# Semantics of programming languages (CS3017)
# Course Notes 2014-2015

Matthew Hennessy
Trinity College Dublin

December 3, 2014

# Contents

# Chapter 1

# Evaluating arithmetic expressions

In this introductory chapter we explain the idea of formal semantics for a programming language using as an example a very simple language for arithmetic expressions *Exp*, involving numerals and two operations, addition and multiplication. Anybody reading these notes will know very well how to evaluate these expressions. But our purpose is to use this language to explain the formalism we will use to give semantics to languages which are much more complicated than *Exp*.

## 1.1 Syntax

The syntax for a very simple language of *arithmetic expressions Exp* is given in Figure 1.1. It uses an auxiliary set of *numerals*, *Nums*, which are syntactic representations of the more abstract set of natural numbers $\mathbb{N}$. The natural numbers 0, 1, 2, ... are mathematical objects which exist in some abstract world of concepts. They have concrete representations in different languages. For example the natural number 5 is represented by the string of symbols *five* in English and the string *cinq* in French; the Romans represented it by the symbol *V*. In our language of arithmetic expressions it will be represented as the corresponding symbol in bold italic font $\mathtt{5}$.

In addition to the numerals the BNF schema in Figure 1.1 also uses two extra symbols, + and ×. Once more most people would know that these symbols are representations for binary mathematical operations on natural numbers, namely *addition* and *multiplication*. Thus the first line of Figure 1.1 says that there are three ways to construct an arbitrary expression $E$ in the language *Exp*:

(i) If $\mathtt{n}$ is an arbitrary numeral then it is also an arithmetic expression. From this we therefore already know that there an infinite number of arithmetic expressions, namely $\mathtt{0}$, $\mathtt{1}$, $\mathtt{2}$, ...

(ii) If we have already constructed two arithmetic expressions $E_1$ and $E_2$ then $E_1 + E_2$ is also an arithmetic expression in *Exp*.

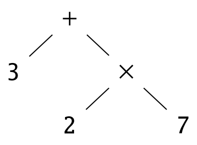
$$E \in Exp ::= \mathtt{n} \mid E + E \mid E \times E$$
$$\mathtt{n} \in Nums ::= \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \ldots$$

Figure 1.1: Syntax: arithmetic expressions

(iii) Similarly if $E_1$, $E_2$ are two expressions in *Exp* then $E_1 \times E_2$ is also an arithmetic expression in *Exp*.

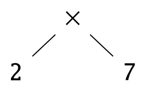
la composición de expresiones es una expresión

Here we take the view that schemas such as that in Figure 1.1 specify the *abstract syntax* of a language, rather than its *concrete syntax*. The latter is concerned with the precise linear sequences of symbols which are valid terms of the language whereas the former describes terms purely in terms of their structure. Another way of saying this is that the schema in Figure 1.1 describes the valid *abstract syntax trees* of the language, rather than linear sequences of symbols. Thus the following is a valid tree in the language *Exp*:

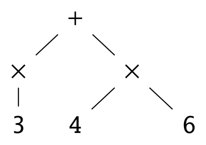


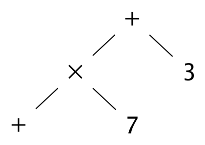
This is because it is formed by condition (ii) above because:

(a) 3 is a valid tree in *Exp*; this follows from condition (i)

(b) the object  is also in *Exp*. This in turn follows by condition (iii) above, because both the objects 2 and 7 are valid trees; these two statements are an instance of condition (i).

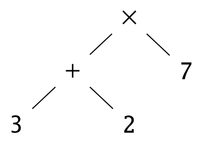
On the other hand trees such as





are not in the language *Exp*; no matter how we try to apply the rules (i) - (iii) above we will not be able to construct either of them.

However it would be tedious to have to continually draw these syntax trees and therefore throughout the notes we use a convention for their linear representation; this consists of using brackets in order to indicate the structure of expressions. Thus in linear representation the valid tree above will be rendered as $3 + (2 \times 7)$. The linear representation $(3 + 2) \times 7$ on the other hand represents a different tree, namely



This linear representation of abstract tress will be rather informal; for example there are many linear expressions, such as $3 + 2 \times 7$, which represent no abstract syntax tree. The over-riding principle will be that given an expression we should always know its structure; how it is constructed using the rules (i) (ii) and (iii) above.

## 1.2 Big-step semantics

Anybody with the least exposure to mathematics will know how to evaluate expressions in the language *Exp*; for example $3+(2\times7)$ evaluates to $17$ while $(3+2)\times7$ evaluates to $35$. However this might not be the case for more complicated languages, and therefore we need general methods for specifying how expressions are to be evaluated, or more abstractly what should be the result of evaluating an expression. We will illustrate these methods using the simple language *Exp*.

One approach would simply be to write a computer programme, an *evaluator* or *interpreter*, which inputs an arithmetic expression and outputs the correct result. However this is unsatisfactory for a number of reasons:

(i) As an explanation it is unnecessarily complicated. Writing the programme would involve all kinds of superfluous decisions about data-structures, and control flow.

(ii) It would also be overly prescriptive; the program would essentially give a specific algorithm for evaluating expressions, thereby offering a bias against other possibilities.

Suppose instead we merely wanted to *specify* what the result should be, rather than how the evaluation should proceed. One way to do this would be to publish a table consisting of all the possible expressions together with the numeral to which they should evaluate. Apart from being incredibly tedious this approach is doomed to failure as there are an infinite number of possible expressions. But as is made clear in the BNF description of the language in Figure 1.1, there is a simple structure to all expressions; this can be exploited to give a simple specification of what the result should be from any algorithm designed to evaluate an arbitrary expression.

But any such specification can only be understood by somebody who is familiar with the abstract arithmetic operations of addition and subtraction. Note that this is also true of *evaluators* or *interpreters*; it would be impossible to implement a program to evaluate expressions if the target language had no way to execute these arithmetic operations.

Suppose we want to evaluate an arbitrary expression $E \in Exp$. According to the description of *Exp* in Figure 1.1 there are three possibilities for the structure of $E$:

$$\text{(B-NUM)}\quad \frac{}{\mathtt{n} \Downarrow \mathtt{n}} \qquad\qquad \text{(B-ADD)}\quad \frac{E_1 \Downarrow \mathtt{n}_1 \qquad E_2 \Downarrow \mathtt{n}_2}{E_1 + E_2 \Downarrow \mathtt{n}_3}\ n_3 = \mathsf{add}(n_1, n_2)$$

evalues

Figure 1.2: Big-step semantics

(i) $E$ is some numeral $\mathtt{n}$: In this case the result of evaluation should obviously be the numeral $\mathtt{n}$ itself.

(ii) $E$ has the structure $E_1 + E_2$ for some (sub)-expressions $E_1$ and $E_2$. In this case the result of evaluating $E$ should be the numeral obtained by applying the binary addition operator to the results obtained from $E_1$ and $E_2$. Spelled out in more detail, if $\mathtt{n}_1$ is the result of evaluating $E_1$ and $\mathtt{n}_2$ is the result of evaluating $E_2$ then the result of evaluating $E$ should be the numeral $\mathtt{n}_3$ where $\mathsf{add}(n_1, n_2) = n_3$.

(iii) $E$ has the structure $E_1 \times E_2$ for some (sub)-expressions $E_1$ and $E_2$. In this case we proceed as in case (ii) but using the multiplication operator $\mathsf{mult}(-,-)$ in place of addition.

Note the use of numbers versus numerals in (ii) and (iii). Both $\mathsf{add}(-,-)$ and $\mathsf{mult}(-,-)$ are abstract mathematical operations on natural numbers; so in (ii) they are applied to the numbers $n_1, n_2$, to obtain the number $n_3$, and the result of the valuation is the corresponding numeral $\mathtt{n}_3$.

The specification given in (i)-(iii) above does not necessarily constitute a precise algorithm for evaluating expressions but it can be used by any reasonably intelligent person to calculate the prescribed result. For example the result of evaluating $(\mathtt{2} + \mathtt{6}) + (\mathtt{2} \times \mathtt{7})$ should be the numeral $\mathtt{22}$. This follows by an application of (ii) because:

(a) $(\mathtt{2} + \mathtt{6}) + (\mathtt{2} \times \mathtt{7})$ has the form $E_1 + E_2$ where $E_1$ is $\mathtt{2} + \mathtt{6}$ and $E_2$ is $\mathtt{2} \times \mathtt{7}$

(b) the result of evaluating $\mathtt{2} + \mathtt{6}$ should be $\mathtt{8}$

(c) the result of evaluating $\mathtt{2} \times \mathtt{7}$ should be $\mathtt{14}$

(d) and $\mathsf{add}(8, 14)$ is the number 22.

Of course this is not the complete justification of why $(\mathtt{2}+\mathtt{6})+(\mathtt{2}\times\mathtt{7})$ should evaluate to $\mathtt{22}$. In addition we need to justify steps (b) and (c) above; these in turn can be justified using applications of the principles (ii) and (iii) respectively.

With some thought the reader should be convinced that these principles, (i), (ii), and (iii), are sufficient to determine the value of any expression from *Exp* no matter how complicated. However they are expressed in natural language (English), which is notoriously prone to mis-interpretation and mis-understanding. For *Exp*, a very simple language, this is not the case, but for more complicated languages it is better to avoid the vagaries of natural language. So instead we propose to replace specifications such as (i) - (iii) above with formal logical systems which do not suffer from the defects of natural language.

The idea is to use logical rules whose general format is given by:

$$\frac{\text{hypothesis} \quad \ldots\text{hypothesis}}{\text{conclusion}}\;\text{name}\quad(\text{side-condition}) \tag{1.1}$$

Each rule has

- at least one conclusion, written underneath the line
- a list, possibly empty, of hypotheses, written above the line
- a side-condition, again possibly empty
- a name with which we can refer to the rule.

The intuition is that if all the hypotheses hold, and the side-condition holds, then the conclusion also holds.

Let us now see how we can recast the informal specification of the semantics above using this form of logical rules. The predicate in which we are interested is: *the expression $E$ should evaluate to the numeral* $\mathtt{n}$. Let us denote this English phrase with a mathematical predicate or *judgement*

$$E \Downarrow \mathtt{n}$$

Now what we want is a set of rules which determine valid instances of this predicate. Two such rules are given in Figure 3.2, corresponding to the informal specifications (i) and (ii) above; the missing third rule can be supplied by the reader to correspond with clause (iii). The first rule, (B-NUM), has no hypothesis and no side condition; such rules are refered to as *axioms*. Thus it says that $\mathtt{n} \Downarrow \mathtt{n}$ for every numeral $\mathtt{n}$; thus it corresponds to the informal specification (i) above. The second rule, (B-ADD), corresponds to the informal specification (ii); it has two hypotheses, namely that $E_1 \Downarrow \mathtt{n}_1$ and $E_2 \Downarrow \mathtt{n}_2$ and one side-condition about natural numbers, $n_3 = \mathsf{add}(n_1, n_2)$. If these hypotheses are known to hold and the side-condition is true then the conclusion $E_1 + E_2 \Downarrow \mathtt{n}_3$ is also true. Imp.

These rules can now be used formally to determine when, for a particular expression $E$ and numeral $\mathtt{n}$, the judgement $E \Downarrow \mathtt{n}$ is valid. Valid judgements are those which can be derived by any sequence of applications of the defining rules. Here is an example of such a derivation, which determines that the judgement $3 + (2 + 1) \Downarrow 6$ is valid, that is, the evaluation of the expression $3 + (2 + 1)$ should evaluate to the numeral $6$.

$$\dfrac{\dfrac{}{3 \Downarrow 3}\,(\text{B-NUM}) \qquad \dfrac{\dfrac{}{2 \Downarrow 2}\,(\text{B-NUM}) \quad \dfrac{}{1 \Downarrow 1}\,(\text{B-NUM})}{(2+1) \Downarrow 3}\,(\text{B-ADD})}{3 + (2 + 1) \Downarrow 6}\,(\text{B-ADD})$$

The derivation is presented as an inverted tree, with the required judgement to be verified, $3 + (2 + 1) \Downarrow 6$, at the root. The tree is generated by applications of the defining

$$\dfrac{\dfrac{\dfrac{}{2 \Downarrow 2}\,(\textsc{b-num}) \qquad \dfrac{}{6 \Downarrow 6}\,(\textsc{b-num})}{(2+6) \Downarrow 8}\,(\textsc{b-add}) \qquad \dfrac{\dfrac{}{2 \Downarrow 2}\,(\textsc{b-num}) \qquad \dfrac{}{7 \Downarrow 7}\,(\textsc{b-num})}{(2 \times 7) \Downarrow 14}\,(\textsc{b-mult})}{(2+6)+(2 \times 7) \Downarrow 22}\,(\textsc{b-add})$$

Figure 1.3: An example derivation in the big-step semantics

rules, with the terminating leaves being generated by axioms. In this example we have three applications of the axiom (B-NUM) and two applications of the rule (B-ADD).

Another example derivation is given in Figure 1.3; it makes reference to the (obvious) missing rule (B-MULT) for dealing with expressions of the form $E_1 \times E_2$. This is a formal justification of the valid judgement $(2+6)+(2 \times 7) \Downarrow 22$ corresponding to the informal justification given in natural language in the clauses (a)-(d) on page 6.

We now sum up what has been achieved in this section. To do so let us introduce the notation

$$\vdash_{big} E \Downarrow \mathtt{n} \tag{1.2}$$

to mean that there is some derivation of the judgement $E \Downarrow \mathtt{n}$ using the three rules (B-NUM), (B-ADD) and (B-MULT). For example, because Figure 1.3 exhibits a derivation of the judgement $2+6)+(2 \times 7) \Downarrow 22$, we can conclude $\vdash_{big} 2+6)+(2 \times 7) \Downarrow 22$. Then we can say that we have given a formal semantics to the language *Exp*. By this we mean that if somebody asks the question: *To what value should the expression E evaluate?* we can answer: $E$ should evaluate to a numeral $\mathtt{n}$ such that $\vdash_{big} E \Downarrow \mathtt{n}$.

Before moving on we should say a few words about the format of the logical rules which we use, in (1.1) above. We have not been very specific about the contents of the various components, *hypothesis*, *conclusion* and *side-condition*. In general the purpose of a rule is to constrain some predicate, the focus of the semantic definition. In this case the predicate is $\Downarrow$, a binary infix predicate between expressions and numerals. Consequently it is natural that the *conclusion*, and very often the *hypotheses*, be particular instances of this predicate; this is the case in the rules (B-ADD) and (B-NUM) in Figure 3.2. On the other hand *side-condition* should concern auxiliary predicates and functions which play a role, but a minor role, in the definition of the main predicate. We have seen that it is not possible to understand the semantics of *Exp* without knowing that the symbols $+$ and $\times$ refer to the mathematical functions $\mathsf{add}(-,-)$ and $\mathsf{mult}(-,-)$ on natural numbers; and in our rules the side-conditions refer to properties of these auxiliary functions. Thus although one might consider an alternative rule such as

$$\overset{(\textsc{b-add.alt})}{\dfrac{E_1 \Downarrow \mathtt{n}_1 \qquad n_3 = \mathsf{add}(n_1, n_2)}{E_1 + E_2 \Downarrow \mathtt{n}_3}} \quad E_2 \Downarrow \mathtt{n}_2$$

the original rule (B-ADD) in Figure 3.2 is to be preferred.

(S-LEFT)
$$\frac{E_1 \rightarrow E_1'}{(E_1 + E_2) \rightarrow (E_1' + E_2)}$$

(S-N.RIGHT)
$$\frac{E_2 \rightarrow E_2'}{(\mathtt{n} + E_2) \rightarrow (\mathtt{n} + E_2')}$$

(S-ADD)
$$\frac{}{(\mathtt{n}_1 + \mathtt{n}_2) \rightarrow \mathtt{n}_3} \quad n_3 = \mathsf{add}(n_1, n_2)$$

Figure 1.4: Small-step semantics: default rules

We should also point out that a rule such as (B-ADD) is actually a *meta-rule*, that is formally represents an infinite number of concrete rules, obtained by instantiating the *meta-variables* $E_1$, $E_2$, $\mathtt{n}_1$, $\mathtt{n}_2$ and $\mathtt{n}_1$. Thus among the many instances of (B-ADD) are

$$\frac{3 \times 7 \Downarrow 4 \qquad 8 \Downarrow 2}{(3 \times 7) + 8 \Downarrow 6} \; 6 = \mathsf{add}(4, 2) \qquad \frac{4 + 2 \Downarrow 9 \qquad 8 + 1 \Downarrow 3}{(3 \times 7) + (8 + 1) \Downarrow 12} \; 12 = \mathsf{add}(9, 3)$$

However the vast majority of these concrete instances are useless; if the premises can not be established then they can not be employed in any valid derivation.

## 1.3 Small-step semantics

The big-step semantics of the previous section is not very constraining; it prescribes what the answer should be when an expression is evaluated but says nothing about how the actual evaluation is to proceed. For example, to evaluate $(3 + 7) + (8 \times 1)$ we know that two additions have to preformed and one multiplication; but the big-step semantics does not decree in what order these are to be carried out. For some languages, for example those with *side-effects*, the order of evaluation is important. In this section we see an alternative semantics for *Exp* in which constraints on the order of the basic operations can be made. In particular it will prescribe, indirectly, that the order of evaluations should be from left to right.

The idea is to design a predicate on expressions which decrees which operation is to be performed first, and then describes the result of performing this operation. This is achieved indirectly by defining judgements of the form

$$E_1 \rightarrow E_2$$

to be read as: *after performing one step of evaluation* of the expression $E_1$ the expression $E_2$ remains to be evaluated; thus this judgement prescribes

- the first operation to be performed, transforming $E_1$ into $E_2$
- the remaining operations to be performed, embodied indirectly in the the residual $E_1$.

The rules defining this small step relation $\rightarrow$ are given in Figure 1.4, although we leave it to the reader to design the two rules, similar to (S-LEFT) and (S-N.RIGHT), for dealing with expressions of the form $E_1 \times E_2$. Let us write

$$\vdash_{sm} E_1 \rightarrow E_2$$

to mean that there is a derivation of the judgement $E_1 \rightarrow E_2$ using these rules. Thus we have

$$\vdash_{sm} (3 + 7) + (8 + 1) \rightarrow 10 + (8 + 1)$$

because of the following derivation:

$$\dfrac{\dfrac{}{3 + 7 \rightarrow 10}\ (\text{S-ADD})}{(3 + 7) + (8 + 1) \rightarrow 10 + (8 + 1)}\ (\text{S-LEFT})$$

As another example we have

$$\vdash_{sm} 10 + (8 + 1) \rightarrow 10 + 9$$

because the following is a valid derivation:

$$\dfrac{\dfrac{}{8 + 1 \rightarrow 9}\ (\text{S-ADD})}{10 + (8 + 1) \rightarrow 10 + 9}\ (\text{S-N.RIGHT})$$

On the other hand we do *not* have

$$\vdash_{sm} (3 + 7) + (8 + 1) \rightarrow (3 + 7) + 9$$

because no matter how inventive we are with the rules in Figure 1.4 we will not be able to construct a derivation of the judgement $(3 + 7) + (8 + 1) \rightarrow (3 + 7) + 9$; the reader is invited to try.

By trying various examples readers should be able to convince themselves that if $\vdash_{sm} E_1 \rightarrow E_2$ then $E_2$ is obtained from $E_1$ by executing the left-most occurrence of an operator, $+, \times$, which has both its operands already evaluated. For example we have

$$\vdash_{sm} (3 + 4) + (5 + 6) \rightarrow 7 + (5 + 6)$$
$$\vdash_{sm} 3 + (4 + (5 + 6)) \rightarrow (3 + (4 + 11)$$
$$\vdash_{sm} (3 + (4 + 5)) + 6 \rightarrow (3 + 9) + 6)$$

How do we use the small-step semantics to evaluate an expression, as in the previous section? We construct derivations again and again until a numeral is obtained. For example we have seen that $\vdash_{sm} (3+7)+(8+1) \rightarrow 10+(8+1)$ and $\vdash_{sm} 10+(8+1) \rightarrow 10+9$. In other words in two steps the expression $(3 + 7) + (8 + 1)$ can be reduced to $10 + 9$;

this we write as $\vdash_{sm} (3+7)+(8+1) \rightarrow^2 \mathtt{10}+9$. More generally for any natural number $k \geq 0$ we write

$$E_0 \rightarrow^k E_k$$

if $E_0$ can be reduced to $E_k$ in $k$ steps; that is, there are intermediate expressions $E_i$ such that

$$\vdash_{sm} E_o \rightarrow E_1 \qquad \vdash_{sm} E_1 \rightarrow E_2 \ldots\ldots \vdash_{sm} E_{k-1} \rightarrow E_k$$

This includes the case when $k$ is 0, when $E_k$ must be the same as $E_0$; that is in 0 steps $E_0$ can only reduce to itself. For example the reader should check the following judgements, by showing that derivations can be obtained for appropriate intermediate expressions:

$$\begin{aligned}(3+(4+5))+6 &\rightarrow^2 12+6\\ 3+(4+(5+6)) &\rightarrow^2 3+15\\ (3+7)+(8+1) &\rightarrow^3 19\\ 3+(4+(5+6)) &\rightarrow^0 3+(4+(5+6))\end{aligned}$$

To fully evaluate an expression we need to indefinitely apply the operations + and × until eventually a final numeral is obtained. Let us write

$$E \rightarrow^* \mathtt{n}$$

to mean that there is some natural number $k \geq 0$ such that $E \rightarrow^k \mathtt{n}$; in other words $E$ can be reduced to the numeral $\mathtt{n}$ in some number $k$ steps. The reader should verify that the following judgements are true, by instantiating the required number $k$:

$$\begin{aligned}(3+7)+(8+1) &\rightarrow^* 19\\ (3+4)+(5+6) &\rightarrow^* 18\\ 3+(4+(5+6)) &\rightarrow^* 18\end{aligned}$$

So just as the big-step semantics associates a value $\mathtt{n}$ to an expression $E$, via the judgements $\vdash_{big} E \Downarrow \mathtt{n}$, the small-step semantics provides an alternative method for doing so, via the slightly more complicated judgements $\vdash_{sm} E \rightarrow^* \mathtt{n}$.

## 1.4 Parallel evaluation

As we have seen, the small-step semantics prescribes a particular order in which the operators in an expression are applied, namely *left-to-right*. Suppose we wish to relax this; suppose we just want to dictate that all the operators are applied but wish to leave the precise sequencing open. One of the roles of a formal semantics is to act as a reference for compiler writers or implementers. Leaving the order of evaluation open could then allow, for example, compiler writers to take advantage of technologies such as multi-core to increase the efficiency of an implementation.

$$\text{(S-LEFT)}\ \frac{E_1 \rightarrow_{ch} E_1'}{(E_1 + E_2) \rightarrow_{ch} (E_1' + E_2)} \qquad \text{(S-RIGHT)}\ \frac{E_2 \rightarrow_{ch} E_2'}{(E_1 + E_2) \rightarrow_{ch} (E_1 + E_2')}$$

$$\text{(S-ADD)}\ \frac{}{(\mathtt{n}_1 + \mathtt{n}_2) \rightarrow_{ch} \mathtt{n}_3}\ n_3 = \mathsf{add}(n_1, n_2)$$

Figure 1.5: Parallel semantics

In Figure 1.5 we give an alternative small-step semantics, with judgements of the form $E_1 \rightarrow_{ch} E_2$, with the subscript referring to *choice*. Two rules are inherited from Figure 1.4 but the rule (S-N.RIGHT) is replaced with the less restrictive (S-RIGHT). The net effect of the presence of the two rules (S-LEFT) and (S-RIGHT) is that when evaluating an expression of the form $E_1 + E_2$ the compiler or interpreter may choose to work on either of $E_1$ or $E_2$. For example we have the derivation:

$$\frac{\overline{\mathtt{8} + \mathtt{1} \rightarrow_{ch} \mathtt{9}}\ \text{(S-ADD)}}{(\mathtt{3} + \mathtt{7}) + (\mathtt{8} + \mathtt{1}) \rightarrow_{ch} (\mathtt{3} + \mathtt{7}) + \mathtt{9}}\ \text{(S-RIGHT)}$$

Using $\vdash_{ch} E_1 \rightarrow_{ch} E_2$ to denote the fact that the judgement $E_1 \rightarrow_{ch} E_2$ can be derived using the rules from Figure 1.5, we therefore have

$$\vdash_{ch} (\mathtt{3} + \mathtt{7}) + (\mathtt{8} + \mathtt{1}) \rightarrow_{ch} (\mathtt{3} + \mathtt{7}) + \mathtt{9} \tag{1.3}$$

in addition to

$$\vdash_{ch} (\mathtt{3} + \mathtt{7}) + (\mathtt{8} + \mathtt{1}) \rightarrow_{ch} \mathtt{10} + (\mathtt{8} + \mathtt{1}) \tag{1.4}$$

Recall from the previous section that this reduction (1.4) is not possible in the standard *left-to-right* semantics. On the other hand note that every application of the rule (S-N.RIGHT) is also an application of the more general (S-RIGHT). This means that any derivation in the *left-to-right* semantics is also a derivation in the *parallel* semantics. It follows that

$$\vdash_{sm} E_1 \rightarrow E_2 \text{ implies } \vdash_{ch} E_1 \rightarrow_{ch} E_2 \tag{1.5}$$

In other words the *parallel* semantics is more general than the *left-to-right*; it allows all the derivations of the *left-to-right* semantics but in addition it allows others such as (1.4) above.

## 1.5 Questions questions

We have now seen three different semantics for the simple language of expressions *Exp*, and various questions arise naturally. For example, intuitively we expect every expression in *Exp* to have a corresponding value. In terms of the big-step semantics we expect the following to be true:

(Q1) For every expression $E$ in *Exp* there exists some numeral $\mathtt{n}$ such that $\vdash_{big} E \Downarrow \mathtt{n}$.

The advantage of a formal semantics is that statements such as (Q1) can be formally proved, or indeed disproved. The predicate $\Downarrow$ between expressions and numerals is formally defined using a set of logical rules, those in Figure 3.2, and therefore (Q1) amounts to a mathematical statement about the mathematical object $\Downarrow$. As such it is either mathematically true or false, which can be demonstrated using standard mathematical techniques. These techniques will be seen in the next chapter.

The same property, often refered to as **Normalisation**, can also be asked of the other two semantics we have seen. These amount to:

(Q2) For every expression $E$ in *Exp* there exists some numeral $\mathtt{n}$ such that $E \rightarrow^{*} \mathtt{n}$.

(Q3) For every expression $E$ in *Exp* there exists some numeral $\mathtt{n}$ such that $E \rightarrow^{*}_{ch} \mathtt{n}$.

Again because these are formal mathematical statements we will see how they can be demonstrated formally.

Another property we would naturally expect of a mechanism for evaluating expressions is a form of *internal consistency*. It would be unfortunate if there was some expression with multiple possible values; that is some expression $E$ such that the first time it is evaluated we would get $E \Downarrow \mathtt{n}_1$ while a subsequent evaluation gives $E \Downarrow \mathtt{n}_2$ where $n_2$ is different than $n_1$. The property which rules out this phenomenon is refered to as **Determinacy**. For each of the three semantics this is defined as follows:

If $\vdash_{big} E \Downarrow \mathtt{n}_1$ and $\vdash_{big} E \Downarrow \mathtt{n}_2$ then $n_1 = n_2$. (Q4)

If $E \rightarrow^{*} \mathtt{n}_1$ and $E \rightarrow^{*} \mathtt{n}_2$ then $n_1 = n_2$. (Q5)

If $E \rightarrow^{*}_{ch} \mathtt{n}_1$ and $E \rightarrow^{*}_{ch} \mathtt{n}_2$ then $n_1 = n_2$. (Q6)

The combination of Normalisation and Determinacy means that each of the semantics we have developed for *Exp* determines one and only one value for every expression.

There are also interesting questions involving the consistency between the different semantics. For example it would be unfortunate if, for some some expression $E$, one semantics gave `20` as the resulting value, while another gave `25`. Ensuring that this can not arise amounts to proving mutual consistency of the different semantics. Specifically it would require proofs for the following mathematical statements:

$\vdash_{big} E \Downarrow \mathtt{n}$ implies $E \rightarrow^{*} \mathtt{n}$ (Q7)

$E \rightarrow^{*}_{ch} \mathtt{n}$ implies $\vdash_{big} E \Downarrow \mathtt{n}$ (Q8)

These, together with (1.5) above, will mean that each of the three different semantics will associate exactly the same value with a given expression $E$.

# Chapter 2

# Induction, in all its forms

We start with a review of *mathematical induction*, a very powerful proof method for proving properties which hold of all natural numbers. Recall that the set of natural numbers $\mathbb{N}$ is infinite so we cannot simply demonstrate that the property in question holds for each particular number. In Chapter 2.2 we then see that this proof method can be generalised to any set of objects which share in some sense a common structure; to be more precise there must be some collection of operations, or *constructors*, with which all objects in the set can be constructed. This more general proof method is called *structural induction*. It is exemplified first by considering the set of *binary trees* but the main application is to the language of arithmetic expressions *Exp* from Chapter 1. We show how all the properties of *Exp* discussed in Chapter 1.5 can be proved using *structural induction*.

However it turns out that in general *structural induction* will not be sufficiently powerful for our purposes. In Chapter 3 we will see a language *While* for which *structural induction* is inadequate. In particular some of the properties discussed in Chapter 1.5 hold also for all programs in *While*, but to prove some of them we will need a more powerful form of induction. This is called *rule induction* and is the topic of Chapter 2.3.

## 2.1 Mathematical Induction

The simplest form of induction is mathematical induction, that is to say, induction over the natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$. The principle can be described thus:

Given a property $P(-)$ of natural numbers, to prove that $P(n)$ holds for *all* natural numbers $n \in \mathbb{N}$, it is enough to

(i) **Base case:** prove that $P(0)$ holds, and

(ii) **Inductive case:** under the assumption that $P(k)$ holds for an arbitrary natural number $k$, prove that the statement $P(k + 1)$ follows.

The **Inductive case:** requires you to provide a *hypothetical argument*; you do not prove that the property $P(k + 1)$ actually holds; instead you show that *if* for some arbitrary and unknown number $k$ the statement $P(k)$ were true then the property $P(k + 1)$ would logically follow. In (ii) $P(k)$ is often referred to as the *inductive hypothesis*, abbreviated to (IH). So it could be rephrased as

(ii) **Inductive case:** Assume the inductive hypothesis (IH) $= P(k)$ for some arbitrary but unknown number $k$. From IH show that the statement $P(k + 1)$ follows.

It should be clear why this principle is valid: if we can prove (i) and (ii) above, then we know

- $P(0)$ holds, by (i).
- Since $P(0)$ holds, $P(1)$ holds, by (ii).
- Since $P(1)$ holds, $P(2)$ holds by (ii).
- Since $P(2)$ holds, $P(3)$ holds by (ii).
- And so on…

Therefore, $P(n)$ holds for any $n$, regardless of the size of $n$.

This conclusion can only be drawn because every natural number can be reached by starting at zero and adding one repeatedly. The two components of the principle of induction can be read as saying

- Prove that the property $P$ is true at the place where you start, that is 0.
- Prove that the operation of adding one *preserves* $P$, that is, if $P(k)$ is true then $P(k + 1)$ is true.

Since every natural number can be *built* by starting at zero and adding one repeatedly, every natural number has the property $P$: as you build the number, $P$ is true of everything you build along the way, and it's still true when you've built the number you're really interested in.

### 2.1.1 An example proof by mathematical induction

Suppose we are set the problem of proving the statement:

$$\boxed{\text{For every natural number } n,\ (8^n - 2^n) \text{ is divisible by 6}}$$

How do we go about it ? Since it is a statement about *every natural number*, then it will probably involve a proof by *mathematical induction*. So

(Step 1:) Identify the precise statement $P(n)$ which needs to be proved. In this case

$P(n)$: $8^n - 2^n$ is divisible by 6

Every proof (by mathematical induction) of a statement

*For every natural number n the statement $P(n)$ is true* always has the same structure. There are always two cases, the **Base case**, and the **Inductive step**. The next step is

(Step 2:) State the **Base case**, namely $P(0)$. In this example this amounts to

$$(8^0 - 2^0) \text{ is divisible by } 6$$

Then

(Step 3:) Use your ingenuity, and repertoire of mathematical facts, to prove the **Base case**. This is usually relatively straightforward. In this example the proof revolves around the two arithmetic facts, $m^0$ is 1 for any $m$, and 0 is divisible by any number, thus divisible by 6.

Having finished the **Base case** we move on to

(Step 4:) State the **Inductive step**. This is *always* the hypothetical inference:

For an arbitrary natural number $k$, $P(k)$ implies $P(k+1)$

Even though at this stage you might not know how you are going to prove this, it is best to unravel the statement. In other words write down

(a) what the inductive hypothesis actually is

(b) what we are required to deduce from it.

In this example we have

(a) We are assuming

$$(8^k - 2^k) \text{ is divisible by } 6 \qquad (IH)$$

for some aribtrary $k$.

(b) We are required to deduce from (IH) that

$$(8^{(k+1)} - 2^{(k+1)}) \text{ is divisible by } 6 \qquad \textit{required statement}$$

The next step is

(Step 5:) Show how the *required statement* follows from the inductive hypothesis (IH).

This is always non-trivial, and requires ingenuity. Normally it means massaging the *required statement* until somewhere inside it you can see the possibility for applying the inductive hypothesis (IH). For this example see the proof given in Figure 2.1.

Having completed the **Base case** and the **Inductive step**, the overall proof is now completed. Finally

(Step 6:) Write up the proof in a coherent manner, showing it's structure, as we have just outlined it.

For an example write-up of a proof see Figure 2.1. The layout used there can be used for *any* proof by mathematical induction.

Let $P(n)$ be the statement $(8^n - 2^n)$ is divisible by 6.
We prove by mathematical induction that the statement $P(n)$ is true, for every natural number $n$.

**Proof:** There are two cases.

**Base case:** We prove $P(0)$ is true; namely $(8^0 - 2^0)$ is divisible by 6.
The proof is by direct calculations, using the fact that $m^0 = 1$ for any number $m$. So

$$\begin{aligned} 8^0 - 2^0 &= 1 - 1 \\ &= 0 \end{aligned}$$

By the definition of *division*, 0 is divisible by 6; it follows that $(8^0 - 2^0)$ is divisible by 6. This is the end of the **Base case**.

**Inductive case:** We have to prove that for an arbitrary natural number $k$, the hypothetical statement $P(k)$ *implies* $P(k+1)$ is true. To this end suppose $P(k)$ is true. So we are assuming, for some arbitrary $k$,

$$(8^k - 2^k) \text{ is divisible by } 6 \qquad (IH)$$

Using this inductive hypothesis (IH) we have to show $P(k+1)$ follows, namely

$$8^{(k+1)} - 2^{(k+1)} \text{ is divisible by } 6 \tag{2.1}$$

First let us manipulate the expression in question:

$$\begin{aligned} 8^{(k+1)} - 2^{(k+1)} &= 8 * 8^k - 2^{(k+1)} \\ &= 8 * (8^k - 2^k) + 8 * 2^k - 2^{(k+1)} \\ &= 8 * (8^k - 2^k) + 8 * 2^k - 2 * 2^k \\ &= 8 * (8^k - 2^k) + 6 * 2^k \end{aligned}$$

So we only have to prove $8 * (8^k - 2^k) + 6 * 2^k$ is divisible by 6. But

(a) by the inductive hypothesis (IH), we know that $(8^k - 2^k)$ is divisible by 6, and therefore $8 * (8^k - 2^k)$ is also divisible by 6

(b) by definition, $6 * 2^k$ is divisible by 6

(c) by properties of addition, if $A$ is divisible by 6, and $B$ is divisible by 6 then so is $A + B$.

Applying (a), (b), and (c) we can conclude $8*(8^k-2^k)+6*2^k$ is divisible by 6. In other words we have established (2.1), from the Inductive Hypothesis (IH).
This is the end of the **inductive case**.

It follows by mathematical induction, that $P(n)$ is true for every natural number $n$.

Figure 2.1: A proof by mathematical induction

### 2.1.2 Defining functions using mathematical induction

As well as using induction to prove properties of natural numbers, we can use it to define functions which operate on natural numbers. Just as proof by induction proves a property $P(n)$ by considering the case of zero and the case of adding one to a number known to satisfy $P$, so definition of a function $f$ by induction works by giving the definition of $f(0)$ directly, and building the value of $f(k+1)$ out of the supposed value of $f(k)$.

Restating this principle, to define a function $f : \mathbb{N} \to X$ it is sufficient to

(i) **Base case:** define $f(0)$ to be some element in the range $X$

(ii) **Inductive step:** show how to calculate $f(k+1)$ in terms of $f(k)$, possibly using additional operations.

Mathematical induction ensures us that if (i) and (ii) are carried out then we are assured that $f$ is indeed defined for every $n \in \mathbb{N}$.

For example suppose we want to define the function $sum : \mathbb{N} \to \mathbb{N}$ where $sum(n)$ returns the sum of the first $n$ natural numbers $0 + 1 + 2 + \ldots + n$. Using induction the function is fully defined by the two clauses

(i) **Base case:** $sum(0) = 0$

(ii) **Inductive step:** $sum(k+1) = sum(k) + k$

In (i) we have identified directly $sum(0)$ while in (ii) we have shown how to calculate $sum(k+1)$ on the assumption that we already know $sum(k)$; specifically this says that to obtain the value of $sum(k+1)$ you add the number $k$ to the supposed value of $sum(k)$.

There is actually a well-known formula for calculating the sum of the first $k$ numbers, namely $\frac{k(k+1)}{2}$; and mathematical induction can be used to prove this assertion.

**Exercise 1** *Use mathematical induction to prove that* $sum(n) = \frac{n(n+1)}{2}$ *for every* $n \in \mathbb{N}$.

□

As another example of the use of mathematical induction let us reconsider the informal notation $E \to^n F$ we have used for executing the small-step semantics of the language *Exp*. We can now formally define these relations as follows:

(i) **Base case:** $E \to^0 F$ whenever $F$ is the same as $E$.

(ii) **Inductive step:** $E \to^{(k+1)} F$ whenever there is some expression $G$ such that $\vdash_{sm} E \to G$ and $G \to^k F$.

In (i) we have explicitly defined the relation $\to^0$ while in (ii) we have shown how $\to^{(k+1)}$ is determined by $\to^k$. Therefore mathematical induction ensures us that the relation $\to^n$ is formally defined for every $n \in \mathbb{N}$.

With this formal definition we can now prove various properties of this evaluation semantics. Here is an example:

**Lemma 1** *For every* $E \in Exp$*, if* $E \to^k F$ *then* $E + G \to^k F + G$ *for any expression* $G$.

**Proof:** We use mathematical induction on the property

$$P(k):\quad E_1 \rightarrow^k E_2 \text{ implies } E_1 + G \rightarrow^k E_2 + G \text{ for any } E_1, E_2$$

We need to show

(a) **Base case:** $P(0)$ is true. This is obvious. For if $E_1 \rightarrow^0 E_2$ then by case (i), the base case, in the definition of $\rightarrow^n$, $E_2$ must actually be $E_1$ and therefore trivially $E_1 + G \rightarrow^0 E_2 + G$.

(b) **Inductive step:** We assume the inductive hypothesis (IH) which says that $P(k)$ is true. From this we have to show that $P(k+1)$ follows.

To this end suppose $E_1 \rightarrow^{(k+1)} E_2$; we have to show that this implies $E_1 + G \rightarrow^{(k+1)} E_2 + G$. From the definition of $\rightarrow^{(k+1)}$ we know that there is some expression $E_3$ such that $\vdash_{sm} E_1 \rightarrow E_3$ and $E_3 \rightarrow^k E_2$. Now (IH) can be applied to the latter to obtain $E_3 + G \rightarrow^k E_2 + G$. Also an application of the rule (S-LEFT) added on to the derivation of the judgement $E_1 \rightarrow E_3$ gives a derivation of $E_1 + G \rightarrow E_3 + G$; that is we obtain $\vdash_{sm} E_1 + G \rightarrow E_3 + G$. Combining these two, using case (ii), the inductive step, in the definition of $\rightarrow^n$, we get the required $E_1 + G \rightarrow^{(k+1)} E_2 + G$. □

**Exercise 2** *Prove that if $E \rightarrow^{k_1} F$ and $F \rightarrow^{k_2} G$ then $E \rightarrow^{(k_1+k_2)} G$.* □

**Exercise 3** *Use mathematical induction on k to prove that, for any numeral* n*, $E \rightarrow^k F$ implies $E + \mathtt{n} \rightarrow^k \mathtt{n} + F$.* □

### 2.1.3 Strong mathematical induction

There is an alternative way to formulate mathematical induction, called *strong* or *complete* mathematical induction, which is sometimes more convenient to use. It also have the advantage that it does not differentiate between the **Base case** and the **Inductive step**.

Suppose $P(-)$ is a property of the natural numbers. In order to prove that $P(n)$ is true for every $n \in \mathbb{N}$ strong mathematical induction says that it is sufficient to do the following:

(i) Assume the inductive hypothesis (IH) which says that $P(k)$ is true for all $k$ strictly less then some arbitrary number $m$

(ii) Show that $P(m)$ follows from (IH).

Despite its name this form of induction is actually no stronger than ordinary mathematical induction; but it is sometimes more convenient to use. The standard example is the proof of the following statement:

> Every number greater than 1 is either a prime number or is the product $p_1 \times p_2 \times \ldots \times p_k$ of prime numbers $p_i$, for $1 < i \leq k$.

