e_k) \Longrightarrow_A v}$$

It is this case which requires us to use the more complicated form of induction rather than structural induction. Here there are $(k+1)$ premises and one conclusion; we may assume $P_A(\rho, e_i, v_i), 1 \leq i \leq k$, and $P_A(\rho[v_1/x_1, \ldots, v_k/x_k], e, v)$ and we have to prove $P_A(\rho, F(e_1, \ldots, e_k), v)$. So suppose $D, \rho \vdash F(e_1, \ldots, e_k) \Longrightarrow_A v'$. We have to show that $v' = v$. Since Rule FunR has to be applied to conclude this, there must be values $v_1', \ldots, v_k'$, such that $D, \rho \vdash e_i \Longrightarrow_A v_i', 1 \leq i \leq k$, and $\rho[v_1'/x_1, \ldots, v_k'/x_k] \vdash e \Longrightarrow_A v'$. Now $P_A(\rho, e_i, v_i)$ implies $v_i' = v_i$ for each $i, 1 \leq i \leq k$, and so $\rho[v_1/x_1, \ldots, v_k/x_k] \vdash e \Longrightarrow_A v'$. But now $P_A(\rho[v_1/x_1, \ldots, v_k/x_k], e, v)$ implies $v' = v$. $\square$

This finishes our exposition of an Evaluation semantics for *Fpl*, a simple yet powerful functional programming language. It contains many of the features common to modern functional languages such as local declarations and user-defined fuctions. Admittedly these functions are first-order in that they may only take as arguments data values or constants. In more realistic languages they may also take functions as arguments and even return them as results. This extension may be accommodated within our framework but the topic is somewhat outside the scope of this introductory text. The interested reader may pursue it in [Kah87].

Further Reading: A language very similar to *Fpl*, called **REC** is given an Evaluatin semantics in Chapter Nine of [Win93]. But the reader should be warned about the difference in notation. For us arrows such as $\Longrightarrow$ tend to be used for Evaluation semantics while in [Win93] $\rightarrow$ is used, sometimes with subscripts and superscripts to distinguish different variations.

Questions

Q1 Use the Evaluation relation in Section 3.3 to evaluate the following expressions, assuming an environment ρ such that $\rho(x) = \mathbf{1}, \rho(y) = \mathbf{3}$ and $\rho(bx) = T$.

$$let\ \ x = (\ let\ \ x = x + \mathbf{2}\ \ in\ \ x * \mathbf{2})\ \ in\ \ y * x$$

$$\begin{array}{l} If\ \ Equal(x,\ let\ \ x = \mathbf{1}\ \ in\ \ y + x) \\ \quad Then\ \ y + \mathbf{2} \\ \quad Else\ \ If\ \ bx \\ \qquad\quad Then\ \ let\ \ x = x * y\ \ in\ \ x * y \\ \qquad\quad Else\ \ let\ \ y = x * y\ \ in\ \ y * y \end{array}$$

Q2 Evaluate the expression $U(\mathbf{2})$ in *Fpl* using the declaration

$$\begin{array}{l} U(x) \Longleftarrow If\ \ Equal(x, \mathbf{0}) \\ \qquad\qquad Then\ \ \mathbf{0} \\ \qquad\qquad Else\ \ \mathbf{1} + U(x - \mathbf{1}) \end{array}$$

Q3 Give an example of a declaration D such that $D, \rho \vdash e \Longrightarrow_A \mathbf{k}$ for for some expression e and some numeral $\mathbf{k}$ if we use call-by-name but $D, \rho \vdash e \Longrightarrow_A \mathbf{k}$ for no $\mathbf{k}$ if we use call-by-value.
If "call-by-value" and "call-by-name" are interchanged, can a similar declaration be found?

Q4 For the language *Exp3* show that if $y \notin FVar(e)$ then $\rho[v'/y] \vdash e[y/x] \Longrightarrow v$ if and only if $\rho[v'/x] \vdash e \Longrightarrow v$.

Use this to deduce that $\rho \vdash let\ \ x = e\ \ in\ \ e' \Longrightarrow v$ if and only if $\rho \vdash let\ \ y = e\ \ in\ \ (e'[y/x]) \Longrightarrow v$ for every $y \notin FVar(e')$.

Q5 Again for the language *Exp3* prove $\rho \vdash e[v'/x] \Longrightarrow v$ if and only if $\rho[v'/x] \vdash e \Longrightarrow v$.

Q6 Generalise the previous question by showing that $\rho \vdash e[e'/x] \Longrightarrow v$ if and only if there is a value v' such that $\rho \vdash e' \Longrightarrow v'$ and $\rho[v'/x] \vdash e \Longrightarrow v$.

Q7 Prove that for the language *Exp4* $\rho \vdash If\ \ T\ \ Then\ \ e\ \ Else\ \ e' \Longrightarrow_A v$ if and only if $\rho \vdash e \Longrightarrow_A v$.

Q8 Extend the abstract syntax of *Fpl* so that user-defined functions may take a mixture of arithmetic and boolean expressions. Extend the Evaluation semantics appropriately.

Q9 In the extended language of Question 8 let D be the declaration

$$\begin{array}{l} IF(bx, y, z) \Longleftarrow If\ \ bx \\ \qquad\qquad\qquad\quad Then\ \ x\ \ Else\ \ y \end{array}$$

Is it true that for all expressions be, e, e', the terms $IF(be, e, e')$ and $If\ \ be\ \ Then\ \ e\ \ Else\ \ e'$ always yield the same results? That is, $D, \rho \vdash IF(be, e, e') \Longrightarrow_A v$ if and only if $D, \rho \vdash If\ \ be\ \ Then\ \ e\ \ Else\ \ e' \Longrightarrow_A v$?

Q10 Suppose D contains the declaration $F(x) \Longleftarrow e'$. Is it true that for every e the two expressions $F(e)$ and $let\ \ x = e\ \ in\ \ e'$ always yield the same results?

Q11 A declaration D is called *closed* if in every definition of the form $F(x_1, \ldots, x_k) \Longleftarrow e$ the body e only uses variables from the list $x_1, \ldots, x_k$. Show that Question 5 generalises to the language *Fpl*, i.e. $D, \rho \vdash e[v'/x] \Longrightarrow v$ if and only if $D, \rho[v'/x] \vdash e \Longrightarrow v$ provided D is closed.
Give a counterexample to this statement when D is not closed.

Q12 Show that the statement in Question 6 does not in general hold for the language *Fpl*.

Q13 The Evaluation rules for local declarations, given in Section 3.2, implement *static binding* or *define-time binding*. In the evaluation of $let\ \ x = e\ \ in\ \ e'$ during the evaluation of e' the value associated or bound to x is the value of e obtained using the global environment. An alternative, called *dynamic binding* is to associate with x the value of e obtained using the environment prevailing when x is required during the evaluation of e'. For example, if we evaluate

$$let\ \ x = x + y\ \ in\ \ (\ let\ \ y = \mathbf{2}\ \ in\ \ x + y)$$

with respect to an environment ρ, such that $\rho(x) = \mathbf{10}, \rho(y) = \mathbf{20}$, then using static binding we obtain **32** whereas dynamic binding yields **14**. Design an Evaluation semantics for *Exp3* which uses dynamic binding.
Hint. Expressions should not be evaluated with respect to environments but functions from *Var* to *Exp3*.

Rule CR
$$\frac{}{D, \rho \vdash \mathbf{n} \Longrightarrow_A \mathbf{n}}$$

Rule VarR
$$\frac{}{D, \rho \vdash x \Longrightarrow_A \rho(x)}$$

Rule IfR
$$\frac{D, \rho \vdash be \Longrightarrow_B T \qquad D, \rho \vdash e \Longrightarrow_A v}{D, \rho \vdash \mathit{If}\ be\ \mathit{Then}\ e\ \mathit{Else}\ e' \Longrightarrow_A v}$$

$$\frac{D, \rho \vdash be \Longrightarrow_B F \qquad D, \rho \vdash e' \Longrightarrow_A v'}{D, \rho \vdash \mathit{If}\ be\ \mathit{Then}\ e\ \mathit{Else}\ e' \Longrightarrow_A v'}$$

Rule OpR
$$\frac{D, \rho \vdash e \Longrightarrow_A v \qquad D, \rho \vdash e' \Longrightarrow_A v'}{D, \rho \vdash e\ op\ e' \Longrightarrow_A Ap(op, v, v')}$$

Rule LocR
$$\frac{D, \rho \vdash e \Longrightarrow_A v \qquad D, \rho[v/x] \vdash e' \Longrightarrow_A v'}{D, \rho \vdash \mathit{let}\ x = e\ \mathit{in}\ e' \Longrightarrow_A v'}$$

Rule FunR
$$\frac{D, \rho \vdash e_i \Longrightarrow_A v_i, 1 \leq i \leq k \qquad D, \rho[v_1/x_1, \ldots, v_k/x_k] \vdash e \Longrightarrow_A v}{D, \rho \vdash F(e_1, \ldots, e_k) \Longrightarrow_A v}$$
whenever $F(x_1, \ldots, x_k) \Longleftarrow e$ occurs in D

Figure 3.9: Evaluation semantics for *Fpl*

Chapter 4

More Languages

In this chapter we give an Evaluation semantics for a selection of different languages, each of a different style. All of the languages considered are simpler than the major popular programming languages which are in use today, such as PASCAL, FORTRAN, PL/1. Our selection is designed to emphasise the versatility of our method of giving an operational semantics while at the same time keeping the languages under investigation relatively simple; each language we examine raises particular problems which require solutions within our general framework. In the first language there are two innovations. The first is that programs no longer evaluate to values; they evaluate to finite sequence of values. The second is that programs use a primitive form of state. The second language pursues the idea of calculating sequences which may now be infinite. In the final section we examine a more standard language which works entirely with states; here a program is in effect a sequence of commands to update a state or memory. In each case we give an Evaluation semantics for the language and show its determinacy and well-definedness — if they have these properties. The precise formulation of what these mean will vary from language to language.

4.1 Using a Calculator

Consider a very simple calculator for evaluating arithmetic expressions (again!). It can do simple arithmetic calculations, print answers and it has a single memory cell. It has the following buttons:

1. *ON* — for switching on the machine
2. *OFF* — for switching off the machine
3. **n** — for each numeral **n** from **0** to **9**
4. *IF* — for making conditional computations
5. *TOTAL* — for printing the value of the expression punched in

6. *LASTANSWER* — gives the value of the previous calculation
7. $+, *, -, div$ — the usual arithmetic operators
8. (,) — for disambiguating expressions.

A typical example of the use of the calculator might be:
PRESS: *ON*
PRESS: $(\mathbf{4}+\mathbf{12}) * \mathbf{2}$
PRESS: *TOTAL* (the calculator prints **32**)
PRESS: $\mathbf{1}+$ *LASTANSWER*
PRESS: *TOTAL* (the calculator prints **33**)
PRESS: *IF LASTANSWER* **+1, 0, 2 + 4**
PRESS: *TOTAL* (the calculator prints **6**)
PRESS: *OFF*

So, to use the calculator, you first switch it on and then you punch in an expression followed by *TOTAL*. Each time *TOTAL* is pressed the value of the expression is printed on the screen. Pressing the *LASTANSWER* button gives the value previously printed. As part of the allowable expressions there is a primitive form of *IF* expression: the choice depends on whether or not the first argument evaluates to zero. In *IF* e_1, e_2, e_3 if the expression e_1 returns the value **0** then the second component e_2 is evaluated and if e_1 returns any other value then the third component e_3 is evaluated. The language for using the calculator, considered as a programming language, is different than that of the last chapter. There each program essentially evaluates to a single value, a numeral, whereas here we obtain a sequence of values displayed on the screen. We are assuming that a program for the calculator consists of a press on the *ON* button, a sequence of meaningful button pushes followed by a push on the *OFF* button. The reason we choose this example language is to demonstrate how our approach to operational semantics can be applied to languages which calculate sequences of values rather than single values.

The makers of the calculator certainly will not supply an abstract syntax for the language. One attempt at this, by Schmidt [Sch86], is given in Figure 4.1. There are three main syntactic categories, *Prog*, *Expseq* and, as usual, *Exp*. A program is simply the *ON* button followed by an *Expseq*. This is a list of expressions separated by the *TOTAL* button, and the sequence must end with the *OFF* button. Expressions are formed in the usual way with numerals and arithmetic operators but, in addition, we can use the *LASTANSWER* button. Again, we emphasise that Figure 4.1 gives the abstract syntax rather than the concrete syntax. So we ignore bracketing, etc., in arithmetic expressions. The net effect of this definition is that a program has the form

$$ON\ e_1\ TOTAL\ e_2\ TOTAL\ \ldots\ TOTAL\ e_k\ TOTAL\quad OFF.$$

1. Syntactic categories

   p in *Prog*

   se in *Expseq*

   e in *Exp*

   op in *Op*

   n in *Num*

2. Definitions

$$\begin{array}{rcl} p & ::= & ON\ \ se \\ se & ::= & e\ \ TOTAL\ \ se \mid e\ \ TOTAL\ \ \ OFF \\ e & ::= & n \mid e\ \ op\ \ e' \mid IF(e,e',e'') \mid \ LASTANSWER \\ op & ::= & + \mid - \mid * \mid \ div \end{array}$$

Figure 4.1: Abstract Syntax for *CalcL*

The intuitive meaning of this program is that the values of the sequence of expressions $e_1, \ldots, e_k$ are printed on the output device.

Let us now see how this meaning can be expressed formally using our structural operational semantics. The first point to notice is that we cannot simply use the relation $\Longrightarrow_A$ of the previous chapter to evaluate expressions. This is because of the possible presence of the *LASTANSWER* button. For example, the value of

$$2 + (3 * \ LASTANSWER \ ) * 6$$

will depend on where in the program it is evaluated, i.e. on the value of *LASTANSWER*. We could say that the value of *LASTANSWER* is supplied by an environment as used in the previous chapter. But, once more, there is a difference because, as the evaluation of a program proceeds, the value of this 'environment' changes. It is preferable to consider the machine as having a primitive state which changes as the computation proceeds; the state records the value of *LASTANSWER* and this is updated each time a *TOTAL* button is pressed. So the computation associated with the program above can be viewed as follows:

1. start up the calculator in some arbitrarily chosen state, say with the value of *LASTANSWER* equal to **0**

2. evaluate the expression e_1 with respect to the state, output the result and update the state
3. evaluate the expression e_2 with respect to the revised state, output the result and update the state
4. continue thus for each expression.

We formalise this using three different evaluation relations, one for each of the main syntactic categories, *Exp, Expseq* and *Prog*. The evaluation relation for expressions, $\Longrightarrow_A$, takes an arithmetic expression and a state (a numeral) and returns a value, once more a numeral; its type is

$$\Longrightarrow_A : < Exp, Num > \longmapsto Num.$$

The evaluation relation for *Expseq*, $\Longrightarrow_S$, takes an element of that syntactic category and a state and returns a sequence of values; its type is

$$\Longrightarrow_S : < Expseq, Num > \longmapsto Num^*.$$

Here Num^* stands for the set of all sequences of elements from *Num*. Finally, the evaluation relation for *Prog*, $\Longrightarrow_P$, simply takes a program and returns the sequence of values computed by the program; its type is

$$\Longrightarrow_P : Prog \longmapsto Num^*.$$

The definitions of the three relations $\Longrightarrow_A$, $\Longrightarrow_S$ and $\Longrightarrow_P$ are given in Figure 4.2. Most of the rules are quite obvious and are derived from the corresponding rules for the evaluation of expressions given in previous chapters. By and large they ignore the state component. This is only used in one rule, which evaluates the expression *LASTANSWER*, or gives the effect of pressing the button marked *LASTANSWER*:

$$\frac{}{(LASTANSWER, l) \ \Longrightarrow_A \ l}$$

This simply says that to obtain the value of *LASTANSWER* look in the state. As has already been pointed out, the *IF* expression does not use boolean values; instead it tests for zero. In $IF(e,e',e'')$ if e evaluates to zero then e' is evaluated and if it evaluates to anything else e'' is evaluated. This is an expedient used by the calculator manufacturer to avoid having too many types around. The definition of $\Longrightarrow_S$, which gives the semantics of sequences of expressions, uses the relation $\Longrightarrow_A$ as one might expect; to evaluate a sequence of expressions one has to know how to evaluate its individual components. There are two kinds of elements in *Expseq* and corresponding to each of these we have an inductive rule for $\Longrightarrow_S$. The first kind of element in *Expseq* has the simple form

$$e\ \ TOTAL\ \ OFF.$$

1. $\Longrightarrow_A : Exp \times Num \longmapsto Num$

Rule CR $\dfrac{}{(\mathbf{n}, l) \Longrightarrow_A \mathbf{n}}$

Rule StR $\dfrac{}{(LASTANSWER, l) \Longrightarrow_A l}$

Rule OpR $\dfrac{(e,l) \Longrightarrow_A v \quad (e',l) \Longrightarrow_A v'}{(e \ op \ e', l) \Longrightarrow_A Ap(op, v, v')}$

Rule IFR $\dfrac{(e,l) \Longrightarrow_A \mathbf{0} \quad (e',l) \Longrightarrow_A v}{(IF(e,e',e''),l) \Longrightarrow_A v}$

$\dfrac{(e,l) \Longrightarrow_A \mathbf{n},\ \mathbf{n} > 0 \quad (e'',l) \Longrightarrow_A v}{(IF(e,e',e''),l) \Longrightarrow_A v}$

2. $\Longrightarrow_S : Expseq \times Num \longmapsto Num^*$

Rule E-sR1 $\dfrac{(e,l) \Longrightarrow_A v}{(e \ \ TOTAL \ \ OFF, l) \Longrightarrow_S < v >}$

Rule E-sR2 $\dfrac{(e,l) \Longrightarrow_A v \quad (se, v) \Longrightarrow_S s}{(e \ \ TOTAL \ \ se, l) \Longrightarrow_S v.s}$

3. $\Longrightarrow_P : Prog \ \longmapsto Num^*$

Rule PrR $\dfrac{(se, \mathbf{0}) \Longrightarrow_S s}{ON \ \ se \Longrightarrow_P s}$

Figure 4.2: Operational semantics for *CalcL*

Intuitively this evaluates the expression e, outputs the result and switches off. So in a given state l this expression evaluates to a sequence of values of length 1. This is expressed in Rule E-sR1 which says

$$(e \ \ TOTAL \ \ OFF, l) \Longrightarrow_S < v >$$

where v is the value of e in the state l , i.e.

$$(e, l) \Longrightarrow_A v.$$

In this context we use $< v >$ to denote the sequence which consists of one element v. More generally, $< v_1, \ldots, v_k >$ denotes the sequence of elements consisting of v_1 followed by v_2, etc. In particular, $<>$ denotes the empty sequence. We will use s, s', etc., to denote typical sequences and finally $v.s$ denotes the sequence obtained by prefixing the sequence s with the value v: this corresponds to the *Cons* operator on lists in languages such as LISP.