Recall that $n$ is a prime number if it is greater than 1 and it can not be broken down into composite numbers; that is if $n = n_1 \times n_2$ then $n_1$ is either 1 or $n$.

To prove this statement we prove the property

$P(n)$: $n$ is either less then or equal to 1, a prime, or else it is the product of primes

by strong mathematical induction. So our inductive hypothesis (IH) is that $P(k)$ is true for every $k$ strictly less than some number $m$. We have to show that $P(m)$ follows from this (IH).

The proof proceeds by a case analysis on $m$:

(i) $m$ is prime: in this case $P(m)$ is immediate.

(ii) $m$ is less than or equal to 1: again the result is trivial.

(iii) The only remaining possibility is that $m$ is greater than 1 and is not a prime; this means that $m$ can be written as $m_1 \times m_2$ where $m_1$ is neither 1 nor $m$ itself. This implies that both $m_1$ and $m_2$ are strictly less than $m$ and therefore (IH) comes into play; we can now assume that both $P(m_1)$ and $P(m_2)$ hold. Since neither can be 1 this means that both are either prime or else a product of primes. Since $m = m_1 \times m_2$ it follows that $m$ is a product of primes. So in this case $P(m)$ also holds.

## 2.2 Structural induction

Here we see a more general form of induction which applies any set of objects which can be viewed structurally; that is there is some collection of constructors which can be used to build all objects in the set. To explain this idea more fully we first give a structural account of the natural numbers, in such a way that mathematical induction, from the previous section, becomes an instance of the more general structural induction.

We then give another example of induction, on binary trees, based on their structure. We apply the same technique to the language of expressions *Exp*, giving us a powerful method of deriving their properties. In the final section we apply this technique to prove interesting properties of the big-step and small-step semantics of *Exp*, for example those discussed in Section 1.5.

### 2.2.1 A Structural View of mathematical induction

We said in the last section that mathematical induction is a valid principle because every natural number can be *built* using 0 as a starting point and the operation of adding one as a method of building new numbers from old. We can turn mathematical induction into a form of structural induction by viewing numbers as expressions generated by the following BNF grammar:

$$N \in Nat ::= \mathsf{zero} \mid \mathsf{succ}(N)$$

Here $\mathsf{succ}$, short for *successor*, should be thought of as the operation of adding one to its argument. Therefore the expression $\mathsf{zero}$ represents the number 0, and 3 is represented by the expression

$$\mathsf{succ}(\mathsf{succ}(\mathsf{succ}(\mathsf{zero})))$$

With this view, it really is the case that a number in *Nat* is built by starting from zero and repeatedly applying succ. Numbers, when thought of like this, are finite, structured objects. The structure can be described as follows.

> A number is either zero, which is indecomposable, or has the form succ($N$), where $N$ is another number.

The principle of induction now says that to prove $P(N)$ for all numbers $N \in Nat$, it suffices to do two things:

(i) **Base case:** Prove that $P(\mathsf{zero})$ holds.

(ii) **Inductive step:** The IH (inductive hypothesis) is that $P(K)$ holds for some arbitrary number $K$. Prove that $P(\mathsf{succ}(K))$ follows from this assumption.

Note that when trying to prove $P(\mathsf{succ}(K))$, the inductive hypothesis tells us that we may assume $P$ holds of the *substructure* of $\mathsf{succ}(K)$, that is, we may assume $P(K)$ holds.

This principle is *identical* to that given on page 14, but written in a structural way. The reason it is valid is the same as before:

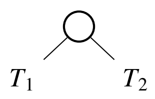
- $P(\mathsf{zero})$ holds,
- so $P(\mathsf{succ}(\mathsf{zero}))$ holds,
- so $P(\mathsf{succ}(\mathsf{succ}(\mathsf{zero})))$ holds,
- so $P(\mathsf{succ}(\mathsf{succ}(\mathsf{succ}(\mathsf{zero}))))$ holds,
- and so on...

That is to say, we have shown that every way of building a number preserves the property $P$, and that $P$ is true of the basic building block zero; so $P$ is true of every number in *Nat*.

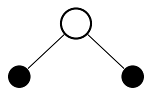
This structural viewpoint, and the associated form of induction, called *structural induction*, is widely applicable.

### 2.2.2 Structural induction for binary trees

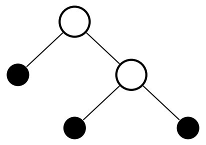
Binary trees are a commonly used data structure. Roughly, a binary tree is either a single *leaf node*, or a *branch node* which has two *sub-trees*. That is, trees take the form of a leaf ● or has the form



where $T_1$ and $T_2$ are the two sub-trees of the bigger tree. For example,



is one such *composite tree*, in which both of the sub-trees are leaf nodes. Another example is



Here the left sub-tree is a single leaf node, while the right sub-tree is the simple composite tree from above.

To make it easier to talk about trees like this, let us introduce a BNF-like syntax for them, similar to that for natural numbers *Nat*, thereby viewing binary trees as a data-structure.

$$T \in \mathbf{bTree} ::= \mathbf{leaf} \mid \mathbf{Branch}(T, T)$$

Note the similarity with *Nat*. There is one seed or starting point; for *Nat* this is zero while for binary trees it is **leaf**. There is also one generator in each case; for *Nat* this is the unary operator succ(−) which takes one argument, while for binary trees it is the binary operator **Branch**(−, −) requiring two arguments.

In this syntax the four trees above are written as

$$\mathbf{leaf},\ \mathbf{Branch}(T_1, T_2),\ \mathbf{Branch}(\mathbf{leaf}, \mathbf{leaf}),\ \mathbf{Branch}(\mathbf{leaf}, \mathbf{Branch}(\mathbf{leaf}, \mathbf{leaf}))$$

respectively.

The principle of *structural induction over binary trees* states that to prove a property $P(T)$ for all trees $T \in$ **bTree**, it is sufficient to do the following two things:

(i) **Base case:** Prove that $P(\mathbf{leaf})$ holds.

(ii) **Inductive step:** The inductive hypothesis IH is that $P(T_1)$ and $P(T_2)$ hold for some arbitrary trees $T_1$ and $T_2$. Then from this assumption prove that $P(\mathbf{Branch}(T_1, T_2))$ follows.

Again, in the inductive step, we require a hypothetical argument; from the assumption that the property holds of $T_1$ and $T_2$ we need to prove that as a logical consequence the property also holds of the tree $\mathbf{Branch}(T_1, T_2)$. The conclusion from the **Base case** and this hypothetical argument is that $P(T)$ is indeed true for every tree $T$ in **bTree**.

To put this another way: to do a proof by induction on the structure of trees, consider all possible cases of what a tree can look like. The grammar above tells us that there are two cases.

- The case of **leaf**. Prove that $P(\mathbf{leaf})$ holds directly.
- The case of $\mathbf{Branch}(T_1, T_2)$. In this case, the inductive hypothesis says that *we may assume that $P(T_1)$ and $P(T_2)$ hold* while we are trying to prove $P(\mathbf{Branch}(T_1, T_2))$. We do not know anything else about $T_1$ and $T_2$: they could be any size or shape, as long as they are binary trees which satisfy $P$.

Using exactly the same principle as before, we may give definitions of functions which take binary trees as their arguments, by induction on the structure of the trees. This applies any function with the type $f$ : **bTree** $\to X$, that is the domain of $f$ must be the set of binary trees but the range can be any set.

As you can probably guess by now, to define a function $f$ which takes an arbitrary binary tree, we must

(i) **Base case:** Define $f$(**leaf**) directly.

(ii) **Inductive step:** Define $f$(**Branch**$(T_1, T_2)$) in terms of $f(T_1)$ and $f(T_2)$, and possibly some other mathematical constructs.

This definition looks like a recursive function definition in a functional programming language, with the proviso that we may make recursive calls only to $f(T_1)$ and $f(T_2)$. That is to say, the recursive calls must be with the *immediate sub-trees* of the tree in which we are interested.

Another way to think of such a function definition is that it says how to build up the value of $f(T)$, in the same way that the tree $T$ is built up, for any tree $T$ in **bTree**. Since any tree can be built starting with some **leaf**s and putting things together using **Branch**$(-,-)$, a definition like this lets us calculate $f(T)$ bit-by-bit.

Here is an example of a pair of inductive definitions over trees, and a proof of a relationship between them. We first define the function $\underline{\mathsf{leaves} : \textbf{bTree} \to \mathbb{N}}$ which returns the number of leaf **leaf**s in a tree.

(i) **Base case:** leaves(**leaf**) = 1.

(ii) **Inductive step:** leaves(**Branch**$(T_1, T_2)$) = leaves$(T_1)$ + leaves$(T_2)$.

We now define another function, branches : **bTree** $\to \mathbb{N}$, which counts the number of **Branch**$(-,-)$ nodes in a tree.

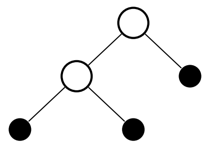
(i) **Base case:** branches(**leaf**) = 0.

(ii) **Inductive step:** branches(**Branch**$(T_1, T_2)$) = branches$(T_1)$ + branches$(T_2)$ + 1.

Let us illustrate how branches works. Consider the tree

**Branch**(**Branch**(**leaf**, **leaf**), **leaf**)

which diagrammatically looks like



This clearly has two branch nodes. Let us see how the function branches calculates this by building this tree up from the bottom.

First, the left sub-tree is built by taking two **leaf**s and putting them together with a **Branch**$(-,-)$. So the left sub-tree has the structure **Branch**$(T_1, T_2)$ where both $T_1$

and $T_2$ are the trivial tree **leaf**. The definition of the function $\mathsf{branches}$ says that the value on a **leaf** is 0, while the value of a **Branch**$(-,-)$ is obtained by adding together the values for the things you're putting together, and adding one. Therefore, the value of $\mathsf{branches}$ on the left sub-tree is $0 + 0 + 1 = 1$.

The value of $\mathsf{branches}$ on the right sub-tree is 0, since this tree is just a **leaf**. The whole tree is built by putting the left and right sub-trees together with a **Branch**$(-,-)$. The definition of $\mathsf{branches}$ again tells us to add together the values for each sub-tree, and add one. Therefore, the overall value is $1 + 0 + 1 = 2$, as we expected.

The purpose of this discussion is of course just to show you how the value of an inductively defined function on a tree is built from the bottom up, in the same way the tree is built. You can also see it as going from the top down, in the usual way of thinking about recursively defined functions: to calculate $f(\mathbf{Branch}(T_1, T_2))$, we break the tree down into its two sub-trees, calculate $f(T_1)$ and $f(T_2)$ with a recursive call, and combine the values of those in some way to get the final value.

Let us now prove, by induction on the structure of trees, that for any tree $T$,

$$\mathsf{leaves}(T) = \mathsf{branches}(T) + 1.$$

Let us refer to this property as $P(T)$. To show that $P(T)$ is true of all binary trees $T \in \mathbf{bTree}$ the principle of induction says that we must do two things.

(i) **Base case:** Prove that $P(\mathbf{leaf})$ is true; that is $\mathsf{leaves}(\mathbf{leaf}) = \mathsf{branches}(\mathbf{leaf}) + 1$.

(ii) **Inductive step:** The inductive hypothesis (IH) is that $P(T_1)$ and $P(T_2)$ are both true, for some arbitrary $T_1$ and $T_2$. So we can assume (IH), namely that

$$\mathsf{leaves}(T_1) = \mathsf{branches}(T_1) + 1 \qquad \text{and} \quad \mathsf{leaves}(T_2) = \mathsf{branches}(T_2) + 1$$

From this assumption we have to derive $P(\mathbf{Branch}(T_1, T_2))$, namely that

$$\mathsf{leaves}(\mathbf{Branch}(T_1, T_2)) = \mathsf{branches}(\mathbf{Branch}(T_1, T_2)) + 1. \qquad (2.2)$$

**Proof:**

(i) **Base case:** By definition, $\mathsf{leaves}(\mathbf{leaf}) = 1 = 1 + \mathsf{branches}(\mathbf{leaf})$ as required, since $\mathsf{branches}(\mathbf{leaf}) = 0$.

(ii) **Inductive step:** By definition, $\mathsf{leaves}(\mathbf{Branch}(T_1, T_2)) = \mathsf{leaves}(T_1) + \mathsf{leaves}(T_2)$. By the inductive hypothesis (IH),

$$\mathsf{leaves}(T_1) = \mathsf{branches}(T_1) + 1 \qquad \text{and} \quad \mathsf{leaves}(T_2) = \mathsf{branches}(T_2) + 1$$

We therefore have

$$\begin{aligned}\mathsf{leaves}(\mathbf{Branch}(T_1, T_2)) &= \mathsf{branches}(T_1) + 1 + \mathsf{branches}(T_2) + 1 \\ &= (\mathsf{branches}(T_1) + \mathsf{branches}(T_2) + 1) + 1 \qquad (2.3)\end{aligned}$$

By definition of the function $\mathsf{branches}$,

$$\mathsf{branches}(\mathbf{Branch}(T_1, T_2)) = \mathsf{branches}(T_1) + \mathsf{branches}(T_2) + 1$$

Plugging this into (2.3) we get the required (2.2) above. □

### 2.2.3 Structural Induction over the language of expressions

The syntax of our illustrative language *Exp* of expressions also gives a collection of structured, finite, but arbitrarily large objects over which induction may be used.

Recall from Figure 1.1 in Chapter 1 that the syntax of *Exp* is given by:

$$E \in Exp ::= \mathtt{n} \in Nums \mid E + E \mid E \times E.$$

Here $\mathtt{n}$ ranges over the numerals $\mathtt{0}$, $\mathtt{1}$, $\mathtt{2}$ and so on. This means that in this language there are in fact an infinite number of seeds or starting points. Contrast this with the structured sets discussed in the two previous sections. For *Nat* in Chapter 2.2.1 there is a unique seed $\mathsf{zero}$ and for **bTree** in Chapter 2.2.2 we also have the unique seed **leaf**. But for *Exp* we have two generators, $(- + -)$ and $(- \times -)$, both binary, while for *Nat* there is only one (unary) generator $\mathsf{succ}(-)$. **bTree** also has only one generator, but this is binary, **Branch**$(-,-)$.

The principle of induction for expressions reflects these differences as follows. If $P$ is a property of expressions, then to prove that $P(E)$ holds for any $E$, it suffices to do the following:

(i) **Base cases:** Prove that $P(\mathtt{n})$ holds for every numeral $\mathtt{n}$.

(ii) **Inductive step:** Here the inductive hypothesis (IH) is that $P(E_1)$ and $P(E_2)$ hold for some arbitrary $E_1$ and $E_2$. Assuming (IH) we must show that both $P(E_1 + E_2)$ and $P(E_1 \times E_2)$ follow.

The conclusion will then be that $P(E)$ is true of *every* expression $E \in Exp$.

Again, this induction principle can be seen as a case-analysis: expressions come in two forms:

- numerals, which cannot be decomposed, so we have to prove $P(\mathtt{n})$ directly for each of them; and
- composite expressions $E_1 + E_2$ and $E_1 \times E_2$, which can be decomposed into sub-expressions $E_1$ and $E_2$. In this case, induction says that we may assume $P(E_1)$ and $P(E_2)$ when trying to prove $P(E_1 + E_2)$ and $P(E_1 \times E_2)$.

Let us now see how this form of structural induction will enable us to prove interesting properties of the big- and small-step semantics for this language of expressions.

As our first example proof we show the big-step semantics always returns at least one answer for every expression.

**Proposition 2 (Normalisation)** *For every expression $E \in Exp$, there is some number $m$ such that $\vdash_{big} E \Downarrow \mathtt{m}$.*

**Proof:** By structural induction on $E$. The property $P(E)$ of expressions we wish to prove is

$P(E)$: there is some number $m$ such that $\vdash_{big} E \Downarrow \mathtt{m}$

The principle of structural induction says that to prove $P(E)$ holds for every expression $E$ we are required to establish two facts:

(i) **Base cases:** $P(\mathtt{n})$ holds for every numeral $\mathtt{n}$.

For any numeral $\mathtt{n}$, the axiom of the big-step semantics, (B-NUM), gives a trivial derivation of $\mathtt{n} \Downarrow \mathtt{n}$. So in this case the required number $m$ is $n$.

(ii) **Inductive step:** The inductive hypothesis (IH) is that $P(E_1)$ and $P(E_2)$ hold for some arbitrary $E_1$ and $E_2$. From IH we are required to prove both $P(E_1 + E_2)$ and $P(E_1 \times E_2)$ follow. We shall consider the case of $E_1 + E_2$ in detail; the case of $E_1 \times E_2$ is similar.

We must show $P(E_1 + E_2)$, namely that for some number $m$ it is the case that $\vdash_{big} (E_1 + E_2) \Downarrow \mathtt{m}$.

By the *inductive hypothesis* (IH) , we may assume that there are numbers $m_1$ and $m_2$ for which the judgements $E_1 \Downarrow \mathtt{m}_1$ and $E_2 \Downarrow \mathtt{m}_2$ are derivable. We can combine these derivations, followed by an application of the rule (B-ADD),

$$\frac{E_1 \Downarrow \mathtt{m}_1 \quad E_2 \Downarrow \mathtt{m}_2}{E_1 + E_2 \Downarrow \mathtt{m}_3}$$

where $m_3 = \mathsf{add}(m_1, m_2)$. to obtain a derivation of $E_1 + E_2 \Downarrow \mathtt{m}_3$. So $m_3$ is the required witness number which makes $P(E_1 + E_2)$ true. □

Proving that the big-step semantics returns exactly one result is only slightly more complicated.

**Proposition 3 (Determinacy)** *For every expression $E \in Exp$, if $\vdash_{big} E \Downarrow \mathtt{m}_1$ and $\vdash_{big} E \Downarrow \mathtt{m}_2$ then $m_1 = m_2$.*

**Proof:** Again we use structural over $E$, this time with the property

$P(E)$: whenever $\vdash_{big} E \Downarrow \mathtt{m}_1$ and $\vdash_{big} E \Downarrow \mathtt{m}_2$ it follows that $m_1 = m_2$.

From the principle of structural induction in order to establish $P(E)$ we need to establish two facts:

(i) **Base cases:** $P(\mathtt{n})$ holds for every numeral $\mathtt{n}$.

So suppose $\mathtt{n} \Downarrow \mathtt{m}_1$ and $\mathtt{n} \Downarrow \mathtt{m}_2$ both have derivations. The only possible rule which can be used to derive these judgements is (B-NUM). From this observation it follows immediately that $m_1$ and $m_2$ must be the same number, namely $n$.

(ii) **Inductive step:** The induction hypothesis is that both $P(E_1)$ and $P(E_2)$ are true. We need to prove that the statements $P(E_1 + E_2)$ and $P(E_1 \times E_2)$ both follow. Here we consider the case $P(E_1 \times E_2)$ in detail; the other case is very similar. But we have to assume that the big-step semantics has some rule to handle multiplicative expressions. Let us assume the obvious one:

(B-MULT)
$$\frac{E_1 \Downarrow \mathtt{n}_1 \qquad E_2 \Downarrow \mathtt{n}_2}{E_1 \times E_2 \Downarrow \mathtt{n}_3} \quad n_3 = \mathsf{mult}(n_1, n_2)$$

where $\mathsf{mult}(-,-)$ is the binary mathematical operation which takes two numbers and returns the result of multiplying them together.

So suppose $(E_1 \times E_2) \Downarrow \mathtt{m}_1$ and $(E_1 \times E_2) \Downarrow \mathtt{m}_2$ are both derivable for some numbers $m_1, m_2$. We need to show that these two numbers coincide. Again we look at how these judgements can be derived; there are only three possible rules, (B-NUM) and (B-ADD) and (B-MULT). So it should be apparent that the derivation of $(E_1 \times E_2) \Downarrow \mathtt{m}_1$ has to involve an application of the last rule. In fact we must have that $m_1 = \mathsf{mult}(k_1, k_2)$ where

(a) $E_1 \Downarrow \mathtt{k}_1$

(b) $E_2 \Downarrow \mathtt{k}_2$

But the same analysis can be applied to the derivation of $(E_1 + E_2) \Downarrow \mathtt{m}_2$; we must have $m_2 = \mathsf{mult}(n_1, n_2)$ where

(c) $E_1 \Downarrow \mathtt{n}_1$

(d) $E_2 \Downarrow \mathtt{n}_2$

Now property $P(E_1)$ applied to (a) and (c) ensures that $k_1 = n_1$ while $P(E_2)$ applied to (b) and (d) gives $k_2 = n_2$. Combining these we get the required $m_1 = m_2$ follows. □

Determinacy and Normalisation combined ensures that the big-step semantics is coherent; it associates precisely one result with every expression in *Exp*. We could address the same issues for the the small-step semantics but instead let us consider the relationship between the two forms of semantics.

**Proposition 4** *For every $E \in Exp$, $\vdash_{big} E \Downarrow \mathtt{m}$ implies $E \rightarrow^* \mathtt{m}$.*

**Proof:** Again we use structural induction on $E$. Recall that we are using $E \rightarrow^* \mathtt{n}$ as a shorthand for the statement *for some number k, $E \rightarrow^k \mathtt{n}$*. Consequently the property of $E$ we have to prove is

$P(E)$: if $\vdash_{big} E \Downarrow \mathtt{m}$ then there is some number $k$ such that $E \rightarrow^k \mathtt{m}$

To prove this, structural induction requires us to establish the following:

(i) **Base cases:** $P(\mathtt{n})$ for every numeral $\mathtt{n}$. This is straightforward. Suppose $\mathtt{n} \Downarrow \mathtt{m}$ is derivable. This must mean that $m = n$ as this judgement can only be derived using the rule (B-NUM). So the required number of steps $k$ is 0 since $\mathtt{n} \rightarrow^0 \mathtt{n}$.

(ii) **Inductive step:** Here we assume the inductive hypothesis IH that both $P(E_1)$ and $P(E_2)$ are true. From this assumption we are required to prove that both $P(E_1 + E_2)$ and $P(E_1 \times E_2)$ follow. As usual we only consider one case, say $P(E_1 + E_2)$.

So suppose $(E_1 + E_2) \Downarrow \mathtt{m}$ is derivable; we have to find some number $k$ such that $(E_1 + E_2) \rightarrow^k \mathtt{m}$.

There is a derivation of the judgement $(E_1 + E_2) \Downarrow \mathtt{m}$ using the rules (B-NUM), (B-ADD) and (B-MULT). Whatever form this derivation takes it must end with an application of (B-ADD), of the form

$$\frac{E_1 \Downarrow \mathtt{m}_1 \quad E_2 \Downarrow \mathtt{m}_2}{(E_1 + E_2) \Downarrow \mathtt{m}}$$

where $m = \mathsf{add}(m_1, m_2)$.

But $P(E_1)$ now tells us that there is some $k_1$ such that $E_1 \rightarrow^{k_1} \mathtt{m}_1$, and we have already seen in Lemma 1 that this in turn means that $(E_1 + E_2) \rightarrow^{k_1} (\mathtt{m}_1 + E_2)$ has a derivation.

Similarly from $P(E_2)$ and $E_2 \Downarrow \mathtt{m}_2$ we know that there is some $k_2$ such that $E_2 \rightarrow^{k_2} \mathtt{m}_2$. From this it is possible to show, using the same technique as that used in the proof of Lemma 1, that $(\mathtt{m}_1 + E_2) \rightarrow^{k_2} (\mathtt{m}_1 + \mathtt{m}_2)$; in fact this is posed as Exercise 8 at the end of this chapter.

Putting both of these executions together we get

$$(E_1 + E_2) \rightarrow^{k_1} (\mathtt{m}_1 + E_2) \rightarrow^{k_2} (\mathtt{m}_1 + \mathtt{m}_2) \rightarrow \mathtt{m}$$

with the final step being an application of the rule (B-ADD). In other words the required $k$ is $(k_1 + k_2 + 1)$. □

The converse of Proposition 4, namely $E \rightarrow^* \mathtt{m}$ implies $\vdash_{big} E \Downarrow \mathtt{m}$, is not so easy to prove. Because of the asymmetry in the premise $E \rightarrow^* \mathtt{m}$ it is hard to come up with a suitable application of structural induction. The proof we propose it somewhat indirect, relying on the following result, which we prove for the *choice* variation of the small-step semantics from Chapter 1.4 which allows the arguments to an operator to be evaluated independently:

**Lemma 5** *Suppose $\vdash_{ch} E \rightarrow_{ch} F$. Then $\vdash_{big} F \Downarrow \mathtt{m}$ implies $\vdash_{big} E \Downarrow \mathtt{m}$.*

**Proof:** Can you do this? Structural induction on $E$ should be used. □

**Proposition 6** *For every $E \in Exp$, $E \rightarrow^*_{ch} \mathtt{m}$ implies $\vdash_{big} E \Downarrow \mathtt{m}$.*

**Proof:** Recall that $E \rightarrow^*_{ch} \mathtt{m}$ actually means that $E \rightarrow^k_{\mathrm{ch}} \mathtt{m}$ for some number $k$. Formally we have not actually defined the relations $E \rightarrow^k_{\mathrm{ch}} F$; however they are defined in exactly the same manner as the relations $E \rightarrow^k F$, in Section 2.1.2. So we prove the following statement to be true using mathematical induction:

$P(k)$ for any expression $E$, $E \rightarrow^k_{\mathrm{ch}} \mathtt{m}$ implies $\vdash_{big} E \Downarrow \mathtt{m}$

from which the result follows.

The statement $P(k)$ will be true for every number $k$ if we can prove the following two facts:

(i) **Base case:** $P(0)$. This case is easy; if $E \rightarrow^0_{\mathrm{ch}} \mathtt{m}$ then $E$ must actually be the numeral $\mathtt{m}$ and one application of the rule (B-NUM) then gives the required derivation of $E \Downarrow \mathtt{m}$.

(ii) **Inductive step:** Here we may assume the inductive hypothesis $P(k)$ to be true. From this assumption we are required to show that $P(k+1)$ follows.

To this end suppose $E \rightarrow_{\mathrm{ch}}^{(k+1)} \mathtt{m}$; from this we need to show how to derive derive $E \Downarrow \mathtt{m}$. By definition we know that $\vdash_{sm} E \rightarrow_{ch} F$ for some expression $F$ such that $F \rightarrow_{\mathrm{ch}}^{k} \mathtt{m}$. But we can apply the inductive hypothesis $P(k)$ to $F$ and we get that $\vdash_{big} F \Downarrow \mathtt{m}$. The required conclusion $E \Downarrow \mathtt{m}$ now follows by the previous lemma, Lemma 5. □

To end this section let us briefly consider how our various formulations of the semantics of the language *Exp* can be used to associated a *meaning* to all expressions in the language.

Because of the results established in the previous section we now know that the various semantics we have proposed for the language *Exp* are consistent, and moreover they agree with each other; the following statements are equivalent:

(1) $\vdash_{big} E \Downarrow \mathtt{n}$

(2) $E \rightarrow^{*} \mathtt{n}$

(3) $E \rightarrow_{ch}^{*} \mathtt{n}$

Proposition 4 ensures that (1) implies (2) while Proposition 6 means that (3) implies (1). The intermediate (2) implies (3) is obvious since the inductive rules which define the left-to-right small-step semantics are included in those which define $\rightarrow_{ch}$.

Consequently it does not actually matter which we use when proposing the *definitive* reference semantics to the language *Exp*. Let the meaning function

$$[\![-]\!] : Exp \rightarrow Nums$$

be defined by letting $[\![E]\!] = \mathtt{n}$, where $E \Downarrow \mathtt{n}$. Proposition 2 (Normalisation) and Proposition 3 (Determinacy) ensure that this is indeed a well-defined function. Moreover we know that the meaning of expressions can be corrected calculated by interpreters which use a left-to-right strategy, and by interpreters which use alternative strategies for deciding the order in which the arguments to an operator should be evaluated.

## 2.3 Rule Induction

The language of expressions *Exp* is particularly straightforward, in the sense that the behaviour of $E$ is completely determined by the behaviour of its components. For this reason structural induction is sufficiently powerful to establish the various properties of the different semantics we have given to *Exp*. This will rarely be the case for more complicated languages, particularly those with recursive or inductive control features. Here we give a brief glimpse of a much more powerful and widely applicable proof technique.

The essential idea is to ignore any structure that objects might have and instead concentrate on the size of the derivations of judgements. For example consider the following simple pair of rules, defining an infix binary relation $\mathsf{D}$ between numbers in $\mathbb{N}$:

$$\text{(AX)}\quad \frac{}{n \,\mathsf{D}\, 0}\; n \in \mathbb{N} \qquad\qquad \text{(PLUS)}\quad \frac{n \,\mathsf{D}\, m}{n \,\mathsf{D}\, (m+n)}$$

Here are two example derivations:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{7 \,\mathsf{D}\, 0}\,\text{(AX)}}{7 \,\mathsf{D}\, 7}\,\text{(PLUS)}}{7 \,\mathsf{D}\, 14}\,\text{(PLUS)}}{7 \,\mathsf{D}\, 21}\,\text{(PLUS)} \qquad\qquad \dfrac{\dfrac{\dfrac{}{2 \,\mathsf{D}\, 0}\,\text{(AX)}}{2 \,\mathsf{D}\, 2}\,\text{(PLUS)}}{2 \,\mathsf{D}\, 4}\,\text{(PLUS)}$$

From these derivations we know that both the judgements $7 \,\mathsf{D}\, 21$ and $2 \,\mathsf{D}\, 4$ are derivable from the rules. But we also know that the size of the derivation of the former is strictly less than that of the latter. So the idea of rule induction is to prove properties of judgements by mathematical induction on the size of their derivations. This makes sense because every derived judgement has some size associated with it; this could be taken to be the size of its smallest proof.

Consequently if we want to prove a statement of the form

$$n \;\mathsf{D}\; m \text{ implies } P(n,m) \tag{2.4}$$

or in other words the property $P(n,m)$ is true whenever $n \,\mathsf{D}\, m$, then we can use mathematical induction on the **size of the derivation** of $n \,\mathsf{D}\, m$. In fact it is more convenient to use strong mathematical induction, as explained in Section 2.1.3.

Let us consider an example. Suppose we want to show:

$$n \,\mathsf{D}\, m \text{ implies } m = n \times k \text{ for some natural number } k \tag{2.5}$$

Incidently this just means that the two rules (AX) and (PLUS) correctly capture the notion of *division*. Let $P(n,m)$ denote the property $m = n \times k$ *for some natural number* $k$. We prove that $n \,\mathsf{D}\, m$ implies $P(n,m)$ by strong mathematical induction on the size of derivation of the judgement $n \,\mathsf{D}\, m$ from the rules (AX) and (PLUS).

So suppose we have a derivation of $n \,\mathsf{D}\, m$. Using strong mathematical induction means that we have as an inductive hypothesis (IH) that

> $P(k_1,k_2)$ is true for any $k_1,k_2$ for which there is a derivation of $k_1 \,\mathsf{D}\, k_2$ whose size is less than the size of this derivation of $n \,\mathsf{D}\, m$.

We have to show that $P(n,m)$ is a logical consequence of (IH).

So we know that $n \,\mathsf{D}\, m$ can be derived using the axioms (AX) and (PLUS). What does this derivation look like? There are two possibilities:

(a) It is simply an application of the axiom (AX). In other words it looks like

$$\frac{}{n \,\mathsf{D}\, m}\,\text{(AX)}$$

But this can only be the case if $m$ is actually 0, and $P(n, 0)$ is trivially true, the required witness $k$ in (2.5) above being 0.

(b) The only other possibility is that the derivation has the form

$$\cfrac{\cfrac{\ldots}{n \mathsf{D}\, m_1}\;\ldots}{n \mathsf{D}\, (m_1 + n)}\;\text{(PLUS)}$$

where $m = m_1 + n$. But this means that the judgement $n \mathsf{D}\, m_1$ also has a derivation from the rules. And moreover the size of this derivation is strictly less than that of $n \mathsf{D}\, m$. So (IH) applies and we know that there is some $k_1$ such that $m_1 = n \times k_1$. Now $P(n, m)$ is an immediate consequence as $m = n \times (k_1 + 1)$; the required witness $k$ in in (2.5) above is $k_1 + 1$.

We have explained **rule induction** by example, but the general principle should be apparent. We ignore the components of the judgements involved and instead carry out (strong) mathematical induction of the size of their derivations.

There is another, more abstract, view of rule induction. We can view (AX) and (PLUS) as properties of binary relations over numbers, which are enjoyed by the relation we are trying to define $(- \mathsf{D} -)$. Formally the rules are used to define the relation by saying:

> $k_1 \mathsf{D}\, k_2$ exactly when there is a derivation of the judgement $k_1 \mathsf{D}\, k_2$ using the two rules (AX) and (PLUS)

An equivalent way to express this is to say that

> $\mathsf{D}$ is the least relation which satisfies the rules (AX) and (PLUS).

By this we mean

(i) The relation $\mathsf{D}$ satisfies the rules; that is

  (a) $n \mathsf{D}\, 0$ for every number $n$

  (b) if $n \mathsf{D}\, m$ for any numbers $n, m$ then $n \mathsf{D}\, (m + n)$ is also true

(ii) if $X$ is any other relation that satisfies the rules (a) and (b) then $D$ is a subset of $X$; that is $n \mathsf{D}\, m$ implies $n\, X\, m$.

It is property (ii) which gives the principle of **rule induction**. For let $P(n, m)$ be some arbitrary property of numbers; this can also be viewed as a binary relation over numbers. So in order to establish the general statement (2.4) above it is sufficient to show that $P(n, m)$ satisfies the two properties (AX) (PLUS). This means we have to prove

(a) $P(n, 0)$ for every number $n$

(b) If $P(n, m)$ for any numbers $n, m$ then $P(n, m + n)$ is also true.

If we now re-examine the actual proof given of the specific instance (2.5) above, we see that it consists precisely in establishing these two properties; although it was expressed within a framework of strong mathematical induction.

So we have seen two slightly different formulations of **rule induction** for a set of inductive rules. To prove that a property $P$ follows from the rules we can

(I) either use strong mathematical induction over the size of derivations

(II) or prove the property $P$ satisfies the inductive rules.

Many people find the former easier to apply.

### 2.3.1 What is going on?

Here we offer a slightly more detailed account of the formal basis for rule induction, *inductively defined sets*.

Let $\mathcal{U}$ be a *universe of discourse*. An *axiom* is of an element of $\mathcal{U}$, while a *rule* takes the form

$$\frac{h_1, h_2, \ldots h_n}{c}$$

where $n > 0$ and

- each $h_i$ an element of $T$, called *hypotheses*
- $c$ an element of $T$, called the *conclusion*.

Then a deductive system $\mathcal{D}$ over the universe $\mathcal{U}$ consists of a set of axioms and rules.

**Example 7** *Rule induction was explained in the previous section using the universe $\mathcal{U}_d$ consisting of all judgements of the form $n \mathsf{D} m$, where $n, m \in \mathbb{N}$. The associated deductive system $\mathcal{D}_d$ consists of*

*(i)* ***Axioms:*** *all judgements of the form $n \mathsf{D}\, 0$ where $n \in \mathbb{N}$*

*(ii)* ***Rules:*** *these are all statements of the form*

$$\frac{n \mathsf{D}\, m}{n \mathsf{D}\, (m+n)}$$

*where both $n, m$ are from $\mathbb{N}$.*

*On the other hand the small-step semantics for* Exp *has as a universe $\mathcal{U}_s$ all judgements of the form $E \to F$, where $E, F$ are expressions from* Exp*. The deductive system $\mathcal{D}_s$is given by*

*(i)* ***Axioms:*** *all judgements of the form*

- $(\mathtt{n}_1 + \mathtt{n}_2) \to \mathtt{n}_3$ *where* $n_3 = \mathsf{add}(n_1, n_1)$
- *or* $(\mathtt{n}_1 \times \mathtt{n}_2) \to \mathtt{n}_3$ *where* $n_3 = \mathsf{mult}(n_1, n_1)$

*(ii)* ***Rules:*** *these take one of the following forms:*

- $$\frac{E_1 \to E_1'}{(E_1 + E_2) \to (E_1' + E_2)}$$

  *for all expressions* $E_1, E_2, E_1', E_2'$

- *or*

$$\frac{E \to E'}{(\mathtt{n} + E) \to (\mathtt{n} + E')}$$

*for all expressions $E, E'$ and all numerals* $\mathtt{n}$ □

The purpose of a deductive system $\mathcal{D}$ is to determine a subset of the universe of discourse $\mathcal{U}$, namely all those which can be derived from the axioms by applying the rules. Let us denote this set by $\mathcal{D}(\mathcal{U})$. So for example:

- The judgements $3 \ \mathsf{D}\ 0$, $4 \ \mathsf{D}\ 12$ and $7 \ \mathsf{D}\ 42$ are all in $\mathcal{D}_d(\mathcal{U}_d)$; they can all be derived using the axioms and inference rule in $\mathcal{D}_d$.
- None of the judgements $0 \ \mathsf{D}\ 6$, $2 \ \mathsf{D}\ 7$, $3 \ \mathsf{D}\ 13$ are in $\mathcal{D}_d(\mathcal{U}_d)$, although they all are in the universe $\mathcal{U}_d$.
- The judgement $(3 + 4) + (2 \times 8) \to 3 + (2 \times 8)$ is in in $\mathcal{D}_s(\mathcal{U}_s)$.
- The judgement $(3 + 4) + (2 \times 8) \to (3 + 4) + 16$ is not in $\mathcal{D}_s(\mathcal{U}_s)$.

To explain the defining characteristics of the set $\mathcal{D}(\mathcal{U})$ we need one more concept. Let $X$ be a subset of the universe $\mathcal{U}$. Then $X$ is said to satisfy the deductive system $\mathcal{D}$ if

(a) for every axiom $a$ in $\mathcal{D}$, $a \in X$

(b) for every rule $\frac{h_1, h_2, \ldots h_n}{c}$ in $\mathcal{D}$, if each hypothesis $h_i$ is in $X$ then so is the conclusion $c$.

**Theorem 8 (Inductive sets)**

*(1) The set $\mathcal{D}(\mathcal{U})$ satisfies the deductive system $\mathcal{D}$.*

*(2) If $X$ is any set which satisfies the deductive system $\mathcal{D}$ then $\mathcal{D}(\mathcal{U}) \subseteq X$.*

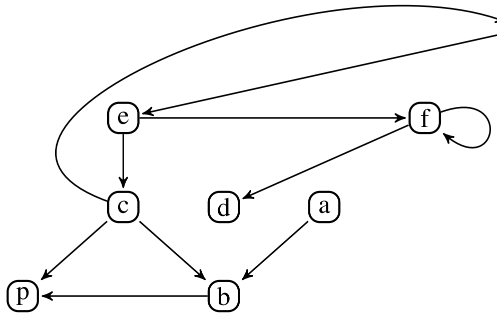
**Proof:** These are relatively straightforward; (1) follows immediately from the definition of $\mathcal{D}(\mathcal{U})$. To prove (2) suppose that $u \in \mathcal{D}(\mathcal{U})$; a proof by strong mathematical induction on the length of the proof of $u$ in the inductive system $\mathcal{D}$ will show that $u \in X$. □

It is property (2) of inductive sets which provides the justification for rule induction.

## 2.4 The reflexive transitive closure of a relation

Up to now we have been using $E \to^* F$ as a shorthand notation for $E \to^k F$ *for some number* $k \geq 0$, where the evaluation relations $\to^k$ have been defined by mathematical induction on $k \in \mathbb{N}$ in Chapter 2.1.2. Here, as an exercise in the use of rule induction, we show how $E \to^* F$, the *reflexive transitive closure of* $E \to F$, can be given an independent definition, and how this informal shorthand notation can be justified.

Recall that a relation (binary) $\mathcal{R}$ with *domain* $D$ and range $E$ is simply a subset of pairs from $D \times E$; if $D$ is the same set as $E$ we say that $\mathcal{R}$ is a (binary) relation over $D$. We use (at least) two different notations to describe elements from a relation:



Representing a relation $\mathcal{S} \subseteq D \times D$ where

- $D$ is the set $\{a,\ b,\ c,\ d,\ f,\ n,\ p\}$
- and $\mathcal{S}$ consists of the pairs

$$\begin{array}{l}(a,b)\\(b,p)\\(c,p),\ (c,b),\ (c,n)\\(e,c),\ (e,f)\\(f,d),\ (f,f)\\(n,e)\end{array}$$

Figure 2.2: Representing a relation

---

- $x \mathrel{\mathcal{R}} y$
- $(x,y) \in \mathcal{R}$

Both say that $\mathcal{R}$ relates $x$ to $y$. Sometimes relations can be described diagrammatically; an example is given in Figure 2.2. Thus we have $e \mathrel{\mathcal{S}} f$ is true while $e \mathrel{\mathcal{S}} p$ is false.

We now formalise the notion of the *reflexive transitive closure* of an arbitrary relation $\mathcal{R} \subseteq D \times D$. This will be denoted by $\mathcal{R}^*$ and if the relation $\mathcal{R}$ is represented diagrammatically as in Figure 2.2 then $\mathcal{R}^*$ has an intuitive explanation: $x \mathrel{\mathcal{R}} y$ precisely when there is a path through the graph starting from $x$ and ending with $y$. So, as we will see

- $e \mathrel{\mathcal{S}^*} p \qquad n \mathrel{\mathcal{S}^*} d \qquad e \mathrel{\mathcal{S}^*} b$ will all be true
- $e \mathrel{\mathcal{S}^*} a \qquad f \mathrel{\mathcal{S}^*} n \qquad p \mathrel{\mathcal{S}^*} b$ will all be false.

The formal definition of *reflexive transitive closure* is given in Figure 2.3. If $\mathcal{R}$ is a relation over the set $D$ then its reflexive transitive closure $\mathcal{R}^* \subseteq D \times D$ is the least

$$\text{(c-Id)}\quad \dfrac{}{x\,\mathcal{R}^*\,x}\;x \in D \qquad\qquad \text{(c-Trans)}\quad \dfrac{x\,\mathcal{R}\,y \qquad y\,\mathcal{R}^*\,z}{x\,\mathcal{R}^*\,z}$$

Figure 2.3: Inductive definition of reflexive transitive closure of $\mathcal{R} \subseteq D \times D$

relation which satisfies the two rules (c-Id) and (c-Trans). In other words $d_1\ \mathcal{R}^* d_2$ if and only if we can find a derivation of $d_1\ \mathcal{R}^* d_2$ using these two rules. In Figure 2.4 we give a derivation of the judgement

$$n\ \mathcal{S}^*\ d$$

where $\mathcal{S}$ is the relation defined in Figure 2.2. Similar judgements can also be given for $e\ \mathcal{S}^*\ p$, $c\ \mathcal{S}^*\ d$ and $e\ \mathcal{S}^*\ b$.

**Exercise 4** *Is it always true that $x\ \mathcal{R}^*\ y$ whenever $x\ \mathcal{R}\ y$?* □

$$\dfrac{n\,\mathcal{S}\,e \qquad \dfrac{e\,\mathcal{S}\,f \qquad \dfrac{f\,\mathcal{S}\,d \qquad \dfrac{}{d\,\mathcal{S}^*\,d}\,\text{(c-Id)}}{f\,\mathcal{S}^*\,d}}{e\,\mathcal{S}^*\,d}\,\text{(c-Trans)}}{n\,\mathcal{S}^*\,d}\,\text{(c-Trans)}$$

Figure 2.4: An example derivation: $n\ \mathcal{S}^*\ d$

### 2.4.1 Alternative formulation

For $\mathcal{R} \subseteq D \times D$, instead of defining the single relation $\mathcal{R}^*$ we define a series of relations $\mathcal{R}^n$, one for every natural number $n$ in $\mathbb{N}$. The definition is by *mathematical induction*, of course, and is only a minor generalisation of the definition of the relations $E \rightarrow^k F$ in Section 2.1.2. First the base case: we define the relation $\mathcal{R}^0$. Then the inductive step: under the hypothesis that we have already defined the relation $\mathcal{R}^k$ we show how to define the relation $\mathcal{R}^{(k+1)}$. Mathematical induction ensures that we will have then defined the relation $\mathcal{R}^n$ for every natural number $n$ in $\mathbb{N}$.

(i) **Base case:** $x\ \mathcal{R}^0\ y$ whenever $x = y$ and $x \in D$

(ii) **Inductive step:** Assume we have already defined the relation $\mathcal{R}^k$. Then the relation $\mathcal{R}^{(k+1)}$ by letting

$$x\ \mathcal{R}^{(k+1)}\ z$$

whenever we can find some element $y \in D$ such that

$$x \,\mathcal{R}\, y \qquad \text{and } y \,\mathcal{R}^k\, z \tag{2.6}$$

So for example, refering to Figure 2.2,

$$a \,\mathcal{S}^2\, p \qquad e \,\mathcal{S}^3\, p \qquad c \,\mathcal{S}^7\, d \text{ and } e \,\mathcal{S}^{10}\, d$$

We now formally show that there is there is an intimate connection between the reflexive transitive closure of a relation, $\mathcal{R}^*$, and the family of relations $\mathcal{R}^n$, $n$ in $\mathbb{N}$.

**Proposition 9** *For every $n$ in $\mathbb{N}$, if $x \,\mathcal{R}^n\, z$ then $x \,\mathcal{R}^*\, z$, for every $x, z$ in the domain of $\mathcal{R}$.*

**Proof:** This has the form of a statement which is amenable to proof by mathematical induction. So let $P(n)$ be the statement

if $x \,\mathcal{R}^n\, z$ then $x \,\mathcal{R}^*\, z$ for every $x, z$ in the domain of $\mathcal{R}$.

To prove that $P(n)$ is true for every $n$ in $\mathbb{N}$ we have to establish two facts:

(i) **Base case:** We prove $P(0)$ is true, namely $x \,\mathcal{R}^0\, z$ implies $x \,\mathcal{R}^*\, z$.

To this end suppose $x \,\mathcal{R}^0\, z$. By definition of $\mathcal{R}^0$ this can only be true if $x = z$ and $x$ is in the domain of of $\mathcal{R}$, say the set $D$. Then applying the rule (c-Id) from Figure 2.3 we can conclude that $x \,\mathcal{R}^*\, z$.
This is the end of the base case.

(ii) **Inductive step:** Here we have to prove for an arbitrary $k$ in $\mathbb{N}$ the hypothetical statement

$P(k+1)$ follows from the inductive hypothesis $P(k)$.

So we are assuming

for any $x, z$ in the domain of $\mathcal{R}$, if $x \,\mathcal{R}^k\, z$ then $x \,\mathcal{R}^*\, z$ (IH)

Using this inductive hypothesis (IH) we have to show that $P(k+1)$ follows, namely $x \,\mathcal{R}^{(k+1)}\, z$ implies $x \,\mathcal{R}^*\, z$.

So suppose $x \,\mathcal{R}^{(k+1)}\, z$ is true.

**Exercise 5** *Finish this proof.* □

The converse to this proposition is also true. The proof uses *Rule induction* on the inductive definition of $\mathcal{R}^*$; let us use the form (II) of rule induction, as explained on page 31. This involves casting the converse in terms of a proposition $P$ which satisfies the inductive rules which define $\mathcal{R}^*$.

**Proposition 10** *If $x \,\mathcal{R}^*\, z$, then there is some number $n$ in $\mathbb{N}$ such that $x \,\mathcal{R}^n\, z$.*

**Proof:** By Rule induction on the judgement $x\ \mathcal{R}^*\ z$. Recall from Figure 2.3 that this relation is defined using two rules; therefore the associated rule induction will have two cases associated with it.

Let $P(x,z)$ be the statement

there is some $n$ in $\mathbb{N}$ such that $x\ \mathcal{R}^n\ z$.

To prove $x\ \mathcal{R}^*\ z$ implies $P(x,z)$ we have to establish that the relation $P$ satisfies the two rules from Figure 2.3 which define $\mathcal{R}^*$.

(a) The rule (c-Id). Here we have to prove that $P(x,x)$ for every $x$ in the domain of $\mathcal{R}$.

This is straightforward since if $x$ is in the domain of $\mathcal{R}$ then by the definition of $\mathcal{R}^0$ we know $x\ \mathcal{R}^0\ x$; so in this case the required $n$ in $\mathbb{N}$ is 0.

(b) The rule (c-Trans). Here we have to prove the hypothetical statement

$x\ \mathcal{R}\ y$ and $P(y,z)$ implies $P(x,z)$

To this end suppose $P(y,z)$ is true. So we are assuming that

there is some natural number $k$ such that $y\ \mathcal{R}^k\ z$ (IH)

Using this hypothesis, and the fact that $x\ \mathcal{R}\ y$, we have to show that $P(x,z)$ follows; we have to show there is some $n$ in $\mathbb{N}$ such that $x\ \mathcal{R}^n\ z$.

The required $n$ in $\mathbb{N}$ is easy to calculate. From (2.6) above in the definition of $\mathcal{R}^n$ we can calculate that $x\ \mathcal{R}^{(k+1)}\ z$, because $x\ \mathcal{R}\ y$ and according to (IH) $y\ \mathcal{R}^k\ z$; so the required $n$ is $(k+1)$. □

**Exercise 6** *Give an alternative proof of Proposition 10 using the form (I) of rule induction.*

**Exercise 7** *Show, using rule induction on the definition of $\mathcal{R}$, that if $x\ \mathcal{R}^*\ y$ and $y\ \mathcal{R}\ z$ then $x\ \mathcal{R}^*\ z$ also holds.*

*Then use this property to show that $x\ \mathcal{R}^*\ y$ and $y\ \mathcal{R}^*\ z$ implies $x\ \mathcal{R}^*\ z$.* □

**Exercise 8** *Use rule induction to prove $E \rightarrow^* F$ implies $\mathtt{n} + E \rightarrow^* \mathtt{n} + F$.* □

# Chapter 3

# The While programming language

The abstract syntax of the imperative programming language *While* is given in Figure 3.1. The main syntactic category is *Com*, for *commands*, and anybody with even minimal exposure to programming should be familiar with the constructs. Here is a sample command, or program:

$$\begin{array}{l} \textsc{l}_2 := \mathtt{1}; \\ \textsc{l}_3 := \mathtt{0}; \\ \mathtt{while}\ \neg\,(\textsc{l}_1 = \textsc{l}_2)\ \mathtt{do} \\ \quad \textsc{l}_2 := \textsc{l}_2 + \mathtt{1}; \\ \quad \textsc{l}_3 := \textsc{l}_3 + \mathtt{1} \end{array}$$

which subsequently we we refer to as $C_1$.

**Exercise 9** *What do you think the command or program $C_1$ does?* □

According to Figure 3.1 the language of commands contains five constructs, which we explain intuitively in turn.

- **Assignments**: These take the form $\textsc{l} := E$ where $E$ is an arithmetic expression and $\textsc{l}$ is the name of some *location* or *variable* in memory. So the language assumes some given set of locations names Locs, and we use $\textsc{l}$, $\textsc{k}, \ldots$ for typical elements. The syntax of commands also depends on a separate language for acceptable arithmetic expressions, $E$. An example abstract syntax for these is also given in Figure 3.1. This in turn uses $\mathtt{n}$ as a meta-variable to range over the set of numerals, *Nums*, used in Chapter 1. Apart from these, we are allowed to use one operator $+$ to construct arithmetic expressions, although others can be easily added such as the multiplication operator $\times$ used in Chapters 1.

  Thus a typical example of an assignment command is

  $$\textsc{k} := \textsc{l} + 2$$

  Intuitively this refers to the command:

$$
\begin{array}{rcl}
C \in \mathit{Com} & ::= & \textsc{l} := E \mid \texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C \\
 & & \mid C\,;C \mid \texttt{while}\ B\ \texttt{do}\ C \mid \texttt{skip} \\
\\
B \in \mathit{Bool} & ::= & \texttt{true} \mid \texttt{false} \mid E = E \mid B\&B \mid \neg\, B \\
\\
E \in \mathit{Arith} & ::= & \textsc{l} \in \mathrm{Locs} \mid \mathtt{n} \in \mathit{Nums} \mid (E + E)
\end{array}
$$

Figure 3.1: The language *While*

  - look up the current value, a numeral, in the location L
  - replace the current value stored in location K by 2 plus the value found in L

- **Sequencing**, $C_1$ ; $C_2$. The intention here should be obvious. First execute the command $C_1$; when this is finished execute the command $C_2$.

- **Test**, `if` $B$ `then` $C_1$ `else` $C_2$. Intuitively this evaluates the Boolean expression $B$; if the resulting value is `true` then the command $C_1$ is executed, if it is `false` $C_2$ is executed. Figure 3.1 contains a separate BNF for the collection of Boolean expressions. This contains the two constants `true`, `false` and two operators, negation represented by $\neg\, B$ and binary conjunction $B\&B$; obviously other Boolean operators can be defined in terms of these two. We also all $E_1 = E_2$ as a Boolean expression, where $E_1$ and $E_2$ can be any arithmetic expressions.

- **Repetition**, `while` $B$ `do` $C$. This is one of the many repetitive control commands found in common sequential programming languages. The intuition here is that the command $C$ is to be repeatedly executed until the Boolean guard $B$ can be evaluated to `false`. Note that this is a somewhat dangerous command; if $B$ always evaluates to `tt` then this command will execute forever, repeating the command $C$ indefinitely.

- **Skip**, `skip`. This construct, the final one, is a bit of a non-entity in that its execution has no effect. We could do without this construct in the language but it will prove to be very useful in Chapter 3.2.

We should point out that in Figure 3.1, as usual, we are describing abstract syntax rather than concrete syntax. If we want to describe a particular command in a linear manner we must ensure that its abstract structure is apparent, by using brackets or as in the example command on page 38 using indentation and white space. For more discussion on this point see page 4 of Chapter 1.

## 3.1 Big-step semantics

In order to design a big-step semantics for the language *While* we need to have an intuition about what we expect commands to do. Following the informal descriptions of the individual constructs above, intuitively we expect a command to execute a sequence of assignments, with the precise sequence depending on the flow of control in the construct, dictated by the evaluation of Boolean expressions in the test and while components. An individual assignment is a transformation on the memory of a machine on which the command is expected to run. A command is expected to start executing relative to an initial memory state, effect a series of updates to the memory, and then halt. Therefore we can describe the overall effect of a command as a transformation from an initial memory state to the terminal memory state. Our big-step semantics will prescribe the allowed transformations, without prescribing in any great detail how the the transformations are to be performed.

Before proceeding further we need to introduce some notation for memory states. An individual memory location holds a value, which for *While*, is a numeral. Therefore a snapshot of the memory, which we refer to as a *state*, is captured completely by a function from locations to numerals:

$$s : \text{Locs} \to \mathit{Nums}$$

$s : \{ \square_1 \mapsto n_1, \square_2 \to n_2, \ldots \} ; \quad Locs := \{ \square_1, \square_2, \ldots, \square_m \}$

We use standard mathematical notation for states, with $s(\textsc{l})$ denoting the numeral currently held in location $\textsc{l}$; the collection of all possible states is denoted by *States*; that is *States* is a convenient notation for the function space Locs $\to$ *Nums*. In addition we need one new piece of notation for modifying states. For any state $s$, the new state $s[\textsc{k} \mapsto n]$ returns the same numeral as the old state $s$ for every location $\textsc{l}$ different from $\textsc{k}$, and for $\textsc{k}$ it returns the numeral $\mathtt{n}$. Formally $s[\textsc{k} \mapsto n]$ is defined by:

$$s[\textsc{k} \mapsto n](\textsc{l}) = \begin{cases} \mathtt{n} & \text{if } \textsc{k} = \textsc{l} \\ s(\textsc{l}) & \text{otherwise} \end{cases}$$

The big-step semantics for *While* has as judgements

$$\langle C, s_i \rangle \Downarrow s_t$$

where $C$ is a command from *Com* and $s_i$, $s_t$ are states. The intention is that this judgement captures the following informal intuition:

> when the command $C$ is run to completion from the initial state $s_i$ it eventually terminates in the state $s_t$.

However the behaviour of commands depends on the behaviour of arithmetic and Boolean expressions, and therefore we can only formalise their behaviour if we already have a formal account of how expressions work. Consider, for example, the commands

- `if` $\textsc{l} = \textsc{k}$ `then` $\textsc{l}_1 := \textsc{k} + \textsc{l}_1$ `else` $\textsc{l}_2 := \textsc{l} + (\textsc{k} + 2)$
- `while` $\neg\,(\textsc{l}_1 = \textsc{l}_2)$ `do` $\textsc{l}_2 := \textsc{l}_2 + 1\,;\textsc{l}_3 := \textsc{l}_3 + 1$

(B-NUM)
$$\frac{}{\langle \mathtt{n}, s\rangle \Downarrow \mathtt{n}}$$

(B-LOC)
$$\frac{}{\langle \textsc{l}, s\rangle \Downarrow s(\textsc{l})}$$

(B-ADD)
$$\frac{\langle E_1, s\rangle \Downarrow \mathtt{n}_1 \qquad \langle E_2, s\rangle \Downarrow \mathtt{n}_2}{\langle E_1 + E_2, s\rangle \Downarrow \mathtt{n}_3} \quad n_3 = \mathsf{add}(n_1, n_2)$$

Figure 3.2: Big-step semantics of arithmetic expressions

In order to explain these commands we need to know how to evaluate expressions such as $\textsc{l} + (\textsc{k} + 2)$ and Boolean expressions $\neg\,(\textsc{l}_1 = \textsc{l}_2)$. Consequently before embarking on commands we have to first give a formal semantics to the auxiliary languages *Arith* and *Bool*, from Figure 3.1. We have already considered arithmetic expressions in detail in the Chapter 1. However here they are a little more complicated as their meaning in general depends on the current state of the command which uses them; we can not know the value of the expression $\textsc{l} + (\textsc{k} + 2)$ without knowing what numerals are currently stored in the locations $\textsc{l}$ and $\textsc{k}$. Imp

So we first give a big-step semantics for both arithmetic expressions and Booleans. The judgements here are or the form

$$\langle E, s\rangle \Downarrow \mathtt{n} \qquad \langle B, s\rangle \Downarrow \mathtt{bv}$$

meaning

> the value of expression $E$ relative to the state $s$ is the numeral $\mathtt{n}$

and

> the (Boolean) value of the Boolean expression $B$ relative to the state $s$ is $\mathtt{bv}$.

Note that the form of these judgements imply that we do not expect the evaluation of expressions to affect the state. We are also using `bv` as a meta-variable which ranges over the possible Boolean values; that is in these judgements `bv` stands for either the value `true` or the value `false`.

The rules for arithmetic expressions are given in Figure 3.2, and are a simple extension of the big-step semantics from Chapter 1; there is one new rule, (B-LOC), for looking up the current value in a location.

**Exercise 10** *Design a big-step semantics for Boolean expressions. Intuitively every Boolean expression should evaluate to either* `true` *or* `false`*. So the rules should be such that for every Boolean expression B and every state s, we can derive either the judgement* $\langle B, s\rangle \Downarrow$ `true` *or* $\langle B, s\rangle \Downarrow$ `false`*. However to express the evaluation rules it*

$$\textsc{(b-skip)}\quad \frac{}{\langle \texttt{skip}, s\rangle \Downarrow s} \qquad\qquad \textsc{(b-assign)}\quad \frac{\langle E, s\rangle \Downarrow \texttt{n}}{\langle \textsc{l} := E, s\rangle \Downarrow s[\textsc{l} \mapsto \texttt{n}]}$$

$$\textsc{(b-seq)}\quad \frac{\begin{array}{l}\langle C_1, s\rangle \Downarrow s_1\\ \langle C_2, s_1\rangle \Downarrow s'\end{array}}{\langle C_1 \,;\, C_2, s\rangle \Downarrow s'}$$

$$\textsc{(b-if.t)}\quad \frac{\begin{array}{l}\langle B, s\rangle \Downarrow \texttt{true}\\ \langle C_1, s\rangle \Downarrow s'\end{array}}{\langle \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, s\rangle \Downarrow s'} \qquad \textsc{(b-if.f)}\quad \frac{\begin{array}{l}\langle B, s\rangle \Downarrow \texttt{false}\\ \langle C_2, s\rangle \Downarrow s'\end{array}}{\langle \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, s\rangle \Downarrow s'}$$

$$\textsc{(b-while.f)}\quad \frac{\langle B, s\rangle \Downarrow \texttt{false}}{\langle \texttt{while } B \texttt{ do } C, s\rangle \Downarrow s} \qquad \textsc{(b-while.t)}\quad \frac{\begin{array}{l}\langle B, s\rangle \Downarrow \texttt{true}\\ \langle C, s\rangle \Downarrow s_1\\ \langle \texttt{while } B \texttt{ do } C, s_1\rangle \Downarrow s'\end{array}}{\langle \texttt{while } B \texttt{ do } C, s\rangle \Downarrow s'}$$

Figure 3.3: Big-step semantics of *While*

*is best to introduce a meta-variable* `bv` *to represent either of these Boolean values, as suggested above. Recall that the rules in Figure 3.2 are facilitated by the use of* `n` *as a meta-variable for the numerals* `0`, `1`, *. . . .* □

These auxiliary judgements are now used in Figure 3.3, containing the defining rules for commands. Basically for each syntactic construct in *Com* we have a particular rule, or pair of rules, which directly formalises the intuition given above, on pages 38 and 39. We look briefly at each of these in turn.

The command $\textsc{l} := E$ is a single statement. Intuitively from start state $s$

- we calculate the current value of the expression $E$, $\langle E, s\rangle \Downarrow \texttt{n}$.
- The final state is then obtained by updating the value in location $\textsc{l}$, $s[\textsc{l} \mapsto \texttt{n}]$

This is the import of the rule (b-assign).

As an example of the application of this rule consider the result of executing the simple command $\textsc{l}_1 := \textsc{l}_2 + 1$ relative to a store in which location $\textsc{l}_1$ contains the value 1 and $\textsc{l}_2$ contains 2. In general let us use $s_{nm}$ to denote a store in which $\textsc{l}_1$, $\textsc{l}_2$ contain

$$\dfrac{\dfrac{\dots}{\langle \mathrm{L}_1 := \mathrm{L}_2 + 1, s_{12}\rangle \Downarrow s_{32}}\ (\textsc{b-assign}) \qquad \dfrac{\dfrac{\dfrac{}{\langle \mathrm{L}_1, s_{32}\rangle \Downarrow 3}\ (\textsc{b-loc}) \qquad \dfrac{}{\langle 2, s_{32}\rangle \Downarrow 2}\ (\textsc{b-num})}{\langle \mathrm{L}_1 + 2, s_{32}\rangle \Downarrow 5}\ (\textsc{b-add})}{\langle \mathrm{L}_2 := \mathrm{L}_1 + 2, s_{32}\rangle \Downarrow s_{35}}\ (\textsc{b-assign})}{\langle \mathrm{L}_1 := \mathrm{L}_2 + 1 \,;\, \mathrm{L}_2 := \mathrm{L}_1 + 2, s_{12}\rangle \Downarrow s_{35}}\ (\textsc{b-seq})$$

Figure 3.4: An application of the rule (b-seq)

the numerals n, m respectively. Then we wish to find some store $s_{nm}$ such that the judgement

$$\langle \mathrm{L}_1 := \mathrm{L}_2 + 1, s_{12}\rangle \Downarrow s_{nm}$$

can be derived. Here is one possible derivation, which uses one application of the rule (b-assign), together with the rules in Figure 3.2 in order to establish required premises.

$$\dfrac{\dfrac{\dfrac{}{\langle \mathrm{L}_1, s_{12}\rangle \Downarrow 2}\ (\textsc{b-loc}) \qquad \dfrac{}{\langle 1, s_{12}\rangle \Downarrow 1}\ (\textsc{b-num})}{\langle \mathrm{L}_2 + 1, s_12\rangle \Downarrow 3}\ (\textsc{b-add})}{\langle \mathrm{L}_1 := \mathrm{L}_2 + 1, s_{12}\rangle \Downarrow s_{32}}\ (\textsc{b-assign})$$

The final state $s_{32}$ emerges because this is the store to which $s_{12}[\mathrm{L}_1 \mapsto 3]$ evaluates.

To calculate the final state which results from executing $C_1 \,;\, C_2$ in initial state $s$

- we first execute $C_1$ in initial state $s$, to obtain the intermediate state $s_1$.
- We then execute $C_2$ from this intermediate state $s_1$ to obtain the final state $s'$. This is then the final state after the successful execution of the composed command $C_1 \,;\, C_2$ from the initial state $s$.

The rule (b-seq) is a direct formalisation of this informal description.

We give an example application of this rule in Figure 3.4 which calculates the result of executing the compound command $\mathrm{L}_1 := \mathrm{L}_2 + 1 \,;\, \mathrm{L}_2 := \mathrm{L}_1 + 2$ from the initial store $s_{12}$. The rule (b-seq) is the last one used in the derivation and is applied to two premises:

(1) $\langle \mathrm{L}_1 := \mathrm{L}_2 + 1, s_{12}\rangle \Downarrow s_{32}$: we have already seen a derivation of this judgement, which is therefore omitted from Figure 3.4

(2) $\langle \mathrm{L}_2 := \mathrm{L}_1 + 2, s_{32}\rangle \Downarrow s_{35}$: this judgement in turn is provided with its own derivation in Figure 3.4.

The effect of executing the test $\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2$ from the state $s$ depends on the value of the Boolean expression $B$ relative to $s$; so it is convenient to express the semantics using two sub-rules, one which can be applied when $\langle B, s\rangle \Downarrow \texttt{true}$ and the other when $\langle B, s\rangle \Downarrow \texttt{false}$. These, (B-IF.T) and (B-IF.F), formalise the obvious intuition that executing $\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2$ amount to the execution of $C_1$ when $B$ is true and $C_2$ when it is false.

The only non-trivial command to consider is $C = \texttt{while}\ B\ \texttt{do}\ C$; the intuitive explanation given on page 39 is not very precise, refering as it does to the *repeated execution of C until ......* We can be a little more precise by considering two sub-cases:

(i) If the Boolean guard $B$ evaluates to $\texttt{false}$ immediately in the initial state $s$, then the body $C$ is never executed and the command immediately terminates with the same the final state, $s$. This is formalised in the rule (B-WHILE.F).

(ii) If $B$ evaluates to $\texttt{true}$ we expect the body $C$ to be executed at least once.

Firming up on exactly what should happen in case (ii) we expect $C$ to successfully terminate in an intermediate state, say $s_1$ and then for the execution of $C$ to be repeated, but this time from the newly obtained state $s_1$. This is formalised in the rule (B-WHILE.T). Note that this inference rule is qualitatively different than all the other rules we have seen so far. Up to now, the behaviour of a compound command is determined entirely by the behaviour of its individual components. For example, according to the rule (B-SEQ), the behaviour of the compound $C_1\ ;\ C_2$ is determined completely by that of individual components, $C_1$ and $C_2$; similarly $\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2$ is explained in the rules (B-IF.T) and (B-IF.F) purely in terms of the behaviour of the individual components $B$, $C_1$ and $C_2$. However this is not the case with the rule (B-WHILE.T); to conclude the judgement $\langle \texttt{while}\ B\ \texttt{do}\ C, s\rangle \Downarrow s'$ we have a premise which still involves the command $\texttt{while}\ B\ \texttt{do}\ C$ itself.

The final possible command is the ineffective $\texttt{skip}$; its execution has no effect on the state and therefore we have the axiom $\langle \texttt{skip}, s\rangle \Downarrow s$ in Rule (B-NOOP).

Let us now look at a sample derivation in the logical system determined by these rules. Consider the command $C_1$ given on page 38. In order to set out the derivation we use the following abbreviations:

| | | |
|---|---|---|
| $C_{11}$ | for | $\mathrm{L}_2 := \texttt{1}\ ;\ \mathrm{L}_3 := \texttt{0}$ |
| $C_{12}$ | for | $\mathrm{L}_2 := \mathrm{L}_2 + \texttt{1}\ ;\ \mathrm{L}_3 := \mathrm{L}_3 + \texttt{1}$ |
| $B$ | for | $\neg\,(\mathrm{L}_1 = \mathrm{L}_2)$ |
| $W$ | for | $\texttt{while}\ \neg\,(\mathrm{L}_1 = \mathrm{L}_2)\ \texttt{do}\ C_{12}$ |

So the command $C_1$ can be alternatively described by $C_{11}\ ; W$. We also use the notation $s_{mnk}$ to denote a state of the memory in which the location $\mathrm{L}_1$ contains the numeral $\texttt{m}$, $\mathrm{L}_2$ contains $\texttt{n}$ and $\mathrm{L}_3$ contains $\texttt{k}$; these are the only locations used by the command $C_1$. With these abbreviations a formal derivation of the judgement

$$\langle C_1, s_{377}\rangle \Downarrow s_{322}$$

Sub-proof **A**:

$$\dfrac{\dfrac{\dfrac{\cdots}{\langle B, s_{332}\rangle \Downarrow \texttt{false}}}{\langle W, s_{332}\rangle \Downarrow s_{332}}\,\text{(B-WHILE.F)} \quad \dfrac{\dfrac{}{\langle \mathrm{L}_2 := \mathrm{L}_2 + \mathtt{1}, s_{321}\rangle \Downarrow s_{331}}\,\text{(B-ASSIGN)} \quad \dfrac{}{\langle \mathrm{L}_3 := \mathrm{L}_3 + \mathtt{1}, s_{331}\rangle \Downarrow s_{332}}\,\text{(B-ASSIGN)}}{\langle C_{12}, s_{321}\rangle \Downarrow s_{332}}\,\text{(B-SEQ)} \quad \dfrac{\cdots}{\langle B, s_{321}\rangle \Downarrow \texttt{true}}}{\langle W, s_{321}\rangle \Downarrow s_{322}}\,\text{(B-WHILE.T)}$$

Sub-proof **B**:

$$\dfrac{\dfrac{}{\langle \mathrm{L}_2 := \mathrm{L}_2 + \mathtt{1}, s_{310}\rangle \Downarrow s_{320}}\,\text{(B-ASSIGN)} \quad \dfrac{}{\langle \mathrm{L}_3 := \mathrm{L}_3 + \mathtt{1}, s_{320}\rangle \Downarrow s_{321}}\,\text{(B-ASSIGN)}}{\langle C_{12}, s_{310}\rangle \Downarrow s_{321}}\,\text{(B-SEQ)}$$

$$\dfrac{\dfrac{\dfrac{}{\langle \mathrm{L}_2 := \mathtt{1}, s_{377}\rangle \Downarrow s_{317}}\,\text{(B-ASSIGN)} \quad \dfrac{}{\langle \mathrm{L}_3 := \mathtt{0}, s_{317}\rangle \Downarrow s_{310}}\,\text{(B-ASSIGN)}}{\langle C_{11}, s_{377}\rangle \Downarrow s_{310}}\,\text{(B-SEQ)} \quad \dfrac{\dfrac{\boxed{\mathbf{A}}}{\langle W, s_{321}\rangle \Downarrow s_{322}}\,\text{(B-WHILE.T)} \quad \dfrac{\cdots}{\langle B, s_{310}\rangle \Downarrow \texttt{true}} \quad \dfrac{\boxed{\mathbf{B}}}{\langle C_{12}, s_{310}\rangle \Downarrow s_{321}}\,\text{(B-SEQ)}}{\langle W, s_{310}\rangle \Downarrow s_{332}}\,\text{(B-WHILE.T)}}{\langle C_1, s_{377}\rangle \Downarrow s_{332}}\,\text{(B-SEQ)}$$

Figure 3.5: An example derivation

is given in Figure 3.5. So if a compiler is to agree with our formal semantics it must ensure that if $C_1$ is executed from the initial state $s_{377}$ it must eventually terminate with $s_{322}$ as the final state.

In the sequel, generalising the notation introduced in page 8 of Chapter 1.2, we write

$$\vdash_{big} \langle C, s_i\rangle \Downarrow s_t$$

to mean that we can derive the judgement $\langle C, s_i\rangle \Downarrow s_t$ using the rules in Figure 3.3. Of course in such a derivation we would also expect to use the rules in Figure 3.2 to construct derivations for arithmetical expressions occurring in the command $C$; and we would also need rules for Boolean expressions.

Intuitively we expect there to be commands in the language *While* which loop, or continue executing indefinitely. Let us see how this is reflected in the big-step semantics. Consider the command

$$\texttt{while}\,(\neg\ \textsc{l} = \mathbb{0})\ \texttt{do}\ \textsc{l} := \textsc{l} + \texttt{1} \tag{3.1}$$

which we denote by *LP* and let $s$ be any state such that $s(\textsc{l}) > 0$. Our intuition says that executing *LP* from the initial state $s$ would lead to non-termination. So it would be unfortunate if we could derive the judgement

$$\vdash_{big} \langle LP, s\rangle \Downarrow s_t \tag{3.2}$$

for some state $s_t$; this would contradict our intuition as this judgement is supposed to capture the idea that command LP, executed from the initial state $s$ eventually terminates, with terminal state $s_t$.

So how do we know that (3.2) is not true for any state $s_t$ ? We can prove it by contradiction. Suppose a judgement $\langle LP, s\rangle \Downarrow s'$ could be derived for some state $s'$ and some state $s$ such that $s(\textsc{l}) > \mathbb{0}$. In fact there might be many such derivations. We zero in on one of these supposed derivations, one which is at least as short as any other such derivation. So suppose the one we choose is a derivation of the judgement $\langle LP, s_1\rangle \Downarrow s_2$ for a some particular states $s_1,\ s_2$ such that $s_1(\textsc{l}) > 0$, and suppose its derivation uses $k$ applications of the rules in Figure 3.3. What this means that if there is any derivation of any other judgement of the form $\langle LP, s\rangle \Downarrow s'$, where $s(\textsc{l}) > 0$ then that derivation must use at least $k$ rules, and possibly more.

So now let us examine the derivation of the $\langle LP, s_1\rangle\Downarrow s_2$, with the shortest derivation, using $k$ rules. How can this judgement be derived? Because $\langle \neg\ \textsc{l} = \mathbb{0}, s_1\rangle \Downarrow \texttt{true}$ every derivation, including this shortest one, must involve an application of the rule (B-WHILE.T). Specifically the structure of the shortest derivation must take the form

$$\dfrac{\dfrac{}{\langle \neg\ \textsc{l} = \mathbb{0}, s_1\rangle \Downarrow \texttt{true}} \qquad \dfrac{}{\langle \textsc{l} := \textsc{l} + 1, s_1\rangle \Downarrow s_3} \qquad \dfrac{\cdots\cdots}{\langle LP, s_3\rangle \Downarrow s_1}}{\langle LP, s\rangle \Downarrow s_1}\ \text{(B-WHILE.T)}$$

Now because $\vdash_{big} \langle \textsc{l} := \textsc{l}+\texttt{1}, s_1\rangle \Downarrow s_3$ we know that $s_3(\textsc{l}) > \mathbb{0}$. And in the above derivation the $\ldots\ldots$ actually provides a derivation for the judgement $\langle LP, s_3\rangle \Downarrow s_1$. Moreover the size of this derivation is actually smaller than that of $\langle LP, s_1\rangle \Downarrow s_2$; it uses strictly fewer than $k$ rules. But this is a contradiction since we assumed that this derivation of $\langle LP, s_1\rangle \Downarrow s_2$ was shortest, with $k$ rules.

**Exercise 11** *Consider the alternative command LP1* = `while true do skip`. *Prove that for any arbitrary state $s$ we can not derive a judgement of the form $\langle LP1, s\rangle \Downarrow s_t$ for any state $s_t$.* □

## 3.2 Small-step semantics

The big-step semantics of the previous section merely specifies what the final state should be when a command is executed from some initial state; it does not put constraints on how the execution from the initial state to the final state is to proceed. Intuitively executing a command involves performing some sequence of *basic operations*, determined by the control flow in the command; the basic operations consist of

(a) updates to the memory, effected by assignment statements

(b) evaluation of Boolean guards, in test or while statements; the results of these evaluations determine the flow of control.

In this section we give a more detailed semantics for *While* which describes, at least indirectly, this sequence of basic operations which should be performed in order to execute a given command.

The judgements in the small-step semantics for *While* take the form

$$\langle C, s\rangle \rightarrow \langle C', s'\rangle$$

meaning:

> one step in the execution of the command $C$ relative to the state $s$ changes the state to $s'$ and leaves the residual command $C'$ to be executed.

Thus the transition from $C$ to $C'$ is achieved by performing the first basic operation, while the execution of the residual $C'$ will determine the remaining basic operations necessary to execute $C$ to completion.

This semantics also depends on how both arithmetic expressions and Booleans are evaluated. But since we are mainly interested in commands our inference rules, in Figure 3.6, are given relative to the big-step semantics of both arithmetics and Booleans. The degenerate command `skip` plays a fundamental role in these rules. Intuitively the execution of `skip` relative to any initial state $s$ involves the execution of *no* basic operations, and thus we would expect that the judgement

$$\langle \texttt{skip}, s\rangle \rightarrow \langle C, s'\rangle$$

can not be derived for any $\langle C, s'\rangle$; indeed the pair $\langle \texttt{skip}, s\rangle$ will indicate a terminal configuration, which requires no further execution.

Let us now briefly look at the rules in Figure 3.6. Executing the command $\textsc{l} := E$ involves performing one basic operation, namely updating the numeral stored in $\textsc{l}$ to be whatever the expression $E$ evaluates to. Thus in (s-ass)

(S-ASS)
$$\frac{\langle E, s\rangle \Downarrow \texttt{n}}{\langle \textsc{l} := E, s\rangle \rightarrow \langle \texttt{skip}, s[\textsc{l} \mapsto \texttt{n}]\rangle}$$

(S-SEQ.LEFT)
$$\frac{\langle C_1, s\rangle \rightarrow \langle C_1', s'\rangle}{\langle C_1 \,;\, C_2, s\rangle \rightarrow \langle C_1' \,;\, C_2, s'\rangle}$$

(S-SEQ.SKIP)
$$\frac{}{\langle \texttt{skip} \,;\, C_2, s\rangle \rightarrow \langle C_2, s\rangle}$$

(S-COND.T)
$$\frac{\langle B, s\rangle \Downarrow \texttt{true}}{\langle \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2, s\rangle \rightarrow \langle C_1, s\rangle}$$

(S-COND.F)
$$\frac{\langle B, s\rangle \Downarrow \texttt{false}}{\langle \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2, s\rangle \rightarrow \langle C_2, s\rangle}$$

(S-WHILE.F)
$$\frac{\langle B, s\rangle \Downarrow \texttt{false}}{\langle \texttt{while}\ B\ \texttt{do}\ C, s\rangle \rightarrow \langle \texttt{skip}, s\rangle}$$

(S-WHILE.T)
$$\frac{\langle B, s\rangle \Downarrow \texttt{true}}{\langle \texttt{while}\ B\ \texttt{do}\ C, s\rangle \rightarrow \langle C \,;\, \texttt{while}\ B\ \texttt{do}\ C, s\rangle}$$

Figure 3.6: Small-step semantics of *While*

- $E$ is evaluated to the numeral n, that is $\langle E, s\rangle \Downarrow \texttt{n}$
- the state $s$ changes to the modified store $s[\textsc{l} \mapsto \texttt{n}]$
- the residual, what remains to be executed is `skip`; that is the command has now been completely executed.

The execution of a statement of the form $C_1 \,;C_1$ is a little more complicated. There are two cases, depending on whether or not there are any basic operations left to be performed in $C_1$. If there is then there will be a judgement of the form $\langle C_1, s\rangle \rightarrow \langle C_1', s'\rangle$, representing the execution of this basic operation. Then the execution of the first step of the compound command $C_1 ; C_2$ is given by the judgement $\langle C_1 ; C_2, s\rangle \rightarrow \langle C_1' ; C_2, s'\rangle$; this is the import of (S-SEQ.LEFT).

However there may be nothing left to execute in $C_1$; although it is not yet apparent, this will only be the case if $C_1$ is precisely the degenerate command `skip`. This accounts for the second rule (S-SEQ.SKIP), which formalises the idea that if $C_1$ has terminated, the execution of $C_2$ should be started. Note that this rule introduces steps into the small-step semantics which do not correspond directly to either (a) or (b) above; these may be considered to be *housekeeping steps*.

**Example 11** *Consider the execution of the compound command* $\text{L}_2$ `:=` `1` ; $\text{L}_3$ `:=` `0` *from the initial state* $s_{377}$*; here we are using the notation for states introduced in the previous section. Using the two rules we have discussed already, we have the following derivation:*

$$\dfrac{\dfrac{}{\langle \text{L}_2 := \mathtt{1}, s_{377}\rangle \to \langle \mathtt{skip}, s_{317}\rangle}\ (\textsc{s-ass})}{\langle \text{L}_2 := \mathtt{1}\,;\, \text{L}_3 := \mathtt{0}, s_{377}\rangle \to \langle \mathtt{skip}\,;\, \text{L}_3 := \mathtt{0}, s_{317}\rangle}\ (\textsc{s-seq.left})$$

*Therefore we can write* $\vdash_{sm} \langle \text{L}_2 := \mathtt{1}\,;\, \text{L}_3 := \mathtt{0}, s_{377}\rangle \to \langle \mathtt{skip}\,;\, \text{L}_3 := \mathtt{0}, s_{317}\rangle$ *which represents the first step in the execution of the compound command, from initial state* $s_{377}$*, an update of the memory.*

*We also have* $\vdash_{sm} \langle \mathtt{skip}\,;\, \text{L}_3 := \mathtt{0}, s_{317}\rangle \to \langle \text{L}_3 := \mathtt{0}, s_{317}\rangle$*, a housekeeping step, because of the derivation*

$$\dfrac{}{\langle \mathtt{skip}\,;\, \text{L}_3 := \mathtt{0}, s_{317}\rangle \to \langle \text{L}_3 := \mathtt{0}, s_{317}\rangle}\ (\textsc{s-seq.skip})$$

*We also have* $\vdash_{sm} \langle \text{L}_3 := \mathtt{0}, s_{317}\rangle \to \langle \mathtt{skip}, s_{310}\rangle$ *because of the derivation consisting of one application of the rule* (s-seq.skip)

$$\dfrac{}{\langle \text{L}_3 := \mathtt{0}, s_{317}\rangle \to \langle \mathtt{skip}, s_{310}\rangle}\ (\textsc{s-seq.skip})$$

*Again this step represents an update to the memory. Recall that we view configurations such as* $\langle \mathtt{skip}, s_{310}\rangle$ *to be terminal, as nothing more needs to be executed. Thus we have executed the command* $\text{L}_2 := \mathtt{1}; \text{L}_3 := \mathtt{0}$ *to completion in three steps, from the start state* $s_{377}$*. Borrowing the notation from Chapter 1 we have*

$$\langle \text{L}_2 := \mathtt{1}\,;\, \text{L}_3 := \mathtt{0}, s_{377}\rangle \to^3 \langle \mathtt{skip}, s_{310}\rangle \qquad \square$$

Returning to our discussion of the inference rules in Figure 3.6, the treatment of the test, `if` $B$ `then` $C_1$ `else` $C_2$ is captured in the two rules (s-cond.t) and (s-cond.f). Depending on what Boolean value $B$ evaluates to, we move on to execute either $C_1$ or $C_2$. Note that with these rules, the evaluation of the Boolean together with the resulting decision is taken to be a single execution step.