The rule for the second kind of expression sequence

$$e \ \ TOTAL \ \ se$$

is slightly more complicated. It essentially says that to evaluate this expression in a given state l:

1. first evaluate the expression e in the state l to the value v, say:

   $(e, l) \Longrightarrow_A v$

2. then evaluate the sequence se in the state v, i.e. the state where *LASTANSWER* has the value v, to the sequence s:

   $(se, v) \Longrightarrow_S s$

3. from these two pieces of information we can conclude that $e \ \ TOTAL \ \ se$ in the state l evaluates to the sequence $v.s$:

   $(e \ \ TOTAL \ \ se, l) \Longrightarrow_S v.s.$

The final relation $\Longrightarrow_P$ is straightforward. The only form a program can take is

$$ON \ \ se$$

where se is in *Expseq*. To evaluate this program we simply evaluate the sequence se in the initial state. We have already decided that the initial state has $\mathbf{0}$ as the value of *LASTANSWER*. So the only rule for *Prog* is

$$ON \ \ se \Longrightarrow_P s$$

where

$$(se, \mathbf{0}) \Longrightarrow_S s.$$

We now look at an example of the use of these rules to evaluate a program. Consider the program

$$\begin{array}{ll} ON\ (\mathbf{4}+\mathbf{12})*\mathbf{2} & TOTAL\ \ \mathbf{1}+\ LASTANSWER \\ & TOTAL\ \ \ IF(\ LASTANSWER\ \ +\mathbf{1},\mathbf{0},\mathbf{2}+\mathbf{4}) \\ & TOTAL \\ & OFF \end{array}$$

To simplify the exposition we let S_1 denote $IF(LASTANSWER + \mathbf{1}, \mathbf{0}, \mathbf{2} + \mathbf{4})$ $TOTAL\ OFF$ and S_2 denote $\mathbf{1} + LASTANSWER\ \ TOTAL\ \ S_1$.

1.	$(\mathbf{1},\mathbf{33}) \Longrightarrow_A \mathbf{1}$	Rule CR
2.	$(LASTANSWER,\mathbf{33}) \Longrightarrow_A \mathbf{33}$	Rule StR
3.	$(LASTANSWER+\mathbf{1},\mathbf{33}) \Longrightarrow_A \mathbf{34}$	Rule OpR to 1, 2
4.	$(\mathbf{2},\mathbf{33}) \Longrightarrow_A \mathbf{2}$	Rule CR
5.	$(\mathbf{4},\mathbf{33}) \Longrightarrow_A \mathbf{4}$	Rule CR
6.	$(\mathbf{2}+\mathbf{4},\mathbf{33}) \Longrightarrow_A \mathbf{6}$	Rule OpR to 4, 5
7.	$(IF(LASTANSWER+\mathbf{1},\mathbf{0},\mathbf{2}+\mathbf{4}),\mathbf{33}) \Longrightarrow_A \mathbf{6}$	Rule IFR to 3, 6
8.	$(S_1,\mathbf{33}) \Longrightarrow_S <\mathbf{6}>$	Rule E-sR1 to 7
9.	$(LASTANSWER,\mathbf{32}) \Longrightarrow_A \mathbf{32}$	Rule StR
10.	$(\mathbf{1},\mathbf{32}) \Longrightarrow_A \mathbf{1}$	Rule CR
11.	$(\mathbf{1}+LASTANSWER,\mathbf{32}) \Longrightarrow_A \mathbf{33}$	Rule OpR to 9, 10
12.	$(S_2,\mathbf{32}) \Longrightarrow_S <\mathbf{33},\mathbf{6}>$	Rule E-sR2 to 8, 11
13.	$(\mathbf{4},\mathbf{0}) \Longrightarrow_A \mathbf{4}$	Rule CR
14.	$(\mathbf{12},\mathbf{0}) \Longrightarrow_A \mathbf{12}$	Rule CR
15.	$(\mathbf{2},\mathbf{0}) \Longrightarrow_A \mathbf{2}$	Rule CR
16.	$(\mathbf{4}+\mathbf{12},\mathbf{0}) \Longrightarrow_A \mathbf{16}$	Rule OpR to 13, 14
17.	$((\mathbf{4}+\mathbf{12})*\mathbf{2},\mathbf{0}) \Longrightarrow_A \mathbf{32}$	Rule OpR to 15, 16
18.	$((\mathbf{4}+\mathbf{12})*\mathbf{2}\ \ TOTAL\ \ S_2,\mathbf{0}) \Longrightarrow_S <\mathbf{32},\mathbf{33},\mathbf{6}>$	Rule E-sR2 to 12, 17
19.	$ON\,(\mathbf{4}+\mathbf{12})*\mathbf{2}\ TOTAL\ S_2$	
	$\Longrightarrow_P <\mathbf{32},\mathbf{33},\mathbf{6}>$	Rule PrR to 18.

What can we say about this semantics for *CalcL*? Because we have less intuition about this language — it is less familiar to us — it is more difficult to say that it is correct in that it captures how we think the calculator should behave. But at least we would expect every program to have a value, i.e. to generate a sequence of numerals and, moreover, a unique sequence of numerals. If we use the calculator in exactly the same manner on two different occasions we should expect exactly the same result. This is in analogy with the languages of the previous chapter. There we expected every expression in the smaller languages *Exp1* to *Exp4* to have a unique value but, in the case of *Fpl*, this was weakened to every expression having at most one value.

The proof of this result for *CalcL* is relatively straightforward. It depends on similar results for elements of the syntactic categories *Exp* and *Expseq*.

Theorem 4.1.1 *For every expression e in Exp and every state l there exists a unique numeral* $\mathbf{n}$ *such that* $(e,l) \Longrightarrow_A \mathbf{n}$.

Proof At this stage the reader should be able to prove this result without any difficulty by structural induction on e. □

Note that here we have to say that the value of an expression depends on the state, which supplies the value of *LASTANSWER*. Note also that here we are combining two results which would have been proved separately in the previous chapter, well-definedness and determinacy. They are:

1. for every e in *Exp* and every state l there exists some numeral $\mathbf{n}$ such that $(e,l) \Longrightarrow_A \mathbf{n}$
2. for every e in *Exp* and state l, if $(e,l) \Longrightarrow_A \mathbf{n}$ and $(e,l) \Longrightarrow_A \mathbf{m}$ then $\mathbf{n} = \mathbf{m}$.

Although in the previous chapter we proved these results separately, in fact their separate proofs can be combined into one proof by structural induction on expressions.

There is a corresponding result for sequences of expressions, once more parametrised on states.

Theorem 4.1.2 *For every $se \in Expseq$ and every state l there exists a unique sequence of numerals s such that* $(se,l) \Longrightarrow_S s$.

Proof This time the proof is by structural induction on elements of *Expseq*. There are two cases:

1. The base case: se has the form $e\ \ TOTAL\ \ OFF$ for some expression e. By the previous result there is a unique $\mathbf{n}$ such that $(e,l) \Longrightarrow_A \mathbf{n}$. So the unique sequence s such that $(se,l) \Longrightarrow_S s$ is the sequence $<\mathbf{n}>$.

2. The inductive case: se has the form $e\ \ TOTAL\ \ se'$ for some $se' \in Expseq$. By induction we may assume that the theorem is true for se', i.e. for any state k there is a unique sequence of numerals s'_k such that

   $$(se',k) \Longrightarrow_S s'_k.$$

   Also invoking the previous result, we know that for a given state l there is a unique numeral $\mathbf{n}$ such that

   $$(e,l) \Longrightarrow_A \mathbf{n}.$$

   The required sequence of numerals is therefore $\mathbf{n}.s'_n$. For $(e,l) \Longrightarrow_A \mathbf{n}$ and $(se',\mathbf{n}) \Longrightarrow_S s'_n$ and, by applying the rule E-sR2, we obtain

$$(se, l) \implies_S \mathbf{n}.s'_n.$$

□

The result for programs now follows since a program is simply a sequence of expressions which is evaluated in the state $\mathbf{0}$.

Theorem 4.1.3 *For every program p there exists a unique sequence of numerals s such that* $p \implies_P s$.

Proof The program p is of the form *ON se* for some $se \in Expseq$. We may apply the previous theorem to obtain a unique sequence of numerals s such that

$$(se, \mathbf{0}) \implies_S s.$$

Now, applying the rule for programs, Rule PrR, we obtain

$$p \implies_P s$$

and, moreover, this is the only sequence which p can produce. □

This result establishes that our semantics is at least consistent or well-defined. We cannot say very much else about it other than that it coincides with whatever intuition we have about how the calculator should work. Its value lies in the fact that it can replace our intuition and act as a formal description of how the calculator should behave.

4.2 A Stream Language

Programs in the language of the previous section could only produce a finite sequence of values; each program is essentially a sequence of arithmetic expressions, each of which is evaluated in turn. Here we increase the power of the language by allowing definitions as in the last section of the previous chapter. The result will be a language for printing finite or infinite sequences of values, which we call *streams* . However, to make sense of this we need to make some changes to the syntax. To keep things relatively simple we drop the presence of the state, reverting to a pure functional language. We also simplify the syntax by omitting the *TOTAL* keyword. Instead e *TOTAL* will simply be e, i.e. we assume that the evaluation of an expression leads automatically to the printing of the resulting value. We keep the simple form of the *IF* statement which does not require a separate category for booleans but, to make it more readable, we use *If* e *Then* e' *Else* e''. Finally, we add to the language the list

constructor *Cons* and the destructors *Hd* and *Tl*. Intuitively *Cons* takes a value and a list and prefixes the list with the value, thereby constructing a new list. *Hd* takes the first element from a list whereas *Tl* takes the tail of a list, i.e. the list obtained by eliminating the first element from its argument. The notation which applied in the previous section to sequences is also used for streams. So $Cons(v, s)$ is the stream $v.s$, $Hd(v.s)$ is the value v and $Tl(v.s)$ is the stream s. Note that both $Hd(<>)$ and $Tl(<>)$ are undefined.

One function declaration could be

$$NN(x) \Longleftarrow Cons(x, NN(x + \mathbf{1})).$$

If we evaluate the expression $NN(\mathbf{0})$ with respect to this declaration, we obtain the *infinite* list of numerals, i.e. a stream. Intuitively we can think of the evaluation of $NN(\mathbf{0})$ proceeding as follows:

$$\begin{array}{lll} NN(\mathbf{0}) & \implies <\mathbf{0}, NN(\mathbf{1})> & \mathbf{0} \text{ is printed} \\ & \implies <\mathbf{0}, \mathbf{1}, NN(\mathbf{2})> & \mathbf{1} \text{ is printed} \\ & \implies <\mathbf{0}, \mathbf{1}, \mathbf{2}, NN(\mathbf{3})> & \mathbf{2} \text{ is printed} \\ & \ldots & \end{array}$$

As another example, consider the definition:

$$\begin{array}{ll} FilEv(x) \Longleftarrow & If \;\; even(Hd\,x) \\ & \quad Then \;\; FilEv(Tl\,x) \\ & \quad Else \;\; Cons((Hd\,x), FilEv(Tl\,x)). \end{array}$$

Here we assume that there is a operator *even* which decides whether or not numerals are even. *FilEv* takes a list of numerals and filters out all those which are even. $FilEv(x)$ examines the first element of the list x, $Hd(x)$. If it is even it is filtered out; i.e. the resulting stream of values is obtained by processing the remainder of the list, $Tl(x)$. If $Hd(x)$ is not even then the result is $Cons(Hd\ x, FilEv(Tl\ x))$. This means that $Hd(x)$ is printed and the subsequent values of the output are obtained by processing *Tl x*. However, this will happen only if the parameter-passing mechanism is call-by-name. Call-by-value would not be of much use in this language because the value of many programs is actually an infinite stream. For example, if call-by-value were used in the evaluation of the expression $FilEv(NN(\mathbf{0}))$, then the parameter $NN(\mathbf{0})$ would have to be evaluated before *FilEv* could be applied. However, the value of $NN(\mathbf{0})$ is the infinite stream $< \mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots >$ and so *FilEv* would never be called and there would be no output produced by $FilEv(NN(\mathbf{0}))$.

The abstract syntax of our new language, *StreamL*, is given in Figure 4.3. At this stage let us not bother about enumerating the set of allowed operators. We will simply assume some reasonable set and when we come to the operational semantics we will assume that the *Apply* mechanism can interpret them correctly. They should certainly include the minimal set $+, -, *, div$, but it may also be convenient to have

1. Syntactic categories

   p in *Prog*

   D in *Dec*

   se in *StExp*

   sx in *StVar*

   e in *Exp*

   x in *Var*

   F in *FVar*

   op in *Op*

   n in *Num*

2. Definitions

$$\begin{array}{rcl}
p & ::= & < se, D > \\
D & ::= & F(sx_1, ..., sx_k) \Longleftarrow se \mid F(sx_1, ..., sx_k) \Longleftarrow se,\ D' \\
se & ::= & \mathit{If}\ e\ \mathit{Then}\ se'\ \mathit{Else}\ se'' \mid \mathit{Cons}(e, se') \mid \mathit{Tl}(se') \mid F(se_1, ..., se_k) \mid \mathit{Nil} \\
e & ::= & x \mid n \mid e\ op\ e' \mid \mathit{Hd}(se)
\end{array}$$

Figure 4.3: Abstract Syntax for *StreamL*

others. The only constraint is the interpretative power we wish to assume of the *Apply* mechanism. The main syntactic category is *Prog* and a program consists of an expression together with a declaration, $< se, D >$. As before, this declaration is supposed to provide an interpretation for the function names which appear in se. We will continue to apply the restriction on programs used in the previous chapter; every function variable appearing in the expression or in the body of a definition must itself have a unique definition in the declaration. There are two types of expression: *Exp* are the arithmetic expressions and, in addition, we have *StExp*, stream expressions. Arithmetic expressions are formed as before except that one can also form an expression by taking the head of a stream expression, $Hd(se)$. *Nil* is one kind of stream expression; it is meant to denote the empty stream $<>$. One can also form streams by the *If* and *Cons* operators or using a user-defined function.

There is some subtlety in these definitions. In $\mathit{If}\ e\ \mathit{Then}\ se_1\ \mathit{Else}\ se_2$ and $Cons(e, se)$, the first argument must be an arithmetic expression. Also function symbols may only be applied to stream expressions because in their declaration variables

Rule CR $\quad \dfrac{}{D \vdash \mathbf{n} \Longrightarrow_A \mathbf{n}}$

Rule OpR $\quad \dfrac{D \vdash e \Longrightarrow_A v \qquad D \vdash e' \Longrightarrow_A v'}{D \vdash e\ op\ e' \Longrightarrow_A Ap(op, v, v')}$

Rule HdR $\quad \dfrac{D \vdash se \overset{v}{\Longrightarrow}_S se'}{D \vdash Hd(se) \Longrightarrow_A v}$

Figure 4.4: Evaluation Semantics for *StreamL*: $\Longrightarrow_A$

from *StVar* rather than *Var* are used. Also these functional expressions represent streams rather than arithmetic expressions as they only appear in the syntactic category *StExp* and not in *Exp*. This means that user-defined functions can only take streams as arguments and also only return streams; they cannot be defined over arithmetic or boolean values. However, the arithmetic expression e can be simulated, albeit somewhat clumsily, by the stream expression $Cons(e, Nil)$. So this restriction does not have much theoretical effect on the power of the language. Therefore, apart from the omission of boolean expressions and the resulting modification of the *If* statement, the language *FpL* can be considered in some sense as a sub-language of *StreamL*.

We now give an operational semantics for programs. Again, we will have a separate evaluation relation for expressions and stream expressions, which we denote by $\Longrightarrow_A$ and $\Longrightarrow_S$ respectively. However, stream expressions may contain user-defined function names and, to make sense of these, we need a declaration. We also need declarations in order to evaluate ordinary arithmetic expressions. This is because $Hd(se)$ is an arithmetic expression and se may contain function names. For example, to evaluate the arithmetic expression $Hd(Ev(Cons(\mathbf{0}, Nil)))$ we need to know the declaration of the function name Ev. As usual we also require environments because expressions may contain variables. However, to make things less complicated we will avoid the use of environments by considering only closed expressions, i.e. those containing no free variables from *Var* or *StVar*. As usual, the value of an arithmetic expression, given a declaration, is a numeral. So the type of $\Longrightarrow_A$ is given by :

$$\Longrightarrow_A\ :\ Dec \longmapsto Exp \longmapsto Num.$$

However, as in the previous chapter, we will tend to write

$$D \vdash e \Longrightarrow_A v$$

Rule ConsR $\dfrac{D \vdash e \implies_A v}{D \vdash Cons(e, se) \overset{v}{\implies}_S se}$

Rule TlR $\dfrac{D \vdash se \overset{v}{\implies}_S se' \quad D \vdash se' \overset{v'}{\implies}_S se''}{D \vdash Tl(se) \overset{v'}{\implies}_S se''}$

Rule IfR $\dfrac{D \vdash e \implies_A \mathbf{0} \quad D \vdash se' \overset{v}{\implies}_S se'''}{D \vdash IF(e, se', se'') \overset{v}{\implies}_S se'''}$

$\dfrac{D \vdash e \implies_A \mathbf{n},\ \mathbf{n} \geq \mathbf{1} \quad D \vdash se'' \overset{v}{\implies}_S se'''}{D \vdash IF(e, se', se'') \overset{v}{\implies}_S se'''}$

Rule FunR $\dfrac{D \vdash se[se_1/sx_1, \ldots, se_k/sx_k] \overset{v}{\implies}_S se}{D \vdash F(se_1, \ldots, se_k) \overset{v}{\implies}_S se}$
whenever $F(sx_1, \ldots, sx_k) \Longleftarrow se$
occurs in D

Figure 4.5: Evaluation Semantics for *StreamL*: $\overset{v}{\implies}_S$

to mean that the expression e evaluates to v under the assumption that the function variables in e are defined in the declaration D. The relation $\implies_A$ is defined in Figure 4.4. The only interesting rule is Rule HdR for evaluating expressions of the form $Hd(se)$. This depends on the evaluation of se and, to understand it, we must see how stream expressions are evaluated.

The technique of the previous section cannot be used because streams may be infinite. For example, we cannot say that to evaluate $Cons(e, se)$ you first evaluate e to v then evaluate se to the stream s and the result of evaluating $Cons(e, se)$ is the stream $v.s$; the stream expression se may denote an infinite stream in which case this rule would never get applied. Intuitively we do expect the value of $Cons(e, se)$ to be the possibly infinite stream $e.se$. But let us look more closely at what we expect to see when we try to evaluate $Cons(e, se)$. We expect the arithmetic expression e to be evaluated, the resulting value to be printed and the evaluation procedure to continue with the analysis of the stream expression se. From this point of view applying $\implies_S$ to a stream expression gives a pair as a result; the first element is a value, i.e. a numeral to be printed, and the second element is another stream expression, embodying the remainder of the result. For example, it will follow from our definition

of $\implies_S$ that

$$D \vdash Cons(\mathbf{1}, se) \implies_S < \mathbf{1}, se > .$$

Here the eventual result of evaluating the stream expression $Cons(\mathbf{1}, se)$ is a stream, the first element of which is $\mathbf{1}$, which is printed immediately, and the remainder is the result of evaluating the stream-expression se. However, to emphasise our view of values being printed out as the stream is being produced, instead of writing

$$D \vdash Cons(\mathbf{1}, se) \implies_S < \mathbf{1}, se >$$

we write

$$D \vdash Cons(\mathbf{1}, se) \overset{\mathbf{1}}{\implies}_S se.$$

More generally,

$$D \vdash se \overset{v}{\implies}_S se'$$

may be thought of as: the stream expression se, using the declaration D to interpret function names, produces the value v and also produces the residual se'. In other words, the first value in the stream associated to se is v. To find out about subsequent values we must apply the definition of $\implies_S$ to se'. If it turns out that

$$D \vdash se' \overset{v'}{\implies}_S se''$$

then we know that the stream associated with se has the form $< v, v', \ldots >$ with subsequent values obtainable from the new residual se''.

Of course, it may be that we cannot apply the rules to continue with the evaluation of se'. In this case the resulting value of the expression se is the stream $< v >$, i.e. a stream of length one. This can happen; for example, when applying the rules to $Cons(\mathbf{1}, Nil)$ we obtain

$$D \vdash Cons(\mathbf{1}, Nil) \overset{\mathbf{1}}{\implies}_S Nil$$

and there is no rule which applies to Nil. So the entire stream produced by $Cons(\mathbf{1}, Nil)$ is $< \mathbf{1} >$. We can apply the same argument to $Cons(\mathbf{1}, F(\mathbf{0}))$ if the declaration D contains the definition $F(x) \Longleftarrow F(x)$ as will be explained later.

Let us sum up on the evaluation of stream expressions. The type of $\implies_S$ is

$$\implies_S \ : \ Dec \longmapsto StExp \longmapsto \langle Num, StExp \rangle.$$

This in turn leads to a derived relation $\overset{v}{\implies}_S$ for each v in Num of type

$$\overset{v}{\implies}_S : Dec \longmapsto StExp \longmapsto StExp$$

i.e. we write $D \vdash se \overset{v}{\implies}_S se'$ as a more graphic rendition of $D \vdash se \implies_S (v, se')$. The inductive definition of $\overset{v}{\implies}_S$ is given in Figure 4.5. The discussion we have gone through should make these rules understandable. For example Rule FunR is taken directly from the previous chapter and, as we have already pointed out, is a call-by-name parameter-passing mechanism. The rule for evaluating $Tl\ se$ should also be intuitively clear. If the first element in the stream corresponding to se is v and its residual is se' then intuitively $Tl\ se$ corresponds to se'. So to calculate the first element of the stream corresponding to $Tl\ se$ and its residual, it is sufficient to calculate them instead for se'. This is the import of Rule TlR.

Let us now look at an example of the use of these axioms to evaluate an expression. Suppose D contains the declaration

$$\begin{aligned} FilEv(sx) \Longleftarrow\ & If\ \ even(Hd\ sx) \\ & Then\ \ FilEv(Tl\ sx) \\ & Else\ \ Cons((Hd\ sx), FilEv(Tl\ sx)). \end{aligned}$$

We look at the evaluation of the expression

$$FilEv(Cons(\mathbf{0}, Cons(\mathbf{1}, Nil))).$$

In order to make the derivation more readable we let $[\mathbf{1}]$ denote $Cons(\mathbf{1}, Nil)$, $[\mathbf{0},\mathbf{1}]$ denote $Cons(\mathbf{0}, [\mathbf{1}])$ and se the body of the definition of $FilEv$, i.e.

$$\begin{aligned} & If\ \ even(Hd\ sx) \\ & \ Then\ \ FilEv(Tl\ sx) \\ & \ Else\ \ Cons((Hd\ sx, FilEv(Tl\ sx)). \end{aligned}$$

In the derivation below we will substitute stream expressions for the variable sx in se. For example, $se[[\mathbf{0},\mathbf{1}]/\ sx\ ]$ represents the result of substituting $Cons(\mathbf{0}, Cons(\mathbf{1}, Nil))$ for sx in the body of $FilEv$.

1.	$[\mathbf{0},\mathbf{1}] \xRightarrow{\mathbf{0}}_S [\mathbf{1}]$	Rule ConsR
2.	$Hd([\mathbf{0},\mathbf{1}]) \Longrightarrow_A \mathbf{0}$	Rule HdR to 1
3.	$even(Hd\ [\mathbf{0},\mathbf{1}]) \Longrightarrow_A \mathbf{0}$	rule for $even$ to 1
4.	$[\mathbf{1}] \xRightarrow{\mathbf{1}}_S Nil$	Rule ConsR
5.	$Tl\ [\mathbf{0},\mathbf{1}] \xRightarrow{\mathbf{1}}_S Nil$	Rule TlR to 1, 4
6.	$Hd(Tl\ [\mathbf{0},\mathbf{1}]) \Longrightarrow_A \mathbf{1}$	Rule HdR to 5
7.	$even(Hd(Tl\ [\mathbf{0},\mathbf{1}])) \Longrightarrow_A \mathbf{1}$	rule for $even$ to 6
8.	$Cons(Hd\ Tl[\mathbf{0},\mathbf{1}], FilEv(Tl\ Tl\ [\mathbf{0},\mathbf{1}])) \xRightarrow{\mathbf{1}}_S$ $FlEv(Tl\ Tl\ [\mathbf{0},\mathbf{1}])$	Rule ConsR to 6
9.	$se[Tl[\mathbf{0},\mathbf{1}]/sx\ ] \xRightarrow{\mathbf{1}}_S FilEv(Tl\ Tl\ [\mathbf{0},\mathbf{1}])$	Rule IfR to 7, 8
10.	$FilEv(Tl[\mathbf{0},\mathbf{1}]) \xRightarrow{\mathbf{1}}_S FilEv(Tl\ Tl\ [\mathbf{0},\mathbf{1}])$	Rule FunR to 9
11.	$se[[\mathbf{0},\mathbf{1}]/sx\ ] \xRightarrow{\mathbf{1}}_S FilEv(Tl\ Tl\ [\mathbf{0},\mathbf{1}])$	Rule IfR 3, 10
12.	$FilEv([\mathbf{0},\mathbf{1}]) \xRightarrow{\mathbf{1}}_S FilEv(Tl\ Tl\ [\mathbf{0},\mathbf{1}])$	Rule FunR to 11.

This means that the first value to be printed when $FilEv$ is called with the parameter $[\mathbf{0},\mathbf{1}]$ is $\mathbf{1}$ and its subsequent values are obtained from $FilEv(Tl(Tl\ [\mathbf{0},\mathbf{1}]))$.

With some patience one can check that $FilEv(Tl(Tl([\mathbf{0},\mathbf{1}])))$ will never again produce a value, essentially because $Tl(Tl([\mathbf{0},\mathbf{1}]))$ is Nil.

The language as it stands is virtually impossible to use. Its serious defects include the lack of the syntactic category *Bool* and the inability to use recursion directly over arithmetic expressions. We have omitted these so as to concentrate on the novel aspects of the language, namely the production of streams. Notice also that the declaration of *NN* discussed informally at the beginning of this section is not definable in the language. This is because it is a declaration of a function which takes as arguments numerals and returns a stream; in our language we are only allowed definitions of functions which take as arguments streams and return streams. As we have already indicated we can simulate these kinds of functions by representing the arithmetic expression e as the stream expression $Cons(e, Nil)$. A function similar in behaviour to *NN* could be defined by

$$NNst(sx) \Longleftarrow Cons(Hd\ \ sx, NNst(Cons(Hd\ \ sx + \mathbf{1}, Tl\ \ sx))).$$

The behaviour of $NN(\mathbf{0})$ is then simulated by the stream expression $NNst(Cons(\mathbf{0}, Nil))$. However, this simulation is rather clumsy and it would be interesting to design an extension of *StreamL* which overcomes all of these defects. This would involve introducing the syntactic category *Bool*, with appropriate definitions, replacing the syntactic category *Dec* with at least three new categories, *AADec*, *SSDec* and *ASDec*, and finally augmenting the definitions associated with the syntactic categories *Exp* and *StExp*. *AADec* would contain definitions of functions whose arguments and results are both numerals, *SSDec* would correspond to the declarations in the present language while *ASDec* would contain declarations of functions whose arguments are numerals and results streams. In the new language one would then be able to write relatively natural programs, including *NN* discussed above, and the language *Fpl* would also be a natural sub-language.

As it stands a program in the language is designed to produce a stream of values. However, because of the function definitions, it is easy to define programs which will not produce a stream or even a part of a stream. We can use an example similar to that used in Section 3.4:

$$F(sx) \Longleftarrow F(sx).$$

No matter what F is applied to, it will not produce any value whatsoever. This is reflected in our rules by the fact that, given any stream expression se, there is no pair $< v, se' >$ such that

$$F(se) \xRightarrow{v}_S se'.$$

This can be formally proved much in the same way as in the previous chapter. However, in this language there is an even simpler example of a program which produces no value, namely the expression Nil. If we examine the rules for stream expressions we see that none can be applied to Nil. So, trivially there is no $< v, se >$ such that

$$Nil \xRightarrow{v}_S se.$$

Intuitively *Nil* represents a terminated or deadlocked program. It has nothing further to produce. On the other hand, a program such as

$$I(sx) \Longleftarrow \mathit{If}\ F\ \mathit{Then}\ \mathit{Nil}\ \mathit{Else}\ \mathit{Tl}(I(sx))$$

gives rise to other intuitive connotations. For example, when applied to $NN(\mathbf{0})$, assuming we have augmented our language to include such definitions, one feels that $I(NN(\mathbf{0}))$ performs a considerable amount of computation. It simply never produces a value; for no $< v, se >$ can we show

$$I(NN(\mathbf{0})) \stackrel{v}{\Longrightarrow}_S se.$$

At the level of our semantics both *Nil*, a program which has terminated, and $I(NN(\mathbf{0}))$ have the same behavioural characteristics. However, if we were to develop a less abstract Computation semantics for our language, for example, the one-step semantics discussed in Chapter 2, then we would expect to see a difference. Here there would be no program or residual C such that

$$Nil \longrightarrow C$$

but there would be residuals such that

$$I(NN(\mathbf{0})) \longrightarrow C.$$

In fact, we would expect to see infinite one-step computations

$$I(NN(\mathbf{0})) \longrightarrow I(Tl\ NN(\mathbf{0})) \longrightarrow I(Tl\ Tl\ NN(\mathbf{0}))\ldots \longrightarrow \ldots$$

We have just seen that some programs may produce no values whatsoever. Other programs produce an infinite sequence of values $v_1, v_2, \ldots$. The simplest example (in the extended language proposed above) is $NN(\mathbf{0})$, where NN is defined by

$$NN(x) \Longleftarrow Cons(x, NN(x + \mathbf{1})).$$

This program produces the infinite sequences $\mathbf{0}, \mathbf{1}, \ldots$. More formally, if D represents the declaration given above, then there is an infinite sequence of stream expressions se_i such that

$$\begin{array}{llllll} NN(0) \stackrel{\mathbf{0}}{\Longrightarrow}_S & se_1 & & & & \\ & se_1 \stackrel{\mathbf{1}}{\Longrightarrow}_S & se_2 & & & \\ & & se_2 \stackrel{\mathbf{2}}{\Longrightarrow}_S & se_3 & & \\ & & & \ldots & & \end{array}$$

There are also programs which can only produce a finite sequence of values. A simple example is

$$Cons(\mathbf{0}, Nil).$$

The only possible move from this is

$$Cons(\mathbf{0}, Nil) \stackrel{\mathbf{0}}{\Longrightarrow}_S Nil.$$

Since *Nil* can produce no value, this program can therefore only produce a sequence of values of length 1. In this case the reason for the absence of further values is the residual *Nil*. Another reason could be the production of a residual which diverges, such as $I(x)$ above.

To end this section we consider the proof of the consistency or the well-definedness of our semantics, restricted to the original language whose abstract syntax appears in Figure 4.3. Consider a sequence of values $v_1, v_2, \ldots$, which may be finite, infinite or even empty. We say that a stream expression *se produces* this sequence of values, with respect to a declaration D, if there is a corresponding sequence of stream expressions $se_1, se_2, \ldots$ such that

$$\begin{array}{llllll} se \stackrel{v_1}{\Longrightarrow}_S & se_1 & & & & \\ & se_1 \stackrel{v_2}{\Longrightarrow}_S & se_2 & & & \\ & & se_2 \stackrel{v_3}{\Longrightarrow}_S & se_3 & & \\ & & & \ldots & & \end{array}$$

Note that if the sequence of values is empty, this means that se produces no value whatsoever. We will prove that for every declaration D and every stream expression se there is a *unique* sequence of values s such that se produces s with respect to D. This is not as difficult as it sounds. The main point is to prove that if a stream expression produces a value, then not only will this value be unique but the resulting residual will also be unique. For convenience we will always assume the presence of some appropriate declaration D.

Theorem 4.2.1 *If* $se \stackrel{v}{\Longrightarrow}_S se_1$ *and* $se \stackrel{v'}{\Longrightarrow}_S se_2$, *then* $v = v'$ *and* $se_1 = se_2$.

Proof Unfortunately we cannot prove this by structural induction on se because of the function declarations in the language. So instead we take advantage of the inductive definition of $\Longrightarrow_S$ and use Rule induction. The proof method has already been explained in Theorem 3.4.2. In fact, since $\Longrightarrow_S$ and $\Longrightarrow_A$ are defined in terms of each other, we must also formulate an inductive hypothesis for $\Longrightarrow_A$. Let $P_S(se, se', v)$ denote the property

$$D \vdash se \stackrel{v'}{\Longrightarrow}_S se'' \text{ implies } v = v' \text{ and } se' = se''$$

and $P_A(e, v)$ denote the property

$$D \vdash e \Longrightarrow_A v' \text{ implies } v = v'.$$

We show that P_S, P_A satisfy the defining rules of $\Longrightarrow_S$ and $\Longrightarrow_A$. Since the latter are the least such relations it will follow that

$$D \vdash e \Longrightarrow_A v \text{ implies } P_A(e, v)$$

i.e.

$$D \vdash e \Longrightarrow_A v \text{ and } D \vdash e \Longrightarrow_A v' \text{ implies } v = v'.$$

But, more importantly,

$$D \vdash se \overset{v}{\Longrightarrow}_S se' \text{ implies } P_S(se, se', v)$$

i.e.

$$D \vdash se \overset{v}{\Longrightarrow}_S se' \text{ and } D \vdash se \overset{v'}{\Longrightarrow}_S se'' \text{ implies } se' = se'' \textit{ and } v = v'.$$

So we must check that P_S, P_A satisfy the seven conditions which constitute Rules ConsR, TlR, IfR, FunR, CR, OpR and HdR. As examples we will consider only two, Rule FunR and Rule HdR. So suppose $P_S(se[se_1/sx_1, \ldots, se_k/sx_k], se', v)$. We must show $P_S(F(se_1, \ldots, se_k), se', v)$ assuming that the declaration

$$F(sx_1, \ldots, sx_k) \Longleftarrow se$$

is in D. So suppose $D \vdash F(se_1, \ldots, se_k) \overset{v'}{\Longrightarrow}_S se''$. We must show $se' = se''$ and $v = v'$. Now the only way to prove $D \vdash F(se_1, \ldots, se_k) \overset{v'}{\Longrightarrow}_S se''$ is to use Rule FunR. So it must be the case that

$$D \vdash se[se_1/sx_1, \ldots, se_k/sx_k] \overset{v'}{\Longrightarrow}_S se''.$$

However, now we can apply the assumption $P_S(se[se_1/sx_1, \ldots, se_k/sx_k], se', v)$ to obtain the required $se' = se''$ and $v = v'$. This means that the condition for the Rule FunR is satisfied.

Now let us consider the other example case, Rule HdR. So we assume $P_S(se, se', v)$ and we must prove $P_A(Hd\ se, v)$. Suppose that $D \vdash Hd\ se \Longrightarrow_A v'$. We have to show that $v = v'$. Once more the only way of proving this is by using an application of Rule HdR. Therefore it must be the case that $D \vdash se \overset{v'}{\Longrightarrow}_S se''$ for some se''. But now we may apply the assumption $P_S(se, se', v)$ to obtain $se' = se''$, which is not of interest, but also $v = v'$, which is what we are required to prove. □

We have called this semantic description of *StreamL* an Evaluation semantics although it could also be argued that it is a Computation semantics. We have not given, nor does there exist, a precise description of what constitutes an Evaluation semantics or a Computation semantics. But intuitively the former defines entire or complete computations from programs to results, whereas the latter defines single

steps of computations, and thereby only defines complete steps indirectly. In *StreamL* these concepts are a little blurred. The final result of evaluating a stream expression is often an infinite stream and therefore the relationship between stream expression and final results cannot be easily defined inductively. Instead we have defined the partial evaluation of stream expressions to values to be immediately printed and residual expressions. This partial evaluation could be viewed as one step of the complete evaluation of the stream expression and in this way our semantics could be construed as a Computation semantics. We have chosen to call it an Evaluation semantics because each partial evaluation may be further analysed into a sequence of more primitive operations which would give rise to a lower-level Computation semantics. However, this example language emphasises that the terms Computation semantics and Evaluation semantics can be applied at different levels of abstraction and the distinction between them is not always very clear.