Finally we come to the interesting construct `while` $B$ `do` $C$. The behaviour depends naturally on the value of the guard $B$ in the current state. Intuitively if this evaluates to `false` then the body $C$ is not to be executed; in short the computation is over. This is formalised in (s-small.f). On the other hand if it is true, $\langle B, s\rangle \Downarrow \mathtt{true}$, then we expect the body to executed at least once, and the execution of the overall command to be repeated. This is conveniently expressed in (s-while.t) by the transition from `while` $B$ `do` $C$ to the command $C$ ; `while` $B$ `do` $C$.

Let us revisit the command $C_1$ on page 38, which re-using the abbreviations on page 44 is equivalently expressed as $C_{11}$ ; $W$. Let us use the small-step semantics to execute it from the initial state $s_{377}$.

Intuitively the first step in this computation is the update of the location $\textsc{l}_2$ with the numeral 1, and this is borne out formally by the following derivation:

$$\cfrac{\cfrac{\cfrac{}{\langle \textsc{l}_2 := \mathtt{1}, s_{377}\rangle \to \langle \mathtt{skip}, s_{317}\rangle}\ (\textsc{s-ass})}{\langle C_{11}, s_{377}\rangle \to \langle (\mathtt{skip}\ ;\ \textsc{l}_3 := \mathtt{1}), s_{317}\rangle}\ (\textsc{s-seq.l})}{\langle C_1, s_{377}\rangle \to \langle (\mathtt{skip}\ ;\ \textsc{l}_3 := \mathbb{0})\ ;\ W, s_{317}\rangle}\ (\textsc{s-seq.l})$$

So we have the judgement $\vdash_{sm} \langle C_1, s_{377}\rangle \to \langle (\mathtt{skip}\ ;\ \textsc{l}_3 := \mathbb{0})\ ;\ W, s_{317}\rangle$.

The second step is the rather uninteresting housekeeping move

$$\vdash_{sm} \langle (\mathtt{skip}\ ;\ \textsc{l}_3 := \mathbb{0})\ ;\ W, s_{317}\rangle \to \langle \textsc{l}_3 := \mathbb{0}\ ;\ W, s_{317}\rangle$$

justified by the formal derivation

$$\cfrac{\cfrac{}{\langle \mathtt{skip}\ ;\ \textsc{l}_3 := \mathbb{0}, s_{377}\rangle \to \langle \textsc{l}_3 := \mathtt{1}, s_{317}\rangle}\ (\textsc{s-seq.s})}{\langle (\mathtt{skip}\ ;\ \textsc{l}_3 := \mathbb{0})\ ;\ W, s_{317}\rangle \to \langle \textsc{l}_3 := \mathbb{0}\ ;\ W, s_{317}\rangle}\ (\textsc{s-seq.l})$$

We leave the reader to check the derivation of the two subsequent moves

$$\vdash_{sm} \langle \textsc{l}_3 := \mathbb{0}\ ;\ W, s_{317}\rangle \to \langle \mathtt{skip}\ ;\ W, s_{311}\rangle \qquad \vdash_{sm} \langle \mathtt{skip}\ ;\ W, s_{310}\rangle \to \langle W, s_{310}\rangle$$

Thus in four steps we have reached the execution of the while command; using the notation of Chapter 1 this is expressed formally as:

$$\langle C_1, s_{377}\rangle \to^4 \langle \mathtt{while}\ \neg\ (\textsc{l}_1 = \textsc{l}_2)\ \mathtt{do}\ C_{12}, s_{310}\rangle \tag{3.3}$$

We are not getting very far.

We have not seen the rules for evaluating Boolean expressions, but let us assume that that they are such that $\langle \neg\ (\textsc{l}_1 = \textsc{l}_2), s_{310}\rangle \Downarrow \mathtt{true}$ can be derived. Then the next step

$$\vdash_{sm} \langle \mathtt{while}\ \neg\ (\textsc{l}_1 = \textsc{l}_2)\ \mathtt{do}\ C_{12}, s_{310}\rangle \to \langle C_{12}\ ;\ W, s_{310}\rangle \tag{3.4}$$

is justified by an application of the rule (s-while.t), in the nearly trivial derivation:

$$\cfrac{\cfrac{}{\langle \neg\ (\textsc{l}_1 = \textsc{l}_2), s_{310}\rangle \Downarrow \mathtt{true}}}{\langle \mathtt{while}\ \neg\ (\textsc{l}_1 = \textsc{l}_2)\ \mathtt{do}\ C_{12}, s_{310}\rangle \to \langle C_{12}\ ;\ W, s_{310}\rangle}\ (\textsc{s-while.t})$$

The command $C_{12}$ consisting of two assignments is now executed, taking four steps

$$\vdash_{sm} \langle C_{12}\ ;\ W, s_{310}\rangle \to^4 \langle \mathtt{while}\ \neg\ (\textsc{l}_1 = \textsc{l}_2)\ \mathtt{do}\ C_{12}, s_{321}\rangle \tag{3.5}$$

and we are back to executing the while command once more; but note the state has changed.

Another round of five derivations gives

$$\langle \mathtt{while}\ \neg\ (\textsc{l}_1 = \textsc{l}_2)\ \mathtt{do}\ C_{12}, s_{321}\rangle \ \to^5\ \langle \mathtt{while}\ \neg\ (\textsc{l}_1 = \textsc{l}_2)\ \mathtt{do}\ C_{12}, s_{332}\rangle \tag{3.6}$$

Now, since presumably $\langle \neg\, (\mathrm{L}_1 \;=\; \mathrm{L}_2), s_{332}\rangle \Downarrow \texttt{false}$ is also derivable, and therefore a near trivial derivation using the rule (s-while.f) justifies the final step

$$\vdash_{sm} \langle \texttt{while} \neg\, (\mathrm{L}_1 = \mathrm{L}_2)\ \texttt{do}\ C_{12}, s_{322}\rangle \;\rightarrow\; \langle \texttt{skip}, s_{332}\rangle \tag{3.7}$$

Combining all the judgements (3.3), (3.4), (3.5), (3.6) and (3.7) we have the complete execution

$$\langle C_1, s_{377}\rangle \;\rightarrow^{15} \langle \texttt{skip}, s_{322}\rangle$$

To end this section let us revisit the non-terminating command $LP = \texttt{while} \neg\, \mathrm{L} = \mathbb{0}\ \texttt{do}\ \mathrm{L} := \mathrm{L} + 1$ discussed on page 46 of Chapter 3.1. Again let $s$ be any state satisfying $s(\mathrm{L}) > 0$. Assuming $\langle \neg\, \mathrm{L} = \mathbb{0}, s\rangle \Downarrow \texttt{true}$ is derivable, an application of the rule (s-while.t) will justify the judgement

$$\vdash_{sm} \langle LP, s\rangle \;\rightarrow\; \langle (\mathrm{L} := \mathrm{L} + 1)\,;\, LP, s\rangle$$

We then have

$$\vdash_{sm} \langle (\mathrm{L} := \mathrm{L} + 1)\,;\, LP, s\rangle \rightarrow \langle (\texttt{skip}\,;\, LP, s_1\rangle \qquad \text{and} \qquad \vdash_{sm} \langle \texttt{skip}\,;\, LP, s_1\rangle \;\rightarrow\; \langle LP, s_1\rangle$$

where $s_1$ is some state which also satisfies $s_1(\mathrm{L}) > \mathbb{0}$. In other words,

$$\langle LP, s\rangle \;\rightarrow^3\; \langle LP, s_1\rangle$$

In three steps we are back where we started.

So in the small-step semantics non-termination is manifest by computation sequences which go on indefinitely. In our particular case:

$$\langle LP, s\rangle \;\rightarrow^3\; \langle LP, s_1\rangle \;\rightarrow^3\; \langle LP, s_2\rangle \;\rightarrow^3\; \ldots\ldots \;\rightarrow^3\; \langle LP, s_k\rangle \;\rightarrow^3\; \ldots$$

where $s(\mathrm{L}) > \mathbb{0}$ and $s_i(\mathrm{L}) > \mathbb{0}$ for every $i$.

**Exercise 12** *Give a small-step semantics to arithmetic and Boolean expressions.* □

**Exercise 13** *Use your small-step semantics of arithmetics and Boolean expressions to rewrite the semantics of commands in Figure 3.6, so that no big-step semantics is used.* □

## 3.3 Properties

In this section we review the two semantics we have given for the language *While* in the previous two sections, Chapter 3.1 and Chapter 3.2. In particular we are interested in the relationship between them, and ensuring that they are self-consistent. Section 2.2.3 serves as a model for the development, and most of the mathematical arguments we used already appear there. However in places we have to use a more complicated form of induction, rule induction, in place of structural induction. But for the moment let us describe structural induction as it applies to commands in *While*. From the BNF definition in Figure 3.1 we see that there are five methods for constructing commands from *Com*. There are two kinds of *seeds* or starting points, and three kinds of constructors:

- **Base cases**:
  - the constant `skip` is a command
  - For every location name $\text{L}$ and arithmetic expression $E$, $\text{L} := E$ is a command.
- **Inductive steps**:
  - If $C_1$ and $C_2$ are commands, then so is $C_1 \,;\, C_2$.
  - If $C_1$ and $C_2$ are commands then `if` $B$ `then` $C_1$ `else` $C_2$ is also a command, for every Boolean expression $B$.
  - If $C$ is a command, then so is `while` $B$ `do` $C$, again for every Boolean expression $B$.

So suppose we wish to prove that some property $P(C)$ is true for every command $C \in$ *Com*. Structural induction will ensure that this will be true provided we prove five separate properties:

- **Base cases**:
  - Prove, in some way or another, that $P($`skip`$)$ is true.
  - Prove that $P(\text{L} := E)$ is true for every location name $\text{L}$ and arithmetic expression $E$.
- **Inductive steps**:
  - Under the assumption that both $P(C_1)$ and $P(C_2)$ are true, for some arbitrary pair of commands $C_1$, $C_2$ prove that $P(C_1 \,;\, C_2)$ follows.
  - Similarly, under the same two assumptions $P(C_1)$ and $P(C_2)$ prove that $P($`if` $B$ `then` $C_1$ `else` $C_2$ is a consequence, for every Boolean expression $B$.
  - Finally, assuming that $P(C)$ is true for some arbitrary command $C$, prove that $P($`while` $B$ `do` $C)$ follows as a logical consequence, again for every Boolean expression $B$.

So these kinds of proofs will be long, with much detail. But normally the details will be fairly mundane and the entire process is open to automatic or semi-automatic software assistance.

But note that in general properties of commands will depend on related properties of the auxiliary arithmetic and Boolean expressions; this is to be expected, as the semantic definitions for commands depend on an a priori semantics for arithmetic and Boolean expressions. In particular we have used a big-step semantics for these auxiliary languages. Here are two particularly useful properties of these semantic definitions.

**Proposition 12** *For every expression $E \in$ Arith and every state $s$*

*(i) (**Normalisation**) there exists some numeral* `n` *such that* $\vdash_{big} \langle E, s\rangle \Downarrow \texttt{n}$

*(ii)* ***(Determinacy)*** *if* $\vdash_{big} \langle E, s\rangle \Downarrow \mathtt{n}_1$ *and* $\vdash_{big} \langle E, s\rangle \Downarrow \mathtt{n}_2$ *then* $n_1 = n_2$.

**Proof:** Both use structural induction on the language *Arith*; the arguments are virtually identical to those used in Chapter 2.2.3 for the slightly simpler language *Exp*.

We have not actually given a big-step semantics for Boolean expressions, but in the sequel we will assume that one has been given and that it also enjoys these properties.

First let us look at the small-step semantics.

**Exercise 14** *Let C be any command different from* `skip`*. Use structural induction to prove that for every state s there is a derivation of the judgement* $\langle C, s\rangle \to \langle C', s'\rangle$ *for some configuration* $\langle C', s'\rangle$. □

**Proposition 13** *For every command* $C \in Com$ *and every state s, if* $\vdash_{sm} \langle C, s\rangle \to \langle C_1, s_1\rangle$ *and* $\vdash_{sm} \langle C, s\rangle \to \langle C_1, s_1\rangle$ *then* $C_1$ *is identical to* $C_2$ *and* $s_1$ *is identical to* $s_2$.

**Proof:** By structural induction on the command $C$. Here the property of commands we want to prove $P(C)$ is:

> for every state $s$, if $\vdash_{sm} \langle C, s\rangle \to \langle C_1, s_1\rangle$ and $\vdash_{sm} \langle C, s\rangle \to \langle C_1, s_1\rangle$ then $C_1 = C_2$ and $s_1 = s_2$.

As explained above, we now have five different statements about $P(-)$ to prove:

(i) A base case, when $C$ is `skip`. Here $P(\mathtt{skip})$ is vacuously true, as it is not possible to derive any judgement of the form $\langle \mathtt{skip}, s\rangle \to \langle D, s'\rangle$, for any pair $\langle D, s'\rangle$.

(ii) Another base case, when $C$ is $\textsc{l} := E$. From Proposition 12 we know that, for a given state $s$, there is exactly one number $n$ such that $\vdash_{big} \langle E, s\rangle \Downarrow \mathtt{n}$. Looking at the collection of rules in Figure 3.6, there is only one possible rule to apply to the pair $\langle \textsc{l} := E, s\rangle$, namely (s-ass). Consequently, if $\vdash_{sm} \langle \textsc{l} := E, s\rangle \to \langle C_1, s_1\rangle$ and $\vdash_{sm} \langle \textsc{l} := E, s\rangle \to \langle C_1, s_1\rangle$ then both $C_1$ and $C_2$ must be `skip`, and both $s_1$ and $s_2$ must be the same state, $s[\textsc{l} \mapsto \mathtt{n}]$.

(iii) An inductive case, when $C$ is $D_1 \,;\, D_2$. Here we are allowed to assume that $P(D_1)$ and $P(D_2)$ are true, and from these we must show that $P(D_1 \,;\, D_2)$ follows. So suppose we have a derivation of both judgements

$$\langle D_1 \,;\, D_2, s\rangle \to \langle C_1, s_1\rangle \qquad \text{and} \qquad \langle D_1 \,;\, D_2, s\rangle \to \langle C_2, s_2\rangle \tag{3.8}$$

Lets do a case analysis on the structure of $D_1$. First suppose it is the trivial command `skip`. Then, looking at the inference rules in Figure 3.6 we see that the only possible rule which can be used to infer these judgements is (s-seq.skip); note in particular that (s-seq.left) can not be used, as an appropriate premise, $\langle \mathtt{skip}, s\rangle \to \langle C', s'\rangle$ can not be found. So both of the above derivations must have exactly the same the form, namely:

$$\frac{}{\langle \mathtt{skip} \,;\, D_2, s\rangle \to \langle D_2, s_1\rangle}\ (\textsc{s-seq.skip})$$

In other words both $C_1$ and $C_2$ are the same command $D_2$, and $s_1$ and $s_2$ are the same state, $s$.

On the other hand if $D_1$ is different than `skip`, a perusal of Figure 3.6 will see that the only possible rule which can be used is (s-seq.left). So the pair of derivations must be of the form

$$\dfrac{\dfrac{\cdots}{\langle D_1, s\rangle \to \langle D_1', s'\rangle}\;?}{\langle D_1 \,;\, D_2, s\rangle \to \langle D_1' \,;\, D_2, s'\rangle}\;\text{(s-seq.skip)} \qquad \text{and} \qquad \dfrac{\dfrac{\cdots}{\langle D_1, s\rangle \to \langle D_1'', s''\rangle}\;??}{\langle D_1 \,;\, D_2, s\rangle \to \langle D_1'' \,;\, D_2, s''\rangle}\;\text{(s-seq.skip)}$$

So that in (3.8) above, $\langle C_1, s_1\rangle$ has the form $\langle D_1' \,;\, D_2, s'\rangle$ and $\langle C_2, s_2\rangle$ the form $\langle D_1'' \,;\, D_2, s''\rangle$.

But to construct these derivations we must already have derivations of both the judgements $\langle D_1, s\rangle \to \langle D_1', s'\rangle$ and $\langle D_1, s\rangle \to \langle D_1'', s''\rangle$. Here we can now apply the first inductive hypothesis, $P(D_1)$, to obtain $D_1'$ is the same as $D_1''$ and $s' = s''$. From this we immediately have our requirement, that $\langle C_1, s_1\rangle$ coincides with $\langle C_2, s_2\rangle$.

Note that in this case we have only used one of the inductive hypotheses, $P(D_1)$.

(iv) Another inductive case, when $C$ is `while` $B$ `do` $D$, for some Boolean expression $B$ and command $D$. Here we are allowed to assume that $P(D)$ is true, and from this hypothesis to demonstrate that $P(C)$ follows. To this end suppose we have derivations of two judgements of the form

$$\langle \texttt{while}\ B\ \texttt{do}\ D, s\rangle \to \langle C_1, s_1\rangle \text{ and } \langle \texttt{while}\ B\ \texttt{do}\ D, s\rangle \to \langle C_2, s_2\rangle \tag{3.9}$$

using the rules from Figure 3.6. These derivations have to use the rules (s-while.f) and (s-while.t), which depend on the semantics of the Boolean expression $B$. So to start with let us look at its evaluation. By Proposition 12, or more correctly the version of this proposition for Boolean expressions, there is exactly one Boolean value `bv` such that the judgement $\langle B, s\rangle \Downarrow \texttt{bv}$ can be derived. There are only two possibilities for `bv`, namely `true` and `false` respectively. Let us look at these two possibilities in turn.

First suppose that $\langle B, s\rangle \Downarrow \texttt{false}$. In this case the rule (s-while.t) can not be used in the derivation of either of the derivations of the judgements in (3.9) above. In fact both can only use (s-while.f) and therefore both have exactly the same derivation, namely:

$$\dfrac{\langle B, s\rangle \Downarrow \texttt{false}}{\langle \texttt{while}\ B\ \texttt{do}\ D, s\rangle \to \langle \texttt{skip}, s\rangle}\;\text{(s-while.f)}$$

So in this case obviously $C_1$ and $C_2$ are the same command, `skip`, and $s_1$ and $s_2$ are the same state, namely $s$.

Now we consider the case when $\langle B, s\rangle \Downarrow \texttt{true}$. In this case both the derivations have to use the rule (s-while.t). But again the derivations have to have exactly the same form, namely:

$$\frac{\langle B, s\rangle \Downarrow \texttt{true}}{\langle \texttt{while}\ B\ \texttt{do}\ D, s\rangle \rightarrow \langle D\,;\texttt{while}\ B\ \texttt{do}\ D, s\rangle}\ (\textsc{s-while.f})$$

So here again we have shown that $C_1$ and $C_2$ in (3.9) above coincide, as they both must be the command $D\,;\texttt{while}\ B\ \texttt{do}\ D$; also $s_1$ and $s_2$ are the same state $s$.

Note that in this case we did not actually need to ever use the inductive hypothesis $P(D)$.

There is one more possibility for $C$, that it is of the form $\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2$; this we leave to the reader to verify.

**Corollary 14** *For every command $C \in Com$, every state $s$ and every natural number $k$, if $\langle C, s\rangle \rightarrow^k \langle C_1, s_1\rangle$ and $\langle C, s\rangle \rightarrow^k \langle C_1, s_2\rangle$ then $C_1$ is identical to $C_2$ and $s_1$ is identical to $s_2$.*

**Proof:** This time we use mathematical induction on the number of steps $k$. The base case, when $k = 0$ is trivial, while the inductive case uses the previous proposition. □

**Exercise 15** *Write out the proof of Corollary 14 in detail.* □

**Theorem 15 (Determinacy)** *For every command $C \in Com$, every state $s$, if $\langle C, s\rangle \rightarrow^* \langle \texttt{skip}, s_1\rangle$ and $\langle C, s\rangle \rightarrow^* \langle \texttt{skip}, s_2\rangle$ then $s_1 = s_2$.*

**Proof:** This is a rather simple consequence of the previous result. We know by definition that

$$\langle C, s\rangle \rightarrow^{k_1} \langle \texttt{skip}, s_1\rangle$$
$$\langle C, s\rangle \rightarrow^{k_2} \langle \texttt{skip}, s_2\rangle$$

for some pair of natural numbers $k_1$, $k_2$. Without loss of generality let us suppose that $k_1 \leq k_2$. Then we actual have

$$\langle C, s\rangle \rightarrow^{k_1} \langle \texttt{skip}, s_1\rangle$$
$$\langle C, s\rangle \rightarrow^{k_1} \langle C', s_2'\rangle \rightarrow^{k_3} \langle \texttt{skip}, s_2\rangle$$

for some $\langle C', s_2'\rangle$, where $k_3$ is the difference between $k_1$ and $k_2$. But by Corollary 14 this must mean that the intermediate command $C'$ must actually be $\texttt{skip}$ and the state $s_2'$ must coincide with $s_1$. But we have already remarked that no small-steps can be taken by the terminal command $\texttt{skip}$. This means that $k_3$ must be 0, and therefore that $s_2$ must be the same as $s_2'$, that is $s_1$.

We now consider the relationship between the two forms of semantics.

**Theorem 16** $\vdash_{big} \langle C, s\rangle \Downarrow s'$ *implies* $\langle C, s\rangle \rightarrow^* \langle \texttt{skip}, s'\rangle$

**Proof:** Similar in style to that of Proposition 4 of Chapter 2.2.3. But because of the complicated inference rule (B-WHILE.T) we can not use structural induction over the command $C$. Instead we use rule induction, as explained in Section 2.3. Specifically, as explained there, we use strong mathematical induction on the *size* of the shortest derivation of the judgement $\langle C, s\rangle \Downarrow s'$.

Recall that $\langle C, s\rangle \to^* \langle \texttt{skip}, s'\rangle$ is a shorthand notation for *there is some natural number k such that* $\langle C, s\rangle \to^k \langle \texttt{skip}, s'\rangle$. So to proceed with the proof let us take this to be the property in which we are interested. Let $P(C, s, s')$ denote the property:

> *there is some natural number k such that* $\langle C, s\rangle \to^k \langle \texttt{skip}, s'\rangle$.

We have to show $\vdash_{big} \langle C, s\rangle \Downarrow s'$ implies $P(C, s, s')$, which we do by rule induction. So let the inductive hypothesis (IH) be:

> $\vdash_{big} \langle D, s_D\rangle \Downarrow s'_D$ implies $P(D, s_D, s'_D)$ whenever the judgement $\langle D, s_D\rangle \Downarrow s'_D$ has a derivation whose size is strictly smaller than the shortest derivation of the judgement $\langle C, s\rangle \Downarrow s'$.

We have to show that from the hypothesis (IH) we can derive $\vdash_{big} \langle C, s\rangle \Downarrow s'$ implies $P(C, s, s')$.

So suppose $\vdash_{big} \langle C, s\rangle \Downarrow s'$, and let us look at the shortest derivation of the judgement $\langle C, s\rangle \Downarrow s'$. There are lots of possibilities for the form of this derivation. To consider them all let us do a case analysis on the structure of $C$. As we know there are five possibilities; we examine a few.

Suppose $C$ is $\boxed{\texttt{skip}}$. Then $P(C, s, s')$ is trivially true, since $\langle \texttt{skip}, s\rangle \to^0 \langle \texttt{skip}, s\rangle$.

Suppose $C$ is the assignment command $\boxed{\mathrm{L} := E}$. Since $\vdash_{big} \langle C, s\rangle \Downarrow s'$ we know that the state $s'$ must be $s[\mathrm{L} \mapsto n]$, where $n$ is the unique number such that $\langle E, s\rangle \Downarrow \mathtt{n}$; we know this is unique from Proposition 12. Then it is easy to use the rule (S-ASS) from Figure 3.6 to show that $\langle C, s\rangle \to^1 \langle \texttt{skip}, s'\rangle$.

Next suppose that $C$ has the structure $\boxed{C_1 ; C_2}$. Then the structure of the derivation of the judgement $\langle C, s\rangle \Downarrow s'$ must be of the form

$$\dfrac{\dfrac{\cdots}{\langle C_1, s\rangle \Downarrow s_1}\;(\textsc{b-?}) \qquad \dfrac{\cdots}{\langle C_2, s_1\rangle \Downarrow s'}\;(\textsc{b-?})}{\langle C_1 ; C_2, s\rangle \Downarrow s'}\;(\textsc{b-seq}) \tag{3.10}$$

for some state $s'$. From this we know that the judgement $\langle C_1, s\rangle \Downarrow s_1$ has a derivation; more importantly the size of this derivation is strictly smaller than the derivation of $\langle C, s\rangle$ we are considering. So the inductive hypothesis kicks in, and we can assume $P(C_1, s, s_1)$ is true; in other words there is some $k_1$ such that $\langle C_1, s\rangle \to^{k_1} \langle \texttt{skip}, s_1\rangle$.

What can we do with this? Well it turns out that this implies $\langle C_1 ; C_2, s\rangle \to^k \langle \texttt{skip} ; C_2, s_1\rangle$; this is posed as Exercise 16 below. So tagging on one application of the rule (S-SEQ.SKIP) we have $\langle C_1 ; C_2, s\rangle \to^{(k_1+1)} \langle C_2, s_1\rangle$.

We have not quite evaluated $\langle C_1 ; C_2, s\rangle$ to completion using the small step semantics but we are getting there; we can now concentrate on running $\langle C_2, s_1\rangle$. Re-examining the proof tree (3.10) above we see that the judgement $\langle C_2, s_1\rangle \Downarrow s'$ also has a derivation, and because of its size (IH) can again be applied, to obtain $P(C_2, s_1, s')$. So we know there is some $k_2$ such that $\langle C_1, s_1\rangle \to^{k_2} \langle \texttt{skip}, s'\rangle$.

We can now put these two sequences of steps together to obtain the required $\langle C_1 ; C_2, s\rangle \to^{k_1+k_2+1} \langle \texttt{skip}, s'\rangle$.

An even more complicated possibility is that $C$ has the form $\boxed{\texttt{while}\ B\ \texttt{do}\ D}$ for some Boolean expression $B$ and command $D$. Here we first concentrate on $B$. Proposition 12, formulated for Booleans means that there is exactly one Boolean value $\texttt{bv}$ such that $\langle B, s\rangle \Downarrow \texttt{bv}$ can be derived. Suppose this is the value $\texttt{false}$. Then the required $\langle C, s\rangle \to^1 \langle \texttt{skip}, s\rangle$ is readily shown, using an application of (s-while.f). The interesting case is when this is the value $\texttt{true}$.

In this case the structure of the derivation of the judgement $\langle C, s\rangle$ must take the form

$$\dfrac{\dfrac{\cdots}{\langle B, s\rangle \Downarrow \texttt{true}}\,(\text{B-?}) \qquad \dfrac{\cdots}{\langle D, s\rangle \Downarrow s_1}\,(\text{B-?}) \qquad \dfrac{\cdots}{\langle \texttt{while}\ B\ \texttt{do}\ D, s_1\rangle \Downarrow s'}\,(\text{B-?})}{\langle \texttt{while}\ B\ \texttt{do}\ D, s\rangle \Downarrow s'}\,(\text{B-WHILE.T}) \tag{3.11}$$

for some state $s_1$. This contains a lot of information. Specifically we know:

(a) The judgement $\langle D, s\rangle \Downarrow s_1$ has a derivation. Moreover its size is strictly less than that of the derivation of $\langle C, s\rangle \Downarrow s'$, and therefore we can apply (IH) above to obtain $P(D, s, s_1)$. That is $\langle D, s\rangle \to^{k_1} \langle \texttt{skip}, s_1\rangle$ for some $k_1$.

(b) The judgement $\langle \texttt{while}\ B\ \texttt{do}\ D, s_1\rangle \Downarrow s'$ also has a judgement, to which (IH) also applies. So we know $\langle \texttt{while}\ B\ \texttt{do}\ D, s_1\rangle \to^{k_2} \langle \texttt{skip}, s\rangle$ for some $k_2$.[1]

We can now combine these two sequences, again using Exercise 16 below, to obtain the required $\langle \texttt{while}\ B\ \texttt{do}\ D, s\rangle \to^k \langle \texttt{skip}, s'\rangle$ for $k = k_1 + k_2 + 2$:

$$\begin{aligned}
\langle \texttt{while}\ B\ \texttt{do}\ D, s\rangle &\to \langle D ; \texttt{while}\ B\ \texttt{do}\ D, s\rangle \\
&\to^{k_1} \langle \texttt{skip} ; \texttt{while}\ B\ \texttt{do}\ D, s_1\rangle \\
&\to \langle \texttt{while}\ B\ \texttt{do}\ D, s_1\rangle \\
&\to^{k_2} \langle \texttt{skip}, s_1\rangle
\end{aligned}$$

There is one more possibility for the structure of $C$, namely $\boxed{\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2}$. We leave this to the reader. □

**Exercise 16** *Use mathematical induction to show that* $\langle C_1, s\rangle \to^k \langle C_1', s'\rangle$ *implies* $\langle C_1 ; C_2, s\rangle \to^k \langle C_1' ; C_2, s'\rangle$. □

This theorem shows that the result of running a command using the big-step semantics can also be obtained using the small-step semantics. We now show that the converse is also true. But the proof is more indirect, via an auxiliary result.

[1]This is where rule induction is essential. With structural induction we would not be able to make this step in the proof.

**Proposition 17** *Suppose* $\vdash_{sm} \langle C, s\rangle \rightarrow \langle C', s'\rangle$. *Then* $\vdash_{big} \langle C', s'\rangle \Downarrow s_t$ *implies* $\vdash_{big} \langle C, s\rangle \Downarrow s_t$.

**Proof:** Similar in style to that of Lemma 5 of Chapter 2.2.1; the proof is by structural induction on $C$. Let $P(C)$ denote the property:

> If $\vdash_{sm} \langle C, s\rangle \rightarrow \langle C', s'\rangle$, then $\vdash_{big} \langle C', s'\rangle \Downarrow s_t$ implies $\vdash_{big} \langle C, s\rangle \Downarrow s_t$.

We are going to prove $P(C)$ for every command $C$. So suppose $\vdash_{sm} \langle C, s\rangle \rightarrow \langle C', s'\rangle$ and $\vdash_{big} \langle C', s'\rangle \Downarrow s_t$. From the definition of the language, in Figure 3.1, we know that there are five possibilities for $C$. But here we look at only one case, the most interesting one, when $C$ has the form `while` $B$ `do` $D$.

In this case the argument depends on the unique Boolean value `bv` such that $\vdash_{big} B \Downarrow$ `bv`. The easy case is when this is `false`. Here the small-step derivation can only use the rule (s-while.t), and therefore takes the form $\langle C, s\rangle \rightarrow \langle \texttt{skip}, s\rangle$. In other words $\langle C', s'\rangle$ must be $\langle \texttt{skip}, s\rangle$. So the big-step judgement $\langle \texttt{skip}, s\rangle \Downarrow s_t$ can only be infered using the rule (b-skip) from Figure 3.3, and so the state $s_t$ must be $s$. But the required $\langle C, s\rangle \Downarrow s_t$ now follows by an application of (b-while.f).

Now suppose $\vdash_{big} \langle B, s\rangle \Downarrow \texttt{true}$. Then the judgement $\langle C, s\rangle \rightarrow \langle C', s'\rangle$ must look like $\langle \texttt{while}\ B\ \texttt{do}\ D, s\rangle \rightarrow \langle D;\texttt{while}\ B\ \texttt{do}\ D, s\rangle$, that is $\langle C', s'\rangle$ must be $\langle D;\texttt{while}\ B\ \texttt{do}\ D, s\rangle$.

Let us know look at the derivation of the big-step judgement $\langle D;\texttt{while}\ B\ \texttt{do}\ D, s\rangle \Downarrow s_t$. This must be constructed using an application of the rule (b-seq), and so we must have a derivation of

(a) $\langle D, s\rangle \Downarrow s_1$

(b) and $\langle \texttt{while}\ B\ \texttt{do}\ D, s_1\rangle \Downarrow s_t$

for some intermediate state $s_1$. But now, because we are assuming $\vdash_{big} \langle B, s\rangle \Downarrow \texttt{true}$, an application of the big-step rule (b-while.t) appended to the derivations of (a) and (b), will give the required derivation of the judgement $\langle \texttt{while}\ B\ \texttt{do}\ D, s\rangle \Downarrow s_t$.

**Exercise 17** *Fill in the remaining four cases in the proof of the previous proposition.*
□

**Theorem 18** $\vdash_{sm} \langle C, s\rangle \rightarrow^* \langle \texttt{skip}, s_t\rangle$ *implies* $\vdash_{big} \langle C, s\rangle \Downarrow s_t$. □

**Proof:** The previous proposition can be generalised to:

> For any natural number $k \geq 0$, $\langle C, s\rangle \rightarrow^k \langle C', s'\rangle$ and $\vdash_{big} \langle C', s'\rangle \Downarrow s_t$ implies $\vdash_{big} \langle C, s\rangle \Downarrow s_t$.

The proof is a straightforward argument by mathematical induction on $k$.

Now suppose $\langle C, s\rangle \rightarrow^* \langle \texttt{skip}, s'\rangle$. Recall that this means there is some natural number $k$ such that $\langle C, s\rangle \rightarrow^k \langle \texttt{skip}, s_t\rangle$. But we also have a trivial derivation to show $\vdash_{big} \langle \texttt{skip}, s_t\rangle \Downarrow s_t$. The required result, $\vdash_{big} \langle C, s\rangle \Downarrow s'$ , now follows trivially from the above generalisation.

**Summing up:** What have we achieved? First we have given two different semantics to a simple language *Com* of imperative commands, a big-step one and a small-step one. Moreover we have shown, in Theorem 16 and Theorem 18, that they coincide on the behaviour they prescribe for commands. Specifically the following statements are equivalent:

- $\vdash_{big} \langle C, s\rangle \Downarrow s_t$
- $\langle C, s\rangle \rightarrow^* \langle \texttt{skip}, s_t\rangle$.

Moreover we have shown that the small-step semantics is *consistent* in the sense of Determinacy, Theorem 15: for every configuration $\langle C, s\rangle$ there is at most one terminal state $s_t$ such that $\langle C, s\rangle \rightarrow^* \langle \texttt{skip}, s_t\rangle$. Incidently the equivalence above also ensures that the big-step semantics is also consistent in this sense.

On page 40 we explained our intuitive understanding of commands, as transformations over states of a computer memory. A command starts from an initial state, makes a sequence of updates to the memory, and ending up eventually with the memory in a terminal state, hopefully. We can now formally describe this transformation, using either of the semantic frameworks.

We use $(\mathit{States} \rightharpoonup \mathit{States})$ to denote the set of *partial* functions from *States* to *States*; we need to consider partial functions rather than total functions because as we have seen commands do not necessarily terminate. Then for every command $C$ in the language *While*, we define the partial function $[\![C]\!]$ over states as follows:

$$[\![C]\!](s) = \begin{cases} s_t, & \text{if } \vdash_{big} \langle C, s\rangle \Downarrow s_t \\ \text{undefined}, & \text{otherwise} \end{cases}$$

Note that this is well-defined; as we have seen for every initial state $s$ there is at most one terminal state $s_t$ such that $\langle C, s\rangle \Downarrow s_t$. Thus this meaning function has the following type:

$$[\![-]\!] : \mathit{Com} \rightarrow (\mathit{States} \rightharpoonup \mathit{States})$$

For example $[\![LP]\!]$, given in (3.1) above, is the partial function which is only defined for states $s$ satisfying $s(\textsc{l}) = \mathbb{0}$. If $s$ is such a state then $[\![LP]\!](s) = s$. In other words $[\![LP]\!]$ is a partial identity function, whose domain is the set of states $s$ such that $s(\textsc{l}) = \mathbb{0}$. On the other hand $[\![LP1]\!]$, where $LP1$ is the command defined on page 47, is the totally undefined function; it has the empty domain.

**Exercise 18** *Describe, using standard mathematical notation, the partial function $[\![C_1]\!]$, where $C_1$ is the command given on page 38.* □

## 3.4 Extensions to the language *While*

In this section we examine various extensions to the basic imperative language *While*, exploring how both big-step and small-step semantic rules can be used to capture the intended behaviour of the added constructs.

$$
\begin{array}{rcl}
B \in \mathit{Bool} & ::= & \ldots \\
\\
E \in \mathit{Arith} & ::= & \ldots \\
C \in \mathit{Com} & ::= & \textsc{l} := E \mid \texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C \\
& \mid & C\,;C \mid \texttt{skip} \mid \texttt{while}\ B\ \texttt{do}\ C \\
& \mid & \texttt{begin}\ [D]\ C\ \texttt{end} \\
\\
D \in \mathit{Dec} & ::= & \texttt{loc}\ \textsc{l} := E
\end{array}
$$

Figure 3.7: The language $\mathit{While}^{block}$, an extension to *While*

### 3.4.1 Local declarations

Many languages allow you to collect code into separate *blocks*, which may contain internal declarations, or local parameters; for example think of methods in Java. Here we examine a simple instance of this general programming construct.

The intuitive idea behind the command

$$\texttt{begin}\ [\texttt{loc}\ \textsc{l} := E]\ C\ \texttt{end}$$

is that

- the location L is *local* to the execution of the command $C$
- the initial value of L for this local execution is obtained from the value of the expression $E$.

Let $C_1$ be the command

$$\textsc{l} := 1\,;\texttt{begin}\ [\texttt{loc}\ \textsc{l} := 2]\ \textsc{k} := \textsc{l}\ \texttt{end}$$

Then in a big-step semantics we would expect, for every state $s$,

$$\langle C_1, s\rangle \Downarrow s_t$$

for some $s_t$. In fact because of the particular command $C_1$ the final values stored in $s_t(\textsc{l})$, $s_t(\textsc{k})$ do not actually depend on the initial state $s$. But we would expect

(i) $s_t(\textsc{k}) = 2$

(ii) $s_t(\textsc{l}) = 1$

The first expectation, (i), is because the local execution of the command $\textsc{k} := \textsc{l}$ is relative to the local declaration $\texttt{loc}\ \textsc{l} := 2$. The second (ii) is because we expect the

$$\frac{\begin{array}{l}\text{(B-BLOCK)}\\ \langle E, s\rangle \Downarrow v\\ \langle C, s[\text{L} \mapsto v]\rangle \Downarrow s'\end{array}}{\langle \texttt{begin [loc}\ \text{L} := E\texttt{]}\ C\ \texttt{end}, s\rangle \Downarrow s'[\text{L} \mapsto s(\text{L})]}$$

Figure 3.8: Big-step rule for blocks

original value of L to be restated when the local execution is finished. This is important for executing commands such as $C_2$:

$$\text{L}_1 := 1\,;\text{L}_2 := 2\,;\texttt{begin [loc}\ \text{L}_1 := 7\texttt{]}\ \text{L}_2 := \text{L}_1\ \texttt{end}\,;\text{K} := (\text{L}_1 + \text{L}_2)$$

Here we would expect the judgement

$$\langle C_2, s\rangle \Downarrow s'_t$$

where $s'_t(\text{K})$ is 8 rather than 14. This is because, intuitively when the block terminates we expect the value stored in the location $\text{L}_2$ to be restored to that which it contained prior to the block executing.

A big-step semantic rule for blocks, (B-BLOCK), is given in Figure 3.8. It says that the only judgements to be made for block commands take the form

$$\langle \texttt{begin [loc}\ \text{L} := E\texttt{]}\ C\ \texttt{end}, s\rangle \Downarrow s_t$$

where the terminal state $s_t$ has the form $s'[\text{L} \mapsto s(\text{L})]$; in other words the value associated with L in the terminal state $s_t$ is exactly the same as in the initial state $s$. Moreover to calculate the terminal state $s_t$ itself we must:

(i) First evaluate the expression $E$ in the initial state, $\langle E, s\rangle \Downarrow v$.

(ii) Then execute the local body $C$ in the initial state $s$ modified so that the value $v$ is associated with the identifier L, $\langle C, s[\text{L} \mapsto v]\rangle \Downarrow s'$.

(iii) The starting value associated with L, namely $s(\text{L})$, is reinstated in the final state, $s_t = s'[\text{L} \mapsto s(\text{L})]$.

**Exercise 19**

*(1) Use this new rule, together with the existing ones for* While*, to find a state $s_t$ such that $\langle C_1, s\rangle \Downarrow s_t$ where the command $C_1$ is given above. Justify your answer by giving a formal derivation using the inference rules.*

*(2) Do the same for the command $C_2$ also given above.*

$$\begin{array}{rcl} B \in \mathit{Bool} & ::= & \ldots \\ E \in \mathit{Arith} & ::= & (E - E) \mid \ldots \\ C \in \mathit{Com} & ::= & \textsc{l} := E \mid \texttt{if } B \texttt{ then } C \texttt{ else } C \\ & \mid & C \,;\, C \mid \texttt{skip} \mid \texttt{while } B \texttt{ do } C \\ & \mid & \texttt{abort} \end{array}$$

Figure 3.9: Another extension to *While*, called $\mathit{While}^{abort}$

*(3) Consider the following command $C_3$:*

```
K := 3 ; L := 2 ; begin loc K := 1;
                         L := K;
                         begin [loc L := 2] K := L + K end
                         L := K + L ; M := L + 1
                   end
```

*What are the final values of the identifiers* L *and* K *after $C_3$ has been executed? In other words if*

$$\langle C, s\rangle \Downarrow s_t$$

*what are the numerals $s_t(\textsc{l})$, $s_t(\textsc{k})$ and $s_t(\textsc{m})$ ?*

*(4) Design a small-step semantics for this extension to* $\mathit{While}^{block}$.
*Note: This is not easy as it requires inventing new notation for changing and reinstating states.* □

### 3.4.2 Aborting computations

Another extension to *While* is given in Figure 3.9. There are two additions. To Booleans we have added the extra construct $(E_1 - E_2)$. The idea here is that this can lead to problems if the value of $E_2$ is greater than that of $E_1$, since the only arithmetic values in the language are the non-negative numerals. In this case the execution in which this evaluation is being carried out should be *aborted*. In order to emphasise this idea of *aborting* an execution we have also added an extra command `abort` to the language. An attempt to execute this command will also lead to an immediate abortion of the execution.