4.3 An Imperative Language

As a final example language in this chapter we consider a simple imperative programming language based on the While statement from languages such as PASCAL and ALGOL. Here a program acts on a store or memory, which can be taken to be a collection of locations or addresses in each of which is stored a value. A program is simply a sequence of commands and each command modifies the store in some way. The simplest command is the assignment statement:

$$x := e.$$

The effect of this command on a store is to replace the existing value stored in the location x by the value of the expression e; it transforms one store into a new store.

The abstract syntax of the language, *WhileL*, is given in Figure 4.6. For the sake of variety we reintroduce the category of boolean expressions but we omit the details of the exact collection of boolean operators (and of arithmetic operators). Excluding these there are four main syntactic categories. The principal one is *Prog* and a program is simply a command. A command can either be an assignment statement, a command with no effect called *skip*, a sequence of commands, a conditional command which uses the *If* statement or, finally, a *While* command. Intuitively *While be Do C* means perform the command C repeatedly so long as the boolean expression *be* remains true. For example, the following program implements multiplication using addition:

$$\begin{array}{l} z := \mathbf{0}; \\ \textit{While } \ Not(Equal(x, \mathbf{0})) \ \textit{ Do} \end{array}$$

1. Syntactic categories

   p in *Prog*

   C in *Com*

   e in *Exp*

   be in *BExp*

   x in *Var*

   bx in *BVar*

   op in *Op*

   bop in *BOp*

   n in *Num*

2. Definitions

$$
\begin{array}{rcl}
p & ::= & C \\
C & ::= & skip \mid x := e \mid C' ; C'' \mid If\ be\ Then\ C'\ Else\ C'' \mid \\
 & & While\ be\ Do\ C' \\
e & ::= & x \mid n \mid e\ op\ e' \\
be & ::= & bx \mid T \mid F \mid be'\ bop\ be'' \mid Not\ be \mid Equal(e, e')
\end{array}
$$

Figure 4.6: Abstract Syntax for *WhileL*

$$z := z + y; x := x - \mathbf{1}.$$

When the program terminates the value stored in x is $\mathbf{0}$, the value stored in y remains unchanged and that in z is y times the original value of x.

The Evaluation semantics of *WhileL* is given in terms of the three relations, $\Longrightarrow_A$, $\Longrightarrow_B$ and $\Longrightarrow_C$, one for each of the syntactic categories *Exp*, *BExp* and *Com*. Strictly speaking we also need a relation for programs but since these are simply commands we can use $\Longrightarrow_C$ to evaluate them. To make sense of the variables we need an environment which assigns values to them. However, in this language variables play a very different role than those in *FpL*. Here they represent locations in a memory or store and a computation proceeds by modifying this store. In effect a program is a sequence of commands for modifying the store. In contrast a program in *Fpl* simply represents a value. The exact value may depend on an environment which gives a meaning to the free variables which may be present in the program.

But a program in *Fpl* can in no way be construed as a sequence of commands for updating its environment. The language *CalcL* used a simple store with only one location . But *WhileL* requires a store which potentially has a location associated with each variable. Mathematically environments and stores are the same kind of objects, namely functions from *Var* to *Num*. However, to emphasise their different roles we introduce a new notation: we use *Store* to denote the set of all stores, i.e. functions from *Var* to *Num*. Typical elements of *Store* are denoted by s, s' and we use the update notation originally introduced for environments: $s[v/x]$ is the same store as s except that the value v is now at location x. We will assume that the store preserves types in that it stores numerals in the locations associated with ordinary arithmetic variables and boolean values (T or F) in the locations associated with boolean variables. Note that according to the syntax of the language we can only assign new values to arithmetic locations because the only form of assignment statement is $x := e$ where x is is an arithmetic variable. There is no inherent reason for this other than to keep the language fairly small.

The evaluation relation $\Longrightarrow_A$ takes a pair consisting of an expression and a store and returns a value, the result of evaluating the expression with respect to the store. It is essentially the same as evaluating expressions with respect to an environment as their evaluation does not affect the store. However, one could easily conceive of languages where these evaluations could have as side-effects modifications of the store. The type of $\Longrightarrow_A$ is given by

$$\Longrightarrow_A : \langle Exp, Store \rangle \longmapsto Num.$$

The definition of $\Longrightarrow_A$ is given in Figure 4.7 and is straightforward. Similarly the type of $\Longrightarrow_B$ is given by

$$\Longrightarrow_B : \langle BExp, Store \rangle \longmapsto \{T, F\}$$

and its definition is equally straightforward. The evaluation relation for commands, $\Longrightarrow_C$, takes a pair consisting of a command and a store, often called a *configuration*, and returns a store; intuitively a command takes a store and returns a modified store. So the type of $\Longrightarrow_C$ should be

$$\Longrightarrow_C : \langle Com, Store \rangle \longmapsto Store.$$

However, with the language *WhileL* "evaluating" a command actually means running it. So $(C, s) \Longrightarrow_C s'$ means that when we *execute* the command C with respect to the store s it eventually halts and the resulting store is s'.

Let us look at the different clauses in its definition. The command $x := e$ simply updates the value stored at location x. So $(x := e, s) \Longrightarrow_C s[v/x]$ where v is the value of e, i.e. $(e, s) \Longrightarrow_A v$. The command skip has no effect and therefore

$$(skip, s) \Longrightarrow_C s.$$

$\Longrightarrow_A : \langle Exp, Store \rangle \longmapsto Num$

Rule CR $\quad \dfrac{}{(\mathbf{n}, s) \Longrightarrow_A \mathbf{n}}$

Rule VarR $\quad \dfrac{}{(x, s) \Longrightarrow_A s(x)}$

Rule OpR $\quad \dfrac{(e, s) \Longrightarrow_A v \quad (e', s) \Longrightarrow_A v'}{(e \; op \; e', s) \Longrightarrow_A Ap(op, v, v')}$

$\Longrightarrow_B : \langle BExp, Store \rangle \longmapsto \{T, F\}$

Rule CR $\quad \dfrac{}{(T, s) \Longrightarrow_B T} \qquad \dfrac{}{(F, s) \Longrightarrow_B F}$

Rule VarR $\quad \dfrac{}{(bx, s) \Longrightarrow_B s(bx)}$

Rule OpR $\quad \dfrac{(be, s) \Longrightarrow_B bv \quad (be', s) \Longrightarrow_B bv'}{(be \; bop \; be', s) \Longrightarrow_B Ap(bop, bv, bv')}$

Rule EqR $\quad \dfrac{(e, s) \Longrightarrow_A v \quad (e', s) \Longrightarrow_A v}{(Equal(e, e'), s) \Longrightarrow_B T} \qquad \dfrac{(e, s) \Longrightarrow_A v \quad (e', s) \Longrightarrow_A v'}{(Equal(e, e'), s) \Longrightarrow_B F}$
if v is different from v'

Rule Not $\quad \dfrac{(be, s) \Longrightarrow_B T}{(Not \; be, s) \Longrightarrow_B F} \qquad \dfrac{(be, s) \Longrightarrow_B F}{(Not \; be, s) \Longrightarrow_B T}$

Figure 4.7: Evaluation semantics for *WhileL*: $\Longrightarrow_A$, $\Longrightarrow_B$

Rule AsR $\quad \dfrac{(e, s) \Longrightarrow_A v}{(x := e, s) \Longrightarrow_C s[v/x]}$

Rule SkipR $\quad \dfrac{}{(skip, s) \Longrightarrow_C s}$

Rule IfR $\quad \dfrac{(be, s) \Longrightarrow_B T \quad (C, s) \Longrightarrow_C s'}{(If \; be \; Then \; C \; Else \; C', s) \Longrightarrow_C s'}$

$$\dfrac{(be, s) \Longrightarrow_B F \quad (C', s) \Longrightarrow_C s'}{(If \; be \; Then \; C \; Else \; C', s) \Longrightarrow_C s'}$$

Rule ComR $\quad \dfrac{(C, s) \Longrightarrow_C s' \quad (C', s') \Longrightarrow_C s''}{(C; C', s) \Longrightarrow_C s''}$

Rule WhileR $\quad \dfrac{(be, s) \Longrightarrow_B F}{(While \; be \; Do \; C, s) \Longrightarrow_C s}$

$$\dfrac{(be, s) \Longrightarrow_B T \quad (C; While \; be \; Do \; C, s) \Longrightarrow_C s'}{(While \; be \; Do \; C, s) \Longrightarrow_C s'}$$

Figure 4.8: Evaluation semantics for *WhileL*: $\Longrightarrow_C$

The compound command $C; C'$ applied to a store s first performs the command C to s to obtain s', say, and then performs C' in this new store to obtain the store s''. So from $(C, s) \Longrightarrow_C s'$ and $(C', s') \Longrightarrow_C s''$ we can conclude that the result of applying $C'; C''$ to the store s is the store s'', i.e. $(C; C', s) \Longrightarrow_C s''$.

We have already seen how *If* works so let us consider the final kind of command, *While be Do C*. To see how this works we consider two cases. If the value of *be* in the store s is F then this command does nothing because C should only be performed until *be* becomes false; if it is false already then there is no need to perform C at all. This is the justification for the first part of the *While* rule:

$$\dfrac{(be, s) \Longrightarrow_B F}{(While \; be \; Do \; C, s) \Longrightarrow_C s}$$

On the other hand, if *be* is true we know that the command C is performed at least once. After performing C the boolean expression *be* is once more tested in the new store to see if the computation is to continue. A convenient way of saying exactly what happens after C is simply to say that the command *While be Do C* is executed once more in the new state. In other words, if *be* is true we execute C and then execute the entire command again. This has the same effect as executing the command C; *While be Do C* with the original store. This justifies the second part of the *While* rule:

$$\frac{\begin{array}{c}(be\,,s) \Longrightarrow_B T\\ (C;\ While\ \ be\ \ Do\ \ C, s) \Longrightarrow_C s'\end{array}}{(While\ \ be\ \ Do\ \ C, s) \Longrightarrow_C s'}$$

Alternatively we could write this as

$$\frac{\begin{array}{c}(be\,,s) \Longrightarrow_B T, (C,s) \Longrightarrow_C s''\\ (While\ \ be\ \ Do\ \ C, s'') \Longrightarrow_C s'\end{array}}{(While\ \ be\ \ Do\ \ C, s) \Longrightarrow_C s'}$$

Both these rules gives exactly the same effect because of the manner in which we evaluate the composition of commands $C_1; C_2$. The complete definition of $\Longrightarrow_C$ is given in Figure 4.8.

Let us now look at an example of the application of these rules to find the meaning of a particular program. We will consider the execution of the simple program

$$\begin{array}{l} z := \mathbf{0};\ While\ \ Not\ \ (Equal(x,\mathbf{0}))\ \ Do\\ \qquad\qquad\qquad\qquad\qquad\qquad z := z + y;\\ \qquad\qquad\qquad\qquad\qquad\qquad x := x - \mathbf{1}\end{array}$$

with respect to a store s such that $s(x) = \mathbf{2}, s(y) = \mathbf{3}$ and $s(z) = \mathbf{7}$. This program calculates $x * y$ by successive addition and places the result in z. For convenience we let W denote the *While* statement which makes up the main part of the program and let C denote its body, namely $z := z + y; x := x - \mathbf{1}$. We will also require notation for new stores which occur as the evaluation proceeds. In general we use $s_{i,j}$ to denote the store modified so that **i** is stored in x and **j** in z. So $s_{i,j}$ is $s[\mathbf{i}/x][\mathbf{j}/z]$. Note that for every i, j $s_{i,j}(y) = \mathbf{3}$. We will actually only use the stores $s_{2,0}, s_{1,3}, s_{2,3}, s_{1,6}, s_{0,6}$. Once more we emphasise to the reader that it is better to try to read this derivation bottom-up rather than from the top-down.

1.	$(\mathbf{0}, s_{0,6}) \Longrightarrow_A \mathbf{0}$	Rule CR
2.	$(x, s_{0,6}) \Longrightarrow_A \mathbf{0}$	Rule VarR
3.	$(Equal(x,\mathbf{0}), s_{0,6}) \Longrightarrow_B T$	Rule EqR to 1, 2
4.	$(Not\ \ Equal(x,\mathbf{0}), s_{0,6}) \Longrightarrow_B F$	Rule NotR to 3
5.	$(W, s_{0,6}) \Longrightarrow_C s_{0,6}$	Rule WhileR to 4
6.	$(y, s_{1,3}) \Longrightarrow_A \mathbf{3}$	Rule VarR
7.	$(z, s_{1,3}) \Longrightarrow_A \mathbf{3}$	Rule VarR

8.	$(z + y, s_{1,3}) \Longrightarrow_A \mathbf{6}$	Rule OpR to 6, 7
9.	$(z := z + y, s_{1,3}) \Longrightarrow_C s_{1,6}$	Rule AsR to 8
10.	$(\mathbf{1}, s_{1,6}) \Longrightarrow_A \mathbf{1}$	Rule CR
11.	$(x, s_{1,6}) \Longrightarrow_A \mathbf{1}$	Rule VarR
12.	$(x - \mathbf{1}, s_{1,6}) \Longrightarrow_A \mathbf{0}$	Rule OpR to 10, 11
13.	$(x := x - \mathbf{1}, s_{1,6}) \Longrightarrow_C s_{0,6}$	Rule AsR to 12
14.	$(C, s_{1,3}) \Longrightarrow_C s_{0,6}$	Rule ComR to 9, 13
15.	$(C; W, s_{1,3}) \Longrightarrow_C s_{0,6}$	Rule ComR to 14, 5
16.	$(\mathbf{0}, s_{1,3}) \Longrightarrow_A \mathbf{0}$	Rule CR
17.	$(x, s_{1,3}) \Longrightarrow_A \mathbf{1}$	Rule VarR
18.	$(Equal(x,\mathbf{0}), s_{1,3}) \Longrightarrow_B F$	Rule EqR to 16, 17
19.	$(Not\ \ Equal(x,\mathbf{0}), s_{1,3}) \Longrightarrow_B T$	Rule NotR to 18
20.	$(W, s_{1,3}) \Longrightarrow_C s_{0,6}$	Rule WhileR to 19, 15
21.	$(z := z + y, s_{2,0}) \Longrightarrow_C s_{2,3}$	Rules VarR, OpR and AsR
22.	$(x := x - 1, s_{2,3}) \Longrightarrow_C s_{1,3}$	by the same sequence of rules
23.	$(C, s_{2,0}) \Longrightarrow_C s_{1,3}$	Rule ComR to 22, 21
24.	$(C; W, s_{2,0}) \Longrightarrow_C s_{0,6}$	Rule ComR to 23, 20
25.	$(Equal(x,\mathbf{0}), s_{2,0}) \Longrightarrow_B F$	Rules VarR, CR and OpR again
26.	$(Not\ \ Equal(x,\mathbf{0}), s_{2,0}) \Longrightarrow_B T$	Rule NotR to 25
27.	$(W, s_{2,0}) \Longrightarrow_C s_{0,6}$	Rule WhileR to 26, 24
28.	$(z := \mathbf{0}, s) \Longrightarrow_C s_{2,0}$	Rule AsR
29.	$(z := \mathbf{0}; W, s) \Longrightarrow_C s_{0,6}$	Rule ComR to 28, 27.

The net result is that if we run the program in the store s it eventually halts with the new store $s_{0,6}$ where the location z now contains the value **6** and x the value **0**.

The consistency or well-definedness of this semantics for the language *WhileL* means that any command, starting in a given state, should halt in a unique state. Because of the *While* statement it is easy to construct programs which intuitively should never halt. One simple example is the command, *Loop*,

$$While\ \ T\ \ Do\ \ skip.$$

In our semantics the non-halting of this program starting from a state s is reflected in the fact that for no state s' can we deduce that

$$(Loop, s) \Longrightarrow_C s'.$$

How, for example, could we make such a deduction? We would have to apply Rule WhileR, in which case we would have to prove

$$(skip; Loop, s) \Longrightarrow_C s'.$$

Now, in order to deduce this, we would have to apply Rule ComR. Since $(skip, s) \Longrightarrow_C s$, we would need to deduce $(Loop, s) \Longrightarrow_C s'$ before Rule ComR could be applied.

So the only way to deduce

$$(Loop, s) \Longrightarrow_C s'$$

is to have a deduction of it already. In other words, it can never be deduced.

Given a command and a store, the most we could therefore show is that when started in this store if it halts then it halts in a unique store:

Theorem 4.3.1 *If* $(C, s) \Longrightarrow_C s'$ *and* $(C, s) \Longrightarrow_C s''$ *then* $s' = s''$.

Because of Rule WhileR this cannot be proved by structural induction on terms. Instead Rule induction obtained from the inductive definition of $\Longrightarrow_C$ must be used. So we need to formulate a predicate $P(C, s)$ which satisfies all the rules defining $\Longrightarrow_C$ and which, in turn, implies the theorem. We have already seen two examples of such predicates and therefore the proof is left to the reader.

Further Reading: The Evaluation semantics for *WhilwL* is also discussed in detail in Chapters Two and Three of [Win93]. But be warned that the notation used is different. For example the language is called **IMP** and the symbol $\rightarrow$ is used for the evaluation relation $\Longrightarrow_C$. It is also discussed in much more detail in Chapter two of [NN92], where it is called *Natural Semantics*; here the language is called **While**, $\rightarrow$ is used to denote the evaluation relation. In later sections of this Chapter it is shown how to extend the semantics to cope with *procedures* and *block structures*.

Questions

Q1 Evaluate the following program in *CalcL* for the calculator:

$$\begin{array}{ll} ON\ \mathbf{3}+\mathbf{7} & TOTAL\ \mathbf{4} -\ LASTANSWER \\ & TOTAL\ IF(LASTANSWER, \mathbf{6}*\mathbf{2}, \mathbf{8}) \\ & TOTAL \\ & OFF \end{array}$$

Q2 Augment the calculator with an equality button, =, such that in the abstract syntax we have an extra clause

$$e ::= \ldots \mid e = e' \mid \ldots$$

Extend the operational semantics for this new construct.

Q3 Replace the single memory cell in the calculator with a stack. In this new calculator the last answer printed is now pushed on to the stack and *LASTANSWER* accesses the top of the stack. In addition it has a new button, called *POP* which pops the internal stack. That is, it eliminates the topmost element of the internal stack. Give an Evaluation semantics for the new calculator.

Q4 Using the Evaluation semantics of Section 4.2, find the first two values printed out by the stream expression $NNst(Cons(\mathbf{0}, Nil))$ where $NNst$ is defined by

$$NNst(sx) \Longleftarrow Cons(Hd\ sx, NNst(Cons(Hd\ sx + \mathbf{1}, sx))).$$

Q5 Extend the language *StreamL* as suggested towards the end of Section 4.2 by allowing boolean expressions and the three different types of declarations, *AADec, SSDec* and *ASDec*. Can you think of a more uniform method for allowing more general declarations?