This extended language contains all of the constructs of the original language *While* and we would not expect our extended rules to change in any way the semantics of these commands, that is any commands which do not use `abort` or the troublesome subtraction operator $(E_1 - E_2)$. But in order to see intuitively what problems can arise consider the following commands:

$$
\begin{array}{ll}
C_1: & \textsc{l} := 1\, ;\, \texttt{abort}\, ;\, \textsc{l} := 2 \\
C_2: & \textsc{l} := 1\, ;\, \texttt{if}\, (\textsc{l} - 7) \leq 4\ \texttt{then}\ \textsc{l} := 4\ \texttt{else}\ \textsc{l} := 3 \\
C_3: & \textsc{l} := 3\, ;\, \texttt{while}\ \textsc{l} > 0\ \texttt{do} \\
 & \qquad\qquad (\,\textsc{l} := (\textsc{l} - 1); \\
 & \qquad\qquad \texttt{abort}; \\
 & \qquad\qquad \textsc{l} := (\textsc{l} - 1)\,) \\
C_4: & \textsc{l} := 3\, ;\, \texttt{while}\ \textsc{l} > 0\ \texttt{do} \\
 & \qquad\qquad \texttt{if}\, (\textsc{l} - 2) > 1\ \texttt{then}\ \textsc{l} := 0\ \texttt{else}\ \textsc{l} := (\textsc{l} - 1)
\end{array}
$$

No matter what initial state $s$ we use we would expect the execution of all of these programs to be aborted. Of course in each case some sub-commands will have been executed and so the state $s$ may have been changed. For example running $C_1$ will result in an aborted computation in which the resulting state $s_t$ satisfies $s_t(\textsc{l}) = 1$.

To differentiate between successful computations and unsuccessful ones we design two judgements

$$\langle C, s\rangle \Downarrow \langle \texttt{skip}, s'\rangle \qquad\qquad \langle C, s\rangle \Downarrow \langle \texttt{abort}, s'\rangle$$

The first says that running $C$ with initial state $s$ leads to a successful (terminating) computation with final state $s'$. For commands $C$ from the base language *While* these judgements should be the same as $\langle C, s\rangle \Downarrow s'$, whose inference inference rules are given in Figure 3.3. This use of `skip` to indicate successful termination is similar to how it is used in the small-step semantics from Figure 3.6 . The second form above says says that running $C$ with state $s$ leads to an unsuccessful or aborted computation, in which the state has changed from $s$ to $s'$.

Of course evaluating arithmetic or Boolean expressions can also be unsuccessful and so we have to amend their big-step semantics as well. Judgements for these will now take the form

(i) $\langle E, s\rangle \Downarrow \texttt{n}$ successful evaluation of $E$ to value `n`

(ii) $\langle E, s\rangle \Downarrow \texttt{abort}$ unsuccessful attempt at evaluating $E$

(iii) $\langle B, s\rangle \Downarrow \texttt{bv}$ successful evaluation of $B$ to the Boolean value `bv`

(iv) $\langle B, s\rangle \Downarrow \texttt{abort}$ unsuccessful attempt at evaluating $B$

The inference rules for arithmetic expressions are given in Figure 3.10, although we have omitted the repetition of the rules (B-NUM), (B-LOC) and (B-ADD) from Figure 3.2. The first two rules (B-MINUS) and (B-MINUS.ABORT) are straightforward; they implement our intuition of what should happen when a minus operation is performed. But the propagation rules (B-PROP.L) and (B-PROP.R) are also important as they allow us to infer judgements such as

$$(3 - 7) + (4 + 1) \Downarrow \texttt{abort} \qquad \text{and} \quad (2 + 3) - (2 - 6) \Downarrow \texttt{abort}$$

The rules use the meta-variable `op` to stand for either of the operators + or −.

(B-MINUS)
$$\frac{\langle E_1, s\rangle \Downarrow \mathtt{n}_1 \qquad \langle E_2, s\rangle \Downarrow \mathtt{n}_2}{\langle E_1 - E_2, s\rangle \Downarrow \mathtt{n}_3} \quad n_3 = \mathsf{minus}(n_1, n_2),\ n_1 \geq n_3$$

(B-MINUS.ABORT)
$$\frac{\langle E_1, s\rangle \Downarrow \mathtt{n}_1 \qquad \langle E_2, s\rangle \Downarrow \mathtt{n}_2}{\langle E_1 - E_2, s\rangle \Downarrow \mathtt{abort}} \quad n_1 < n_2$$

(B-PROP.L)
$$\frac{\langle E_1, s\rangle \Downarrow \mathtt{abort}}{\langle E_1 \;\mathsf{op}\; E_2, s\rangle \Downarrow \mathtt{abort}}$$

(B-PROP.R)
$$\frac{\langle E_2, s\rangle \Downarrow \mathtt{abort}}{\langle E_1 \;\mathsf{op}\; E_2, s\rangle \Downarrow \mathtt{abort}}$$

Figure 3.10: Extra rules for arithmetic expressions in *While*$^{abort}$

**Exercise 20** *Give the inference rules for the judgements for Boolean expressions of While*$^{abort}$ *in (iii) and (iv) above.* □

The rules for commands are given in Figure 3.11. The execution of the one-instruction command (L := $E$), in the rules (B-ASSIGN.S) and (B-ASSIGN.F), depends on whether the evaluation of $E$ is successful; note that in the latter case the state remains unchanged. To execute $C_1$ ; $C_2$ we first evaluate $C_1$. If this is successful, with terminal state $s_1$, we continue with the execution of $C_2$ with $s_1$ as an initial state; this may or may not abort, and to cover both possibilities in rule (B-SEQ.R) we use the meta-variable `r` to range over both `skip` and `abort`. If on the other hand the attempted execution of $C_1$ is unsuccessful then the rule (B-SEQ.F) allows us to conclude that the execution of ($C_1$ ; $C_2$) is also unsuccessful. The rules for executing (`if` $B$ `then` $C_1$ `else` $C_2$ are adapted in a similar manner from those in Figure 3.3, with a new rule for when the evaluation of the Boolean $B$ is unsuccessful.

Finally, for the command (`while` $B$ `do` $C$) we could also have adapted the rules (B-WHILE.T) and (B-WHILE.F) from Figure 3.3. Instead, for the sake of variety we use the rule (B-WHILE.UN), an *unwinding rule*. It says that the result of executing (`while` $B$ `do` $C$) is exactly the same as the execution of the command `if` $B$ `then` (`while` $B$ `do` $C$) `else skip`.

**Exercise 21** *Use the inference in Figure 3.3 to execute the four commands $C_i$ given on page 63 relative to an arbitrary initial state s; the behaviour should not actually depend on the values stored in s.*

## 3.4.3 Adding parallelism

In the new language *While*$^{par}$ the idea of the new construct ($C_1$ `par` $C_2$) is that, intuitively, the individual commands $C_1$ and $C_2$ be executed in parallel, with no particular preference being given to one or the other; this means that their executions are to be interleaved, which will lead to non-deterministic behaviour. For example consider the

(B-SKIP)
$$\frac{}{\langle \texttt{skip}, s\rangle \Downarrow \langle \texttt{skip}, s\rangle}$$

(B-ABORT)
$$\frac{}{\langle \texttt{abort}, s\rangle \Downarrow \langle \texttt{abort}, s\rangle}$$

(B-ASSIGN.S)
$$\frac{\langle E, s\rangle \Downarrow \texttt{n}}{\langle \textsc{l} := E, s\rangle \Downarrow \langle \texttt{skip}, s[\textsc{l} \mapsto \texttt{n}]\rangle}$$

(B-ASSIGN.A)
$$\frac{\langle E, s\rangle \Downarrow \texttt{abort}}{\langle \textsc{l} := E, s\rangle \Downarrow \langle \texttt{abort}, s\rangle}$$

(B-SEQ.S)
$$\frac{\langle C_1, s\rangle \Downarrow \langle \texttt{skip}, s_1\rangle \quad \langle C_2, s_1\rangle \Downarrow \langle \texttt{r}, s'\rangle}{\langle C_1 \,;\, C_2, s\rangle \Downarrow \langle \texttt{r}, s'\rangle}$$

(B-SEQ.A)
$$\frac{\langle C_1, s\rangle \Downarrow \langle \texttt{abort}, s'\rangle}{\langle C_1 \,;\, C_2, s\rangle \Downarrow \langle \texttt{abort}, s'\rangle}$$

(B-IF.T)
$$\frac{\langle B, s\rangle \Downarrow \texttt{true} \quad \langle C_1, s\rangle \Downarrow \langle \texttt{r}, s'\rangle}{\langle \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2, s\rangle \Downarrow \langle \texttt{r}, s'\rangle}$$

(B-IF.A)
$$\frac{\langle B, s\rangle \Downarrow \texttt{abort}}{\langle \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2, s\rangle \Downarrow \langle \texttt{abort}, s\rangle}$$

(B-IF.F)
$$\frac{\langle B, s\rangle \Downarrow \texttt{false} \quad \langle C_2, s\rangle \Downarrow \langle \texttt{r}, s'\rangle}{\langle \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2, s\rangle \Downarrow \langle \texttt{r}, s'\rangle}$$

(B-WHILE.UN)
$$\frac{\langle \texttt{if}\ B\ \texttt{then}\ (C \,;\, \texttt{while}\ B\ \texttt{do}\ C)\ \texttt{else}\ \texttt{skip}, s\rangle \Downarrow \langle \texttt{r}, s'\rangle}{\langle \texttt{while}\ B\ \texttt{do}\ C, s\rangle \Downarrow \langle \texttt{r}, s'\rangle}$$

Figure 3.11: Big-step inference rules for commands in *While*$^{abort}$

command $C$:

$$\textsc{l} := \texttt{0}\ \texttt{par}\ (\textsc{l} := 1 \,;\, \textsc{l} := \textsc{l} + 1) \tag{3.12}$$

Then the single assignment $\textsc{l} := \texttt{0}$ can be executed

- before the compound command $\textsc{l} := 1 \,;\, \textsc{l} := \textsc{l} + 1$ is executed
- after it has been executed
- or in between the execution of the sub-commands $\textsc{l} := 1$ and $\textsc{l} := \textsc{l} + 1$ .

$$\begin{array}{rcl}
B \in \mathit{Bool} & ::= & \ldots \\
E \in \mathit{Arith} & ::= & \ldots \\
C \in \mathit{Com} & ::= & \textsc{l} := E \mid \texttt{if}\, B\, \texttt{then}\, C\, \texttt{else}\, C \\
& \mid & C \,;\, C \mid \texttt{skip} \mid \texttt{while}\, B\, \texttt{do}\, C \\
& \mid & C\, \texttt{par}\, C
\end{array}$$

Figure 3.12: *While*$^{par}$: adding parallelism to *While*

So when the command (3.12) has terminated the final value associated with the location L can either be `2`, `0` or `1`.

Because of this interleaving of operations it would be very difficult to give a big-step semantics for the language *While*$^{par}$. The problem is exemplified by the same command (3.12). Using the existing big-step semantics for *While* we know

$$\langle \textsc{l} := \mathtt{0}, s\rangle \Downarrow s[\textsc{l} \mapsto \mathtt{0}] \qquad \langle \textsc{l} := \mathtt{1} \,;\, \textsc{l} := \textsc{l} + \mathtt{1}, s\rangle \Downarrow s[\textsc{l} \mapsto \mathtt{2}]$$

But how can we use these two judgements to deduce that when $C$ is executed that the identifier L might have the value `1` associated with it?

Instead we show how the small-step semantics of *While* can be adapted for *While*$^{par}$. Rules for the new construct are given in Figure 3.13. The first two, (S-LPAR), (S-RPAR), say that the next step in the execution of $C_1$ `par` $C_2$ can be either a step from $C_1$ or a step from $C_2$, while the second pair of rules handle the termination of either sub-command; recall we use the configuration $\langle \texttt{skip}, s\rangle$ to indicate an execution which has terminated.

**Exercise 22** *Use the rules in Figure 3.13, together with those in Figure 3.6 to find all states $s'$ such that $\langle C, s\rangle \rightarrow^* \langle \texttt{skip}, s'\rangle$, where C is given in (3.12) above. This should not depend on the initial state s.*

**Exercise 23** *Do the same for the command $C_2$:*

$$(\textsc{l} := \textsc{k} + \mathtt{1})\ \texttt{par}\ (\textsc{k} := \textsc{l} + \mathtt{1} \,;\, \textsc{k} := \textsc{k} + \textsc{l})$$

*relative to an initial state s satisfying* $s(\textsc{l}) = s(\textsc{k}) = 0$.

In *While*$^{par}$ communication between parallel commands is via the state; information passes between concurrent commands by allowing them to share variables or identifiers. Within such a framework it is very difficult to limit interference between commands and many real programming languages have constructs for alleviating this problem; constructs such as *semaphores*, *locks*, *critical regions*, etc.. Here we briefly examine one such construct, **Conditional critical regions**.

$$\text{(S-LPAR)}\ \frac{\langle C_1, s\rangle \to \langle C_1', s'\rangle}{\langle C_1\ \texttt{par}\ C_2, s\rangle \to \langle C_1'\ \texttt{par}\ C_2, s'\rangle} \qquad \text{(S-RPAR)}\ \frac{\langle C_2, s\rangle \to \langle C_2', s'\rangle}{\langle C_1\ \texttt{par}\ C_2, s\rangle \to \langle C_1\ \texttt{par}\ C_2', s'\rangle}$$

$$\text{(S-LPARS)}\ \frac{}{\langle \texttt{skip}\ \texttt{par}\ C, s\rangle \to \langle C, s\rangle} \qquad \text{(S-RPARS)}\ \frac{}{\langle C\ \texttt{par}\ \texttt{skip}, s\rangle \to \langle C, s\rangle}$$

Figure 3.13: Rules for parallelism

We add to *While<sup>par</sup>* the construct

$$\texttt{await}\ B\ \texttt{protect}\ C\ \texttt{end}$$

The intuition is that this command can only be executed when the Boolean $B$ is true, and then the entire command $C$ is to be executed to completion without interruption or interference. For example consider the command $D_1$:

$$x := \texttt{0}\ \texttt{par}\ \texttt{await}\ x = \texttt{0}\ \texttt{protect}\ x := \texttt{1}\ ;\ x := x + \texttt{1}\ \texttt{end} \tag{3.13}$$

This should be a deterministic program; if it is executed relative to a state $s$ then it will terminate and the only possible terminal state is $s[x \mapsto \texttt{2}]$.

As another example consider $D_2$ defined by

$$\begin{array}{l} \quad(\texttt{await}\ \texttt{true}\ \texttt{protect}\ \textsc{l} := \texttt{1}\ ;\ \textsc{l} := \textsc{k} + \texttt{1}\ \texttt{end}) \\ \texttt{par} \\ \quad(\texttt{await}\ \texttt{true}\ \texttt{protect}\ \textsc{k} := \texttt{2}\ ;\ \textsc{k} := \textsc{l} + \texttt{1}\ \texttt{end}) \end{array} \tag{3.14}$$

Here the two Boolean guards, `true`, are vacuous, so which protected command is executed first is chosen non-deterministically. But they are executed in isolation, without interference from each other. For example if executed with an initial state $s$ satisfying $s(\textsc{l}) = s(\textsc{k}) = \texttt{0}$ then there are only two possible terminal states; the first has $\textsc{l}, \textsc{k}$ containing $\texttt{1}, \texttt{3}$ respectively, while the second has $\texttt{2}, \texttt{1}$.

There is a bit of a trick in the required rule for this new command, as it uses the reflexive transitive closure of the small-step relation $\to$ in the hypothesis:

$$\text{(B-AWAIT)}\ \frac{\langle B, s\rangle \Downarrow \langle \texttt{tt}, s_1\rangle \qquad \langle C, s_1\rangle \to^* \langle \texttt{skip}, s'\rangle}{\langle \texttt{await}\ B\ \texttt{protect}\ C\ \texttt{end}, s\rangle \to \langle \texttt{skip}, s'\rangle}$$

**Exercise 24** *Use this rule, together with those from Figure 3.6, to give formal derivations confirming the expected behaviour of the two commands $D_1, D_2$ in (3.13) and (3.14) above.*

# Chapter 4

# A simple functional language

In this chapter we develop a very simple language based on function definitions, which may be considered as the beginnings of the core of a functional programming language such as ML or Haskell. The starting point is the language of arithmetic expressions *Exp* from Chapter 1, where we developed a big-step and a small-step semantics for it. Moreover Chapter 2.2.3 was devoted to proving properties of these semantic definitions. Here we gradually extend the language, in three distinct steps, until the core functional language is reached. For each intermediate language we give a big-step and a small-step semantics, and extend the proofs in Chapter 2.2.3 to establishing their properties.

In the first section we extend *Exp* with a construct for *local declarations*, a feature common to most programming languages. Proving properties of their big-step and small-step semantics provides us with some opportunities to use rule induction from Chapter 2.3; however we will also see that alternative forms of induction can also be used for this language.

We then add Boolean expressions to the language, Chapter 4.2, to obtain $Exp_B$. But unlike in the language *While* from Chapter 3, we do not have separate syntactic categories for arithmetics and Booleans. Instead arithmetic operators and Boolean operators may be arbitrarily applied to arguments. In the resulting language *run-time* errors may occur, in the sense that unlike *Exp* there are expressions whose evaluations get stuck. This is another common phenomenon in programming languages, and provides us with the opportunity to introduce another topic of interest, namely *typing*. In Chapter 4.3 we discuss this concepts in detail, explaining topics such as *typechecking*, *progress* and *preservation*, via the rather simple language $Exp_B$.

In the final section 4.4 we add user-defined functions, to obtain the core functional language which we call *Fpl*.

$$\begin{aligned} E \in \mathit{Exp}_{loc} \quad ::= \quad & x \in \mathit{Vars} \mid \mathtt{n} \in \mathit{Nums} \mid (E + E) \mid (E \times E) \\ & \mid \mathtt{let}\ x = E\ \mathtt{in}\ E \end{aligned}$$

Figure 4.1: Syntax of *let expressions*

## 4.1 Local declarations

Let us start by reconsidering the language of arithmetic expressions *Exp* from Chapter 1. Consider the evaluation of the expression

$$(\mathtt{1} + \mathtt{2}) \times ((\mathtt{1} + \mathtt{2})\ +\ \mathtt{4}) \tag{4.1}$$

By following the big-step or small-step semantics there is considerable danger that the sub-expression $(\mathtt{1} + \mathtt{2})$ is actually evaluated twice. In order to avoid this possibility let us extend the language by a new construct which allows the sharing of sub-expressions. The idea is to replace the expression (4.1) with

$$E_0 : \mathtt{let}\ x = (\mathtt{1} + \mathtt{2})\ \mathtt{in}\ x \times (x + \mathtt{4}) \tag{4.2}$$

More generally we introduce into the language the new form of expression

$$\mathtt{let}\ x = E_1\ \mathtt{in}\ E_2$$

called a *local declaration*. Here $E_2$ is referred to as the *body* of the expression while $x = E_1$ is called the *declaration*; the variable $x$ is declared (locally) to be the expression $E_1$. So in (4.2) above $x = (\mathtt{1} + \mathtt{2})$ is the *declaration* which is *local* to the *body*, which in this case is the expression expression $x \times (x + \mathtt{4})$. So when evaluating this body the variable $x$ stands for it's *declared* value, that is the value of $(\mathtt{1} + \mathtt{2})$. In other words to evaluate the expression (4.2):

(1) first evaluate the expression $(\mathtt{1} + \mathtt{2})$ in order to get the declared value of $x$
(2) then evaluate the body of the expression, with $x$ standing for its declared value.

The abstract syntax for the extended language $\mathit{Exp}_{loc}$ is given in Figure 4.1 and uses a set *Vars* of variables. The extra syntax appears to be quite simple, but in fact the introduction of variables into the language makes the situation unexpectedly complex.

The first problem is that many expressions do not now naturally evaluate to any value. Consider

$$E_1 : \mathtt{let}\ y = (\mathtt{2} + \mathtt{3})\ \mathtt{in}\ (y \times y + z \times z) \tag{4.3}$$

Here $y$ is *declared* to be standing for the value of the expression $(\mathtt{2} + \mathtt{3})$, and the body of $E_1$, namely $(y \times y + z \times z)$, is meant to be evaluated relative to this declaration. However the variable $z$ also occurring in the body has no corresponding declaration; so $E_1$ can not actually be evaluated. Nevertheless expressions such as $E_1$ are useful as they can be used in the construction of more expressions such as

$$E_2 : \texttt{let}\ z = (1 + 2)\ \texttt{in}\ E_1$$

Intuitively $E_2$ can be evaluated, to the value 34, as we end up evaluating the expression $y \times y + z \times z$ relative to the declaration that $z$ stands for the value of $(1 + 2)$ and $y$ for that of $(2 + 3)$, both of which can in turn be evaluated. However

$$E_3 : \texttt{let}\ z = (w + 3)\ \texttt{in}\ E_1$$

can not be evaluated; to evaluate $(y \times y + z \times z)$ relative to the declaration that $y$ stands for the value of $(2 + 3)$ and $z$ for that of $(w + 3)$ we need to know what the variable $w$ stands for.

The second problem is that in nested declarations variables may be used in multiple roles. To discuss this let us introduce some informal notation. In the expression $E_1$, or rather in the body of $E_1$ in (4.3) above, we say that the variable $y$ has a *bound* occurrence, as in the declaration it is bound to the expression $(2+3)$; on the other hand $z$ occurs *free* as, intuitively, it is not governed by any declaration. Consider for example

$$E_4 : \texttt{let}\ z = 2 + z\ \texttt{in}\ (z \times z \times y)$$

Here $y$ has a free occurrence and $z$ has two free occurrences in the body of the declaration; but also it has a free occurrence in the declared value $(2 + z)$. To evaluate $E_4$ we have to evaluate the body $(z \times z \times y)$, relative to some declaration of $y$, and the declaration that $z$ stands for the expression $(2 + z)$; but here in turn, in this expression we need to know what this occurrence of $z$ stands for.

A related problem is multiple declarations for the same variable.

$$E_5 : \texttt{let}\ x = 1\ \texttt{in}\ \texttt{let}\ x = (1 + 2)\ \texttt{in}\ (x \times (x + 4)) \tag{4.4}$$

Should this evaluate to 5 or 21?

The crucial concept, which we need to define formally, is the set of variables which occur free in an expression, that is which have an occurrence which is not governed by a declaration.

**Definition 19 (Free variables)** *The set of free variables in an expression E, written fv(E), is defined by structural induction on E, as follows:*

*(i)* $fv(x) = \{x\} \qquad fv(\mathtt{n}) = \{\}$
*(ii)* $fv(\texttt{let}\ y = E_1\ \texttt{in}\ E_2) = fv(E_1) \cup (fv(E_2) - \{y\})$
*(iii)* $fv(E_1\ \mathsf{op}\ E_2) = fv(E_1) \cup fv(E_2)$

*where* op *is either of the arithmetic operators,* $+, \times$. □

So for example $fv(E_1) = \{z\}$, $fv(E_3) = \{w\}, fv(E_4) = \{y, z\}$, while $fv(E_2) = \emptyset$. Intuitively if $x \in fv(E)$ then in order to evaluate $E$ we need, at least, to declare some value to associate to $x$. Consequently we can only evaluate expressions which have no free variables.

**Definition 20 (Programs)** *We define a* program *to be any term $E$ in the language $Exp_{loc}$ such that $fv(E) = \emptyset$.*

*To emphasise the difference between programs and expressions, we use $P, Q, \ldots$ to represent arbitrary programs.* □

Note that for an expression $E$ of the form $\mathtt{let}\ x = E_1\ \mathtt{in}\ E_2$ to be a program the sub-expression $E_1$ must be a program, since by definition $fv(E_1) \subseteq fv(E)$. But $E_2$ need not be a program; it is allowed to have $x$ as a free variable.

## 4.1.1 Big-step semantics

The big-step semantics of $Exp_{loc}$ take the form of judgements

$$P \Downarrow \mathtt{n}$$

where $P$ is a program in $Exp_{loc}$ and $\mathtt{n}$ is a numeral; as with the previous arithmetic expressions in Chapter 1 this is supposed to mean:

the evaluation of the program $P$ results in the value $\mathtt{n}$.

The inference rules for the judgements are in Figure 4.2, with the rules (B-NUM), (B-ADD), and the missing rule for multiplication, taken from the big-step semantics for *Exp*. The new rule for *let*-expressions (B-LET) states that in order to evaluate the program $\mathtt{let}\ x = P\ \mathtt{in}\ E$ we must

(1) first evaluate the program $P$ to a value say $\mathtt{m}$

(2) then evaluate the body $E$, with $x$ declared to be $\mathtt{m}$.

In fact rather than worrying about what it means to evaluate an expression relative to a declaration (2) is replaced by

(2) then evaluate the program which results from replacing the variable $x$ in the body $E$ by the value $\mathtt{m}$, that is evaluate the program $E\{\mathtt{m}/x\}$.

So for example $(\mathtt{let}\ x = (\mathtt{1} + \mathtt{2})\ \mathtt{in}\ (x \times (x + \mathtt{4}))) \Downarrow \mathtt{21}$ because $(\mathtt{1} + \mathtt{2}) \Downarrow \mathtt{3}$ and $(\mathtt{3} \times (\mathtt{3} + \mathtt{4})) \Downarrow \mathtt{21}$, that is $(x \times (x + \mathtt{4})\{\mathtt{3}/x\} \Downarrow \mathtt{21}$.

Unfortunately what it actually means to substitute a value $\mathtt{m}$ for a variable in an expression $E$ is not very straightforward. For example, refering to (4.2) above, $E_0\{\mathtt{1}/x\}$ should not result in $\mathtt{let}\ \mathtt{1} = (\mathtt{1}+\mathtt{2})\ \mathtt{in}\ (\mathtt{1}\times(\mathtt{1}+\mathtt{4}))$ for the simple reason that this is not a valid expression in the language. Nor should it result in $\mathtt{let}\ x = (\mathtt{1}+\mathtt{2})\ \mathtt{in}\ (\mathtt{1}\times(\mathtt{1}+\mathtt{4}))$. Intuitively the occurrence of $x$ in the body of $E_0$, $(x \times (x + \mathtt{4})$. refers to the value of the declared expression $(\mathtt{1} + \mathtt{2})$ in the declaration $E$, and so the substitution $\{\mathtt{1}/x\}$ should have no effect on the body of $E_0$. Incidentally one consequence of this decision is that the evaluation of $E_5$ in (4.4) above will result in the value $\mathtt{21}$; inner declarations will have precedence over outer ones.

It turns out that when we perform a substitution for a variable we should only physically substitute its free occurrences. The formal definition is as follows, where we allow substitution by arbitrary programs:

$$\frac{}{\mathtt{n} \Downarrow \mathtt{n}}\ \text{(B-NUM)} \qquad \frac{P_1 \Downarrow \mathtt{n}_1 \qquad P_2 \Downarrow \mathtt{n}_2}{(P_1 + P_2) \Downarrow \mathtt{n}_3}\ \text{(B-ADD)}\quad n_3 = \mathsf{add}(n_1, n_2)$$

$$\frac{\begin{array}{l} P \Downarrow \mathtt{m} \\ E\{^{\mathtt{m}}/_{x}\} \Downarrow \mathtt{n} \end{array}}{\mathtt{let}\ x = P\ \mathtt{in}\ E \Downarrow \mathtt{n}}\ \text{(B-LET)} \qquad \frac{P_1 \Downarrow \mathtt{n}_1 \qquad P_2 \Downarrow \mathtt{n}_2}{(P_1 \times P_2) \Downarrow \mathtt{n}_3}\ \text{(B-MULT)}\quad n_3 = \mathsf{mult}(n_1, n_2)$$

Figure 4.2: Big-step semantics of *let expressions*

$$\frac{\dfrac{}{\mathtt{1} \Downarrow \mathtt{1}}\ \text{(B-NUM)} \qquad \dfrac{\dfrac{\dfrac{}{\mathtt{2} \Downarrow \mathtt{2}}\ \text{(B-NUM)} \quad \dfrac{}{\mathtt{1} \Downarrow \mathtt{1}}\ \text{(B-NUM)}}{(\mathtt{2} + z)\{^{\mathtt{1}}/_{z}\} \Downarrow \mathtt{3}}\ \text{(B-ADD)} \qquad \dfrac{\dfrac{}{\mathtt{3} \Downarrow \mathtt{3}}\ \text{(B-NUM)} \quad \dfrac{}{\mathtt{4} \Downarrow \mathtt{4}}\ \text{(B-NUM)}}{(z + \mathtt{4})\{^{\mathtt{3}}/_{z}\} \Downarrow \mathtt{7}}\ \text{(B-ADD)}}{(\mathtt{let}\ z = (\mathtt{2} + z)\ \mathtt{in}\ z + \mathtt{4})\{^{\mathtt{1}}/_{z}\} \Downarrow \mathtt{7}}\ \text{(B-LET)}}{\mathtt{let}\ z = \mathtt{1}\ \mathtt{in}\ (\mathtt{let}\ z = (\mathtt{2} + z)\ \mathtt{in}\ z + \mathtt{4}) \Downarrow \mathtt{7}}\ \text{(B-LET)}$$

Figure 4.3: An example derivation

**Definition 21 (Substitution)** *Let $E$ be an arbitrary expression in $Exp_{loc}$ and $P$ a program. Then $E\{^{P}/_{x}\}$, the result of substituting the program $P$ for all free occurrences of the variable $x$ in the expression $E$ is defined by structural induction on $E$ as follows:*

*(i)* $x\{^{P}/_{x}\} = P$
*(ii)* $y\{^{P}/_{x}\} = y$, *if $y$ is a variable different from $x$*
*(iii)* $(E_1\ \mathsf{op}\ E_2)\{^{P}/_{x}\} = (E_1\{^{P}/_{x}\})\ \mathsf{op}\ (E_2\{^{P}/_{x}\})$
*(iv)* $(\mathtt{let}\ x = E_1\ \mathtt{in}\ E_2)\{^{P}/_{x}\} = \mathtt{let}\ x = (E_1\{^{P}/_{x}\})\ \mathtt{in}\ E_2$
*(v)* $(\mathtt{let}\ y = E_1\ \mathtt{in}\ E_2)\{^{P}/_{x}\} = \mathtt{let}\ y = (E_1\{^{P}/_{x}\})\ \mathtt{in}\ (E_2\{^{P}/_{x}\})$, *if $y$ is a variable different from $x$* □

The crucial clause is (iv). For example substituting $(\mathtt{1} + \mathtt{2})$ for $x$ in the expression $\mathtt{let}\ x = (x + \mathtt{3})\ \mathtt{in}\ (x \times x)$ results in the expression $\mathtt{let}\ x = ((\mathtt{1} + \mathtt{2}) + \mathtt{3})\ \mathtt{in}\ (x \times x)$; the occurrences of $x$ in the body, $(x \times x)$, are already bound in the declaration $\mathtt{let}\ x = (x + \mathtt{3})\ \mathtt{in}\ \ldots$ and are therefore left untouched.

An example derivation is given in Figure 4.3, where this subtlety plays a role, in the evaluation of repeated declarations of the same variable $z$ in $\mathtt{let}\ z = \ldots\ \mathtt{in}\ (\mathtt{let}\ z = \ldots\ \mathtt{in}\ \ldots)$. The tricky point in the derivation is the observation that $(\mathtt{let}\ z = (\mathtt{2} + z)\ \mathtt{in}\ z + \mathtt{4})\{^{\mathtt{1}}/_{z}\}$ turns out to be $(\mathtt{let}\ z = (\mathtt{2} + z)\{^{\mathtt{1}}/_{z}\}\ \mathtt{in}\ z + \mathtt{4}$. Therefore the derivation of $(\mathtt{let}\ z = (\mathtt{2} + z)\ \mathtt{in}\ z + \mathtt{4})\{^{\mathtt{1}}/_{z}\} \Downarrow \mathtt{7}$ using an application of the rule (B-LET) requires the hypotheses $(\mathtt{2} + z)\{^{\mathtt{1}}/_{z}\} \Downarrow \mathtt{3}$ and $(z + \mathtt{4})\{^{\mathtt{3}}/_{z}\} \Downarrow \mathtt{7}$.

In Chapter 2.2.3 we proved various interesting properties of the big-step semantics of expressions from the language *Exp* using structural induction. Here we show how these can be extended to $Exp_{loc}$.

**Proposition 22 (Determinacy)** *For every program P, if* $\vdash_{big} P \Downarrow \mathtt{m}$ *and* $\vdash_{big} P \Downarrow \mathtt{n}$ *then* $\mathtt{m} = \mathtt{n}$. □

**Proof:** The proof of the corresponding result for *Exp* was by structural induction on expressions. But here structural induction can not be used. For if $P$ is the let expression $\mathtt{let}\ x = Q\ \mathtt{in}\ E$ then with structural induction we will be able to assume the required property of $Q$, that is $Q \Downarrow \mathtt{k}_1$ and $Q \Downarrow \mathtt{k}_2$ implies $k_1 = k_2$. But we have no assumption about $E$ because in general $E$ will not be a program; normally it will have free occurrences of $x$.

We prove the result by Rule induction. For every program $Q$ and numeral $\mathtt{k}$ let $\mathcal{P}(Q, \mathtt{k})$ be the property:

for every number $m$, if $Q \Downarrow \mathtt{m}$ then $m = k$.

We use Rule induction to prove that $\vdash_{big} P \Downarrow \mathtt{n}$ implies $\mathcal{P}(P, \mathtt{n})$.

To do so we assume the inductive hypothesis (IH):

$\mathcal{P}(Q, \mathtt{k})$ is true for every pair $(Q, \mathtt{k})$ which has a proof of $\vdash_{big} Q \Downarrow \mathtt{k}$ smaller than a proof of $\vdash_{big} P \Downarrow \mathtt{n}$.

From this assumption we need to prove that $P \Downarrow \mathtt{n}$ implies $\mathcal{P}(P, \mathtt{n})$.

So suppose $\vdash_{big} P \Downarrow \mathtt{n}$. To show that $\mathcal{P}(P, \mathtt{n})$ let us further assume that $\vdash_{big} P \Downarrow \mathtt{m}$; from these two assumptions we need to show that $m = n$. To so so let us look a the derivation of $P \Downarrow \mathtt{n}$, and in particular the last rule used. According to Figure 4.2 there are four possibilities. Let us look at the most interesting, (B-LET)*:* $P$ has the form $\mathtt{let}\ x = P_1\ \mathtt{in}\ E$ and the last rule used is $\dfrac{P_1 \Downarrow \mathtt{n}_1 \quad E\{^{\mathtt{n}_1}/_x\} \Downarrow \mathtt{n}}{P \Downarrow \mathtt{n}}$ for some number $n_1$. But the proof of the judgement $P \Downarrow \mathtt{m}$ must also use (B-LET), and therefore we must have $\dfrac{P_1 \Downarrow \mathtt{m}_1 \quad E\{^{\mathtt{m}_1}/_x\} \Downarrow \mathtt{m}}{P \Downarrow \mathtt{m}}$ for some number $m_1$.

Now the inductive hypothesis (IH) applies to the pair $(P_1, \mathtt{n}_1)$; so $n_1$ and $m_1$ must be the same number. This is turn means that $E\{^{\mathtt{n}_1}/_x\}$ and $E\{^{\mathtt{m}_1}/_x\}$ must be the same program. Moreover (IH) applies to the pair $(E\{^{\mathtt{n}_1}/_x\}, \mathtt{n})$, which means that $m$ must be equal to $n$.

There are three other possibilities for the last rule used in the derivation of the judgement $P \Downarrow \mathtt{n}$, namely (B-NUM), (B-ADD), and (B-MULT); each are handled in exactly the same way as the case we have just seen. □

**Proposition 23 (Normalisation)** *For every program P in* $Exp_{loc}$ *there is a value* $\mathtt{n}$ *such that* $\vdash_{big} P \Downarrow \mathtt{n}$.

**Proof:**[Outline] Once more we can not use structural induction here because the subcomponents of the program $\mathtt{let}\ x = Q\ \mathtt{in}\ E$ are not necessarily programs. Luckily there is an easy alternative. Let $|P|$ be the number of symbols occurring in the program $P$. Then there is an easy proof using mathematical induction on $|P|$ that there is always some $\mathtt{n}$ such that $P \Downarrow \mathtt{n}$. The proof proceeds by examining the possible forms that $P$ can take, and uses the fact that $|E| = |E\{^{\mathtt{k}}/_x\}|$. □

**Exercise 25** *Fill in the details of the proof of Proposition 23.* □

Let us now revisit the discussion at the beginning of this section. Local declarations were introduced in order to avoid the repeated evaluation of a sub-expression with multiple occurrences, such as $(1 + 2)$ in (4.1) above. We now prove that this is indeed the case. First a lemma.

**Lemma 24** *Suppose* $\vdash_{big} P \Downarrow \mathtt{n_p}$. *Then for any expression containing at most one free variable x,* $\vdash_{big} E\{^{P}/_{x}\} \Downarrow \mathtt{n}$ *if and only if* $\vdash_{big} E\{^{\mathtt{n_p}}/_{x}\} \Downarrow \mathtt{n}$.

**Proof:** For what is by the now the standard reason this also can not be proved by structural induction on $E$. So how can this be proved?

Now suppose $E$ is an expression as in the statement of the lemma, with at most one free variable $x$. This variable may have multiple occurrences and so the program $P$ may also have multiple occurrences in the expression $E\{^{P}/_{x}\}$, which means that when evaluating $E\{^{P}/_{x}\}$ the program $P$ may be evaluated multiple times. However in ($\mathtt{let}\ x = P\ \mathtt{in}\ E$) it is evaluated exactly once; moreover the evaluation of this latter expression gives the correct result:

**Proposition 25** *Suppose E is an expression which contains at most one free variable x. Then* $\vdash_{big} E\{^{P}/_{x}\} \Downarrow \mathtt{n}$ *if and only if* $\vdash_{big} (\mathtt{let}\ x = P\ \mathtt{in}\ E) \Downarrow \mathtt{n}$.

**Proof:** Follows immediately from Lemma 24. Normalisation tells us that there exists some numeral $\mathtt{n_p}$ such that $\vdash_{big} P \Downarrow \mathtt{n_p}$; moreover Determinacy means that it is unique.

Now suppose $\vdash_{big} E\{^{P}/_{x}\} \Downarrow \mathtt{n}$. Then by the lemma there is a derivation of $E\{^{\mathtt{n_p}}/_{x}\} \Downarrow \mathtt{n}$. The rule (B-LET) from Figure 4.2 now allows us to construct a derivation of that ($\mathtt{let}\ x = P\ \mathtt{in}\ E) \Downarrow \mathtt{n}$.

Conversely suppose $\vdash_{big} (\mathtt{let}\ x = P\ \mathtt{in}\ E) \Downarrow \mathtt{n}$. This can only be inferred by an application of this rule (B-LET) and so we must be also able to derive $E\{^{\mathtt{n_p}}/_{x}\} \Downarrow \mathtt{n}$. Once more the lemma now allows us to conclude that $\vdash_{big} E\{^{P}/_{x}\} \Downarrow \mathtt{n}$. □

## 4.1.2 Small-step semantics

The judgements for the small-step semantics of $Exp_{loc}$ take the form

$$P \rightarrow P'$$

and the inference rules, given in Figure 4.4, are mostly inherited from those for *Exp*. The new rules, (S-LET) and (S-LET.SUB), say that in order to evaluate the expression $\mathtt{let}\ x = P\ \mathtt{in}\ E$

- first evaluate $P$ to some value $\mathtt{n}$
- then start evaluating $E$, in which the free variable $x$ has been replaced by the value $\mathtt{n}$.

**Exercise 26** *Supply the rules missing from Figure 4.4 for the programs of the form* $(P_1 \times P_2)$. □

(S-LEFT)
$$\frac{P_1 \to P_1'}{(P_1 + P_2) \to (P_1' + P_2)}$$

(S-ADD)
$$\frac{}{(\mathtt{n}_1 + \mathtt{n}_2) \to \mathtt{n}_3} \quad n_3 = \mathsf{add}(n_1, n_2)$$

(S-N.RIGHT)
$$\frac{P_2 \to P_2'}{(\mathtt{n} + P_2) \to (\mathtt{n} + P_2')}$$

(S-LET)
$$\frac{P \to P'}{\mathtt{let}\ x = P\ \mathtt{in}\ E \to \mathtt{let}\ x = P'\ \mathtt{in}\ E}$$

(S-LET.SUBS)
$$\frac{}{\mathtt{let}\ x = \mathtt{n}\ \mathtt{in}\ E \to E\{^{\mathtt{n}}/_{x}\}}$$

Figure 4.4: Small-step semantics of *let expressions*

**Proposition 26 (Progress)** *For every program $P$ in $\mathsf{Exp}_{loc}$, either $P$ is a value or there is some program $P'$ such that $\vdash_{sm} P \to P'$.*

**Proof:** Here we can use structural induction on $P$. □

**Corollary 27** *For every program $P$ in $\mathsf{Exp}_{loc}$, there exists some numeral $\mathtt{n}$ such that $P \to^* \mathtt{n}$.*

**Proof:** This is a simple consequence of the previous proposition. First a proof by structural induction on $P_1$ will show that if $P_1 \to P_2$ then $|P_2| < |P_1|$; here we are using $|P|$ to mean the number of symbols used in $P$. This means that as the small-step semantics is repeatedly applied the size of the program decreases with each step. This can not go on forever.

So let $k$ be such that $P \to^k P_1$ for some $P_1$ but $P \to^{(k+1)} P_2$ for no $P_2$. Progress, Proposition 26, means that $P_1$ must be a value, that is some numeral $\mathtt{n}$. □

**Theorem 28** *For every program $P$ in $\mathsf{Exp}_{loc}$, $\vdash_{big} P \Downarrow \mathtt{n}$ implies $P \to^* \mathtt{n}$.*

**Proof:** See the corresponding proof for arithmetic expressions, Proposition 4 in Chapter 2.2.3 □

**Theorem 29** *For every program $P$ in $\mathsf{Exp}_{loc}$, $P \to^* \mathtt{n}$ implies $\vdash_{big} P \Downarrow \mathtt{n}$.*

**Proof:** Again a minor extension of the corresponding proof for arithmetic expressions, Proposition 6 in Chapter 2.2.3, although that result is for the *choice* variation of the small-step semantics. □

## 4.2 Adding Boolean expressions

.

$$\begin{aligned}
E \in Exp_B \quad &::= \quad v \\
&\quad\ \mid (E + E) \mid (E \times E) \mid \texttt{let}\ x = E\ \texttt{in}\ E \\
&\quad\ \mid E \ \mathsf{and}\ E \mid \neg\, E \mid E = E \mid \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \\
v \in Val \quad &::= \quad x \in Vars \mid \mathtt{n} \in Nums \mid \texttt{tt} \mid \texttt{ff}
\end{aligned}$$

Figure 4.5: Adding Booleans; the language $Exp_B$

(S-IF)
$$\frac{P \to P'}{\texttt{if}\ P\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \to \texttt{if}\ P'\ \texttt{then}\ P_1\ \texttt{else}\ P_2}$$

(S-IF.T)
$$\frac{}{\texttt{if}\ \texttt{tt}\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \to P_1}$$

(S-IF.F)
$$\frac{}{\texttt{if}\ \texttt{ff}\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \to P_2}$$

(S-AND.LEFT)
$$\frac{P_1 \to P_1'}{(P_1\ \mathsf{and}\ P_2) \to (P_1'\ \mathsf{and}\ P_2)}$$

(S-AND.RIGHT)
$$\frac{P_2 \to P_2'}{(\mathsf{v}\ \mathsf{and}\ P_2) \to (\mathsf{v}\ \mathsf{and}\ P_2')}$$

(S-CONJ)
$$\frac{}{(\mathsf{bv}_1\ \mathsf{and}\ \mathsf{bv}_2) \to \mathsf{bv}} \quad bv = \mathsf{conj}(bv_1, bv_2)$$

(S-NOT)
$$\frac{P \to P'}{\neg\, P_1 \to \neg\, P'}$$

(S-NOT.T)
$$\frac{}{\neg\, \texttt{tt} \to \texttt{ff}}$$

(S-NOT.F)
$$\frac{}{\neg\, \texttt{ff} \to \texttt{tt}}$$

Figure 4.6: Extra small-step inference rules for $Exp_B$

In Figure 4.5 we give the syntax for an extension of the language $Exp_{loc}$ with Boolean operators and values. For convenience there is now a separate syntactic class of values. Recall that in *Exp* the numerals $\mathtt{n}$ are syntactic representations for the natural numbers $n$. In the same way we have two syntactic representations `tt`, `ff` for the two Boolean values *true* and *false*. Then the syntactic class of expressions is extended with two new operators, Boolean conjunction and negation $E\ \mathsf{and}\ F$, $\neg\, E$, an equality operator, $E_1 = E_2$, for any two arithmetic expressions, and a branching construct, `if` $E$ `then` … `else` …. Note that unlike the language of commands *While*, whose BNF is given in Figure 3.1 of Chapter 3, we do not have separate syntactic classes for Booleans and arithmetics; this will make life a little more interesting.

The small-step semantics for the extended language is by now straightforward; the extra rules for the new constructs are given in Figure 4.4. The rules for the `if then else` construct formalises the following informal idea:

- To start executing the program `if` $P$ `then` $P_1$ `else` $P_2$, execute one step of the program $P$, rule (S-IF).
- On the other hand, if $P$ is already evaluated to a value
  - if this value is `tt` then start evaluating $P_1$, rule (S-IF.T)
  - if it is `ff` then start evaluating $P_2$, rule (S-IF.F).

The evaluation of the program $P_1$ and $P_2$ proceeds in a left-to-right fashion, in a manner similar to the evaluation of $P_1 + P_2$. The third rule, (S-CONJ), uses the function $\mathsf{conj}(bv_1, bv_2)$ which operates on Boolean values; if both $bv_1$ and $bv_2$ are *true* then this returns *true*, otherwise it returns *false*. Note that for these rules we are using $bv$ as a meta-variable over Boolean values, and `bv` to range over their corresponding syntactic representations `tt` and `ff`. The evaluation of $\neg\ P$ proceeds using the same strategy.

**Exercise 27** *Design a big-step semantics for the language $Exp_B$.* □

**Exercise 28** *Prove that your big-step semantics agrees with the small-step semantics given in Figure 4.6. That is prove*

*(i)* $P \Downarrow \mathtt{v}$ *implies* $P \rightarrow^* \mathtt{v}$
*(ii)* $P \rightarrow^* \mathtt{v}$ *implies* $P \Downarrow \mathtt{v}$. □

In extending the language $Exp_{loc}$ to the language $Exp_B$ one significant property is lost; it is no longer the case that for every program $P$ there exists some value `v` such that $P \rightarrow^* \mathtt{v}$. For example take $P$ to be `if (3 + 4) then 6 else 1`. Using the rule (S-IF) this can be reduced to the program `if 7 then 6 else 1` but now neither of the rules (S-IF.T) or (S-IF.F) can be applied and therefore the computation is stuck. The rules assume that in programs of the form `if` $P_1$ `then` $P_2$ `else` $P_3$ the sub-program $P_1$ will always reduce to a Boolean value, but this is not necessarily the case.

This situation arises quite frequently in programming languages; programs can be syntactically correct with respect to the BNF of the language but their execution leads to run-time errors. But many of these run-time errors can be eliminated by a syntactic analysis of the program prior to execution. This is the topic of the next section.

## 4.3 Typing

In the language $Exp_B$ we expect the program $(P_1 + P_2)$ to return a numeral, since + is a symbol representing addition. Similarly we expect $(Q_1$ and $Q_2)$ to return a Boolean value, either `tt` or `ff`. But equally well, in the former we expect both $P_1$ and $P_2$ to evaluate to numerals while in the latter we expect $Q_1$ and $Q_2$ to evaluate to Booleans; otherwise the corresponding operations, arithmetic addition and Boolean conjunction respectively, can not be performed.

These remarks form the basis of a method for syntactically analysing programs to ensure that when they are executed run-time errors do not occur. The kind of errors which can be captured include

- attempting to apply an arithmetic operation to a Boolean value
- attempting to apply a Boolean operation to a numeral
- having to execute a program of the form `if n then` $P_2$ `else` $P_3$; that is finding an arithmetic value in a place where a Boolean value is expected.

The basic idea of the syntactic analysis is straightforward. We classify programs into three kinds:

(1) those which should return an arithmetic value
(2) those which should return a Boolean value
(3) those which are can not be evaluated

We say the first are programs *with the type* int while those in (2) the type bool.

The analysis would then involve trying to decide the type of a given program. More concretely it would take the form of a *type inference algorithm* which would input an arbitrary program in $Exp_B$ and return one of three possible values, the type int, the type bool, or the token wrong. We would expect such algorithms to be correct, in the sense that:

(a) if int is returned for program $P$, then $P$ is guaranteed to evaluate to a numeral
(b) if bool is returned, it is guaranteed to evaluate to a Boolean.

This correctness criterion is not very onerous. Here is a very simple algorithm ($A$) which satisfies it:

(1) Input a program $P$.
(2) If $P$ is a numeral return the answer int.
(3) If $P$ is one of the Boolean values `tt` or `ff` return the answer bool.
(4) Otherwise return the token wrong.

This algorithm is obviously correct; every program to which it assigns a type evaluates to a value of that type. But it assigns types to very few programs and so is not very useful.

One can imagine more useful algorithms, which manage to assign types to more programs by analysing their structure but remain correct. We will not pursue directly the development of such algorithms; instead we look at the question of how comprehensive we could expect them to be. For what class of programs should they be able to assign types?

### 4.3.1 Typechecking

This refers to the problem of deciding exactly what types *should* be assigned to what programs. In other words typechecking can be used to evaluate the usefulness of type inference algorithms.

Typechecking can be carried out structurally; the program $(P_1 \times P_2)$ should have the type int, provided both $P_1$ and $P_2$ can also have the type int. But the structural analysis of programs of the form `let` $x$ `=` $P_1$ `in` $E$ leads to complications. The type, if any, which should be assigned to this program depends on that assigned to the sub-program

**Types** : $\tau ::= \mathsf{int} \mid \mathsf{bool}$ **Type environments:** $\epsilon \mid \Gamma, x\!:\!\tau$

**Environment look-up:**

$$\text{(TY-LOOK1)}\ \frac{}{\Gamma, x\!:\!\tau \vdash x\!:\!\tau} \qquad \text{(TY-LOOK2)}\ \frac{\Gamma \vdash x_2\!:\!\tau_1}{\Gamma, x_1\!:\!\tau_1 \vdash x_2\!:\!\tau_2}\ x_1 \neq x_2$$

Figure 4.7: Types and environments

$P_1$; but it also depends on the structure of $E$ which in general is not a program. For example if $E$ is the expression $(2 + x)$ then whether or not it should be assigned a type depends on what type can be assigned to $P_1$; if $P_1$ can be assigned the type $\mathsf{int}$ then $E$ also should also be assigned $\mathsf{int}$ whereas if $P_1$ is assigned $\mathsf{bool}$ then $E$ should not be assigned any type.

This dependence of the type of $E$ on that of $P_1$ can be conveniently expressed in terms of assumptions about the free variable $x$ occurring in $E$. For example the previous discussion can be rephrased as saying

- assuming $x$ stands for an arithmetic value, $E$ should be assigned the type $\mathsf{int}$
- assuming $x$ stands for a Boolean value, $E$ should not be assigned any type.

As the structural analysis of an expression proceeds these assumptions about free variables build up. For example to decide what type should be assigned to

$$\texttt{let}\ x = P\ \texttt{in}\ \texttt{let}\ y = E_1\ \texttt{in}\ E_2$$

we need to analyse the structure of $E_2$, and this analysis will depend on assumptions about the variables $x$ and $y$. Consequently our inference rules for typechecking expressions in $Exp_B$ will take the form

$$\Gamma \vdash E : \tau \tag{4.5}$$

where $\Gamma$ is a *type environment*, and $\tau$ is an allowed type. For the simple language $Exp_B$ the only possible types are $\mathsf{int}$, for arithmetics, and $\mathsf{bool}$ for Booleans, while $\Gamma$ is essentially a list of type associations between variables and types, $x\!:\!\tau$.

These are defined formally in Figure 4.7; there $\epsilon$ represents the empty list. A method is also given for checking which type, if any, is currently associated with a variable in $\Gamma$, written $\Gamma \vdash x\!:\!\tau$; essentially the rules scan $\Gamma$ from right to left looking for an occurrence of $x$.

The inference rules for typechecking expressions, with judgements of the form (4.5) above, are given in Figure 4.8. Intuitively $\Gamma \vdash E : \tau$ means that if all the free variables occurring in $E$ are replaced with values of the type dictated by the environment $\Gamma$ then the resulting program should execute completely to a value. Moreover if $\tau$ is $\mathsf{int}$ then the result will be a numeral, whereas if it is $\mathsf{bool}$ then it will be a Boolean value.

(TY-BOOL)
$$\frac{}{\Gamma \vdash \mathtt{tt} : \mathsf{bool}}$$

(TY-BOOL)
$$\frac{}{\Gamma \vdash \mathtt{ff} : \mathsf{bool}}$$

(TY-INT)
$$\frac{}{\Gamma \vdash \mathtt{n} : \mathsf{int}}$$

(TY-EQ)
$$\frac{\Gamma \vdash E_1 : \tau \quad \Gamma \vdash E_2 : \tau}{\Gamma \vdash E_1 = E_2 : \mathsf{bool}}$$

(TY-A.OP)
$$\frac{\Gamma \vdash E_1 : \mathsf{int} \quad \Gamma \vdash E_2 : \mathsf{int}}{\Gamma \vdash E_1 \mathit{op}\, E_2 : \mathsf{int}}$$

(TY-B.OP)
$$\frac{\Gamma \vdash E_1 : \mathsf{bool} \quad \Gamma \vdash E_1 : \mathsf{bool}}{\Gamma \vdash E_1 \mathit{bop}\, E_2 : \mathsf{bool}}$$

(TY-IF)
$$\frac{\begin{array}{l}\Gamma \vdash E : \mathsf{bool} \\ \Gamma \vdash E_1 : \tau \quad \Gamma \vdash E_2 : \tau\end{array}}{\Gamma \vdash \mathtt{if}\ E\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2 : \tau}$$

(TY-LET)
$$\frac{\Gamma \vdash E_1 : \tau_1 \qquad \Gamma, x{:}\tau_1 \vdash: E_2 : \tau}{\Gamma \vdash \mathtt{let}\ x = E_1\ \mathtt{in}\ E_2 : \tau}$$

Figure 4.8: Typing inference rules for $Exp_B$

Most of the rules are straightforward. The rules (TY-BOOL) and (TY-INT) assign the appropriate types to values, while (TY-A.OP) assigns $\mathsf{int}$ to an expression of the form $E_1 + E_2$ or $E_1 \times E_2$, provided the same type can be applied to $E_1$ and $E_2$; in the rule (TY-A.OP) is a meta-variable for an arithmetic operator. The rule (TY-B.OP) for Boolean operators is similar; the rule of expressions of the form $\neg\ E$ is omitted but is a unary version of (TY-B.OP). The rule (TY-EQ) requires both $E_1$ and $E_2$ to have the same type in the expression $(E_1 = E_2)$. Similarly to assign a type to $\mathtt{if}\ E\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2$, according to (TY-IF) both of $E_1$ and $E_2$ must have the same type, and of course we must be able to assign to $E$ the type $\mathsf{bool}$. The only non-trivial rule is for let-expressions $\mathtt{let}\ x = E_1\ \mathtt{in}\ E_2$; according to (TY-LET) this can be assigned the type $\tau$ provided $E_2$ can be assigned that type under an augmented list of assumptions. $\Gamma$ is increased by the assumption $x{:}\tau_1$, where $\tau_1$ is any type which can be inferred for $E_1$.

For example consider the expression

$$\mathtt{let}\ x = 1 + z\ \mathtt{in}\ (\mathtt{if}\ x = \mathtt{0}\ \mathtt{then}\ z\ \mathtt{else}\ x + 1) \tag{4.6}$$

This has one free variable, $z$, and therefore it can only be typechecked relative to an environment which has some type association with $z$. Let $\Gamma_z$ be a type environment such that the judgement

$$\Gamma_z \vdash z : \mathsf{int}$$

can be inferred using the two rules (TY-LOOK1) and (TY-LOOK2) from Figure 4.7; we will usually use (TY-LP) to refer to this procedure, of scanning an environment right to left for an entry.

$$\dfrac{\dfrac{\begin{array}{c}\Gamma_z \vdash 1 : \mathsf{int}\\ \Gamma_z \vdash z : \mathsf{int}\end{array}}{\Gamma_z \vdash 1 + z : \mathsf{int}}\,(\textsc{ty-a.op}) \qquad \dfrac{\dfrac{\begin{array}{c}\Gamma_{zx} \vdash \mathtt{x} : \mathsf{int}\\ \Gamma_{zx} \vdash \mathtt{0} : \mathsf{int}\end{array}}{\Gamma_{zx} \vdash x = \mathtt{0} : \mathsf{bool}}\,(\textsc{ty-eq}) \quad \dfrac{}{\Gamma_{zx} \vdash z : \mathsf{int}}\,(\textsc{ty-lp}) \quad \dfrac{\begin{array}{c}\Gamma_{zx} \vdash x : \mathsf{int}\\ \Gamma_{zx} \vdash 1 : \mathsf{int}\end{array}}{\Gamma_{zx} \vdash x + 1 : \mathsf{int}}\,(\textsc{ty-add})}{\Gamma_{zx} \vdash \mathtt{if}\ x = \mathtt{0}\ \mathtt{then}\ z\ \mathtt{else}\ x + 1 : \mathsf{int}}\,(\textsc{ty-if})}{\Gamma_z \vdash \mathtt{let}\ x = 1 + z\ \mathtt{in}\ (\mathtt{if}\ x = \mathtt{0}\ \mathtt{then}\ z\ \mathtt{else}\ x + 1) : \mathsf{int}}\,(\textsc{ty-let})$$

Figure 4.9: Inferring a typing judgement

In Figure 4.9 we show that the expression in (4.6) can be assigned the type int relative to such an $\Gamma_z$. In the derivation we have used $\Gamma_{zx}$ as a shorthand for the augmented environment $\Gamma_z, x : \mathsf{int}$, and have omitted some instances of the use of the rule (ty-lp).

The fact that type environments are lists, scanned from right-to-left, is of significance for the typing of *let* expressions, as the following example shows.

**Example 30** *Consider the program* $\mathtt{let}\ x = \mathtt{tt}\ \mathtt{in}\ (\mathtt{let}\ x = \mathtt{2}\ \mathtt{in}\ x + x)$. *This can be assigned the type* int*, because of the derivation:*

$$\dfrac{\dfrac{}{\vdash \mathtt{tt} : \mathsf{bool}}\,(\textsc{ty-bool}) \qquad \dfrac{\dfrac{}{x{:}\mathsf{bool} \vdash \mathtt{2} : \mathsf{int}}\,(\textsc{ty-int}) \quad \dfrac{\dfrac{}{x{:}\mathsf{bool},\ x{:}\mathsf{int} \vdash x : \mathsf{int}}\,(\textsc{ty-look1})}{x{:}\mathsf{bool},\ x{:}\mathsf{int} \vdash x + x : \mathsf{int}}\,(\textsc{ty-a.op})}{x{:}\mathsf{bool} \vdash \mathtt{let}\ x = \mathtt{2}\ \mathtt{in}\ x + x : \mathsf{int}}\,(\textsc{ty-let})}{\epsilon \vdash \ \mathtt{let}\ x = \mathtt{tt}\ \mathtt{in}\ (\mathtt{let}\ x = \mathtt{2}\ \mathtt{in}\ x + x) : \mathsf{int}}\,(\textsc{ty-let})$$

*This accords with our decision that inner declarations have precedence over outer ones; the inner declaration* $x = \mathtt{2}$ *is the one which is ultimately used when the body,* $(x + x)$ *is evaluated. And note that in the type inference above the application of the look-up rule* (ty-look1) *returns the correct type association for x.*

*The reader should check that the program* $\mathtt{let}\ x = \mathtt{tt}\ \mathtt{in}\ (\mathtt{let}\ x = \mathtt{2}\ \mathtt{in}\ (x\ \mathsf{and}\ x)\ )$ *can not be assigned a type using the inference rules, which again accords with our intuition.* □

Despite the subtlety of this example, in general typing derivations are independent of many minor variations in the environments used. The more interesting ones are collected in the following proposition.

**Proposition 31 (Sanity)**

*(1) (Strengthening) If x is not in* $\mathsf{fv}(E)$ *then* $\Gamma_1, x{:}\tau_x, \Gamma_2 \vdash E : \tau$ *implies* $\Gamma_1, \Gamma_2 \vdash E : \tau$.

*(2) (Weakening)* $\Gamma \vdash E : \tau$ *implies* $x{:}\tau_x, \Gamma \vdash E : \tau$.

*(3) (Fresh weakening)* $\Gamma \vdash E : \tau$ *implies* $\Gamma, x{:}\tau_x \vdash E : \tau$*, provided x does not occur in E.*

*(4) (Permutation)* $\Gamma_1, x{:}\tau_x, y{:}\tau_y, \Gamma_2 \vdash E : \tau$ *implies* $\Gamma_1, y{:}\tau_y, x{:}\tau_x, \Gamma_2 \vdash E : \tau$*, provided x is different than y.*

*(5) (Repetition)* $\Gamma_1, x{:}\tau_1, \Gamma_2, x{:}\tau_2, \Gamma_3 \vdash E : \tau$ *implies* $\Gamma_1, \Gamma_2, x{:}\tau_2, \Gamma_3 \vdash E : \tau$

**Proof:** Each result is proved by structural induction on $E$. Since there are eleven different possibilities for the structure of $E$ these proofs are rather long. But the only case which does not follow trivially is when $E$ is a variable. □

One interesting consequence of (Strengthening) and (Weakening) pertains to the typing of programs, which have no free variables; their typing is independent of the environment used:

> For every program $P$, $\epsilon \vdash P$ if and if if there is some environment $\Gamma$ such that $\Gamma \vdash P$.

**Exercise 29** *Prove this property for all programs P.*

**Exercise 30** *Prove, using counter-examples, that in each of (1), (3) and (4) of Proposition 31 the side-condition is necessary.* □

The ability to assign a type to an expression, means that we expect the expression, when transformed into a program by replacing its free variables with values, to evaluate fully to a value of that type. Since informally we expect this value to be unique, one would also expect the type which can be assigned to the expression to be unique.

**Proposition 32 (Type uniqueness)** *If* $\Gamma \vdash E : \tau_1$ *and* $\Gamma \vdash E : \tau_2$ *then* $\tau_1 = \tau_2$.

**Proof:** A straightforward argument by structural induction on $E$; essentially there is only one possible inference rule from Figure 4.8 which can be applied to any given expression $E$.

But the most important property of the type inference rules is that, in some sense, typing is preserved by substitution:

**Theorem 33 (Substitution lemma)** *Suppose* $\Gamma, x{:}\tau_x \vdash E : \tau$, *where E is any term in the language* $Exp_B$. *Then* $\epsilon \vdash P : \tau_x$ *implies* $\Gamma \vdash E\{^P\!/\!_x\} : \tau$.

**Proof:** By structural induction on $E$, which means that there are eleven cases to consider in all.

Let us first look at the most difficult case; suppose $E$ has the form `let` $y = E_1$ `in` $E_2$. Since $\Gamma, x{:}\tau_x \vdash E : \tau$ we know

(i) $\Gamma, x{:}\tau_x \vdash E_1 : \tau_1$ for some type $\tau_1$
(ii) $\Gamma, x{:}\tau_x, y{:}\tau_1 \vdash E_2 : \tau$

Structural induction applied to (i) gives

(i') $\Gamma \vdash E_1\{^P\!/\!_x\} : \tau_1$

We now do a case analysis. First suppose that $x$ and $y$ are different variables. So here $E\{^P\!/\!_x\}$ is actually `let` $y = (E_1\{^P\!/\!_x\})$ `in` $(E_2\{^P\!/\!_x\})$. Moreover (Permutation), from Proposition 31, applied to (ii) gives $\Gamma, y{:}\tau_1 x{:}\tau_x, \vdash E_2 : \tau$, to which structural induction can be applied to obtain (ii') $\Gamma, y : \tau_1 \vdash E_2\{^P/_x\} : \tau$

An application of (TY-LET) to (i') and (ii') now gives the required $\Gamma \vdash E\{^P/_x\} : \tau$.

Now suppose that $x$ and $y$ are the same; here $E\{^P/_x\}$ works out to be `let` $y$ = $(E_1\{^P/_x\})$ `in` $E_2$. In this case (Repetition) can be applied to (ii) to obtain

(ii') $\Gamma, x : \tau_1 \vdash E_2 : \tau$

and once more the rule (TY-LET) applied to (i') and (ii') gives $\Gamma \vdash E\{^P/_x\} : \tau$.

We have now finished one of eleven cases of the possible structure of $E$. Luckily all others are straightforward; we look briefly at two.

Suppose $E$ is a variable. If this variable is $x$ then $\tau$ must coincide with $\tau_x$ the required result, $\Gamma \vdash P : \tau$, follows by (Weakening) from $\epsilon \vdash P : \tau_x$. If it is different, say $y$, then the argument is equally trivial since $E\{^P/_x\}$ is $y$ and $\Gamma \vdash y : \tau$ follows from (Strengthening).

As an example of an inductive case suppose $E$ is $E_1 + E_2$, in which case we know that $\tau$ must be the type $\mathsf{int}$. Then by the hypothesis we have

(i) $\Gamma, x : \tau_x \vdash E_1 : \mathsf{int}$
(ii) $\Gamma, x : \tau_x \vdash E_1 : \mathsf{int}$

since the judgement $\Gamma, x : \tau_x \vdash E : \mathsf{int}$ can only be established using the rule (TY-ADD). We can apply induction to both of these to obtain

(i') $\Gamma \vdash E_1\{^P/_x\} : \mathsf{int}$
(ii') $\Gamma \vdash E_1\{^P/_x\} : \mathsf{int}$

Now an application of (TY-ADD) gives the required $\Gamma \vdash E\{^P/_x\} : \mathsf{int}$.

All remaining inductive cases are equally mechanical. □

**Exercise 31** *Design a type inference algorithm, which inputs an arbitrary program from* $\mathsf{Exp}_B$ *and returns*

- *the type* int *if* $\epsilon \vdash P : \mathsf{int}$ *can be inferred from the type inference rules*
- *the type* bool *if* $\epsilon \vdash P : \mathsf{bool}$ *can be inferred from the type inference rules*
- *the token* wrong *otherwise.* □

In the remainder of this chapter we abbrviate the judgement $\epsilon \vdash P : \tau$ to simply $\vdash P : \tau$, leaving the empty type environment understood.

## 4.3.2 Typed programs don't go wrong

The purpose of assigning types to programs is to ensure that run-time errors will never occur; well-typed programs can be safely executed. For the language $Exp_B$ a run-time error occurs when the evaluation of a program to a value becomes stuck; we have seen the example $P$ = `if (3 + 4) then 6 else 1` on page 77; after one computation step the evaluation of $P$ can not continue. In this section we explain why the type inference system from Figure 4.8 ensures that all programs in $Exp_B$ which are assigned a type by the type inference system can be safety executed to completion, to obtain a value.

This idea of safety is usually encapsulated in the slogan

$$\boxed{\text{Safety } = \text{Progress } + \text{Preservation}}$$

We have already come across the idea of *Progress*, stated in Proposition 26 for the language $Exp_{loc}$. Intuitively it means that if a program has not terminated then it can continue evaluating.

**Theorem 34 (Progress)** *Let $P$ be a program in $Exp_B$ such that $\vdash P : \tau$. Then either $P$ is a value or there is some program $Q$ such that $P \rightarrow Q$.*

**Proof:** Straightforward structural induction on $P$. □

The second property, *Preservation*, means that whenever a program can be assigned a type, if it takes a computation step then the residual program can also be assigned the same type.

**Theorem 35 (Preservation)** *Let $P$ be a program in $Exp_B$ such that $\vdash P : \tau$. Then $P \rightarrow Q$ implies $\vdash Q : \tau$.*

**Proof:** Here again the proof is by structural induction, this time on $P$. Let us start with the difficult case, when $P$ has the form $\texttt{let}\ x = P_1\ \texttt{in}\ E$. Since $P$ has type $\tau$ we know

(i) $\vdash P_1 : \tau_x$ for some type $\tau_x$, such that
(ii) $x{:}\tau_x \vdash E$

According to the small-step semantics for $Exp_B$ in Figure 4.6 there are two possibilities for $Q$:

(a) $Q$ is $\texttt{let}\ x = P_1'\ \texttt{in}\ E$, where $P_1 \rightarrow P_1'$. Here the inductive hypothesis applied to (i) ensures that $\vdash P_1' : \tau_x$ and an application of the typing rule (TY-LET) to this and (ii) gives the required $\vdash Q : \tau$.
(b) $P_1$ is a value, $\mathtt{v}$ and $Q$ is $E\{\mathtt{v}/x\}$. Here the result $\vdash Q : \tau$ follows from (i) and (ii) by the Substitution lemma, Theorem 33.

There are many other possibilities for the structure of $P$. The base cases, when $P$ is a value, are all trivial since values can not make an evaluation step. Moreover all of the inductive cases are purely mechanical. Suppose for example that $P$ has the structure $\texttt{if}\ P\ \texttt{then}\ P_1\ \texttt{else}\ P_2$. Here we know

(i) $\vdash P : \mathsf{bool}$
(ii) $\vdash P_1 : \tau$
(iii) $\vdash P_2 : \tau$

Moreover examining the small-step semantics in Figure 4.6 we know that there are three possibilities for $Q$:

(a) $Q$ is $\texttt{if}\ P'\ \texttt{then}\ P_2\ \texttt{else}\ P_2$, where $P \rightarrow P'$. Applying Structural induction to (i) we obtain $\vdash P' : \mathsf{bool}$, and this together with (ii) and (iii) can be used with an application of the typing rule (TY-IF) to give the required $\vdash Q : \tau$.
(b) $P$ is the value $\texttt{tt}$ and $Q$ is $P_1$. The required result is given in (ii).
(c) Finally, $Q$ can be $P_2$, if $P$ is the $\texttt{ff}$, where the result is given in (iii). □

These two generic results, Progress and Preservation, when applied to the language $Exp_B$ ensure that every well typed program evaluates completely to value:

**Corollary 36** *Let $P$ be a well-typed program in $Exp_B$, that is $\vdash P : \tau$ for some type $\tau$. Then there exists some value $\mathsf{v}$ such that $P \rightarrow^* \mathsf{v}$. Moreover if $\tau$ is the type $\mathsf{int}$ then $\mathsf{v}$ is a numeral, whereas if it is $\mathsf{bool}$ it is a Boolean.*

**Proof:** We use the same strategy as in Corollary 27 in Section 4.1. We know that if $P \rightarrow P'$ then $|P'| < |P|$, where $|P|$ counts the number of symbols in $P$. Thus there must be some $k$ such that $P \rightarrow^k P_k$ but $P \rightarrow^{(k+1)} Q$ for no $Q$. Preservation, applied repeated means that $\vdash P^k : \tau$. Since $P_k \not\rightarrow$, Progress gives that $P_k$ is a value; moreover we know this value if of the same type as $P$. □

**Exercise 32** *Give an example of a program $P$ such that $P \rightarrow^* \mathsf{v}$ for some value $\mathsf{v}$, but which can not be typed, that is $\vdash P : \tau$ for no type $\tau$.* □

## 4.4 User-defined functions

The language defined so far, $Exp_B$ appears to be very limited; there are only two arithmetic operations and two Boolean operations. Nevertheless it is still a very powerful language. One can show that any computable function can be expressed by some expression in $Exp_B$.

**Exercise 33** *Let $\mathsf{monus} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ be the binary function such that $\mathsf{monus}(n, m)$ returns $0$ whenever $m$ is greater than $n$ and returns the result of subtracting $m$ from $n$ otherwise.*

*Show that there is some expression $E(x, y)$ in the language $Exp_B$ which implements the function $\mathsf{monus}$. This means that for every pair of numerals $\mathtt{n}$, $\mathtt{m}$ the expression $E(\mathtt{n}, \mathtt{m})$ should evaluate to the numeral correpsonding to the natural number $\mathsf{monus}(n, m)$.* □

Nevertheless the language would be more convenient if the user were allowed to define such functions directly and use these functions subsequently to define even more functions. Thus we would allow expressions of the form

$$\mathtt{if}\ \mathsf{max}(\mathtt{3}, \mathtt{4})\ \mathtt{then}\ \mathsf{fac}(\mathtt{6})\ \mathtt{else}\ \mathsf{rem}(\mathtt{10}, \mathtt{3}) \tag{4.7}$$

where $\mathsf{max}$, $\mathsf{fac}$, $\mathsf{rem}$ are user-declared function names. Of course when we come to evaluate such expressions we will have to know how to handle calls to these functions. So with every function name we will assume a *declaration* such as

$$\mathsf{max}(x, y) \quad \Leftarrow \mathtt{if}\ \mathsf{less}(x, y)\ \mathtt{then}\ y\ \mathtt{else}\ x$$

Then when evaluating the Boolean condition in (4.7) we end up evaluating the *body* in the declaration of $\mathsf{max}$, with the formal parameters, $x$ and $y$ instantiated by the actual parameters $\mathtt{3}$ and $\mathtt{4}$. That is we end up evaluating the expression $\mathtt{if}\ \mathsf{less}(\mathtt{3}, \mathtt{4})\ \mathtt{then}\ \mathtt{4}\ \mathtt{else}\ \mathtt{3}$. This in turn will require the evaluation of the expression $\mathsf{less}(\mathtt{3}, \mathtt{4})$, which in turn will depend on some declaration of the function $\mathsf{less}$.

$$
\begin{array}{rcl}
E \in \mathit{Fpl} & ::= & v \\
& & \mid (E + E) \mid (E \times E) \mid \texttt{let}\ x = E\ \texttt{in}\ E \\
& & \mid E\ \mathsf{and}\ E \mid \neg\, E \mid E = E \mid \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \\
& & f(F_1, \ldots, F_k), k > 0, \quad f \in \mathit{Fnames} \\
v \in \mathit{Val} & ::= & x \in \mathit{Vars} \mid \mathtt{n} \in \mathit{Nums} \mid \mathtt{tt} \mid \mathtt{ff}
\end{array}
$$

Figure 4.10: The language *Fpl*

$$
\begin{array}{rcl}
f_1(x_1, \ldots, x_{k_1}) & \Leftarrow & E_1 \\
\ldots & \Leftarrow & \ldots \\
f_i(x_1, \ldots, x_{k_i}) & \Leftarrow & E_i \\
\ldots & \Leftarrow & \ldots \\
f_n(x_1, \ldots, x_{k_n}) & \Leftarrow & E_n
\end{array}
$$

Sanity conditions:

(1) $E_i$ can only use fellow function names $f_1, \ldots, f_n$
(2) $E_i$ can only use variables $x_1, \ldots, x_{i_k}$, all of which are distinct
(3) number of arguments in $f(x_1, \ldots, x_k)$ determined by name $f$

Figure 4.11: Declaration set *D*

The extended syntax is given in Figure 4.10. The language *Fpl* is obtained from $Exp_B$ by adding one extra clause to the BNF definition. We now allow expressions of the form $f(E_1, \ldots, E_k)$ where $f$ comes from some predefined set of function names *Fnames*. In addition we assume some collection of function definitions, one for each function name used; we call such a collection a *declaration set*. A typical declaration set would look like:

$$
\begin{array}{rcl}
\mathsf{even}(x) & \Leftarrow & \texttt{if}\ x = 0\ \texttt{then}\ 0\ \texttt{else}\ \mathsf{minus}(\mathsf{odd}(\mathsf{minus}(x, 1)), 1) \\
\mathsf{odd}(x) & \Leftarrow & \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ \mathsf{even}(\mathsf{minus}(x, 1)) + 1 \\
\mathsf{minus}(x, y) & \Leftarrow & \texttt{if}\ x = y\ \texttt{then}\ \mathtt{0}\ \texttt{else} \\
& & \qquad \texttt{let}\ z = \mathsf{minus}(x, y + 1)\ \texttt{in}\ (1 + z) \\
\mathsf{less}(x, y) & \Leftarrow & \texttt{if}\ x = \mathtt{0}\ \texttt{then}\ \mathtt{tt}\ \texttt{else}\ \texttt{if}\ y = \mathtt{0}\ \texttt{then}\ \mathtt{ff}\ \texttt{else} \\
& & \qquad \mathsf{less}(\mathsf{minus}(x, 1), \mathsf{minus}(y, 1)) \\
\mathsf{rem}(x, y) & \Leftarrow & \texttt{if}\ (\mathsf{less}(x, y)\ \mathsf{and}\ \neg\ x = y)\ \texttt{then}\ x\ \texttt{else} \\
& & \qquad \mathsf{rem}(\mathsf{minus}(x, y), y)
\end{array}
\tag{4.8}
$$

Notice that mutual dependencies are allowed. For example less uses a call to the function minus in its body, even calls odd, which in turn uses a call to even. In the sequel we use $D_{\text{ex}}$ to refer to this specific set of declarations.

The format for declaration sets is given in Figure 4.11. The first sanity condition (1) says that the declaration set must be self-contained; any function named used must have an associated declaration. The second (2) merely says that whenever formal parameters are replaced in the body of any declaration, the resulting expression is closed, that is contains no free variables. The third says that each function symbol $f$ has an explicit number associated with it, telling how many arguments it expects. This is often refered to as the *arity* of $f$.

## 4.4.1 Big-step semantics

*Fpl* is an extension of the language $Exp_B$, and thus we have the same problems with the management of free and bound variables as before. But the techniques developed in Section 4.1 are easily extended to the larger language. For example to calculate the set of free-variables of an expression $E$ from *Fpl* we just add to the clauses in the definition of $fv(E)$ given in Section 4.1 the extra clause

(iv) $fv(f(E_1, \ldots, \ldots, E_n)) = fv(E_1) \cup \ldots \cup fv(E_n)$

while for substitution we add

(vi) $f(E_1, \ldots, E_n)\{P/x\} = f(E_1\{P/x\}, \ldots, E_n\{P/x\})$

As this suggests we will continue to use $P$ to denote an arbitrary program from *Fpl*, that is an expression with no free variables.

The result of evaluating a program $P$ in *Fpl* depends on the declaration set associated with the function names. For if this contains the declaration

$$\mathsf{arb}(x, y) \quad \Leftarrow \mathtt{if}\ x = y\ \mathtt{then}\ \mathtt{tt}\ \mathtt{else}\ x$$

then the result of evaluating the expression $\mathsf{arb}(\mathtt{tt}, \mathtt{ff})$ should be $\mathtt{tt}$, while if it contains the declaration

$$\mathsf{arb}(x, y) \quad \Leftarrow \mathtt{if}\ x = y\ \mathtt{then}\ \mathtt{ff}\ \mathtt{else}\ y$$

then the result should be $\mathtt{ff}$.

Consequently the judgements for the big-step semantics for *Fpl* take the form

$$P \Downarrow_D \mathsf{v}$$

where $P$ is a program, $D$ is a declaration set and $\mathsf{v}$ is a value. The inference rules for all of the program constructs except function calls are inherited from $Exp_B$; indeed the only interest is in how to evaluate calls to user-declared functions.

Consider for example the evaluation of the expression $\mathsf{minus}(5, 4)$ relative to the declaration set $D_{ex}$ given above. The obvious evaluation rule should dictate:

(1) Look up the declaration of the function minus.

(B-VAL)
$$\frac{}{\mathtt{v} \Downarrow^{\alpha}_{D} \mathtt{v}}$$

(B-ADD)
$$\frac{P_1 \Downarrow^{\alpha}_{D} \mathtt{n}_1 \qquad P_2 \Downarrow^{\alpha} \mathtt{n}_2}{(P_1 + P_2) \Downarrow^{\alpha}_{D} \mathtt{n}_3} \quad n_3 = \mathsf{add}(n_1, n_2)$$

(B-LET)
$$\frac{\begin{array}{l} P \Downarrow^{\alpha}_{D} \mathtt{m} \\ E\{\mathtt{m}/x\} \Downarrow^{\alpha}_{D} \mathtt{n} \end{array}}{\mathtt{let}\ x = P\ \mathtt{in}\ E \Downarrow^{\alpha}_{D} \mathtt{n}}$$

(B-MULT)
$$\frac{P_1 \Downarrow^{\alpha} \mathtt{n}_1 \qquad P_2 \Downarrow^{\alpha} \mathtt{n}_2}{(P_1 \times P_2) \Downarrow^{\alpha}_{D} \mathtt{n}_3} \quad n_3 = \mathsf{mult}(n_1, n_2)$$

(B-AND)
$$\frac{P_1 \Downarrow^{\alpha}_{D} \mathtt{bv}_1, \quad P_2 \Downarrow^{\alpha}_{D} \mathtt{bv}_2}{P_1\ \mathsf{and}\ P_2 \Downarrow^{\alpha}_{D} \mathtt{bv}} \quad \mathsf{bv} = \mathsf{conj}(\mathsf{bv}_1, \mathsf{bv}_2)$$

(B-NOT)
$$\frac{P \Downarrow^{\alpha} \mathtt{bv}}{\neg\, P \Downarrow^{\alpha}_{D} \mathtt{bv}'} \quad bv' = \mathsf{neg}\, bv$$

(B-EQ.T)
$$\frac{P_1 \Downarrow^{\alpha}_{D} \mathtt{bv}_1, \quad P_2 \Downarrow^{\alpha}_{D} \mathtt{bv}_2}{P_1 = P_2 \Downarrow^{\alpha}_{D} \mathtt{tt}} \quad bv_1 = bv_2$$

(B-EQ.F)
$$\frac{P_1 \Downarrow^{\alpha}_{D} \mathtt{bv}_1, \quad P_2 \Downarrow^{\alpha}_{D} \mathtt{bv}_2}{P_1 = P_2 \Downarrow^{\alpha}_{D} \mathtt{ff}} \quad bv_1 \neq bv_2$$

(B-IF.T)
$$\frac{P \Downarrow^{\alpha}_{D} \mathtt{tt}, \quad P_2 \Downarrow^{\alpha}_{D} \mathtt{v}}{\mathtt{if}\ P\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2 \Downarrow^{\alpha}_{D} \mathtt{v}}$$

(B-IF.F)
$$\frac{P \Downarrow^{\alpha}_{D} \mathtt{ff}, \quad P_2 \Downarrow^{\alpha}_{D} \mathtt{v}}{\mathtt{if}\ P\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2 \Downarrow^{\alpha}_{D} \mathtt{v}}$$

(B-EAGER)
$$\frac{\begin{array}{l} P_1 \Downarrow^{\mathtt{E}}_{D} \mathtt{v}_1, \ldots, P_k \Downarrow^{\mathtt{E}}_{D} \mathtt{v}_k \\ F\{\mathtt{v}_1/x_1\} \ldots \{\mathtt{v}_k/x_k\} \Downarrow^{\mathtt{E}}_{D} \mathtt{v} \end{array}}{f(P_1, \ldots, P_k) \Downarrow^{\mathtt{E}}_{D} \mathtt{v}} \quad D(f(x_1, \ldots, x_k)) = F$$

(B-LAZY)
$$\frac{F\{\mathtt{P}_1/x_1\} \ldots \{\mathtt{P}_k/x_k\} \Downarrow^{\mathtt{L}}_{D} \mathtt{v}}{f(P_1, \ldots, P_k) \Downarrow^{\mathtt{L}}_{D} \mathtt{v}} \quad D(f(x_1, \ldots, x_k)) = F$$

Figure 4.12: Big-step semantics of *Fpl*

(2) Substitute the actual parameters 5 and 4, in for the formal parameters $x$ and $y$, respectively, in the body of the definition to obtain a program.
(3) evaluate this resulting program.