Q6 Using the Evaluation semantics of Section 4.3, execute the following program using the store s, where $s(x) = \mathbf{0}, s(y) = \mathbf{4}, s(z) = \mathbf{1}$:

$$\begin{array}{l} While\ \ Not(Equal(x, y))\ Do \\ \qquad \begin{array}{ll} If\ \ even(x) & Then\ \ z := z * x \\ & Else\ \ skip; \\ x := x + \mathbf{1} & \end{array} \end{array}$$

You may assume that *even* is interpreted as expected by the *Apply* mechanism.

Q7 Augment the language *WhileL* with a new command of the form *Repeat C Until be* . This command repeatedly executes C until *be* becomes true. It differs from *While Not be Do C* in that the command C is always executed at least once. Extend the Evaluation semantics to this new construct.

Q8 Use the Evaluation semantics of the previous question to evaluate

$$\begin{array}{ll} Repeat & z := z * x; \\ & x := x - \mathbf{1} \\ & Until\ \ x = \mathbf{0} \end{array}$$

in a store s where $s(z) = \mathbf{1}, s(x) = \mathbf{2}$.

Q9 Extend the language *WhileL* with a new type of command $swap(x, y)$. Intuitively the effect of this command on a store is to interchange the values of the variables x and y. Extend the Evaluation semantics to this new construct.

Q10 Extend the language *WhileL* with a parallel assignment command of the form $x, y := e, e'$. Extend the Evaluation semantics to this new construct.
Give an example to show that $x, y := e, e'$ does not always behave in the same way as $x := e; y := e'$.

Q11 Extend the language *WhileL* with a *case* command as in PASCAL. A new syntactic category, which we call *Guarded Lists*, is required, whose abstract syntax is given by

$$G := n : C \mid n : C, G$$

and a new command is introduced:

$$case\ \ e\ \ of\ \ G\ \ end.$$

Intuitively the effect of this command is to evaluate the expression e to a numeral and then execute the command in the list G corresponding to this numeral.
Extend the Evaluation semantics to this new construct. It should include an evaluation relation $\Longrightarrow_G$ for guarded lists.

Q12 Extend the language *WhileL* by adding an extra clause to the definition of the abstract syntax of expressions and boolean expressions:

$$\begin{aligned} e &::= \ldots \mid \ \ Run\ \ C\ \ Then\ \ e \mid \ \ldots \\ be &::= \ldots \mid \ \ Run\ \ C\ \ Then\ \ be \mid \ \ldots . \end{aligned}$$

Intuitively to evaluate the expression $Run\ \ C\ \ Then\ \ e$ in a store s you first evaluate the command C in s and then evaluate e in the resulting store.
Give a new operational semantics for this language. Note that the types of $\Longrightarrow_A$ and $\Longrightarrow_B$ will have to change as the evaluation of expressions will now affect the store.

Chapter 5

Computation Semantics

This chapter is devoted entirely to Computation semantics. In the first two sections we give a Computation semantics to the languages *Fpl* and *WhileL* which we have already encountered in previous chapters. Computation semantics describes the evaluation or execution of programs in terms of primitive one-step operations. Frequently there is considerable choice as to what these operations should be and in these two sections we will see some examples of the choices which have to be made. In the final section a very different kind of Computation semantics is seen. It occurs as part of a Concrete operational semantics for the language *Fpl*. This Concrete semantics is in terms of an interpreter for *Fpl* which runs on an *abstract machine*. The functioning of this *abstract machine* will be described by defining a Computation semantics for it. In other words, a Concrete operational semantics for the high-level language *Fpl* will be given, using a Computation semantics for a lower-level system, namely the *abstract machine*.

5.1 Computation Semantics for *Fpl*

In this section we re-examine the language *Fpl* from Figure 3.8. We give it a one-step operational semantics or Computation semantics which prescribes the sequences of primitive operations to which the evaluation of an expression may give rise. The approach is similar to that in Section 2.3 where a Computation semantics is given for the simple language *Exp*. Indeed, *Fpl* is an extension of *Exp* and so in this section we simply extend the computation semantics given there. This is formalised as a binary relation $\longrightarrow : Exp \longmapsto Exp$, and

$$e \longrightarrow e'$$

means that one primitive operation may be applied in the expression e and the resulting expression, which may remain to be evaluated, is e'. Intuitively a primitive operation is the application of one of the predefined operator symbols $+, -, *, div$ to a tuple of values. The formal definition of $\longrightarrow$ is given in Figure 2.3 and we wish to extend this definition to the more complicated language *Fpl*.

The expressions in the language *Fpl* may contain free variables and therefore, once more, we need to work with respect to an environment. So this indicates that we need to define a relation with the type

$$\longrightarrow : ENV \longmapsto Fpl \ \longmapsto Fpl.$$

The presence of free variables may then be accommodated by adding to the rules of Figure 2.3 the extra rule:

$$\frac{}{\rho \vdash v \ \longrightarrow \ \rho(v)}$$

which says that to evaluate a variable you look it up in the environment. The other rules in Figure 2.3 remain the same except that environments must be mentioned.

Another difference between *Exp* and *Fpl* is the presence of user-defined functions. As with the Evaluation semantics, to explain these functions we need to work with respect to a declaration D. In other words, we need a relation parametrised on declarations. However, we are evaluating arithmetic expressions and boolean expressions and we need two relations, $\longrightarrow_A$ for arithmetic expressions and $\longrightarrow_B$ for boolean expressions. So

$$D, \rho \vdash e \ \longrightarrow_A \ e'$$

means that, assuming the declaration D and the environment ρ in one step of the computation, the arithmetic expression e may evolve to the arithmetic expression e'. The statement

$$D, \rho \vdash be \ \longrightarrow_B \ be'$$

has a similar meaning but for boolean expressions.

The inductive definition of $\longrightarrow_A$ is given in Figure 5.1 and, as discussed above, is an extension of the relation defined in Figure 2.3.

The first two rules in the definition of $\longrightarrow_A$, Rules VarRc and OpRc, have already been explained. The third concerns the evaluation of *If* expressions and it also uses the auxiliary relation $\longrightarrow_B$ whose definition is given in Figure 5.2. This auxilary relation is very straightforward because of the simplicity of boolean expressions. Note that in Rule EqRc the relation $\longrightarrow_A$ is used and so the definitions of $\longrightarrow_B$ and $\longrightarrow_A$ are dependent on each other. We will concentrate our explanation on the rules for $\longrightarrow_A$, after which those for $\longrightarrow_B$ should be easily understandable.

Let us now explain Rule IfRc. Intuitively, to evaluate $\mathit{If}\ be\ \ \mathit{Then}\ e\ \ \mathit{Else}\ e'$ we must first evaluate be and then, depending on the result, evaluate either e or e'. That is, the first step in any computation from $\mathit{If}\ be\ \ \mathit{Then}\ e\ \ \mathit{Else}\ e'$ should make progress towards the evaluation of be. So if $be \ \longrightarrow_B \ be'$ then the first step in the computation is from $\mathit{If}\ be\ \ \mathit{Then}\ e\ \ \mathit{Else}\ e'$ to $\mathit{If}\ be'\ \ \mathit{Then}\ e\ \ \mathit{Else}\ e'$. This explains the first part of the rule. However, it may be that the boolean expression be is already evaluated.

Rule VarRc $$\frac{}{D, \rho \vdash x \ \longrightarrow_A \ \rho(x)}$$

Rule OpRc $$\frac{}{D, \rho \vdash v \ \ op \ \ v' \ \longrightarrow_A \ Ap(op, v, v')}$$

$$\frac{D, \rho \vdash e \ \longrightarrow_A \ e''}{D, \rho \vdash e \ \ op \ \ e' \ \longrightarrow_A \ e'' \ \ op \ \ e'}$$

$$\frac{D, \rho \vdash e' \ \longrightarrow_A \ e''}{D, \rho \vdash e \ \ op \ \ e' \ \longrightarrow_A \ e \ \ op \ \ e''}$$

Rule IfRc $$\frac{D, \rho \vdash be \ \longrightarrow_B \ be'}{D, \rho \vdash \mathit{If}\ be\ \ \mathit{Then}\ e\ \ \mathit{Else}\ e' \ \longrightarrow_A \ \mathit{If}\ be'\ \ \mathit{Then}\ e\ \ \mathit{Else}\ e'}$$

$$\frac{}{D, \rho \vdash \mathit{If}\ T\ \ \mathit{Then}\ e\ \ \mathit{Else}\ e' \ \longrightarrow_A \ e}$$

$$\frac{}{D, \rho \vdash \mathit{If}\ F\ \ \mathit{Then}\ e\ \ \mathit{Else}\ e' \ \longrightarrow_A \ e'}$$

Rule LocRc $$\frac{D, \rho \vdash e \ \longrightarrow_A \ e''}{D, \rho \vdash \ \mathit{let}\ x = e\ \ \mathit{in}\ e' \ \longrightarrow_A \ \mathit{let}\ x = e''\ \ \mathit{in}\ e'}$$

$$\frac{}{D, \rho \vdash \ \mathit{let}\ x = v\ \ \mathit{in}\ e' \ \longrightarrow_A \ e'[v/x]}$$

Rule FunRc $$\frac{D, \rho \vdash e_i \ \longrightarrow_A \ e_i'}{D, \rho \vdash F(e_1, \ldots, e_i, \ldots, e_k) \ \longrightarrow_A \ F(e_1, \ldots, e_i', \ldots, e_k)}$$

$$\frac{}{D, \rho \vdash F(v_1, \ldots, v_k) \ \longrightarrow_A \ e[v_1/x_1, \ldots, v_k/x_k]}$$

whenever $F(x_1, \ldots, x_k) \Longleftarrow \ e$ occurs in D

Figure 5.1: Computation semantics for *Fpl*: $\longrightarrow_A$

Rule VarRc $$\frac{}{D, \rho \vdash bx \longrightarrow_B \rho(bx)}$$

Rule OpRc $$\frac{}{D, \rho \vdash bv \;\; bop \;\; bv' \longrightarrow_B Ap(bop, bv, bv')}$$

$$\frac{D, \rho \vdash be \longrightarrow_B be''}{D, \rho \vdash be \;\; bop \;\; be' \longrightarrow_B be'' \;\; bop \;\; be'}$$

$$\frac{D, \rho \vdash be' \longrightarrow_B be''}{D, \rho \vdash be \;\; bop \;\; be' \longrightarrow_B be \;\; bop \;\; be''}$$

Rule NotRc $$\frac{D, \rho \vdash be \longrightarrow_B be'}{D, \rho \vdash Not \;\; be \longrightarrow_B Not \;\; be'}$$

$$\frac{}{D, \rho \vdash Not \;\; T \longrightarrow_B F}$$

$$\frac{}{D, \rho \vdash Not \;\; F \longrightarrow_B T}$$

Rule EqRc $$\frac{D, \rho \vdash e \longrightarrow_A e''}{D, \rho \vdash Equal(e, e') \longrightarrow_B Equal(e'', e')}$$

$$\frac{D, \rho \vdash e' \longrightarrow_A e''}{D, \rho \vdash Equal(e, e') \longrightarrow_B Equal(e, e'')}$$

$$\frac{v = v'}{D, \rho \vdash Equal(v, v') \longrightarrow_B T}$$

$$\frac{v \neq v'}{D, \rho \vdash Equal(v, v') \longrightarrow_B F}$$

Figure 5.2: Computation semantics for *Fpl*: $\longrightarrow_B$

This case is covered by the other two parts: if it has been evaluated to T, then the expression we are currently evaluating is $\mathit{If}\ T\ \ \mathit{Then}\ e\ \ \mathit{Else}\ e'$ and the first step in the computation is to e. If it has been evaluated to F, it is $\mathit{If}\ F\ \ \mathit{Then}\ e\ \ \mathit{Else}\ e'$ and the first step is to e'.

The rule for local declarations is somewhat similar. To evaluate $let\ \ x = e\ \ in\ \ e'$ with respect to the environment ρ, we first evaluate e to some value v and then evaluate e' on the assumption that x is v. So the computation should proceed by first evaluating e, if it needs to be evaluated. In this case the first step of the computation is from $let\ \ x = e\ \ in\ \ e'$ to $let\ \ x = e''\ \ in\ \ e'$ where the first step in the evaluation of e is from e to e''. If, on the other hand, e has already been evaluated, then the expression we wish to evaluate is of the form $let\ \ x = v\ \ in\ \ e'$ for some value v. The computation could then proceed by evaluating e' with respect to the modified environment $\rho[v/x]$. However, we will model this by first syntactically substituting the value v for all free occurrences of x in e' to obtain $e'[v/x]$ and then evaluating this modified expression with respect to the original environment ρ. So in this case the first step in the computation is from $let\ \ x = v\ \ in\ \ e'$ to $e'[v/x]$. In this way the environment with respect to which a computation takes place will not change as the computation proceeds. Another method for handling local declarations which does not involve substitution will be discussed later.

The rule for user-defined functions, FunRc, is also similar. The first sub-rule covers the case when at least one of the parameters is unevaluated, the second when all the parameters are values. In the latter case the first step is from $F(v_1, \ldots, v_k)$ to $e[v_1/x_1, \ldots, v_k/x_k]$ when D contains the declaration $F(x_1, \ldots, x_k) \Longleftarrow e$. This corresponds to a call to the body of a definition. Once more, instead of evaluating the body e in a modified environment, the environment remains unchanged but the actual parameters $v_1, \ldots, v_k$ are syntactically substituted for the parameters $x_1, \ldots, x_k$ in e. On the other hand, if any of the parameters are unevaluated, then the computation proceeds by evaluating an arbitrary parameter. So, in this case, a first step is from $F(e_1, \ldots, e_i, \ldots, e_k)$ to $F(e_1, \ldots, e_i', \ldots, e_k)$, where a first step in the evaluation of e_i is from e_i to e_i'.

Let us now see an example of the use of these rules. Consider the declaration

$$\begin{array}{lll} H(x,y) \Longleftarrow & \mathit{If}\ \ Equal(x,y) & \\ & \mathit{Then}\ \ x & \\ & \mathit{Else}\ \ \mathit{If}\ \ Gt(x,y) & \\ & \qquad \mathit{Then}\ \ H(x-y, y) & \\ & \qquad \mathit{Else}\ \ H(y,x) & \end{array}$$

Here we assume another primitive function symbol Gt similar to $Equal$. It takes two arithmetic arguments and returns T if the first is strictly greater than the second and false otherwise. We leave it to the reader to incorporate it into the Computation semantics with rules similar to those for $Equal$. We will compute the value of the expression $H(\mathbf{15},\mathbf{25})$ with respect to this declaration. For convenience we use e to denote the sub-expression

$$\mathit{If}\ Gt(x,y)\ \ \mathit{Then}\ \ H(x-y,y)\ \ \mathit{Else}\ \ H(x,y).$$

Because there are no free variables in $H(\mathbf{15},\mathbf{25})$ the computation will not depend on the values contained in the environment. Also the declaration is fixed and therefore for convenience we will abbreviate $D,\rho \vdash e \longrightarrow_A e'$ to $e \longrightarrow_A e'$. If we examine the rules for $\longrightarrow_A$ we see that the only one applicable to the expression $H(\mathbf{15},\mathbf{25})$ is the second part of Rule FunRc. So the first step in the computation is

$$H(\mathbf{15},\mathbf{25}) \longrightarrow_A \mathit{If}\ \mathit{Equal}(\mathbf{15},\mathbf{25})\ \ \mathit{Then}\ \ \mathbf{15}\ \ \mathit{Else}\ \ (e[\mathbf{15}/x,\mathbf{25}/y]).$$

To find the second step in the computation we must see what rule is applicable to the expression $\mathit{If}\ \ \mathit{Equal}(\mathbf{15},\mathbf{25})\ldots$. Since the boolean expression is not evaluated, the only rule which applies is the first part of Rule IfRc. So the second step has the form

$$\mathit{If}\ \mathit{Equal}(\mathbf{15},\mathbf{25})\ \ \mathit{Then}\ \ \ldots \longrightarrow_A \mathit{If}\ be'\ \ \mathit{Then}\ \ \ldots$$

where

$$\mathit{Equal}(\mathbf{15},\mathbf{25}) \longrightarrow_B be'.$$

By examining the rules for $\longrightarrow_B$ we see that the final part of Rule EqRc gives $\mathit{Equal}(\mathbf{15},\mathbf{25}) \longrightarrow_B F$. So the second step in the computation is

$$\mathit{If}\ \mathit{Equal}(\mathbf{15},\mathbf{25})\ \ \mathit{Then}\ \ \ldots \longrightarrow_A \mathit{If}\ F\ \ \mathit{Then}\ \ \ldots$$

The third step is obtained by applying the final part of Rule IfRc to obtain

$$\mathit{If}\ F\ \ \mathit{Then}\ \ \mathbf{15}\ \ \mathit{Else}\ \ e[\mathbf{15}/x,\mathbf{25}/y] \longrightarrow_A e[\mathbf{15}/x,\mathbf{25}/y].$$

The next step is obtained by examining the expression $e[\mathbf{15}/x,\mathbf{25}/y]$. Once more this is an *If* expression of the form $\mathit{If}\ Gt(\mathbf{15},\mathbf{25})\ \ \mathit{Then}\ \ldots$. Since the boolean expression is unevaluated, the applicable rule is the first part of Rule IfRc. Applying this we obtain the step

$$\mathit{If}\ Gt(\mathbf{15},\mathbf{20})\ \ \mathit{Then}\ \ \ldots \longrightarrow_A \mathit{If}\ F\ \ \mathit{Then}\ \ \ldots$$

assuming appropriate rules for Gt. The fifth step is obtained by applying the final part of Rule IfRc to obtain the step

$$\mathit{If}\ F\ \ \mathit{Then}\ \ H(\mathbf{15}-\mathbf{25},\mathbf{25})\ \ \mathit{Else}\ \ H(\mathbf{25},\mathbf{15}) \longrightarrow_A H(\mathbf{25},\mathbf{15}).$$

So, in five computation steps, each individual step being derived using the rules from Figures 5.1 and 5.2, we have reduced $H(\mathbf{15},\mathbf{25})$ to $H(\mathbf{25},\mathbf{15})$. Using the notation introduced in Section 2.3, we may write this as

$$H(\mathbf{15},\mathbf{25}) \longrightarrow_A^* H(\mathbf{25},\mathbf{15}).$$

Another five computation steps which, once more, simply evaluate the boolean expressions and branch accordingly, give rise to

$$H(\mathbf{25},\mathbf{15}) \longrightarrow_A^* H(\mathbf{25}-\mathbf{15},\mathbf{15}).$$

The next step in the computation is

$$H(\mathbf{25}-\mathbf{15},\mathbf{15}) \longrightarrow_A H(\mathbf{10},\mathbf{15})$$

by applying the first part of Rule FunRc. So at this stage the computation is

$$H(\mathbf{15},\mathbf{25}) \longrightarrow_A^* H(\mathbf{10},\mathbf{15})$$

and consists of 11 steps. Another 11 steps will give rise to

$$H(\mathbf{10},\mathbf{15}) \longrightarrow_A^* H(\mathbf{5},\mathbf{10}).$$

and a further 11 steps to

$$H(\mathbf{5},\mathbf{10}) \longrightarrow_A^* H(\mathbf{5},\mathbf{5}).$$

The computation now proceeds as:

$$\begin{aligned} H(\mathbf{5},\mathbf{5}) &\longrightarrow_A \mathit{If}\ \mathit{Equal}(\mathbf{5},\mathbf{5})\ \ \mathit{Then}\ \ \mathbf{5}\ \ \mathit{Else}\ \ \ldots \\ &\longrightarrow_A \mathit{If}\ T\ \ \mathit{Then}\ \ \mathbf{5}\ \ \mathit{Else}\ \ \ldots \\ &\longrightarrow_A \mathbf{5}. \end{aligned}$$

So, in 36 computation steps, we have

$$H(\mathbf{15},\mathbf{25}) \longrightarrow_A^* \mathbf{5}.$$

Using the evaluation semantics of Chapter 3, the reader may also check that

$$H(\mathbf{15},\mathbf{25}) \Longrightarrow_A \mathbf{5}$$

and it is instructive to compare these two derivations. The first consists of many individual computation steps, i.e. applications of $\longrightarrow_A$, each being derived from the defining rules for $\longrightarrow_A$. These individual derivations are usually quite short and uninvolved. For example, in the computation just carried out, none of these derivations have depth greater than two. The second, using $\Longrightarrow_A$, consists of only one derivation using the defining rules for $\Longrightarrow_A$. This derivation is quite long and involved. The sequence of individual steps using $\longrightarrow_A$ is "encoded" in its structure. For this reason the Evaluation semantics is considered more abstract than the Computation semantics. The Evaluation semantics tells one what an implementation should do without being too explicit as to how to do it. By examining derivations within the Evaluation semantics it can be seen which primitive operations have to be performed and often the order in which they should be performed. The Computation semantics is more explicit. It may be viewed as an attempt to make more explicit the sequence of primitive operations which lie implicit within the derivations of the Evaluation semantics.

For a given Evaluation semantics there may be many different Computation semantics, each representing a different method for ordering these implicit primitive operations. As an example there are many different variations we could introduce into our Computation semantics. One obvious possibility is with respect to the *If* statements. In our semantics the decision to follow the *Then* branch or the *Else* branch takes one computation step. We could abstract away from this step thereby saying that the decision is taken instantaneously. This would be the case if we replaced the second and third sub-rules in Rule IfRc by

$$\frac{D, \rho \vdash e \;\longrightarrow_A\; e''}{D, \rho \vdash \mathit{If}\ T\ \ \mathit{Then}\ e\ \ \mathit{Else}\ e' \;\longrightarrow_A\; e''}$$

and

$$\frac{D, \rho \vdash e' \;\longrightarrow_A\; e''}{D, \rho \vdash \mathit{If}\ F\ \ \mathit{Then}\ e\ \ \mathit{Else}\ e' \;\longrightarrow_A\; e''}$$

Similarly, it takes one computation step to call a user-defined function. This could be eliminated by replacing the second sub-rule of Rule FunRc with

$$\frac{D, \rho \vdash e[v_1/x_1, \ldots, v_k/x_k] \;\longrightarrow_A\; e'}{D, \rho \vdash F(v_1, \ldots, v_k) \;\longrightarrow_A\; e'}$$

Another variation on the Computation semantics concerns the treatment of local declarations. In our existing rules we have used syntactic substitution of values for variables in expressions and it could be argued that such tampering with the expressions under evaluation should be avoided. Environments were introduced to handle the association between variables and values and it may be preferable if part of the role of environments were not obscured in this way. One way of modifying the rules for local declarations so that substitution is avoided is to replace the second part of Rule LocRc by the two rules

$$\frac{D, \rho[v/x] \vdash e \;\longrightarrow_A\; e'}{D, \rho \vdash\ \mathit{let}\ x = v\ \ \mathit{in}\ e \;\longrightarrow_A\; \mathit{let}\ x = v\ \ \mathit{in}\ e'}$$

and

$$\frac{}{D, \rho \vdash\ \mathit{let}\ x = v\ \ \mathit{in}\ v' \;\longrightarrow_A\; v'}$$

Substitution is also used in Rule FunRc to usurp some of the role of environments and, by analogy with Rule LocRc, it is tempting to replace the second part of it by

$$\frac{D, \rho[v_1/x_1, \ldots, v_k/x_k] \vdash e \;\longrightarrow_A\; e'}{D, \rho \vdash F(v_1, \ldots, v_k) \;\longrightarrow_A\; e'}$$

whenever $F(x_1, \ldots, x_k) \Longleftarrow\ e$ occurs in D.

However, this is incorrect in that it leads to very different computations and, moreover, computations which are counter-intuitive. For example, let D be the declaration

$$F(x, y) \Longleftarrow x + y$$

and consider the computations from $F(\mathbf{1}, \mathbf{2})$ in an environment ρ where $\rho(y) = \mathbf{6}$. If we use the proposed rule, together with the existing rules (in particular Rule VarRc), we obtain as a first step in the computation

$$D, \rho \vdash F(\mathbf{1}, \mathbf{2}) \;\longrightarrow_A\; \mathbf{1} + y.$$

Unfortunately the second step, from $\mathbf{1} + y$, will be

$$D, \rho \vdash \mathbf{1} + y \;\longrightarrow_A\; \mathbf{1} + \mathbf{6}$$

and the final step

$$D, \rho \vdash \mathbf{1} + \mathbf{6} \;\longrightarrow_A\; \mathbf{7}.$$

However, intuitively the evaluation of $F(\mathbf{1}, \mathbf{2})$ should lead to $\mathbf{3}$ and not $\mathbf{7}$. The problem occurs because after the first step the proper environment for evaluating the body of F, namely $\rho[\mathbf{1}/x, \mathbf{2}/y]$, is lost and subsequent steps are evaluated erroneously with respect to the original environment ρ. There is a very close link between the semantics of function calls and local declarations. The careful reader will have noticed that, in both the Evaluation semantics and our first Computation semantics, $F(e')$ is interpreted in the same way as the expression $\mathit{let}\ \ x = e'\ \ \mathit{in}\ \ e$ where e is the body in the declaration of F. This semantic equivalence can be exploited to give a substitution-free rule for function calls: the second part of Rule FunRc is replaced by

$$\frac{}{D, \rho \vdash F(v_1, \ldots, v_k) \;\longrightarrow_A\; \mathit{let}\ x_1 = v_1\ \ \mathit{in}\ \ \ldots\ldots\ \mathit{let}\ x_k = v_k\ \ \mathit{in}\ e}$$

provided $F(x_1, \ldots, x_k) \Longleftarrow e$ is a definition in the declaration D.

This adjustment to Rule FunRc is satisfactory because the correct modification to the environment for evaluating the body of F, namely $\rho[v_1/x_1, \ldots, v_k/x_k]$, is carried along in the syntax, in the local declarations $\mathit{let}\ \ x_1 = v_1\ \ \mathit{in}\ \ \ldots\ \mathit{let}\ \ x_k = v_k\ \ \mathit{in}\ \ \ldots$ as long as they are required.

We end this section with one more comment on our Computation semantics. In the above example the computation is completely deterministic. For each expression e which arises during the computation, apart from the final one, there is a unique next-state, i.e. a unique expression e' such that $D, \rho \vdash e \;\longrightarrow_A\; e'$. This is not always the case as the Computation semantics is in general non-deterministic. The non-determinism arises in function expressions such as $e + e'$ or $F(e, e')$. If both e and e' are unevaluated, the semantics states that either may be evaluated first. Indeed, their evaluation may be interleaved in any order. In actual implementations it is more usual to evaluate the parameters of a function from left to right. But our semantics essentially leaves this decision up to the implementer. We set as an exercise at the end of this chapter the problem of modifying our Computation semantics so that parameters may only be evaluated in this more restricted manner. A similar exercise for the sub-language *Exp* was given at the end of Chapter 2.

5.2 The Language *WhileL*

We have already seen an Evaluation semantics for the language *WhileL*. In this section we design a Computation semantics. The basic operation for this language is the assignment statement, $x := e$. It acts on a store s and changes it to $s[v/x]$, where v is the value of the expression e with respect to the store s. So a Computation semantics for *WhileL* must describe the basic operations a command can perform and implicitly the semantics will also dictate an ordering between them. To execute an assignment statement a store is required and therefore the Computation semantics is expressed as a relation over configurations

$$\longrightarrow_C : \langle Com, Store\rangle \longmapsto \langle Com, Store\rangle.$$

The statement

$$(C, s) \longrightarrow_C (C', s')$$

means that the command C may execute a basic operation, an assignment statement, with respect to the store s; this operation will change the store s to s' and C' is the remainder of the command C still to be executed.

When designing the Computation semantics a number of choices have to be made. One concerns whether or not choices in *If* commands and *While* commands take a computation step. In the previous section where we discussed a Computation semantics for *Fpl* the resolution of these choices did constitute a step and for the sake of variety in this section, we assume that for *If* statements they are instantaneous. But for technical reasons it is important to assume that the choice in a *While* statement to terminate or to execute the body takes one computation step. Let us also assume that we are not concerned with how arithmetic and boolean expressions are computed. For this reason we will not define computation relations, $\longrightarrow_A$, $\longrightarrow_B$ for these expressions. Instead we will use the evaluation relations $\Longrightarrow_A$ and $\Longrightarrow_B$ from the previous chapter. So the rule for the assignment statement is:

$$\frac{(e, s) \Longrightarrow_A v}{(x := e, s) \longrightarrow_C (skip, s[v/x])}$$

All the rules are given in Figure 5.3.

Rule AsRc $\quad \dfrac{(e, s) \Longrightarrow_A v}{(x := e, s) \longrightarrow_C (skip, s[v/x])}$

Rule IfRc $\quad \dfrac{(be, s) \Longrightarrow_B T, (C, s) \longrightarrow_C (C'', s')}{(If\ be\ Then\ C\ Else\ C', s) \longrightarrow_C (C'', s')}$

$\quad \dfrac{(be, s) \Longrightarrow_B F, (C', s) \longrightarrow_C (C'', s')}{(If\ be\ Then\ C\ Else\ C', s) \longrightarrow_C (C'', s')}$

Rule ComRc $\quad \dfrac{(C, s) \longrightarrow_C (C'', s')}{(C; C', s) \longrightarrow_C (C''; C', s')}$

$\quad \dfrac{(C, s)\surd, (C', s) \longrightarrow_C (C'', s')}{(C; C', s) \longrightarrow_C (C'', s')}$

Rule WhileRc1 $\quad \dfrac{(be, s) \Longrightarrow_B F}{(While\ be\ Do\ C, s) \longrightarrow_C (skip, s)}$

Rule WhileRc2 $\quad \dfrac{(be\ , s) \Longrightarrow_B T}{(While\ be\ Do\ C, s) \longrightarrow_C (C; While\ be\ Do\ C, s)}$

Figure 5.3: Computation semantics for *WhileL*

Rule Skipt $\quad \dfrac{}{(skip, s)\surd}$

Rule IfRt $\quad \dfrac{(be, s) \Longrightarrow_B T, (C, s)\surd}{(If\ be\ Then\ C\ Else\ C', s)\surd}$

$\quad \dfrac{(be, s) \Longrightarrow_B F, (C', s)\surd}{(If\ be\ Then\ C\ Else\ C', s)\surd}$

Rule ComRt $\quad \dfrac{(C, s)\surd, (C', s)\surd}{(C; C', s)\surd}$

Figure 5.4: Termination predicate for *WhileL*

Let us examine the rule for *If* commands. We are assuming the instantaneous evaluation of boolean expressions and of the resolution of choices. So if the configuration (C, s) can perform a basic operation and thereby be transformed into (C'', s'), i.e. $(C, s) \longrightarrow_C (C'', s')$, and *be* evaluates to true with respect to the store s, i.e. $(be, s) \Longrightarrow_B T$, then we may conclude that in one step $(\mathit{If}\ be\ \mathit{Then}\ C\ \mathit{Else}\ C', s)$ can move to (C'', s'), i.e. $(\mathit{If}\ be\ \mathit{Then}\ C\ \mathit{Else}\ C', s) \longrightarrow_C (C'', s')$. Similarly, if $(be, s) \Longrightarrow_B F$ and $(C', s) \longrightarrow_C (C'', s')$ we can also conclude that $(\mathit{If}\ be\ \mathit{Then}\ C\ \mathit{Else}\ C', s) \longrightarrow_C (C'', s')$.

The rule for *While* commands is quite different. Intuitively if *be* evaluates to T with respect to the store s, the computation from *While be Do C* should proceed by first executing C and subsequently executing *While be Do C*. We are assuming that this decision to continue with the execution takes one computation step. So in this case the first step from *While be Do C* is

$$(\mathit{While}\ be\ \mathit{Do}\ C, s) \longrightarrow_C (C; \mathit{While}\ be\ \mathit{Do}\ C, s).$$

On the other hand, if the boolean expression *be* evaluates to F then the computation is finished. We model this by one final computation step to the configuration (skip, s):

$$(\mathit{While}\ be\ \mathit{Do}\ C, s) \longrightarrow_C (\mathit{skip}, s).$$

No rules apply to the configuration (skip, s); it is effectively the halt configuration with the final state s.

The rules for the composition operator ; are the most interesting. The computation from $C_1; C_2$ should proceed by first executing C_1 and then C_2. So the first step in this computation should be the first step in the computation from C_1. This explains the first sub-rule, namely that from

$$(C_1, s) \longrightarrow_C (C_1', s')$$

we can conclude

$$(C_1; C_2, s) \longrightarrow_C (C_1'; C_2, s').$$

But it may be that there is no computation from C_1 and so if this were the only rule for the composition operator ; then there would be no move from $(C_1; C_2, s)$. This arises, for example, in

$$\mathit{skip}; x := \mathbf{1}$$

and

$$(\mathit{If}\ T\ \mathit{Then}\ \mathit{skip}\ \mathit{Else}\ y := \mathbf{2}); x := \mathbf{1}.$$

The above rule does not apply to these commands but intuitively they do give rise to computations, namely, updating the value of the location x to be $\mathbf{1}$. The subcommands *skip* and *If T Then skip Else* $y := \mathbf{2}$ have *terminated* and for this reason the second sub-command i.e. the sub-command after ; , may start executing. In general

whether a command has terminated or not depends not only on the command itself but also on the current store. For example,

$$\mathit{If}\ x = \mathbf{0}\ \mathit{Then}\ \mathit{skip}\ \mathit{Else}\ y := \mathbf{1}$$

is terminated if in the current store $\mathbf{0}$ is stored in the location x but is not terminated otherwise. Let us indicate that a configuration (C, s) has terminated by writing

$$(C, s)\surd.$$

Then whenever $(C, s)\surd$, the computation from $(C; C', s)$ may proceed by executing C'. So the first step in the computation from $(C; C', s)$ is the first step from (C', s). From $(C, s)\surd$ and $(C', s) \longrightarrow_C (C'', s')$ we may infer $(C; C', s) \longrightarrow_C (C'', s')$. This is the second part of Rule ComRc.

The definition of the termination predicate $\surd$ is given in Figure 5.4. It may be viewed as a definition by structural induction and the rules are very straightforward. Note that there are no rules which apply to *While* commands. This is because every *While* command can always make at least one step, the decision as to whether or not to go once more around the loop.

Let us now look at an application of these rules to execute a command. We examine the same program as in Section 4.3:

$$\begin{array}{l} z := \mathbf{0}; \\ \mathit{While}\ \mathit{Not}\ (\mathit{Equal}(x, \mathbf{0}))\ \mathit{Do} \\ \qquad\qquad z := z + y; \\ \qquad\qquad x := x - \mathbf{1}. \end{array}$$

We also assume the same store s, with $s(x) = \mathbf{2}$, $s(y) = \mathbf{3}$ and $s(z) = \mathbf{7}$ and use the notation for stores whereby $s_{i,j}$ is used to denote the store $s[\mathbf{i}/x][\mathbf{j}/z]$. As before, we also use W to denote the *While* command and C its body. We know $(\mathbf{0}, s) \Longrightarrow_A \mathbf{0}$ and therefore by Rule AsRc we may infer

$$(z := \mathbf{0}, s) \longrightarrow_C (\mathit{skip}, s_{2,0}).$$

Applying the first part of Rule ComRc, we obtain the first step of the overall program

$$(z := \mathbf{0}; W, s) \longrightarrow_C (\mathit{skip}; W, s_{2,0}).$$

To obtain the next step we have to see which rule is applicable to $\mathit{skip}; W$. Now $(\mathit{skip}, s)\surd$ and therefore the second part of Rule ComRc applies. In order to apply it we must find the first move from W. To discover this we will need to apply the rule *WhileRc*. Since $(\mathit{Not}(\mathit{Equal}(x, \mathbf{0})), s) \Longrightarrow_B T$ its first move is

$$(W, s_{2,0}) \longrightarrow_C (C; W, s_{2,0})$$

So, applying Rule ComRc, the second step from our program is

$$(\mathit{skip}; W, s_{2,0}) \longrightarrow_C (C; W, s_{2,0}).$$

The third computation step is from the configuration $(C; W, s_{2,0})$ and to obtain it we must apply Rule ComRc to a move from $(C, s_{2,0})$. Applying Rule AsRc we obtain

$$(z := z + y, s_{2,0}) \longrightarrow_C (\mathit{skip}, s_{2,3})$$

and applying Rule ComRc

$$(C, s_{2,0}) \longrightarrow_C (\mathit{skip}; x := x - \mathbf{1}, s_{2,3}).$$

So another application of ComRc gives the third step

$$(C; W, s_{2,0}) \longrightarrow_C ((\mathit{skip}; x := x - \mathbf{1}); W, s_{2,3}).$$

The next step is obtained by an application of Rule AsRc

$$(x := x - \mathbf{1}, s_{2,3}) \longrightarrow_C (\mathit{skip}, s_{1,3}),$$

followed by an application of the second part of Rule ComRc

$$(\mathit{skip}; x := x - \mathbf{1}, s_{2,3}) \longrightarrow_C (\mathit{skip}, s_{1,3})$$

and followed, finally, by an application of the first part of Rule ComRc to obtain

$$((\mathit{skip}; x := x - \mathbf{1}); W, s_{2,3}) \longrightarrow_C (\mathit{skip}; W, s_{1,3}).$$

So in four steps the program has progressed from $(z := \mathbf{0}; W, s)$ to $(\mathit{skip}; W, s_{1,3})$. As before, we use $\longrightarrow_C^*$ to denote multiple steps. So,

$$(z := \mathbf{0}; W, s) \longrightarrow_C^* (\mathit{skip}; W, s_{1,3}).$$

In three more steps we obtain

$$(\mathit{skip}; W, s_{1,3}) \longrightarrow_C^* (\mathit{skip}; W, s_{0,6}).$$

A final application of the second part of Rule WhileRc followed by Rule ComRc leads to

$$(\mathit{skip}; W, s_{0,6}) \longrightarrow_C (\mathit{skip}, s_{0,6}).$$

There is no rule which applies to *skip* and therefore the computation is terminated; it can proceed no further. So the complete computation is

$$(z := \mathbf{0}; W, s) \longrightarrow_C^* (\mathit{skip}, s_{0,6}).$$

In the previous chapter we have also seen the derivation of

$$(z := \mathbf{0}; W, s) \Longrightarrow_C s_{0,6}$$

using the Evaluation semantics $\Longrightarrow_C$ and a similar comparison may be made between the derivations of $\Longrightarrow_C$ and sequences of derivations of $\longrightarrow_C$ as between $\Longrightarrow_A$ and $\longrightarrow_A$ in the previous section. In the latter, individual derivations of single steps are relatively short and uncomplicated, and a computation consists of a sequence of these steps. In the former, a complete computation is modelled by a single complicated derivation within the structure of which the single steps or primitive operations may be discerned.