So we end up evaluating the program

$$\mathtt{if}\ 5 \le 4\ \mathtt{then}\ 0\ \mathtt{else}\ \mathsf{less}(\mathsf{minus}(5, 1), \mathsf{minus}(4, 1))$$

However suppose we have to evaluate the more complicated expression $\mathsf{minus}(\mathsf{rem}(24, 7), \mathsf{even}(8))$. Here the arguments to the function $\mathsf{minus}(-, -)$ are not values but other programs,

$\mathsf{rem}(24, 7)$ and $\mathsf{even}(8)$ respectively. There are (at least) two reasonable strategies to follow:

**Eager:** Here the parameters are evaluated before the function is called:

(1) First evaluate the actual parameters $\mathsf{rem}(24, 7)$ and $\mathsf{even}(8)$ to values; in this case we will get $\mathtt{3}$ and $\mathtt{0}$ respectively.
(2) Substitute the resulting values, in this case $\mathtt{3}$ and $\mathtt{0}$, in for the formal parameters $x$ and $y$, respectively, in the body of the definition to obtain a program.
(3) evaluate this resulting program.

In the end we have to evaluate the program

$$\begin{aligned}&\mathtt{if}\ \mathtt{3} = \mathtt{0}\ \mathtt{then}\ \mathtt{tt}\ \mathtt{else}\ \mathtt{if}\ \mathtt{0} = \mathtt{0}\ \mathtt{then}\ \mathtt{ff}\ \mathtt{else}\\ &\qquad\qquad \mathtt{let}\ z = \mathsf{minus}(3, 1 + 1)\ \mathtt{in}\ (1 + z)\end{aligned}$$

**Lazy:** Here the actual parameters are not evaluated prior to the call to the function:

(1) Look up the declaration of the function $\mathsf{minus}$.
(2) Substitute the programs $\mathsf{rem}(24, 7)$ and $\mathsf{even}(8)$ in the body of the definition for the formal parameters $x$ and $y$ respectively.
(3) Evaluate the resulting program.

Here we end up evaluating

$$\begin{aligned}&\mathtt{if}\ \mathsf{rem}(24, 7) = \mathtt{0}\ \mathtt{then}\ \mathtt{tt}\ \mathtt{else}\\ &\qquad\qquad \mathtt{if}\ \mathsf{even}(8) = \mathtt{0}\ \mathtt{then}\ \mathtt{ff}\ \mathtt{else}\\ &\qquad\qquad \mathtt{let}\ z = \mathsf{minus}(\mathsf{rem}(24, 7), \mathsf{even}(8) + 1)\ \mathtt{in}\ (1 + z)\end{aligned}$$

Thus for *Fpl* we have two different big-step semantics, one for each of these evaluation strategies. The judgements are of the form

(i) $P \Downarrow^{\mathrm{E}}_D \mathsf{v}$: eager evaluation
(ii) $P \Downarrow^{\mathrm{L}}_D \mathsf{v}$: lazy evaluation

The inference rules are given in Figure 4.12, where all of the rules inherited from $Exp_B$ are common to both strategies; in these we use the annotation $\alpha$ to indicate either the eager or the lazy judgement. The only difference is in the evaluation of function application, with the rules (B-EAGER) and (LAZY); in these rules the notation $D(f(x_1, \ldots, x_k)) = F$ means that the declaration set $D$ contains a declaration of the form $f(x_1, \ldots, x_k) \Leftarrow F$ for some term $F$.

An example inference of the judgement

$$\mathsf{minus}(4, 3) \Downarrow^{\mathrm{E}}_D 1$$

is given in Figure 4.13; to save space we have omitted some applications of the trivial rule (B-NUM) and we have used $B(\mathtt{n}_1, \mathtt{n}_2)$ as an abbreviation of the program

$$\mathtt{if}\ \mathtt{n}_1 = \mathtt{n}_2\ \mathtt{then}\ \mathtt{0}\ \mathtt{else}\ \mathtt{let}\ z = \mathsf{minus}(\mathtt{n}_1, \mathtt{n}_2 + 1)\ \mathtt{in}\ (1 + z),$$

$$\dfrac{\dfrac{\dfrac{\cdots}{(4 = 3) \Downarrow^{\mathrm{E}}_{D} \mathtt{ff}}\,\textsc{(b-eq)} \qquad \dfrac{\dfrac{\dfrac{}{(3 + 1) \Downarrow^{\mathrm{E}}_{D} 4}\,\textsc{(b-add)} \quad \dfrac{\dfrac{}{(4 = 4) \Downarrow^{\mathrm{E}}_{D} \mathtt{tt}}\,\textsc{(b-eq)} \quad \dfrac{}{\mathtt{0} \Downarrow^{\mathrm{E}}_{D} \mathtt{0}}\,\textsc{(b-num)}}{B(4, 4) \Downarrow^{\mathrm{E}}_{D} \mathtt{0}}\,\textsc{(b-if.t)}}{\mathsf{ms}(4, 3 + 1) \Downarrow^{\mathrm{E}}_{D} \mathtt{0}}\,\textsc{(b-eager)} \quad \dfrac{\cdots}{(1 + \mathtt{0}) \Downarrow^{\mathrm{E}}_{D} 1}\,\textsc{(b-add)}}{\mathtt{let}\ z = \mathsf{ms}(4, 3 + 1)\ \mathtt{in}\ 1 + z \Downarrow^{\mathrm{E}}_{D} 1}\,\textsc{(b-let)}}{\dfrac{B(4, 3) \Downarrow^{\mathrm{E}}_{D} 1}{\mathsf{ms}(4, 3) \Downarrow^{\mathrm{E}}_{D} 1}\,\textsc{(b-eager)}}\,\textsc{(b-if.f)}$$

Figure 4.13: Example big-step derivation for *Fpl*

that is the body of the declaration of $\mathsf{minus}$ in which the actual values $\mathtt{n}_1$ and $\mathtt{n}_2$ replace the formal parameters $x$ and $y$ respectively. We also abbreviate $\mathsf{minus}(-, -)$ with $\mathsf{ms}(-, -)$.

With the introduction of user-defined functions into the language we lose the property of strong normalisation; that is we now longer have the property that for every program $P$ there exists a value $\mathtt{v}$ such that $P \Downarrow^{\mathrm{E}}_{D} \mathtt{v}$ ( or indeed or $P \Downarrow^{\mathrm{L}}_{D} \mathtt{v}$). As a simple example consider the declaration

$$\mathsf{loop}(x) \quad \Leftarrow \mathtt{if}\ x = \mathtt{0}\ \mathtt{then}\ \mathtt{tt}\ \mathtt{else}\ \mathsf{loop}(x + 1)$$

Then it is easy to derive the judgement $\mathsf{loop}(\mathtt{0}) \Downarrow^{\mathrm{E}}_{D} \mathtt{tt}$. But to derive $\mathsf{loop}(1) \Downarrow^{\mathrm{E}}_{D} \mathtt{tt}$ it would be necessary to first derive $\mathsf{loop}(2) \Downarrow^{\mathrm{E}}_{D} \mathtt{tt}$; this in turn would require a prior derivation of $\mathsf{loop}(3) \Downarrow^{\mathrm{E}}_{D} \mathtt{tt}$, and so on. In other words if $n > 0$ then there is no value $\mathtt{v}$ such that the judgement $\mathsf{loop}(\mathtt{n}) \Downarrow^{\mathrm{E}}_{D} \mathtt{v}$ can be derived using the inference rules of Figure 4.12.

**Exercise 34** *Give a formal proof of this property of the function* $\mathsf{loop}$*, for both the eager and lazy semantics. Consult the argument given on page 46 in Chapter 3.1.* □

**Exercise 35** *Relative to the set of declarations $D_{ex}$ in (4.8) above, does the expression* $\mathsf{minus}(x, y)$ *correctly implement the function* $\mathsf{monus}$ *from Exercise 33 ?*

Let us end this section with a brief comparison between the eager and lazy semantics. Refering back to the derivation in Figure 4.13, it is easy to adapt it so as to derive $\mathsf{minus}(4, 3) \Downarrow^{\mathrm{L}}_{D} 1$, and indeed the actual derivation would not be very different. However in general the derivations in the lazy semantics are much larger than in the eager semantics, because of the duplication of unevaluated programs. For example in the derivation of the judgement

$$\mathsf{minus}(\mathsf{rem}(13, 5), 2) \Downarrow^{\mathrm{L}}_{D} 1$$

the program $\mathsf{rem}(13, 5)$ is evaluated at least twice, once in the equality test, and again in the subsequent call to $\mathsf{minus}$. Indeed in this subsequent call to $\mathsf{minus}$ it will again be evaluated in the equality test, and again in the following call to $\mathsf{minus}$. We thus get a cascading number of re-evaluations of the original parameter $\mathsf{rem}(13, 5)$.

However it is important to realise that our inference rules merely give a reference formal semantics, and do not represent an implementation proposal. And in fact there are many standard implementation techniques for the lazy semantics which avoid this duplication of evaluations.

The lazy semantics also has certain advantages. First it can be used to simulate the eager semantics, using the $\mathtt{let}\ x = \ldots\ \mathtt{in}\ \ldots$ construct. For example to evaluate a program $P$ eagerly it is sufficient to replace each function call $f(Q_1, \ldots, Q_n)$ in $P$ with the program

$$\mathtt{let}\ x_1 = Q_1\ \mathtt{in}\ \ldots \mathtt{let}\ x_n = Q_n\ \mathtt{in}\ f(x_1, \ldots, x_n)$$

There are also certain cases in which the eager semantics provides no answer while lazy semantics returns a value. As a simple example suppose we have the declaration

$$\mathsf{proj2}(x, y) \Leftarrow y$$

Then it is easy to check that

$$\mathsf{proj2}(\mathsf{loop}(2), \mathbb{0}) \Downarrow^{\mathsf{L}}_D \mathbb{0}$$

whereas there is no value $\mathtt{v}$ such that $\mathsf{proj2}(\mathsf{loop}(2), \mathbb{0}) \Downarrow^{\mathsf{E}}_D \mathtt{v}$. The general problem is that in the eager semantics arguments to functions are always evaluated, whether or not they are actually needed.

Finally we can prove that the lazy semantics is guaranteed to provide any answers which the eager semantics can provide. First a preliminary result.

**Lemma 37** *Suppose $P \Downarrow^{\mathsf{L}}_D \mathtt{w}$. Then for every expression $E$ containing at most one free variable $x$, $E\{\mathtt{w}/x\} \Downarrow^{\mathsf{L}} \mathtt{v}$ implies $E\{P/x\} \Downarrow^{\mathsf{L}} \mathtt{v}$.*

**Proof:** Here we need to use rule induction. For the purposes of the proof let us fix a program $P$ and a value $\mathtt{w}$ such that $P \Downarrow^{\mathsf{L}}_D \mathtt{w}$. Then let $\mathcal{P}(E, \mathtt{v})$ be the predicate which says: $E\{P/x\} \Downarrow^{\mathsf{L}} \mathtt{v}$. We prove $E\{\mathtt{w}/x\} \Downarrow^{\mathsf{L}}_D \mathtt{v}$ implies $\mathcal{P}(E, \mathtt{v})$ by induction on the size of the derivation of the judgement $E\{\mathtt{w}/x\} \Downarrow^{\mathsf{L}}_D \mathtt{v}$ using the rules in Figure 4.12.

The proof now proceeds by an analysis of this derivation, and in particular the last rule used. In all there are eleven possibilities. One possibility is that the last rule used is (B-VAL), so that the expression $E$ is $x$ and so the values $\mathtt{v}$ and $\mathtt{w}$ coincide. In this case $\mathcal{P}(E, \mathtt{v})$ is trivial.

The most interesting case is when the rule (B-LAZY) is used. Here suppose for convenience that $E$ has the form $f(G)$, a function which takes only one argument. Then we know $f$ has a declaration in $D$, say $D(f(x_1)) = F$, and the judgement $F\{(G\{\mathtt{w}/x\})/x_1\} \Downarrow^{\mathsf{L}}_D \mathtt{v}$ can be derived; moreover the size of its derivation is strictly smaller than that of $f(G\{\mathtt{w}/x\}) \Downarrow^{\mathsf{L}}_D \mathtt{v}$. Also because $x_1$ is the only variable allowed in the declaration $F$, the expression $F\{(G\{\mathtt{w}/x\})/x_1\}$ can be written as $(F\{G/x_1\})\{\mathtt{w}/x\}$.

So we can apply induction to obtain $(F\{G/x_1\})\{P/x\} \Downarrow^{\mathsf{L}}_D \mathtt{v}$. Once more this can rewritten as $F\{(G\{P/x\})/x_1\}$, because $x$ can not appear in $F$, and so an application of the rule (B-LAZY) gives the required $f(G\{P/x\}) \Downarrow^{\mathsf{L}}_D \mathtt{v}$.

$$\frac{P_i \rightarrow^{\mathtt{E}}_D P_i', \; 1 \leq i \leq k}{f(P_1, \ldots, P_i, \ldots, P_k) \rightarrow^{\mathtt{E}}_D f(P_1, \ldots, P_i', \ldots, P_k)} \;\text{(S-EAGER.ARG)}$$

$$\frac{}{f(\mathtt{v}_1, \ldots, \mathtt{v}_k) \rightarrow^{\mathtt{E}}_D F\{^{\mathtt{v}_1}/_{x_1}\} \ldots \{^{\mathtt{v}_k}/_{x_k}\}} \;\text{(S-EAGER.APP)} \quad D(f(x_1, \ldots, x_k)) = F$$

$$\frac{}{f(\mathtt{P}_1, \ldots, \mathtt{P}_k) \rightarrow^{\mathtt{L}}_D F\{^{\mathtt{P}_1}/_{x_1}\} \ldots \{^{\mathtt{P}_k}/_{x_k}\}} \;\text{(S-LAZY)} \quad D(f(x_1, \ldots, x_k)) = F$$

Figure 4.14: Small-step semantics of *Fpl*: function application

There are still nine other cases to consider, but all are completely straightforward. □

**Exercise 36** *Show that the result in this lemma is also true for the eager semantics.* □

**Theorem 38** *For every program in Fpl, $P \Downarrow^{\mathtt{E}}_D \mathtt{v}$ implies $P \Downarrow^{\mathtt{L}}_D \mathtt{v}$.*

**Proof:** Here again we use rule induction. Let $\mathcal{P}(P, \mathtt{v})$ denote the predicate $P \Downarrow^{\mathtt{L}}_D \mathtt{v}$. We prove $P \Downarrow^{\mathtt{E}}_D \mathtt{v}$ implies $\mathcal{P}(P, \mathtt{v})$ by induction on the size of the derivation of the judgement $P \Downarrow^{\mathtt{E}}_D \mathtt{v}$.

The proof proceeds by an analysis of this derivation, and in particular the last rule used. In all there are eleven possibilities, corresponding to each of the rules in Figure 4.12. Here we examine only one, the most interesting case (B-EAGER): $P$ has the form $f(Q)$, the function $f$ has a declaration in $D$, that is $D(f(x)) = F$ for some $F$, and we know both $Q \Downarrow^{\mathtt{E}}_D \mathtt{w}$ can be derived for some value $w$, and $F\{^{\mathtt{v}}/_x\} \Downarrow^{\mathtt{E}}_D \mathtt{v}$ can also be derived; for the sake of simplicity we are assuming that the function $f$ only takes one argument.

At this point we can use induction, since the derivations of $Q \Downarrow^{\mathtt{E}}_D \mathtt{w}$ and $F\{^{\mathtt{w}}/_x\} \Downarrow^{\mathtt{E}}_D \mathtt{v}$ are strictly smaller than that of $f(Q) \Downarrow^{\mathtt{E}}_D \mathtt{v}$. This gives us

(a) $Q \Downarrow^{\mathtt{L}}_D \mathtt{w}$

(b) $F\{^{\mathtt{w}}/_x\} \Downarrow^{\mathtt{L}}_D \mathtt{v}$

But now we can apply the previous lemma to (b) to obtain $F\{^{Q}/_x\} \Downarrow^{\mathtt{L}}_D \mathtt{v}$, and an application of (B-LAZY) gives the required $f(Q) \Downarrow^{\mathtt{L}}_D \mathtt{v}$.

### 4.4.2 Small-step semantics

Here we also have two evaluation rules, for the eager and lazy strategies, and both are relative to a declaration set for function names. The semantics uses the judgements of the form

$$P \rightarrow^{\text{E}}_{D} Q \qquad\qquad P \rightarrow^{\text{L}}_{D} Q$$

respectively, and both inherit all the inference rules from Figure 4.6. We just need to add rules for evaluating function applications; these are given in Figure 4.14.

First let us consider eager evaluation. Intuitively to execute the program $f(P_1, \dots, P_n)$:

(i) Start evaluating any of the actual parameters $P_1, \dots, P_n$.

(ii) If all parameters $P_i$ are already evaluated, say to the values $\mathsf{v}_i$, look up the definition of $f$ in the declaration set, say F.

(iii) Substitute the actual value parameters $\mathtt{v_i}$ in for the corresponding formal parameter in the body of the declaration of $f$, namely $F$.

(iv) Start executing the resulting program.

The rule (s-eager.arg) in Figure 4.14 corresponds to (i) above, whereas (ii) - (iv) are formalised in the rule (s-eager.app).

Eager evaluation is easier. To execute $f(P_1, \dots, P_n)$:

(i) Look up the definition of $f$ in the declaration set, to obtain the declaration $F$.
(ii) Substitute the actual parameters $P_i$ in for the corresponding formal parameter in the body of the declaration of $f$, namely $F$.
(iii) Start executing the resulting program.

This is formalised in one rule, (s-lazy).

**Exercise 37** *Show that this small-step semantics for Fpl is consistent with the big-step semantics. That is prove, for $\alpha = \text{E}$ or $\text{L}$,*

- $P \Downarrow^{\alpha}_{D} \mathsf{v}$ *implies* $P \rightarrow^{\alpha *}_{D} \mathsf{v}$
- *Conversely,* $P \rightarrow^{\alpha *}_{D} \mathsf{v}$ *implies* $P \Downarrow^{\alpha}_{D} \mathsf{v}$. □

**Exercise 38** *The small-step eager semantics in Figure 4.14 does not really specify left-to-right evaluation, as the actual parameters to a function can be evaluated in any order. Modify the rules so that evaluation is from left to right.* □

## 4.4.3 Typing functions

The functional language *Fpl* is an extension of $Exp_B$, and so inherits all of the problems of run-time errors discussed in Section 4.2; moreover the introduction of functions introduces even more. For example the execution of the program $\mathsf{minus}(\mathsf{less}(1, 2), \mathsf{even}(3))$ will lead to a run-time error because the function $\mathsf{minus}$ expects both its arguments to be numerals, and $\mathsf{less}(1, 2)$ returns a Boolean. Luckily the approach of Section 4.3 to eliminating this errors via types is easily extended to the current language *Fpl*.

In $Exp_B$ there are already function symbols, such as + and $\mathsf{and}$, and the type inference rules in Figure 4.8 dictate the manner in which these operations may be applied in

**Types:**

$$\begin{array}{rcl} \tau & ::= & \mathsf{base} \mid \tau_f \\ \mathsf{base} & ::= & \mathsf{int} \mid \mathsf{bool} \\ \tau_f \in \mathsf{function} & ::= & (\mathsf{base}_1, \ldots, \mathsf{base}_k) \to \mathsf{base},\ k \geq 1 \end{array}$$

**Type environments:**

$$\Gamma \ ::= \ \mid \Gamma,\ x : \mathsf{base} \ \mid \ \Gamma,\ f : \tau_f$$

**Environment look-up:**

$$\text{(TY-LOOK1)}\quad \frac{}{\Gamma, u:\tau \vdash u:\tau} \qquad\qquad \text{(TY-LOOK2)}\quad \frac{\Gamma \vdash u_2:\tau_2}{\Gamma, u_1:\tau \vdash u_2:\tau_2}\ u_1 \neq u_2$$

Figure 4.15: Types and environments for *Fpl*

expressions. For example the rule (TY-A.OP) says that the function + may only be applied to arguments which are, or evaluate to, numerals, while (TY-B.OP) says that and may only be applied to Booleans. But (TY-A.OP) also dictates that expressions formed with +, that is of the form $(E_1 + E_2)$ can only be used where arithmetic expressions are expected. So for example $(E_1 + E_2) \times 3$ is allowed, whereas $\neg\,(E_1 + E_2)$ is incorrect.

We need similar rules for each of the function symbols used in *Fpl.* For each $f$ we need to know:

(i) the type of arguments to which it can be applied, the input types, and their number
(ii) the type of value an application of $f$ returns, its output type.

For example we would expect that less should only be applied to two arguments, each of type int, and that it will return a value of type bool. So we associate with less the type $(\mathsf{int}, \mathsf{int}) \to \mathsf{bool}$. More generally we need to associate with each function symbol a type of the form

$$(\mathsf{base}_1, \ldots, \mathsf{base}_k) \to \mathsf{base}$$

where $\mathsf{base}_i$ are the input types and base is the output type; incidently here $k$ is the *arity* of the function symbol, dictating how many arguments it expects. Since we only have two kinds of values in the language, all of these types can only be either int for numerals, or bool for Booleans. This revision of types, and type environments required for *Fpl* is reported in Figure 4.15; the environment look-up rules are inherited directly from Figure4.7.

In principle one might try to calculate the appropriate types for the function symbols from the corresponding definitions in a declaration set $D$. But here we avoid this algorithmic issue and confine our attention to typechecking; that is given a proposed set of types for the function symbols how do we check that there are in fact appropriate. So we extend the type inference systems from Section 4.3.1 to *Fpl.* As before we will

$$\text{(TY-APP)}\quad \frac{\begin{array}{l}\Gamma \vdash P_1 : \tau_1\\ \dots\\ \Gamma \vdash P_k : \tau_k\\ \Gamma \vdash f : (\tau_1, \dots \tau_k) \to \tau\end{array}}{\Gamma \vdash f(P_1, \dots, P_k) : \tau} \qquad\qquad \text{(TY-DEC)}\quad \frac{\begin{array}{l}x_1 : \tau_1, \dots, x_k : \tau_k \vdash F : \tau\\ \Gamma \vdash f : (\tau_1, \dots \tau_k) \to \tau\end{array}}{\Gamma \vdash f(x_1, \dots, x_k) \Leftarrow F}$$

Figure 4.16: Typing functions

$$\dfrac{\dfrac{\dots}{\Gamma_{\text{ex}} \vdash \mathsf{less} : (\mathsf{int}, \mathsf{int}) \to \mathsf{bool}}\,\text{(TY-LP)} \quad \dfrac{\begin{array}{l}\dfrac{}{\Gamma_{\text{ex}} \vdash \mathsf{ms} : (\mathsf{int}, \mathsf{int}) \to \mathsf{int}}\,\text{(TY-LP)}\\ \Gamma_{\text{ex}} \vdash 3 : \mathsf{int}\\ \Gamma_{\text{ex}} \vdash 2 : \mathsf{int}\end{array}}{\Gamma_{\text{ex}} \vdash \mathsf{ms}(3, 2) : \mathsf{int}}\,\text{(TY-APP)} \quad \dfrac{\begin{array}{l}\dfrac{}{\Gamma_{\text{ex}} \vdash \mathsf{rem} : (\mathsf{int}, \mathsf{int}) \to \mathsf{int}}\,\text{(TY-LP)}\\ \Gamma_{\text{ex}} \vdash 7 : \mathsf{int}\\ \Gamma_{\text{ex}} \vdash 4 : \mathsf{int}\end{array}}{\Gamma_{\text{ex}} \vdash \mathsf{rem}(7, 4) : \mathsf{int}}\,\text{(TY-APP)}}{\Gamma_{\text{ex}} \vdash \mathsf{less}(\mathsf{ms}(3, 2), \mathsf{rem}(7, 4)) : \mathsf{bool}}\,\text{(TY-APP)}$$

Figure 4.17: An example type inference for *Fpl*

have judgements of the form

$$\Gamma \vdash E$$

where $E$ is an expression in *Fpl* and $\Gamma$ is an environment which contains type associations for function names, in addition to those for variables, as in Section 4.3.1. The rules for forming type environments, and how to look-up type associations, are given in Figure 4.15; these are a very minor extension of the the corresponding rules for $Exp_B$, in Figure 4.7. Note that in Figure 4.15 $u$ is a meta-variable which now ranges over both variables $x$ and function symbols $f$.

The type inference system for *Fpl* uses all of the rules for $Exp_B$ in Figure 4.8; we only need an extra rule for typing the use of function symbols. The rule (TY-APP) in Figure 4.16 says that in order to assign a type to a function application

$$\Gamma \vdash f(P_1, \dots, P_k) : \tau$$

it is necessary to

(i) assign a type to the functions symbol, $\Gamma \vdash f : (\tau_1, \dots \tau_k) \to \tau$

(ii) and assign to each of the arguments to the function the appropriate type, $\Gamma \vdash P_i : \tau_i$.

**Example 39** *Suppose $\Gamma_{ex}$ contains the following type associations:*

$$\begin{aligned}\mathsf{even} &: \mathsf{int} \to \mathsf{int}\\ \mathsf{odd} &: \mathsf{int} \to \mathsf{int}\\ \mathsf{minus} &: (\mathsf{int}, \mathsf{int}) \to \mathsf{int}\\ \mathsf{less} &: (\mathsf{int}, \mathsf{int}) \to \mathsf{bool}\\ \mathsf{rem} &: (\mathsf{int}, \mathsf{int}) \to \mathsf{int}\end{aligned}$$

*Then we can use the inference rules to derive the judgement*

$$\Gamma_{ex} \vdash \mathsf{less}(\mathsf{minus}(\mathtt{3}, \mathtt{2}), \mathsf{rem}(\mathtt{10}, \mathtt{4})) : \mathsf{bool}$$

*The structure of the derivation is given in Figure 4.17, where again we abbreviate* $\mathsf{minus}(-, -)$ *with* $\mathsf{ms}(-, -)$*: We also use* (TY-LP) *to refer to the repeated use of the look-up rules* (TY-LOOK1) *and* (TY-LOOK2) *in order to scan the type environment from right to left to search for an entry, as in Section 4.3.1. We have also omitted explicit references to applications of the simple rule* (TY-INT). □

However the single rule (TY-APP) is not sufficient. We also have to ensure that when a call is actually made to a function the subsequent behaviour is in accord with its declared type. For example when we call the function $\mathsf{less}$ on the arguments $\mathsf{minus}(\mathtt{3}, \mathtt{2})$ and $\mathsf{rem}(\mathtt{7}, \mathtt{4})$ we have to ensure that the body, or the declaration associated with the function, treats the arguments as integers, and returns a Boolean value as a result. In other words we have to also typecheck the function definitions.

The rule (TY-DEC) in Figure 4.16 says that in order to typecheck a declaration

$$\Gamma \vdash f(x_1, \ldots, x_k) \Leftarrow F$$

it is necessary to:

(i) have a type association for the function symbol, $\Gamma \vdash f : (\tau_1, \ldots \tau_k) \to \tau$
(ii) ensure that the type of body of the function $F$ accords with this type. This means that, assuming the formal parameters $x_i$ have the appropriate type $\tau_i$, the body $F$ can be assigned the correct output type, namely $\tau$.

For example consider the declaration of the function $\mathsf{less}$ in the declaration set $D_{\text{ex}}$:

$$\mathsf{less}(x, y) \Leftarrow \mathtt{if}\ x = \mathtt{0}\ \mathtt{then}\ \mathtt{tt}\ \mathtt{else}\ \mathtt{if}\ y = \mathtt{0}\ \mathtt{then}\ \mathtt{ff}\ \mathtt{else}$$
$$\mathsf{less}(\mathsf{minus}(x, \mathtt{1}), \mathsf{minus}(y, \mathtt{1}))$$

This declaration is well-typed relative to the environment $\Gamma_{\text{ex}}$, as the outline derivation of the judgement

$$\Gamma_{\text{ex}} \vdash \mathsf{less}(x, y) \Leftarrow F$$

in Figure 4.18 demonstrates. $F$ is used as an abbreviation for the body of $\mathsf{less}$, which in turn uses $G$ as a further abbreviation for the inner `if then else` construct; we also use the standard abbreviation $\mathsf{ms}(-, -)$ for $\mathsf{minus}(-, -)$.

It is appropriate, when working with expressions using function symbols which are defined in a declaration set $D$, to build in to the typechecking inference a check that all of the function declarations are well-typed. So let us introduce some notation for this.

$$
\dfrac{
\begin{array}{c}
\dfrac{}{\Gamma_{\text{xy}} \vdash x = \mathbb{0} : \mathsf{bool}}\,(\textsc{ty-eq}) \\
\dfrac{}{\Gamma_{\text{xy}} \vdash \mathtt{tt} : \mathsf{bool}}\,(\textsc{ty-bool})
\end{array}
\quad
\dfrac{
\begin{array}{c}
\dfrac{}{\Gamma_{\text{xy}} \vdash y = \mathbb{0} : \mathsf{bool}}\,(\textsc{ty-eq}) \\
\dfrac{}{\Gamma_{\text{xy}} \vdash \mathtt{ff} : \mathsf{bool}}\,(\textsc{ty-bool})
\end{array}
\quad
\dfrac{
\dfrac{
\begin{array}{l}
\dfrac{}{\Gamma_{\text{xy}} \vdash \mathsf{ms} : (\mathsf{int}, \mathsf{int}) \to \mathsf{int}}\,(\textsc{ty-lp}) \\
\Gamma_{\text{xy}} \vdash x : \mathsf{int} \\
\Gamma_{\text{xy}} \vdash 1 : \mathsf{int}
\end{array}
}{\Gamma_{\text{xy}} \vdash \mathsf{ms}(x, 1) : \mathsf{int}}\,(\textsc{ty-app})
\quad
\dfrac{
\begin{array}{l}
\dfrac{}{\Gamma_{\text{ex}} \vdash \mathsf{ms} : (\mathsf{int}, \mathsf{int}) \to \mathsf{int}}\,(\textsc{ty-lp}) \\
\Gamma_{\text{xy}} \vdash y : \mathsf{int} \\
\Gamma_{\text{xy}} \vdash 1 : \mathsf{int}
\end{array}
}{\Gamma_{\text{xy}} \vdash \mathsf{ms}(y, 1) : \mathsf{int}}\,(\textsc{ty-app})
}{\Gamma_{\text{xy}} \vdash \mathsf{less}(\mathsf{ms}(x, 1), \mathsf{ms}(y, 1)) : \mathsf{bool}}
}{\Gamma_{\text{xy}} \vdash G : \mathsf{bool}}\,(\textsc{ty-if})
}{\Gamma_{\text{xy}} \vdash F : \mathsf{bool}}\,(\textsc{ty-if})
\quad
\dfrac{}{\Gamma_{\text{ex}} \vdash \mathsf{less} : (\mathsf{int}, \mathsf{int}) \to \mathsf{bool}}\,(\textsc{ty-lp})
}{\Gamma_{\text{ex}} \vdash \mathsf{less}(x, y) \Leftarrow F}\,(\textsc{ty-dec})
$$

Abbreviations:

- $F$ stands for `if` $x = \mathbb{0}$ `then` `tt` `else` $G$
- $G$ stands for `if` $y = \mathbb{0}$ `then` `ff` `else` $\mathsf{less}(\mathsf{ms}(x, 1), \mathsf{ms}(y, 1))$
- $\Gamma_{\text{xy}}$ stands for the environment $\Gamma_{\text{ex}}, x : \mathsf{int}, y : \mathsf{int}$

Figure 4.18: Typechecking a declaration

---

**Definition 40** *We write $\Gamma \vdash_D E$ whenever*

*(i)* $\Gamma \vdash E$
*(ii)* $\Gamma \vdash f(x_1, \ldots, x_n) \Leftarrow F$*, for every function declaration in the declaration set D.* □

It is this form of typing judgement which should be used for programs in the language *Fpl*. Moreover the results in Section 4.3.2 for the typing judgements for the language *Exp*$_B$ can be extended to *Fpl* without too much difficulty.

**Exercise 39** *Prove a* Progress *result for this typechecking system for Fpl, corresponding to Theorem 34. That is for every program P in Fpl which is not a value, prove that* $\vdash_D P : \tau$ *implies there is some program Q such that* $P \to_D^{\alpha} Q$*. The proof should apply to both the eager and the lazy evaluation strategies.* □

**Exercise 40** *Prove a* Preservation *result for Fpl, corresponding to Theorem 35.* □

To prove these exercises it will be necessary to generalise *Sanity*, Proposition 31, and the *Substitution lemma*, Theorem 33, in Section 4.3.2 to *Fpl*. □

# Chapter 5

# Functions as data

The language *Fpl* of the previous chapter leaves much to be desired as a prototypical core functional programming language. There is limited access for the programmer to function definitions. All required functions must first be defined globally as part of a *declaration* rather than being declared dynamically as required as part of a program. More importantly the type of functions allowed are very restrictive. They can only manipulate data of basic types; more specifically for *Fpl* data of type $\mathsf{int}$ and $\mathsf{bool}$. But much of the power of functional programming languages comes from functions being able to manipulate other functions as data.

Here we address these issues by considering a language *Lambda* in which functions can act on and return other functions as results, in much the same way as the functions in *Fpl* can use integers and Booleans. The essential ingredient is a notation, called *lambda notation* for describing functions, which we first describe informally. In mathematics there are numerous methods for describing functions. For example suppose we wanted to describe a function which takes in an integer $n$ and returns the result of adding 2 to it. This could be described by a diagram such as

$$x \ \longmapsto x+2 \tag{5.1}$$

Here $x$ on the left hand side, stands for an arbitrary input; the $\longmapsto$ refers in some intuitive way to *is operated on by ...* and the mathematical formula $x+2$ describes the transformation which needs to take place on the input $x$ in order to obtain the output. In our language this function will be described by the new expression

$$\lambda x\,.x + \mathsf{2} \tag{5.2}$$

which can be understood in much the same manner: $x$ in $\lambda x\,.-$ represents the argument to the function while the $x+\mathsf{2}$ describes what needs to be done to the argument in order to produce the result. As another example consider

$$\lambda f\,.f\;\mathsf{3} \tag{5.3}$$

This also describes a function. It takes an argument, represented by $f$; presumably by the use of this particular variable we are hinting the argument is going to be treated as

$$\begin{array}{rcl}
M \in \mathit{Lambda} & ::= & v \\
 & & \mid (M_1 + M_2) \\
 & & \mid M_1 \text{ and } M_2 \mid M_1 \text{ or } M_2 \mid M_1 = M_2 \mid M_1 < M_2 \\
 & & \mid \texttt{if } M_1 \texttt{ then } M_2 \texttt{ else } M_3 \mid \texttt{let } x = M_1 \texttt{ in } M_2 \\
 & & \mid M_1 M_2 \\
v \in \mathit{Val} & ::= & x \in \mathit{Vars} \mid \texttt{n} \in \mathit{Nums} \mid \texttt{tt} \mid \texttt{ff} \mid \lambda x.M
\end{array}$$

Figure 5.1: The language *Lambda*

a function. And indeed the formaula for describing the result to be returned reenforces this intuition; the result is to be obtained by applying the argument, represented by $f$, to the value 3.

One use of the function (5.2) is to apply it to the function described in (5.2). In our new language this will be represented by the expression

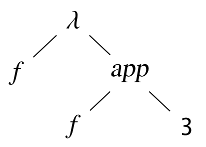
$$(\lambda f.f\ 3)\ (\lambda x.x + 2) \tag{5.4}$$

This is beginning to look a bit complicated; but in fact it is just another instance of a function, in this case $\lambda f.f(3)$, being applied to an argument, in this case $\lambda x.x+2$. This *function application* will be a central ingredient of our new language. In (5.3) it was used in the expression $f(3)$ to represent what should happen to the function $f$; in (5.4) it is used in exactly the same manner, except the actual function description is a little more complicated, as is the argument.

## 5.1 The language *Lambda*

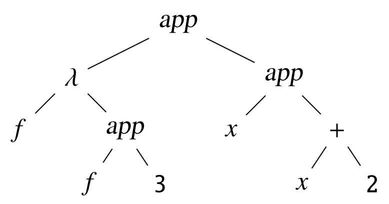
The starting point for our new language is the simple language $Exp_B$ from Chapter 4.2 rather than the more complicated *Fpl* from Chapter 4.4. The abstract syntax for expressions in *Lambda* is given in Figure 5.1 and has only two extra clauses to that for $Exp_B$ from Figure 4.5. The first states that for any expression $M$, the expression $\lambda x.M$ is a value in the language. As explained in the Introduction to this chapter this represents a function; we designate it to be a value rather than simply an expression as we want such functions to be treated in exactly the same way as the other values in the language `tt`, `ff`, `0`, `1`, .... The second new construct is for *function application*, $M_1\ M_2$. Here the intention is that the expression $M_1$ should evaluate to a function, and that resulting function be applied to $M_2$, or possibly whatever value to which $M_2$ evaluates.

It is important to keep in mind that the BNF definition in Figure 5.1 describes the abstract syntax of the language; to make sense of most expressions we will need to either describe them as an abstract syntax tree, or to describe concrete linear representations using brackets as necessary to describe their structure. Thus



(5.5)

is a value in *Lambda* which we have represented linearly in (5.3) above as the expression $\lambda f\,.f(3)$. Note that in the abstract syntax tree (5.5) above we have deliberately introduced a token *app* to represent explicitly the function application which is only implicit in the expression $f\,3$; more precisely function application is represented implicitly by the white space between the $f$ and the 3. Similarly the expression represented linearly in (5.4) above is given in terms of abstract syntax by:



(5.6)

Our intention is to give a small-step semantics to *Lambda* by building on that of in Chapter 4.2. Recall from there that the presence of variables caused considerable problems, due to the presence of local declarations, Chapter 4.1. For example the expression

$$\texttt{let}\; z = 2 + 3\; \texttt{in}\; (z + z + y),$$

which is also a valid expression in *Lambda*, could not be evaluated to any value; in the body, $z+z+y$, although we know that $z$ will be replaced by the value 5 we do not know to what $y$ refers. Similarly the expression

$$(\lambda x\,.x + 2 + z)\,4$$

will not be meaningful because when the function $\lambda x\,.x{+}2{+}z$ is applied to the argument 4 we do not know what $z$ stands for.

The solution is just an extension of that used in Chapter 4.1; we need to take into account occurrences of free and bound variables.

**Definition 41 (Free variables)** *For every M in Lambda, let fv(M) be the set of variables defined by structural induction as follows:*

*(i)* $fv(M_1 M_2) = fv(M_1) \cup fv(M_2)$
*(ii)* $fv(\lambda x\,.M) = fv(M) - \{x\}$
*(iii)* $fv(x) = \{x\}$ $\qquad fv(\texttt{n}) = fv(\texttt{tt}) = fv(\texttt{ff}) = \{\}$
*(iv)* $fv(\texttt{let}\; y = M_1\; \texttt{in}\; M_2) = fv(M_1) \cup (fv(M_2) - \{y\})$
*(v)* $fv(M_1\; \texttt{op}\; M_2) = fv(M_1) \cup fv(M_2)$, *where* op *ranges all of the arithmetic and Boolean operators*

*(vi)* $fv(\texttt{if } M_1 \texttt{ then } M_2 \texttt{ else } M_3) = fv(M_1) \cup fv(M_2) \cup fv(M_3)$ □

Note that only the clauses (i) and (ii) are new; the other four clauses are inherited directly from the corresponding definition in the previous Chapter.

**Exercise 41** *Calculate the set of free variables for the following expressions:*

*(i)* $\lambda x\,.\texttt{let } y = \texttt{2} + x \texttt{ in } f\,(f\,y)$

*(ii)* $\texttt{let } x = \lambda y\,.x\,y \texttt{ in } x\,(y + \texttt{2})$

*(iii)* $\lambda x\,.\texttt{let } x = x + \texttt{2 in } (\texttt{let } y = f\ x \texttt{ in } \lambda f\,.((f\ y)\ x))$ □

We also extend the other concepts originally developed in Chapter 4.1 for the language $Exp_{\mathrm{loc}}$.