Intuitively the predicate $\surd$ represents *termination* and one property which one would expect of programs is that they can do a computation step if and only if they are not terminated. If this were not the case there would be something drastically wrong with our definitions. As a consistency check on these definitions we therefore prove that this is indeed the case for our formal notion of termination and computation step.

Theorem 5.2.1 *For every configuration (C, s) exactly one of the following holds:*

1. *the configuration is terminated, i.e. $(C, s)\surd$*
2. *it has a move, i.e. there is a configuration (C', s') such that $(C, s) \longrightarrow_C (C', s')$.*

Proof The proof is by structural induction on commands. According to the abstract syntax of *WhileL* there are five different kinds of command and therefore there are five cases to consider.

1. C is *skip*. In this case $(C, s)\surd$ and since no rule applies to *skip* there can be no C' such that $(\mathit{skip}, s) \longrightarrow_C (C', s')$ for any s, s'.

2. C is $x := e$. This case is the opposite to the previous one.

3. C is $C'; C''$. By induction we may assume that exactly one possibility applies to (C', s). If it has a move and is not terminated, then by the first part of Rule ComRc $(C'; C'', s)$ also has a move. Also it is not terminated because, according to Rule ComRt $(C'; C'', s)$ is only terminated if both (C', s) and (C'', s) are terminated. So suppose the second possibility applies, namely that $(C', s)\surd$ and it has no move. We may now apply induction to (C'', s). If $(C'', s)\surd$ and has no move, then by Rule ComRt $(C'; C'', s)\surd$ and $(C'; C'', s)$ has no move. The latter follows since to derive a step from $(C'; C'', s)$ the second part of Rule ComRc must be applied, which is not possible. On the other hand, if (C'', s) has a move and not $(C'', s)\surd$ then by Rule ComRc $(C'; C'', s)$ has a move and also $(C'; C'', s)$ does not terminate.

4. C is *If be Then* C' *Else* C''. In this case if $(be, s) \Longrightarrow_B T$ we apply induction to (C', s) and if $(be, s) \Longrightarrow_B F$ we apply it to (C'', s).

5. C is *While be Do* C'. Regardless of *be* and C', in this case C does not terminate since no rule in the definition of $\surd$ applies to *While* commands. Also, for every store s one part of the rule WhileRc applies to (C, s) since either $(be, s) \Longrightarrow_B T$ or $(be, s) \Longrightarrow_B F$. So there is always a move from (C, s).

□

Further Reading: A slightly different Computation semantics for *While* is given in the second section of Chapter Two of [NN92], but again you should be warned about the different notation used; First Computation Semantics is called *Structured Operational Semantics*, the language itself is called **While** and the next step relation $\longrightarrow_C$ is denoted by $\Rightarrow$. This Chapter also contains detailed proofs of the properties of the semantics and the relationship between the Computation and Evaluation semantics.

5.3 An Abstract Machine for *Fpl*

In this section we give a somewhat unusual example of an Computation semantics. We have already seen two semantics for *Fpl*, a rather abstract Evaluation semantics and a more detailed Computation semantics. Here we give an even more concrete semantics in the form of an *abstract machine*. This is, in effect, a formal interpreter for the language which runs on a hypothetical machine. It is a much more detailed semantics than the previous two; not only does it determine the order in which primitive operations are to be performed, it suggests concrete implementation methods for analysing programs to determine this order and how to perform the sequences of operations.

Let us begin by re-examining the simple language *Exp*. We have already seen a Concrete operational semantics for *Exp* in Section 2.1 in the form of a compiler for the *STACK*-machine. Here we use the same technique except that it is expressed as an interpreter. We augment the *STACK*-machine so that it has not only a stack for holding values and an arithmetic unit for carrying out operations but also a *control unit*. The control unit holds the program or expression we wish to evaluate. As the evaluation or computation proceeds the expression in the control unit changes and we will shortly see that in general it contains a list of expressions. We call the machine an *abstract machine* because we do not fill in many of the details of how it is to be constructed or implemented. It could be a physical device or, more likely, implemented in some low-level language. The only description of the machine, or rather states or configurations of the machine, will be in terms of tuples $< S, C >$

where S is a stack of values and C is the current control. These descriptions are somewhat abstract as they ignore many details which would be present in any actual implementation. However, they are sufficiently detailed to enable us to define how the machine should work. To evaluate an expression e the machine is started in the state $< \epsilon, e >$, where ϵ represents the empty stack. The machine then proceeds from state to state until a final state is reached. These are states where the control components have been exhausted, i.e. states of the form $< v, \epsilon >$ where the stack contains one value v and the control unit contains the empty list of expressions. (Here it is convenient to use ϵ to represent the empty list or sequence.) Then the value of the original expression e is v.

To describe the functioning of the machine it is therefore sufficient to define a next-state relation $\longrightarrow$ between states:

$$< S, C > \longrightarrow < S', C' >$$

means that if the abstract machine is in the state $< S, C >$ in one step it can move to the state $< S', C' >$. In other words we describe the functioning of the machine by giving a Computation semantics for it. In fact the machine will be deterministic so that the relation $\longrightarrow$ will be a partial function, i.e. for every state $< S, C >$ there will be at most one state $< S', C' >$ such that $< S, C > \longrightarrow < S', C' >$. Before defining the relation $\longrightarrow$ we should be precise about the exact form states may take. The easiest way to define the allowable states is to give the abstract syntax of a language for describing them. This is given in Figure 5.5 and assumes an abstract syntax for *Exp*. So e refers to an arbitrary arithmetic expression, *op* an arbitrary arithmetic operator symbol and **n** an arbitrary numeral. The main syntactic category is *States*, each element of which is of the form $< S, C >$ where S is an element of *Stack* and C an element of *Control*. An element of *Stack* can be a sequence or stack of values, possibly empty. Here we use the usual dot notation, . , for sequences and, since we are interpreting *Exp*, the only values required are numerals. As stated above, we use ϵ to represent the empty stack or more generally the empty sequence. So, for example, the sequence $x.\epsilon$ is the sequence which consists of exactly one element x. An element of *Control* is also a sequence, each element of which is either an expression from *Exp* or a special constant. To interpret *Exp* the only special constants required are the operator symbols.

The definition of $\longrightarrow$ is given in Figure 5.6 and is relatively straightforward. It consists of three rules, none of which requires premises, and are therefore expressed in a simple form. The first rule says that if the expression to be evaluated is already a value, i.e. if the expression on top of the control unit is simply a numeral, then it is moved to the value stack; the value of a numeral is itself. The second rule analyses expressions. If the expression on top of the control unit is e *op* e' then it is replaced by the three elements e, e' and *op*. Intuitively this says that in order to evaluate e *op* e' we must first evaluate the two objects e, e' and then apply *op* to the two

1. Syntactic categories

   st in *States*

   C in *Control*

   S in *Stack*

   $cons$ in *Constants*

   v in *Values*

2. Definitions

$$
\begin{array}{rcl}
st & ::= & < S, C > \\
S & ::= & \epsilon \mid v.S \\
C & ::= & \epsilon \mid e.C \mid cons.C \\
cons & ::= & op \\
v & ::= & \mathbf{n}
\end{array}
$$

Figure 5.5: States of the abstract machine for *Exp*

results. When op appears at the top of the control unit the time has come to apply it. By the construction of the machine the values of e and e' will at this stage be at the top of the value stack. So the third rule states that these two values, v and v', are replaced by the value $Ap(op, v, v')$. Once more, note that this description of the machine avoids the issue of how operations are performed; it merely assumes that there is a mechanism, the operation Ap, for carrying them out correctly.

Here is an example of the use of the machine to evaluate $(\mathbf{3} * \mathbf{4}) + (\mathbf{8} - \mathbf{2})$. Each state is obtained from the previous one by an application of the rule noted on the right-hand side.

$$
\begin{array}{ll}
\text{Rule Val} & < S, v.C > \longrightarrow < v.S, C > \\[1em]
\text{Rule Anlm} & < S, e \;\; op \;\; e'.C > \longrightarrow < S, e.e'.op.C > \\[1em]
\text{Rule Opm} & < v'.v.S, op.C > \longrightarrow < Ap(op, v, v').S, C >
\end{array}
$$

Figure 5.6: A simple abstract machine

$< \epsilon, (\mathbf{3} * \mathbf{4}) + (\mathbf{8} - \mathbf{2}) >$	
$< \epsilon, \mathbf{3} * \mathbf{4}.\mathbf{8} - \mathbf{2}.+ >$	Rule Anlm
$< \epsilon, \mathbf{3}.\mathbf{4}.*.\mathbf{8} - \mathbf{2}.+ >$	Rule Anlm
$< \mathbf{3}, \mathbf{4}.*.\mathbf{8} - \mathbf{2}.+ >$	Rule Val
$< \mathbf{4}.\mathbf{3}, *.\mathbf{8} - \mathbf{2}.+ >$	Rule Val
$< \mathbf{12}, \mathbf{8} - \mathbf{2}.+ >$	Rule Opm
$< \mathbf{12}, \mathbf{8}.\mathbf{2}.-.+ >$	Rule Anlm
$< \mathbf{8}.\mathbf{12}, \mathbf{2}.-.+ >$	Rule Val
$< \mathbf{2}.\mathbf{8}.\mathbf{12}, -.+ >$	Rule Val
$< \mathbf{6}.\mathbf{12}, + >$	Rule Opm
$< \mathbf{18}, \epsilon >$	Rule Opm.

The relation between the compiler *COMP* in Section 2.1 and the abstract machine should be apparent from this example. The compiler translates an expression into a sequence of constants and then this sequence is evaluated on the *STACK*-machine. The abstract machine interleaves these two processes. It translates expressions into much the same sequence of constants, using Rule Anlm, and evaluates the sequence using the Rules Valm and Opm. But the processes of translation and evaluation are intertwined. The relation with the Computation semantics is more interesting. According to this semantics, a first step in the evaluation of $(\mathbf{3}*\mathbf{4})+(\mathbf{8}-\mathbf{2})$ is

$$(\mathbf{3} * \mathbf{4}) + (\mathbf{8} - \mathbf{2}) \;\longrightarrow\; \mathbf{12} + (\mathbf{8} - \mathbf{2}).$$

It takes the abstract machine five steps to implement this. It is represented by

$$< \epsilon, (\mathbf{3} * \mathbf{4}) + (\mathbf{8} - \mathbf{2}) > \longrightarrow^{*} < \mathbf{12}, \mathbf{8} - \mathbf{2}.+ > .$$

More generally, each step at the level of the Computation semantics is reflected in the abstract machine by an application of Rule Opm. But these rules are interspersed with long sequences of applications of Rules Anlm and Val which are used to analyse expressions and to prepare for the next application of an operator symbol. Note, however, that there are some steps at the level of the Computation semantics which have no counterpart in the abstract machine. A typical example is

$$(\mathbf{3} * \mathbf{4}) + (\mathbf{8} - \mathbf{2}) \;\longrightarrow\; (\mathbf{3} * \mathbf{4}) + \mathbf{6}.$$

This is because the abstract machine, unlike the computation semantics, implements a left-to-right evaluation strategy.

The inferences in the Computation semantics, although much simpler than those in the Evaluation semantics, are more complicated than those for the abstract machine. Indeed, inferences for the rules governing the latter are the simplest possible; they always have depth one. So the extra steps which occur during an evaluation on the abstract machine reflect the structure of the inferences at the level of the Computation semantics.

We will now see how to modify this machine so as to interpret *Fpl*, the language defined in Section 3.4. We examine in turn the extra features of *Fpl* and see what extensions are required. The abstract syntax for the resulting machine is given in Figure 5.7 and the operational semantics in Figure 5.8, 5.9 and 5.10. As we go through each of the constructs in *Fpl* these will be explained. Let us first consider variables. To evaluate any expression containing variables we need an environment which associates with each variable a value. For the relatively abstract Computation and Evaluation semantics this is represented as a function from variables to values. Here we wish to be a little more concrete and will therefore represent environments as finite lists. Of course an environment is in general an infinite object as it associates with every variable, from a possibly infinite set of variables, a value. However, any particular expression will only ever contain a finite number of variables and therefore to evaluate it it will be sufficient to know only a finite part of the environment. So representing an environment as a finite list rather than a function is not very restrictive and it also gives an indication of how environments may be implemented on real machines. The abstract syntax of environments for variables is then given by:

$$env ::= \epsilon \mid (x, v).env.$$

To look up the value associated with a variable we search the list from left to right. Let us use

$$env, x \vdash v$$

to indicate that according to the environment env the value associated with x is v. The relation $\vdash$ is captured, at least for arithmetic variables, by the rules:

$$\frac{}{(x,v).env, x \vdash v} \qquad \frac{env, x \vdash v}{(y,v').env, x \vdash v}\ \text{if } x \neq y$$

The effect of these rules is as follows: to look up the value of a variable x in an environment env the environment env is searched from left to right until an entry of the form (x, v) is found; the required value of x is then the value v.

The abstract machine must now be augmented with a third component, namely an environment. So a state has the form $< S, env, C >$ and the next-state relation

1. Syntactic categories

   st in *States*

   C in *Control*

   S in *Stack*

   env in *Env*

   a in *Assoc*

   $cons$ in *Constants*

   F in *FunVar*

   v in *Values*

   bv in *BValues*

2. Definitions

$$\begin{array}{rcl}
st & ::= & < S, env, C > \\
S & ::= & \epsilon \mid v.C \mid bv.C \\
C & ::= & \epsilon \mid e.C \mid cons.S \\
cons & ::= & op \mid bop \mid < x, e > \mid if\ (e, e') \mid not \mid equal \mid pop \mid F \\
env & ::= & \epsilon \mid a.env \\
a & ::= & (x, v) \mid (bx, bv) \mid (F, (x, e)) \\
v & = & \mathbf{n} \\
bv & = & tt \mid ff
\end{array}$$

Figure 5.7: States of the abstract machine for *Fpl*

has the form

$$< S, env, C > \longrightarrow < S', env', C' > .$$

This will not radically change the definition of $\longrightarrow$ as the environment component is mostly ignored. It is only used in the one new rule

$$\frac{env, x \vdash v}{< S, env, x.C > \longrightarrow < v.S, env, C >}$$

which places the value of the variable x on the stack.

To interpret the full language *Fpl* no more additions will be made to the machine. Instead we will simply augment the definition of *Constants* and extend $\longrightarrow$

appropriately. The treatment of local declarations is a case in point. To evaluate

$$let\ \ x = e\ \ in\ \ e'$$

we must first evaluate e in the current environment, env say, and then evaluate e' in a modification of env. The first rule, Rule Locm1, analyses local declarations into its components:

$$< S, env,\ let\ \ x = e\ \ in\ \ e'.C > \longrightarrow < S, env, e. < x, e' > .C > .$$

Here we introduce a new constant $< x, e' >$ which will be treated like e' except that to evaluate it in an environment env we must actually evaluate e' in the modified environment $(x, v).env$ where the new value associated with x, namely v, is obtained from the top of the stack. The machine is constructed so that this value picked from the top of the stack is in fact the result of evaluating e. This is expressed by the following two rules:

$$< v.S, env, < x, e > .C > \longrightarrow < S, < x, v > .env, e.pop.C >$$

$$< S, < x, v > .env, pop.C > \longrightarrow < S, env, C > .$$

The new constant pop is used to clear the environment of the temporary association (x, v) when it is no longer needed. These two rules are called Rule Locm2 and Rule Pop respectively in Figure 5.9.

A similar technique may be used to implement function definitions. For simplicity let us assume that all function symbols have arity one, so a declaration consists of a sequence of definitions of the form

$$F(x) \Longleftarrow e'.$$

We can keep this information in the environment by storing pairs $(F, (x, e'))$. The presence of such a pair in the environment means that we are assuming a declaration which contains the definition $F(x) \Longleftarrow e'$. The user-defined function symbols can be handled by the abstract machine by the analysis rule

$$< S, env, F(e).C) \ \longrightarrow < S, env, e.F.C >$$

and the application rule

$$\frac{env, F \vdash (x, e')}{< v.S, env, F.C > \longrightarrow < S, (x, v).env, e'.pop.C >}$$

In the application rule when F occurs in the control it means that the function F is to be applied to the value at the top of the stack, v. So if $env, F \vdash (x, e')$, i.e. $F(x) \Longleftarrow e'$ occurs in the declaration, this means that the expression e' should be evaluated in

Rule Opm1	$< S, env, e\ \ op\ \ e'.C > \longrightarrow < S, env, e.e'.op.C >$
	$< S, env, be\ \ \ bop\ \ be'.C > \longrightarrow < S, env, be.be'.bop.C >$
Rule Notm1	$< S, env, Not\ \ be.C > \longrightarrow < S, env, be.not.C >$
Rule Eqm1	$< S, env, Equal(e, e').C > \longrightarrow < S, env, e.e'.equal.C >$
Rule Ifm1	$< S, env, If\ \ be\ \ \ Then\ \ e\ \ Else\ \ e'.C > \longrightarrow < S, env, be.\,if\ \ (e, e').C >$
Rule Funm1	$< S, env, F(e).C > \longrightarrow < S, env, e.F.C >$
Rule Locm1	$< S, env,\ let\ \ x = e\ \ in\ \ e'.C > \longrightarrow < S, env, e. < x, e' > .C >$

Figure 5.8: Analysis rules for the abstract machine

the environment modified by the association (x, v). The machine is so constructed that this value v is the value of the original expression e to which the symbol F was applied.

The constructors associated with boolean expressions may be handled with similar techniques. First we need to be able to store boolean associations (bx, bv) in environments and be able to look them up. Then in the analysis phase we introduce new constants:

$$< S, env, If\ \ be\ \ \ Then\ \ e\ \ Else\ \ e'.C > \longrightarrow < S, env, be.\,if\ \ (e, e').C >$$
$$< S, env, Equal(e, e').C > \longrightarrow < S, env, e.e'.equal.C >$$
$$< S, env, Not\ \ be.C > \longrightarrow < S, env, be.not.C >$$

In the case of $Equal(e, e')$ a new constant $equal$ is introduced. When the time comes to process this constant then the values of e and e' are at the top of the stack. So they are replaced by tt if they are the same and ff otherwise:

$$< v.v'.S, env, equal.C > \longrightarrow < tt.S, env, C > \ \text{ if } v = v'$$
$$< v.v'.S, env, equal.C > \longrightarrow < ff.S, env, C > \ \text{ if } v \neq v'.$$

Similar reasoning justifies the application rules for the other two constants $if\ \ (e, e')$ and not:

$$< tt.S, env,\ if\ \ (e, e').C > \longrightarrow < S, env, e.C >$$
$$< ff.S, env,\ if\ \ (e, e').C > \longrightarrow < S, env, e'.C >$$
$$< tt.S, env, not.C > \longrightarrow < ff.S, env, C >$$
$$< ff.S, env, not.C > \longrightarrow < tt.S, env, C > .$$

Rule Opm2 $\quad < v'.v.S, env, op.C > \longrightarrow < Ap(op, v, v').S, env, C >$

$$< bv'.bv.S, env, bop.C > \longrightarrow < Ap(bop, bv, bv').S, env, C >$$

Rule Varm $\quad \dfrac{env, x \vdash v}{< S, env, x.C > \longrightarrow < v.S, env, C >}$

$$\frac{env, bx \vdash bv}{< S, env, bx\,.C > \longrightarrow < bv\,.S, env, C >}$$

Rule Valm $\quad < S, env, v.C > \longrightarrow < v.S, env, C >$

$$< S, env, bv.C > \longrightarrow < bv.S, env, C >$$

Rule Notm2 $\quad < tt.S, env, not.C > \longrightarrow < ff.S, env, C >$

$$< ff.S, env, not.C > \longrightarrow < tt.S, env, C >$$

Rule Eqm2 $\quad \dfrac{v = v'}{< v.v'.S, env, equal.C > \longrightarrow < tt.S, env, C >}$

$$\frac{v \neq v'}{< v.v'.S, env, equal.C > \longrightarrow < ff.S, env, C >}$$

Rule Ifm2 $\quad < tt.S, env, \; if \;\; (e, e').C > \longrightarrow < S, env, e.C >$

$$< ff.S, env, \; if \;\; (e, e').C > \longrightarrow < S, env, e'.C >$$

Rule Funm2 $\quad \dfrac{env, F \vdash (x, e)}{< v.S, env, F.C > \longrightarrow < S, (x, v).env, e.pop.C >}$

Rule Locm2 $\quad < v.S, env, < x, e > .C > \longrightarrow < S, (x, v).env, e.pop.C >$

Rule Pop $\quad < S, (x, v).env, pop.C > \longrightarrow < S, env, C >$

Figure 5.9: Application rules for the abstract machine

$(F, (x, e)).env, F \vdash (x, e)$

$(x, v).env, x \vdash v \qquad\qquad (bx, bv).env, bx \vdash bv$

$\dfrac{env, F \vdash (x, e)}{a.env, F \vdash (x, e)}$
if a does not bind F

$\dfrac{env, x \vdash v}{a.env, x \vdash v}$
if a does not bind x

$\dfrac{env, bx \vdash bv}{a.env, bx \vdash bv}$
if a does not bind bx

Figure 5.10: Lookup rules for environments for an abstract machine

This completes our description of the abstract machine for *Fpl*. As has already been stated the complete abstract syntax is gathered in Figure 5.7 and all the rules defining the Computation semantics in Figures 5.8, 5.9 and 5.10. In the latter we use the phrases "a does not bind F", etc. to mean that a associates a value or expression to some variable other than F.

In view of the preceding discussion, these definitions should now be understandable to the reader. Let us consider an example. Let D be the single definition

$$\begin{array}{lll} Ev(x) \Longleftarrow & \mathit{If}\ \ \mathit{Equal}(x, \mathbf{0}) & \\ & \quad \mathit{Then}\ \ \mathbf{0} & \\ & \quad \mathit{Else}\ \ \ \mathit{If}\ \ \mathit{Equal}(x, \mathbf{1}) & \\ & \qquad\qquad \mathit{Then}\ \ \mathbf{1}\ \ \mathit{Else}\ \ Ev(x - \mathbf{2}). & \end{array}$$

This function returns $\mathbf{0}$ if its input is even and $\mathbf{1}$ if it is odd. For convenience, let e denote the body of the declaration and e_1 the expression

$$\mathit{If}\ \ \mathit{Equal}(x, \mathbf{1})\ \ \mathit{Then}\ \ \mathbf{1}\ \ \mathit{Else}\ \ Ev(x - \mathbf{2}).$$

We evaluate the expression $Ev(\mathbf{2})$ with respect to D using the abstract machine. To do so we assume that in the environment env we have the association $(Ev, (x, e))$. The machine is started in the initial configuration $< \epsilon, env, Ev(\mathbf{2}) >$. The description of the subsequent computation may be found in Figure 5.11; we omit the arrow $\longrightarrow$ and simply list the states through which the machine proceeds together with the corresponding rule.

The final state is $< \mathbf{0}, env, \epsilon >$ and therefore the value of $Ev(\mathbf{2})$ is $\mathbf{0}$. The computation consists of a large number of steps each of which is obtained by a trivial derivation using the rules which define $\longrightarrow$. It consists of waves of analysis, where

$< \epsilon, env, Ev(\mathbf{2}) >$	
$< \epsilon, env, \mathbf{2}.Ev >$	Rule Fun1
$< \mathbf{2}, env, Ev >$	Rule Valm
$< \epsilon, (x, \mathbf{2}).env, e.pop >$	Rule Funm2
$< \epsilon, (x, \mathbf{2}).env, Equal(x, \mathbf{0}).\ if\ \ (\mathbf{0}, e_1).pop >$	Rule Ifm1
$< \epsilon, (x, \mathbf{2}).env, x.\mathbf{0}.equal.\ if\ \ (\mathbf{0}, e_1).pop >$	Rule Eqm1
$< \mathbf{2}, (x, \mathbf{2}).env, \mathbf{0}.equal.\ if\ \ (\mathbf{0}, e_1).pop >$	Rule Varm
$< \mathbf{0}.\mathbf{2}, (x, \mathbf{2}).env, equal.\ if\ \ (\mathbf{0}, e_1).pop >$	Rule Valm
$< F, (x, \mathbf{2}).env,\ if\ \ (\mathbf{0}, e_1).pop >$	Rule Eqm2
$< \epsilon, (x, \mathbf{2}).env, e_1.pop >$	Rule Ifm2
$< \epsilon, (x, \mathbf{2}).env, Equal(x, \mathbf{1}).\ if\ \ (\mathbf{1}, Ev(x - \mathbf{2})).pop >$	Rule Ifm1
$< \epsilon, (x, \mathbf{2}).env, x.\mathbf{1}.equal.\ if\ \ (\mathbf{1}, Ev(x - \mathbf{2})).pop >$	Rule Eqm1
$< \mathbf{2}, (x, \mathbf{2}).env, \mathbf{1}.equal.\ if\ \ (\mathbf{1}, Ev(x - \mathbf{2})).pop >$	Rule Varm
$< \mathbf{1}.\mathbf{2}, (x, \mathbf{2}).env, equal.\ if\ \ (\mathbf{1}, Ev(x - \mathbf{2})).pop >$	Rule Valm
$< F, (x, \mathbf{2}).env,\ if\ \ (\mathbf{1}, Ev(x - \mathbf{2})).pop >$	Rule Eqm2
$< \epsilon, (x, \mathbf{2}).env, Ev(x - \mathbf{2}).pop >$	Rule Ifm2 .
$< \epsilon, (x, \mathbf{2}).env, x - \mathbf{2}.Ev.pop >$	Rule Funm1
$< \epsilon, (x, \mathbf{2}).env, x.\mathbf{2}. - .Ev.pop >$	Rule Opm1
$< \mathbf{2}, (x, \mathbf{2}).env, \mathbf{2}. - .Ev.pop >$	Rule Varm
$< \mathbf{2}.\mathbf{2}, (x, \mathbf{2}).env, -.Ev.pop >$	Rule Valm
$< \mathbf{0}, (x, \mathbf{2}).env, Ev.pop >$	Rule Opm2
$< \epsilon, (x, \mathbf{0}).(x, \mathbf{2}).env, e.pop.pop >$	Rule Funm2
$< \epsilon, (x, \mathbf{0}).(x, \mathbf{2}).env, Equal(x, \mathbf{0}).\ if\ \ (\mathbf{0}, e_1).pop.pop >$	Rule Ifm1
$< \epsilon, (x, \mathbf{0}).(x, \mathbf{2}).env, x.\mathbf{0}.equal.\ if\ \ (\mathbf{0}, e_1).pop.pop >$	Rule Eqm1
$< \mathbf{0}, (x, \mathbf{0}).(x, \mathbf{2}).env, \mathbf{0}.equal.\ if\ \ (\mathbf{0}, e_1).pop.pop >$	Rule Varm
$< \mathbf{0}.\mathbf{0}, (x, \mathbf{0}).(x, \mathbf{2}).env, equal.\ if\ \ (\mathbf{0}, e_1).pop.pop >$	Rule Valm
$< T, (x, \mathbf{0}).(x, \mathbf{2}).env,\ if\ \ (\mathbf{0}, e_1).pop.pop >$	Rule Eqm2
$< \epsilon, (x, \mathbf{0}).(x, \mathbf{2}).env, \mathbf{0}.pop.pop >$	Rule Ifm2
$< \mathbf{0}, (x, \mathbf{0}).(x, \mathbf{2}).env, pop.pop >$	Rule Valm
$< \mathbf{0}, (x, \mathbf{2}).env, pop >$	Rule Pop
$< \mathbf{0}, env, \epsilon >$	Rule Pop

Figure 5.11: A computation of the abstract machine

the analysis rules are used, followed by applications, where the application rules are used. Once more we can see a direct analogy with the derivation in the Computation semantics of

$$Ev(\mathbf{2}) \ \longrightarrow_A^* \ \mathbf{0}$$

which may be derived using the next-state relation $\longrightarrow_A$ defined in Section 5.1. In the latter there are far fewer steps, most of which correspond to the use of application rules in the Computation semantics of the abstract machine. The use of the analysis rules do not appear directly at the level of the Computation semantics of *Fpl* but are reflected indirectly in the derivation structure of the individual steps.

This completes our description of a Concrete operational semantics for the language *Fpl*. We have now given three different operational semantics for this language, each at different levels of abstraction. The Evaluation semantics is the most abstract and it simply formalises the result of evaluating programs without indicating how any computation for producing the result might proceed. The Computation semantics, on the other hand, describes — at least in some detail — possible computations which lead from the program to the result. Of course, these are rather abstract computations in that they only prescribe the primitive operations a program can perform or, more generally, the sequences of primitive operations which it can perform. Finally the Concrete operational semantics in this section is the most detailed semantics. It provides an abstract machine on which the language can be interpreted. Although this is a description of the architecture of a machine, it is still a relatively abstract description. Unnecessary detail is avoided and the actual behaviour of the machine is defined at the level of Computation semantics. It is possible to show that these three different semantic descriptions are consistent with each other. They merely represent three complementary views, at different levels of abstraction, of the same essential behaviour of programs. In Section 2.3 we have already seen how to relate the Computation semantics and the Evaluation semantics for the simple language *Exp*. There we proved

$$e \implies v \quad \text{if and only if} \quad e \rightsquigarrow v,$$

where $e \rightsquigarrow v$ means that if we execute e according to the Computation semantics we will eventually obtain the value v. This result can be extended to the language *Fpl* by showing that for any arithmetic expression e from *Fpl*,

$$D, \rho \vdash e \implies_A v \quad \text{if and only if} \quad D, \rho \vdash e \rightsquigarrow v.$$

A similar result relating the Evaluation semantics and the Computation semantics of *WhileL* may also be proved. To provide the link with the Concrete operational semantics one would have to show in addition that $D, \rho \vdash\vdash e \implies_A v$ if and only if whenever the machine is started in the configuration $< \epsilon, env, e >$, where the environment env contains the definitions in the declaration D, the machine will eventually stop and the final configuration will be $< v, env, \epsilon >$. The proof of these statements is not particularly straightforward and in this introductory text they will not be attempted. However, the more mathematically inclined reader may be tempted, in which case a review of Section 2.3. would be very helpful.

Questions

Q1 Use the Computation semantics of *Fpl* to evaluate the expression Fib($\mathbf{2}$) using the declaration

$$\begin{aligned} Fib(x) \Longleftarrow\ & \textit{If}\ \ Lt(x, \mathbf{2}) \\ & \textit{Then}\ \ x \\ & \textit{Else}\ \ Fib(x - \mathbf{1}) + Fib(x - \mathbf{2}) \end{aligned}$$

Here *Lt* is a boolean function which compares two numerals and returns true if the first is strictly less than the second and false otherwise.

Q2 Use the alternative Computation semantics for *Fpl* outlined at the end of Section 5.1, which does not use substitution, to evaluate the expression $H(\mathbf{15}, \mathbf{25})$ with respect to the declaration of $H(x, y)$ given in that section.

Q3 Let $D, \rho \vdash e \rightsquigarrow_A v$ if $D, \rho \vdash e \longrightarrow_A^* v$, i.e. if using the Computational semantics the expression e eventually evaluates to the value v. Show that $D, \rho \vdash e \ \Longrightarrow_A \ v$ implies $D, \rho \vdash e \rightsquigarrow_A v$.
Hint Use Rule induction on the definition of $\Longrightarrow_A$ over *Fpl*. Question 11 from Chapter 3 is also required.

Q4 Prove the converse of the previous question, that $D, \rho \vdash e \rightsquigarrow_A v$ implies $D, \rho \vdash e \ \Longrightarrow_A \ v$.
Hint Use the proof technique employed in Section 2.3 to prove the same result for the language *Exp*.

Q5 Use the Computation semantics of *WhileL* to evaluate the program, previously used in Section 4.3:

$$\begin{aligned} & z := \mathbf{0}; \\ & \textit{While}\ \ \textit{Not}\ \ (\textit{Equal}(x, \mathbf{0}))\ \textit{Do} \\ & \qquad\qquad z := z + y; \\ & \qquad\qquad x := x - \mathbf{1} \end{aligned}$$

with respect to the same store s, where $s(x) = \mathbf{2}, s(y) = \mathbf{3}$ and $s(z) = \mathbf{7}$.

Q6 Add to the language *WhileL* the new command

$$\textit{do}\ \ e\ \ \textit{times}\ \ C.$$

Intuitively this command executes C n times, where n is the value of the expression e. Extend the Computation semantics to this new command.

Q7 Augment the language *WhileL* with the variant of the *While* command:

$$\textit{Repeat}\ \ C\ \ \textit{Unless}\ \ be.$$

(See Question 7 of Chapter 4 for an intuitive explanation of this command.) Extend the Computation semantics to this new construct.

Q8 Extend the Computation semantics to the commands introduced in Questions 11 and 12 at the end of Chapter 4, namely the case statement

$$\textit{case}\ \ e\ \ \textit{of}\ \ G\ \ \textit{end}$$

and the expressions

$$\textit{Run}\ \ C\ \ \textit{Then}\ \ e\ \text{ and }\ \textit{Run}\ \ C\ \ \textit{Then}\ \ be.$$

Q9 Prove that the Computation semantics and the Evaluation semantics for *WhileL* coincide. This involves showing

$$(C, s) \ \Longrightarrow_C \ s' \quad \text{implies} \quad (C, s) \longrightarrow_C^* s'$$

and the converse

$$(C, s) \longrightarrow_C^* s' \ \text{ implies } \ (C, s) \ \Longrightarrow_C \ s',$$

where $(C, s) \longrightarrow_c^* s'$ means $(C, s) \longrightarrow_C^* (C', s')$ for some command C' such that $(C', s')\surd$.

Q10 Show the computation of the abstract machine which results from evaluating $U(\mathbf{2})$, where U is defined by as in Question 2 at the end of Chapter 3.

Q11 Do likewise for $Fib(\mathbf{2})$ where *Fib* is defined in Question 1 above.

Bibliography

[Gun92] Carl Gunter. *Semantics of Programming Languages.* MIT Press, 1992.

[Kah87] G. Kahn. Natural Semantics. Rapport de recherche no. 601, INRIA, Sophia-Antipolis, France, 1987.

[NN92] H. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction.* Wiley, 1992.

[P.A83] P.Aczel. *An introduction to inductive definitions.* North-Holland, 1983.

[Plo81] G. Plotkin. A structural approach to operational semantics. Report daimi fn-19, University of Aarhus, 1981.

[Sch86] D. Schmidt. *Denotational Semantics.* Allyn and Bacon, 1986.

[Sch88] D. Schmidt. *Denotational Semantics: a methodoilogy for language development.* Allyn and Brown, 1988.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages.* MIT Press, 1993.