**Definition 42 (Programs)** *We define a* program *to be any term M in the language* Lambda *such that* $fv(M) = \emptyset$.

*Again we use the meta-variables $P, Q, R$ to range over programs.* □

The definition of substitution, originally defined for the language $Exp_{\mathrm{loc}}$ is extended in the obvious manner to *Lambda*.

**Definition 43 (Substitution)** *Let M be an arbitrary expression in* Lambda *and P a program. Then $M\{^P/_x\}$, the result of substituting the program P for all free occurrences of the variable x in the expression M is defined by structural induction on M as follows:*

*(i)* $x\{^P/_x\} = P$
*(ii)* $y\{^P/_x\} = y$, *if y is a variable different from x*
*(iii)* $(M_1 \texttt{ op } M_2)\{^P/_x\} = (M_1\{^P/_x\}) \texttt{ op } (M_2\{^P/_x\})$, *where* op *ranges over all the operators, arithmetic and Boolean*
*(iv)* $(\texttt{let } x = M_1 \texttt{ in } M_2)\{^P/_x\} = \texttt{let } x = (M_1\{^P/_x\}) \texttt{ in } M_2$
*(v)* $(\texttt{let } y = M_1 \texttt{ in } M_2)\{^P/_x\} = \texttt{let } y = (M_1\{^P/_x\}) \texttt{ in } (M_2\{^P/_x\})$, *if y is a variable different from x*
*(vi)* $(\texttt{if } M_1 \texttt{ then } M_2 \texttt{ else } M_3)\{^P/_x\} = \texttt{if } M_1\{^P/_x\} \texttt{ then } M_2\{^P/_x\} \texttt{ else } (M_3\{^P/_x\})$
*(vii)* $(M_1\, M_2)\{^P/_x\} = (M_1\{^P/_x\})\,(M_2\{^P/_x\})$
*(viii)* $(\lambda x\,.M)\{^P/_x\} = \lambda x\,.M$
*(ix)* $(\lambda y\,.M)\{^P/_x\} = \lambda y\,.(M\{^P/_x\})$, *if y is different from x.* □

Only the last three clause are new, dealing with the new constructs. Clause (viii) is the most important; in the abstraction $\lambda x\,.M$ there are no free occurrences of the variable $x$ and therefore the substitution of $P$ for $x$ should leave it unchanged.

The small-step semantics of *Lambda* will now be expressed in terms of two relations between programs

$$P \to_{\mathrm{E}} Q \qquad\qquad P \to_{\mathrm{L}} Q$$

representing one step in the eager evaluation of $P$, and the lazy evaluation of $P$ respectively. We will sometimes use

$$P \to Q$$

Common rule: (S-APP)
$$\frac{P \to P'}{PQ \to P'Q}$$

Eager: (S-CBV.A)
$$\frac{Q \to_{\mathrm{E}} Q'}{(\lambda x\,.M)Q \to_{\mathrm{E}} (\lambda x\,.M)Q'}$$
(S-CBV)
$$\frac{}{(\lambda x\,.M)v \to_{\mathrm{E}} M\{^{v}\!/\!_{x}\}}$$

Lazy: (S-CBN)
$$\frac{}{(\lambda x\,.M)Q \to_{\mathrm{L}} M\{^{Q}\!/\!_{x}\}}$$

Figure 5.2: Function application: small-step semantics

to indicate that the reduction from $P$ to $Q$ can be deduced in both the *eager* and the *late* semantics. The defining rules for both semantics inherit all the rules for the sublanguage $Exp_B$ given in Figure 4.4 and Figure 4.6 of the previous Chapter. Here we only have to discuss how the two new constructs are handled, function abstraction $\lambda x\,.M$ and function application $M_1\, M_2$. In fact programs of the form $\lambda x\,.M$ are already values and therefore do not require any evaluation rules.

So let us consider the evaluation of a program of the form $M_1\, M_2$. Because this is a program, containing no occurrences of free-variables we may assume that both $M_1$ and $M_2$ are also programs, and so let us use the correct meta-variables, and assume it has the form $P\,Q$ where $P$ and $Q$ denote arbitrary programs. Here intuitively $P$ is expected to be a function and $P\,Q$ means that we are applying that function to the argument $Q$. But to apply $P$ to an argument it must first look like a function; that is it should be of the form $\lambda x\,.M$ for some variable $x$ and expression $M$. So the first rule in the semantics,(S-APP) in Figure 5.2, is concerned with trying to reduce $P$ until it does look like a function. Intuitively this rule is applied repeatedly obtaining $P\,Q \to P_1\,Q \to \ldots \to P_k\,Q$ until eventually, hopefully, we get to an expression of the form $(\lambda x\,.M)\,Q$; that is some $P_n$ is actually a function abstraction.

At this stage we need to evaluate an expression of the form $(\lambda x\,.M)\,Q$, which is indeed a function applied to an argument. How we proceed now depends on whether we wish to dictate *eager* or *lazy* evaluation, as discussed in the previous chapter. Let us first discuss the latter. This requires only one extra rule, (S-CBN) in Figure 5.2. This says that the first step in the lazy evaluation of $(\lambda x\,.M)\,Q$ is to replace the formal parameter of the function definition, $x$, by the actual parameter, $Q$, in the body of the function, $M$: $(\lambda x\,.M)Q \to_{\mathrm{L}} M\{^{Q}\!/\!_{x}\}$. The computation will then continue by the evaluation of the resulting program $M\{^{Q}\!/\!_{x}\}$.

With eager evaluation functions can only be applied to arguments which are already values. Thus the rule (S-CBV.A) allows the evaluation of the argument expression $Q$:

$(\lambda x\,.M)Q \rightarrow_{\mathrm{E}} (\lambda x\,.M)Q'$ provided $Q$ can be reduced to $Q'$, that is $Q \rightarrow_{\mathrm{E}} Q'$. This rule can be applied again and again until $Q$ is reduce to a value $v$; this may be an arithmetic value $\mathtt{n}$, a Boolean value or even another function abstraction $\lambda y\,.N$. It is a feature of our rules that none can be applied to any value. Therefore when we have a function application of the form $(\lambda x\,.M)\,v$ the only applicable rule will be (s-cbv). This now acts in the same way as the more general lazy rule. The actual parameter $v$ is substituted for the formal parameter $x$ in the body of the function $M$, $(\lambda x\,.M)v \rightarrow_{\mathrm{E}} M\{^{v}\!/\!_{x}\}$. The computation will now continue with the evaluation of the program $M\{^{v}\!/\!_{x}\}$.

Let us see an example evaluation, in which use use the eager rules. Consider the program $(\lambda f\,.f\;3)\;(\lambda x\,.x + 2)$ discussed above in (5.4); recall that this program is expressed in concrete syntax; the actual program we are evaluating is given in (5.6) above. The program pattern matches $\lambda f\,.M\;\;v$ since $\lambda x\,.x{+}2$ is a value. Therefore we can apply the rule (s-cbv) which requires no premise. So we have the trivial proof

$$\frac{}{(\lambda f\,.f\;3)\;(\lambda x\,.x + 2) \rightarrow_{\mathrm{E}} (\lambda x\,.x + 2)3}\;\text{(s-cbv)}$$

which means we can assert the judgement:

$$\vdash_{sm} (\lambda f\,.f\;3)\;(\lambda x\,.x + 2) \rightarrow_{\mathrm{E}} (\lambda x\,.x + 2)\;3$$

Another application of the same rule gives

$$\vdash_{sm} (\lambda x\,.x + 2)\;3 \rightarrow_{\mathrm{E}} 3 + 2$$

Finally an application of (s-add) from Figure 4.4 gives $\vdash_{sm} 3{+}2 \rightarrow_{\mathrm{E}} 5$. So using the notation of Chapter 1 we have

$$(\lambda f\,.f\;3)\;(\lambda x\,.x + 2) \rightarrow^{3}_{\mathrm{E}} 5$$

Let us now compare the language *Lambda* with *Fpl* from Chapter 4. In *Fpl* functions are defined a priori and then used, whereas in *Lambda* they are defined on the fly. In *Fpl* functions are given names, and they can then be invoked by using these names. For example if a declaration set $D$ contained the declaration

$$\mathsf{double}(x) \Leftarrow x + x$$

then any expression of the form $\mathsf{double}(P)$ may appear in a program to be executed. On the other hand functions in *Lambda* appear to anonymous. However they can readily be given names in local declarations. For example the above named declaration of the function double can be mimicked in *Lambda* by

$$\mathtt{let}\;\mathsf{double} = \lambda x\,.x + x\;\mathtt{in} \qquad \ldots\mathsf{double}(P_1)\ldots\mathsf{double}(P_2)\ldots$$

Anywhere within the body of the local declaration calls can also be made to the function named double.

In *Fpl* functions can take multiple arguments whereas in *Lambda* they can only take one. But there is a well known mechanism, called *currying*, for transforming the former into the latter which we explain by an example. Consider the declaration

$$\mathsf{max}(x, y) \Leftarrow \mathtt{if}\; x < y\;\mathtt{then}\; y\;\mathtt{else}\; x$$

in *Fpl* of the binary function max. In *Lambda* this can be rendered as the unary function

$$\lambda x\,.\,(\lambda y\,.\mathtt{if}\ x < y\ \mathtt{then}\ y\ \mathtt{else}\ x) \qquad (5.7)$$

which takes an argument n, to be bound to $x$, and returns the function

$$\lambda y\,.\mathtt{if}\ \mathtt{n} < y\ \mathtt{then}\ y\ \mathtt{else}\ \mathtt{n}$$

In turn when this is applied to an argument m we end up evaluating the program

$$\mathtt{if}\ \mathtt{n} < \mathtt{m}\ \mathtt{then}\ \mathtt{m}\ \mathtt{else}\ \mathtt{n}$$

Thus, for example, if we let $P$ denote the expression

$$\begin{array}{ll} \mathtt{let}\ \mathsf{max} = \lambda y\,.\mathtt{if}\ \mathtt{n} < y\ \mathtt{then}\ y\ \mathtt{else}\ \mathtt{n} \\ \mathtt{in} \qquad (\mathsf{max}\ 2)\ 3 \end{array}$$

then one can check, regardless of whether we use the early or late semantics, that $P \rightarrow^* 3$. In three computation steps we obtain the value 5.

Curried functions tend to lead to a proliferation of brackets in concrete syntax. But instead of reverting to abstract synax more standard conventions may be used. For example in terms such as (5.7) above we tend to omit the brackets, assuming that function abstractions hold sway as far to the right as possible. In general means that $\lambda x\,.M\,N$ stands for $\lambda x\,.(M\,N)$ rather than $(\lambda x\,.M)(N)$. The main syntactic convention for concrete syntax is that *function application associates to the left*. By this we mean that the series of function applications

$$M_1\, M_2\, M_3 \ldots\, M_k$$

is an abbreviation for

$$(\ldots (M_1\, M_2)\, M_3 \ldots)\, M_k$$

Thus for example we write

$$\lambda x\,.\lambda y\,.\lambda z\,.x\,y\,z \qquad \text{in place of} \quad \lambda x\,.\lambda y\,.\lambda z\,.(x\,y)\,z$$

Curried functions can also be made somewhat less mysterious by using more meaningful variable names. For example in

$$\lambda F\,.\lambda x\,.\lambda f\,.\,F\,f\,x$$

the use of $f$ indicates that the third argument is meant to be some function, while the first, bound to $F$ is some *functional*, that is a function which manipulates functions in some manner. However such hints can only ever be approximate. When in doubt use lots of brackets to explain the abstract syntax of the term in question, or use abstract syntax trees themselves.

The main characteristic of function definitions in *Fpl* is that they are *recursive*. For example consider the definition

$$\mathsf{fac}(x) \Leftarrow \mathtt{if}\ x = 0\ \mathtt{then}\ 1\ \mathtt{else}\ x \times \mathsf{fac}(x \dot{-} 1)$$

in a declaration set $D$, where we are assuming access to a multiplication operation and a subtraction operator suitable for numerals called *monus*. Recall from Exercise 33 that $\mathtt{m} \dot{-} \mathtt{n}$ returns $\mathtt{0}$ if $n > m$ and $\mathtt{m} - \mathtt{n}$ otherwise. Then, as we have seen in Chapter 4, $\mathsf{fac}(3) \rightarrow^*{}_D$ $\mathtt{6}$; there are in all three calls to the definition of $\mathsf{fac}$ in the evaluation. Suppose we try to duplicate this in *Lambda* using a local declaration such as

$$\begin{aligned} &\mathtt{let}\ \mathsf{fac} = \lambda x\,.\mathtt{if}\ x = \mathtt{0}\ \mathtt{then}\ \mathtt{1}\ \mathtt{else}\ x \times \mathsf{fac}(x \dot{-} \mathtt{1}) \\ &\ \mathtt{in}\quad\ \ \mathsf{fac}(3) \end{aligned} \tag{5.8}$$

Assuming access to the multiplication and monus operations this expression, which we denote by $M$, is indeed a valid expression in *Lambda*. But it is not a program and therefore can not be executed; calculations show that $fv(M) = \{\mathsf{fac}\}$. The free occurrence is in the binding of the name in (5.8) to the abstraction $\mathtt{if}\ x = \mathtt{0}\ \mathtt{then}\ \mathtt{1}\ \mathtt{else}\ x \times \mathsf{fac}(x \dot{-} \mathtt{1})$; here the occurrence of $\mathsf{fac}$ is free. Thus local declarations in *Lambda* can mimic the declarations of functions in *Fpl* but their interpretation is not recursive.

**Exercise 42** *For each of the following programs find a value to which they can be reduced using the lazy small-step semantics.*

*(i)* $(\lambda f\,.\lambda x\,.f(f\,x))\,(\lambda y\,.y + \mathtt{2})\,\mathtt{0}$

*(ii)* $(\lambda x\,.\lambda y\,.\lambda z\,.x\,y\,z)(\lambda t\,.\lambda f\,.t)\,v\,w$

*(iii)* $(\lambda x\,.\mathtt{let}\ x = \lambda y\,.x + y\ \mathtt{in}\ x\ \mathtt{2})\ \mathtt{3}$

*(iv)* $P\ (\lambda y\,.\lambda z\,.z + y)\ \mathtt{4}$*, where* $P$ *denotes the program* $\lambda f\,.\lambda x\,.(\lambda g\,.g\ \mathtt{2})\ f(x)$*.* □

Consider the program $\lambda x\,.x\,x$, which is often abbreviated to $\Delta$. This is a little strange in that it takes an argument and applies that argument to itself. But it can be employed successfully. For example

$$\Delta\ \lambda y\,.y \rightarrow (\lambda y\,.y)(\lambda y\,.y) \rightarrow \lambda y\,.y$$

So when applied to the identity function it successfully produces a value, which happens to be the identity function again. Note that these are valid reductions in both the *eager* and the *late* semantics.

But things get more interesting if we try to apply $\Delta$ to itself. Using the small-step semantics, in particular either of the rules (S-CBV) or (S-CBN), we can derive $\vdash_{sm} \Delta\ \Delta \rightarrow \Delta\ \Delta$. In one step from the program $\Delta\ \Delta$ we get a reduction back to itself. This means of course that we have an infinite diverging computation

$$\Delta\ \Delta \rightarrow \Delta\ \Delta \rightarrow \Delta\ \Delta \rightarrow \ldots \rightarrow \Delta\ \Delta \rightarrow \Delta\ \Delta \rightarrow \ldots \tag{5.9}$$

This sets the language *Lambda* apart from the sub-language $Exp_B$ as in the latter we can not have infinite computations. This can be seen by completing the following exercise:

**Exercise 43** *We say a program* $P$ *in* $\mathrm{Exp}_B$ *is* blocked *if there is no* $P'$ *such that* $P \rightarrow P'$*. Prove that for every* $P$ *in* $\mathrm{Exp}_B$ *that there is some blocked* $Q$ *such that* $P \rightarrow^* Q$*.*

*Because the small-step semantics of* $\mathrm{Exp}_B$ *is deterministic this means that there can be no infinite computations in* $\mathrm{Exp}_B$*.* □.

The infinite computation in (5.9) may be looked upon as a bug in the language. But instead we will look on the bright side and consider it a feature. It turns out that variations on self-application can be used to define arbitrary recursive functions.

Consider the more complicated *combinator* $\lambda x.\lambda y.y(xxy)$ which we abbreviate to $A$ and let $\Theta$ denote the program $A\,A$, which applies $A$ to itself.[1]. Then for any program $P$ in *Lambda* we have the following reductions, in the lazy semantics:

$$\begin{aligned}\Theta\,P &\rightarrow_{\text{L}} (\lambda y.yAAy)\,P \\ &\rightarrow_{\text{L}} P\,(\Theta\,P)\end{aligned} \tag{5.10}$$

This may not look very interesting, but we can use this property of $\Theta$ to express recursive functions in *Lambda*. To see this let us look at programs $P$ with the particular form $\lambda f.\lambda x.B$; here we should look on the code $B$ as the body of a recursive definition, where $f$ is the *recursion variable* and $x$ is the formal parameter to the resulting recursive function. Let $\textsc{rec}f.B$ abbreviate the combinator $\Theta(\lambda f.\lambda x.B)$. Repeating (5.10) we have

$$\textsc{rec}f.B \quad \rightarrow^*_{\text{L}} \quad (\lambda f.\lambda x.B)\,(\textsc{rec}f.B) \tag{5.11}$$

$$\rightarrow^*_{\text{L}} \quad \lambda x.B\{^{\textsc{rec}f.B}/_{f}\} \tag{5.12}$$

In other words the combinator $\textsc{rec}f.B$ acts very much like a recursive function, whose behaviour is determined by the code $B$.

**Example 44** *Suppose we want to define a program which calculates the factorial function. Then we can use the above schema, using* $\textsc{rec}f.F$ *where F is the body*

$$\texttt{if} \ = \texttt{0}\ \texttt{then}\ 1\ \texttt{else}\ x \times f\,(x \dot{-} 1) \tag{5.13}$$

*Here we are assuming access to the arithmetic operators* $\times$ *and* $\dot{-}$. *Then using (5.11) we have the computation*

$$\textsc{rec}f.F \qquad \rightarrow^*_{\text{L}} \quad \lambda x.\texttt{if}\ x = \texttt{0}\ \texttt{then}\ 1\ \texttt{else}\ x \times (\textsc{rec}f.F)\,(x \dot{-} 1)$$

*From this it should be obvious why we can use* $\textsc{rec}f.F$ *to calculate factorials. Let us see an example, by calculating the factorial of* 3.

$$\begin{aligned}(\textsc{rec}f.F)\ 3 \ &\rightarrow^*_{\text{L}} \ (\lambda x.\texttt{if}\ x = \texttt{0}\ \texttt{then}\ 1\ \texttt{else}\ x \times (\textsc{rec}f.F)\,(x \dot{-} 1))\ 3 \\ &\rightarrow_{\text{L}} \ \texttt{if}\ 3 = \texttt{0}\ \texttt{then}\ 1\ \texttt{else}\ 3 \times (\textsc{rec}f.F)\,(3 \dot{-} 1) \\ &\rightarrow_{\text{L}} \ 3 \times (\textsc{rec}f.F)\,(3 \dot{-} 1) \\ &\rightarrow^*_{\text{L}} \ 3 \times (\lambda x.\texttt{if}\ x = \texttt{0}\ \texttt{then}\ 1\ \texttt{else}\ x \times (\textsc{rec}f.F)\,(x \dot{-} 1))\,(3 \dot{-} 1) \\ &\rightarrow^*_{\text{L}} \ 3 \times 2 \times (\textsc{rec}f.F)(2 \dot{-} 1) \\ &\rightarrow^*_{\text{L}} \ 3 \times 2 \times (\lambda x.\texttt{if}\ x = \texttt{0}\ \texttt{then}\ 1\ \texttt{else}\ x \times (\textsc{rec}f.F)\,(x \dot{-} 1))\,(2 \dot{-} 1) \\ &\rightarrow^*_{\text{L}} \ 3 \times 2 \times 1 \times (\lambda x.\texttt{if}\ x = \texttt{0}\ \texttt{then}\ 1\ \texttt{else}\ x \times (\textsc{rec}f.F)\,(x \dot{-} 1))\,(1 \dot{-} 1) \\ &\rightarrow^*_{\text{L}} \ 3 \times 2 \times 1 \times 1 \rightarrow^*_{\text{L}} 6\end{aligned} \tag{5.14}$$

[1]$\Theta$ is called Turings combinator

**Exercise 44** *Recall that formally the only arithmetic opearation in the language* Lambda*, according to Figure 5.1, is infix addition +. Here we show that many more can be coded up. Let $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$ be a partial function over the natural numbers, which takes k arguments. We say f is* implementable *in Lambda if there is a program $P_f$ such that $P_f\ \mathtt{n_1}\ \mathtt{n_2}\ \ldots\ \mathtt{n_k} \to^* \mathtt{m}$ if and only if $f(n_1, \ldots n_k) = m$, for all natural numbers $n_1, \ldots, n_k$.*

*Show that the following functions are implementable.*

*(i)* pred : $\mathbb{N} \to \mathbb{N}$*, defined by* pred$(n) = 0$*, if n is* 0 *and is* $n-1$ *otherwise.*

*This is quite difficult. You may just want to assume it and use it in the following questions.*

*(ii)* isEven : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ *given by* isEven$(n) = 0$ *whenever n is an even number and* 1 *otherwise.*

*Can you define this withut using* pred *?*

*(iii) The modified* minus *function over the naturals, often called* monus*: $n \dot{-} m = 0$ if $m > n$ and otherwise $n - m$, where $-$ represents standard* minus *on all integers.*

*Here use the function* pred *from the previous question.*

*(iv) The multiplication function,* mult : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$*;* mult$(n, m)$ *returns the result of multiplying n by m.* □

**Exercise 45** *Consider the following program in* Lambda*:*

$$\Theta(\lambda f.\lambda x.\mathtt{if}\ x = \mathtt{0}\ \mathtt{then}\ \mathtt{0}\ \mathtt{else}\ (f\ (x \dot{-} 1)\ + 1)$$

*What does it do?*

*Find a simpler program in* Lambda*, one which does not involve any self application, which behaves the the same way.* □

Our method for defining recursive functions in *Lambda* only works when we use the lazy operational semantics. To see why consider again the term abbreviated as $\textsc{rec}f.B$. Unravelling this abbreviation it is immediate that it is not a value. Therefore the reduction from (5.11) to (5.12) above is not a reduction in the eager semantics. In general the reduction

$$(\lambda f.Q)\ P \to_{\text{E}} Q\{^P\!/\!_f\}$$

can only be inferred from the rules in Figure 5.2, by Rule (s-cbv) in particular, if $P$ is a value. If $P$ is not a value then only rule (s-cbv.a) can be applied. Thus going back to (5.11) above, in the eager semantics we have

$$\begin{aligned}
\textsc{rec}f.B &\to^*_{\text{E}} (\lambda f.\lambda x.B)\ (\textsc{rec}f.B)\\
&\to^*_{\text{E}} (\lambda f.\lambda x.B)\ ((\lambda f.\lambda x.B)\ (\textsc{rec}f.B))\\
&\to^*_{\text{E}} (\lambda f.\lambda x.B)\ ((\lambda f.\lambda x.B)\ ((\lambda f.\lambda x.B)\ (\textsc{rec}f.B)))\\
&\to^*_{\text{E}} \ldots\ldots
\end{aligned}$$

So under the eager semantics the combinator $\textsc{rec} f.B$ diverges, regardless of what the code $B$ is.

However the fixpoint combinator can be adapted to the eager evaluation strategy so that this endless unwinding of the recursive definition is avoided. Let $A_v$ denote the combinator $\lambda x.\lambda y.y\ \lambda z.(xxy)z$ and $\Theta_v$ denote $A_v A_v$. Then mimicing (5.10) above we have, for any program $P$:

$$\begin{aligned}\Theta_v\, P &\rightarrow_{\text{E}} (\lambda y.y\ \lambda z.(A_v\, A_v y)z)\, P\\ &\rightarrow_{\text{E}} P\ \lambda z.(\Theta_v\, P)z\end{aligned}$$

Here the presence of the value $\lambda z.(\Theta_v\, P)z$ in place of $\Theta_v\, P$, avoids the indefinite unwinding of the combinator. In general the replacement of a combinator $Q$ with the value $\lambda z.Q\, z$ is referred to as *$\eta$-expansion*.

To see how this combinator can now be used to define recursive definitions in the eager semantics let us consider again the instance of $P$, of the form $\lambda f.\lambda x.F$ where $F$ is the body of the recursive definition of the factorial function given in (5.13) above. In analogy with the lazy case let $\textsc{rec}_{\text{V}} f.B$ be an abbreviation for the combinator $\Theta_v(\lambda f.\lambda x.F)$. Then in analogy with the lazy execution given in Example 44 we have:

$$\begin{aligned}(\textsc{rec}_{\text{V}} f.F)\ 3 \quad &\rightarrow_{\text{E}}^{*} \quad (\lambda x.\texttt{if}\ x = \mathtt{0}\ \texttt{then}\ 1\ \texttt{else}\ x \times (\lambda z.(\textsc{rec}_{\text{V}} f.F)z)\,(x \dot{-} 1))\ 3\\ &\rightarrow_{\text{E}} \quad \texttt{if}\ 3 = \mathtt{0}\ \texttt{then}\ 1\ \texttt{else}\ 3 \times (\lambda z.(\textsc{rec}_{\text{V}} f.F)z)\,(3 \dot{-} 1)\\ &\rightarrow_{\text{E}} \quad 3 \times (\lambda z.(\textsc{rec}_{\text{V}} f.F)z)\,(3 \dot{-} 1)\\ &\rightarrow_{\text{E}} \quad 3 \times (\lambda z.(\textsc{rec}_{\text{V}} f.F)z)\ 2\\ &\rightarrow_{\text{E}} \quad 3 \times (\textsc{rec}_{\text{V}} f.F)\ 2\\ &\rightarrow_{\text{L}}^{*} \quad 3 \times 2 \times (\lambda z.(\textsc{rec}_{\text{V}} f.F)z)(2 \dot{-} 1)\\ &\rightarrow_{\text{E}} \quad 3 \times 2 \times (\lambda z.(\textsc{rec}_{\text{V}} f.F)z)\ 1\\ &\rightarrow_{\text{E}} \quad 3 \times 2 \times (\textsc{rec}_{\text{V}} f.F)1\\ &\rightarrow_{\text{E}}^{*} \quad 3 \times 2 \times 1 \times (\lambda z.(\textsc{rec}_{\text{V}} f.F)z)(1 \dot{-} 0)\\ &\rightarrow_{\text{E}} \quad 3 \times 2 \times 1 \times (\lambda z.(\textsc{rec}_{\text{V}} f.F)z)\ \mathtt{0}\\ &\rightarrow_{\text{E}} \quad 3 \times 2 \times 1 \times (\textsc{rec}_{\text{V}} f.F)\mathtt{0}\\ &\rightarrow_{\text{E}}^{*} \quad 3 \times 2 \times 1 \times 1 \rightarrow_{\text{E}}^{*} 6\end{aligned}$$

We have concentrated here on the Turing fixpoint combinator $\Theta$ but another popular one is Currys $Y$ combinator, defined as $\lambda f.(\lambda x.f(xx))\,(\lambda x.f(xx))$. Unlike the unwinding property of Turings combinator, in (5.10) above, we do not have the reduction

$$Y\, P \rightarrow_{\text{L}}^{*} P\,(Y\, P)$$

Instead we have

$$\begin{aligned}\Theta\, P &\rightarrow_{\text{L}} \lambda x.P\,(x\, x)\ \lambda x.P\,(x\, x)\\ &\rightarrow_{\text{L}} P\ (\lambda x.P\,(x, x))\ (\lambda x.P\,(x, x))\end{aligned}$$

However this weaker property is also sufficient to implement recursive definitions; see the following exercise.

**Exercise 46** *Use the Y combinator to show that the multiplication function is implementable:* $\mathsf{mult} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$*, defined by* $\mathsf{mult}(n, m) = n \times m$*.* □

**Exercise 47** *Design a big-step semantics for* Lambda*. The judgements should take the form* $P \Downarrow_{\text{E}} v$ *for eager evaluation and* $P \Downarrow_{\text{L}} v$ *for lazy evaluation, where P ranges over programs and v ranges over values.*

*Then prove that this semantics is consistent with the small-step semantics. This involves proving two statements:*

*(i)* $P \to^* v$ *implies* $P \Downarrow v$

*(ii)* $P \Downarrow v$ *implies* $P \to^* v$*.*

*regardless of whether we use the eager or lazy semantic relations.* □

We have seen that the eager and lazy semantics are equally powerful when it comes to implementing recursion. But we have also seen in Chapter 4 for the language *Fpl* there is a difference between the results they produce. In *Fpl* there are functions which diverge under the eager strategy but return a value in the lazy case. These examples can be reproduced in *Lambda*.

Consider a program of the form $(\lambda x\,.\mathbb{0})Q$ where $Q$ is *any program*. Then $(\lambda x\,.\mathbb{0})Q \to_{\text{L}} \mathbb{0}$, regardless of the structure of $Q$; this follows by an application of the rule (S-CBN) in Figure 5.2. In particular let $Q$ be any program which never terminates; the prototypical example we have seen is $(\Delta\Delta)$, where $\Delta$ abbreviates $\lambda xx\;x$. Then we have

$$(\lambda x\,.\mathbb{0})(\Delta\Delta) \to_{\text{L}} \mathbb{0};$$

However this program diverges under the eager strategy. The program $(\Delta\Delta)$ is not a value and therefore the only eager reduction rule applicable to $(\lambda x\,.\mathbb{0})(\Delta\Delta)$. is (S-CBV.A). In particular since $(\Delta\Delta) \to_e (\Delta\Delta)$, the only eager reduction we have is $(\lambda x\,.\mathbb{0})(\Delta\Delta) \to_{cbv} (\lambda x\,.\mathbb{0})(\Delta\Delta)$. So the only possible execution sequence is

$$(\lambda x\,.\mathbb{0})(\Delta\Delta) \to_{\text{E}} (\lambda x\,.\mathbb{0})(\Delta\Delta) \to_{\text{E}} \ldots \to_{\text{E}} (\lambda x\,.\mathbb{0})(\Delta\Delta) \to_{\text{E}} \ldots$$

In other words, under the eager strategy the program $(\lambda x\,.\mathbb{0})(\Delta\Delta)$ never reduces to a value.

In *Fpl* the converse was not true. There if the eager strategy produces a value then the lazy strategy also produces exactly the same value, although it may take some steps to converge. This was proved in Theorem 38 in Chapter 4.4.1. Somewhat surprisingly this is no langer true for *Lambda*. Here is a counter-example:

$$\begin{array}{lcl}(\lambda x\,.\lambda y\,.x)\,(Id\,\mathbb{0}) & \to^*_{\text{E}} & \lambda y\,.\mathbb{0} \\ (\lambda x\,.\lambda y\,.x)\,(Id\,\mathbb{0}) & \to^*_{\text{L}} & \lambda y\,.(Id\,\mathbb{0})\end{array}$$

where $Id$ is an abbreviation for the identity function $\lambda x\,.x$. Under the eager strategy the argument $(Id\,\mathbb{0})$ is first evaluated to the value $\mathbb{0}$ and then an application of the rule (S-CBV) produces the final value $\lambda y\,.\mathbb{0}$. But under the lazy evaluation the rule (S-CBN) is immediately applied to the unevaluated argument $(Id\,\mathbb{0})$, to produce a different final value $\lambda y\,.(Id\,\mathbb{0})$.

**Simple types** : $\sigma ::= \mathsf{base} \mid (\sigma_1 \to \sigma_2)$ **Type environments:** $\Gamma ::= \epsilon \mid \Gamma, x\!:\!\tau$

$\mathsf{base} ::= \mathsf{int} \mid \mathsf{bool}$

**Environment look-up** :

$$\text{(TY-LOOK1)}\;\frac{}{\Gamma, x\!:\!\tau \vdash x\!:\!\sigma} \qquad \text{(TY-LOOK2)}\;\frac{\Gamma \vdash x_2\!:\!\sigma_1}{\Gamma, x_1\!:\!\sigma_1 \vdash x_2\!:\!\sigma_2}\; x_1 \neq x_2$$

Figure 5.3: Simple types for *Lambda*; and type environments

So in general the eager and lazy strategies can produce different values. Intuitively, at least in this case, although the values are syntactically different, there is some sense that they *behave in the same way* as functions. If $\lambda y\,.\mathtt{0}$ and $\lambda y\,.(Id\,\mathtt{0})$ are ever employed in larger programs to produce base values, then they will always produce the same one, namely $\mathtt{0}$. But sorting out what exactly the phrase *behaves in the same way* should mean is beyond the scope of this chapter.

## 5.2 Typing *Lambda*

In $Exp_B$ the only possible values were either arithmetics, that is numerals, or Boolean; for these we have the types int and bool respectively. In *Lambda* we have many more possible values; these new values are of the form $\lambda x\,.M$, representing functions. Examples include

$$\lambda x\,.x + x \qquad \lambda f\,.\lambda x\,.(f\,x) + x \qquad \lambda x\,.(\lambda y\,.y\,y)\,(\lambda y\,.y\,y)$$

In order to extend the type checking systems developed in Chapter 4.3 we must invent appropriate types for these functional values. These, called *simple types*, are given in Figure 5.3 and are ranged over by $\sigma$. Examples include

(i) $\mathsf{int} \to \mathsf{int}$ : functions from integers to integers, such as $\lambda x\,.x + x$. Speaking more strictly these functions act on numerals and return numerals.

(ii) $(\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{bool})$: functionals which take integer functions and construct functions from integers to Booleans. A typical example is $\lambda f\,.f\,(1) = 1$.

(iii) $(\mathsf{int} \to \mathsf{int}) \to \mathsf{int} \to (\mathsf{int} \to \mathsf{int})$: functionals which take an integer function and then an integer, and constructs another integer function. An example would be $\lambda f\,.\lambda x\,.\,\lambda y\,.f(x + y)$.

In Figure 5.3 we also define type environments, lists of associations between variables and types, and two rules for looking up the type associated with a variable in an environment; this is just a repeat of Figure 4.7 from Chapter 4.3 but now using simple types rather than just base types.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_{xf} \vdash f : \mathsf{int} \to \mathsf{int}}\,{\scriptstyle (\text{TY-LOOK})} \qquad \dfrac{}{\Gamma_{xf} \vdash x : \mathsf{int}}\,{\scriptstyle (\text{TY-LOOK})}}{\Gamma_{xf} \vdash f(x) : \mathsf{int}}\,{\scriptstyle (\text{TY-APP})} \qquad \dfrac{}{\Gamma_{xf} \vdash 1 : \mathsf{int}}\,{\scriptstyle (\text{TY-INT})}}{x : \mathsf{int},\ f : \mathsf{int} \to \mathsf{int} \vdash f(x) + 1 : \mathsf{int}}\,{\scriptstyle (\text{TY-ADD})}}{x : \mathsf{int} \vdash \lambda f\,.f(x) + 1 : (\mathsf{int} \to \mathsf{int}) \to \mathsf{int}}\,{\scriptstyle (\text{TY-ABS})}}{\varepsilon \vdash P : \mathsf{int} \to (\mathsf{int} \to \mathsf{int}) \to \mathsf{int}}\,{\scriptstyle (\text{TY-ABS})}$$

Figure 5.4: A typing judgement for $P = \lambda x\,.\lambda f\,.f(x) + 1$

The rules for typechecking, to infer judgements of the form

$$\Gamma \vdash M : \sigma$$

inherit all of the rules used in Chapter 4.3 for the sub-language $Exp_B$ which were given in Figure 4.8. We only need two new rules for the extra constructs of function abstraction and function application:

$$\overset{(\text{TY-APP})}{\dfrac{\Gamma \vdash M : \sigma_1 \to \sigma_2, \qquad \Gamma \vdash N : \sigma_1}{\Gamma \vdash M\,N : \sigma_2}} \qquad\qquad \overset{(\text{TY-ABS})}{\dfrac{\Gamma, x : \sigma_1 \vdash M : \sigma_2}{\Gamma \vdash \lambda x\,.M : \sigma_1 \to \sigma_2}}$$

The first, (TY-APP), is straightforward. If we know that, under the assumptions in $\Gamma$,

- $M$ is an expression which is expected to return a value of type $(\sigma_1 \to \sigma_2)$ , that is a function which takes an argument of type $\sigma_1$ and returns a value of type $\sigma_2$
- $N$ is an expression which is expected to return a value of type $\sigma_1$

then the expression $M\,N$ is well-typed; it denotes a function of type $(\sigma_1 \to \sigma_2)$ applied to an argument of type $\sigma_1$, and should therefore evaluate to a value of the return type $\sigma_2$.

The rule (TY-ABS) is for function abstraction. It states that $\sigma_1 \to \sigma_2$ is a correct type to assign to the expression $\lambda x\,.M$ provided under the extra assumption that $x$ will always have values of type $\sigma_1$ assigned to it, we can demonstrate that the expression $M$ will return a value of type $\sigma_2$.

To see how these inference rules are used consider the program $\lambda x\,.\lambda f\,.f(x) + 1$, which we abbreviate to $P$. A derivation of the judgement

$$\varepsilon \vdash P : \mathsf{int} \to (\mathsf{int} \to \mathsf{int}) \to \mathsf{int}$$

is given in Figure 5.4. In this derivation we use $\Gamma_{xf}$ as an abbreviation for the type environment generated during the construction of the derivation, $x : \mathsf{int},\ f : \mathsf{int} \to \mathsf{int}$.

We also use (LOOKUP) to denote the systematic scanning of the type environment for the type of a variable; this notation has previously been used in Chapter 4.

However this is not the only type which can be assigned to $P$. Intuitively the only constraint in the body of the abstraction is that the expression $f(x)$ returns an integer. This can be achieved with the assumption that $f$ be a function of type $(\sigma \to \mathsf{int})$ for any type $\sigma$. And indeed one can check that for any simple type $\sigma$ it is possible to derive the judgement

$$\varepsilon \vdash P : \sigma \to (\sigma \to \mathsf{int})$$

**Exercise 48**

*(i) Let* $\Gamma_x$ *denote the type environment consisting of one entry,* $x : \mathsf{int}$. *Is it possible to derive a typing judgement* $\Gamma_x \vdash \lambda y\,.y\; x\; (\lambda x\,.x \;\texttt{or}\; \texttt{tt}) : \sigma$

*for some simple type* $\sigma$*?*

*(ii) Let P denote the program* $\lambda f\,.\lambda g\,.f\;(g\;\texttt{tt})\;+\;1$. *What kind of simple types can be assigned to P?*

*That is what is the set of simple types* $\sigma$ *such that typing judgements* $\vdash P : \sigma$ *can be derived?* □

**Theorem 45 (Well-typed programs don't go wrong)** *Suppose* $\vdash P : \sigma$ *for some program P in Lambda and simple type* $\sigma$.

*(i) (**Progress**) Either P is a value or there is some program Q such that* $\vdash_{sm} P \to Q$.

*(ii) (**Preservation**) If* $\vdash_{sm} P \to Q$ *then* $\vdash Q : \sigma$. □

**Exercise 49** *Prove that Theorem 45 is true for both the eager and late small-step semantics.*

*This will involve developing a series of auxiliary results for the typechecking system for Lambda, similar to those proved for* $Exp_B$ *in Chapter 4.3.2.* □

There are two issues (at least) with the typechecking system we have given for *Lambda*. The first is the failure of *type uniqueness*. Recall from Proposition 32 that for any expression $E$ in $Exp_B$, if $\Gamma \vdash E : \tau_1$ and $\Gamma \vdash E : \tau_2$ then $\tau_1 = \tau_2$. This is no longer true for *Lambda*. The simplest example is the identity function $\lambda x\,.x$, although we have already come across many others. It is easy to check that for any simple type $\sigma$ we can derive the judgement $\vdash \lambda x\,.x : \sigma \to \sigma$; regardless of the actual type $\sigma$ this judgement can be derived by one application of the rule (TY-ABS).

A priori this may not be a problem. But it turns out that many type related techniques, including type inference algorithms, depend on type uniqueness. To recover this property for *Lambda* one approach is to extend the language of types, so that for example $\lambda x\,.x$ will have a unique type from this augmented set. These types get complicated and they are way out of the scope of these course notes. But see part V of [2]. A second possible approach is to modify the language *Lambda* in some manner so as to recover type uniqueness. This we will do in the following section.

The presence of types also impacts on the fixpoint combinators described in Chapter 5.1. The essential problem is that programs using self-application can not be assigned a type. Suppose there were a simple type $\sigma$ such that the judgement

$$\Gamma \vdash \lambda x\,.x : \sigma$$

could be inferred from the typechecking rules, for some $\Gamma$. If this is the case then in fact we can also infer $\vdash \lambda x\,.x : \sigma$ because of the general property of Strengthening; that is the type can be inferred from the empty type environment. The only way that this judgement can be inferred is by an application of the rule (TY-ABS), which requires the type $\sigma$ to be of the form $\sigma_1 \to \sigma_2$ such that the judgement

$$x : \sigma_1 \vdash x\,x : \sigma_2$$

can be inferred. In turn this can only be inferred by an application of (TY-APP) which requires the existence of another simple type $\sigma_3$ such that both of the judgements

$$x : \sigma_1 \vdash x : \sigma_3 \to \sigma_2 \qquad \text{and} \qquad x : \sigma_1 \vdash x : \sigma_3 \tag{5.15}$$

can be inferred. As we have seen we do not have type uniqueness in general for *Lambda*. But this does hold for variables; one can show that for any variable $y$, if $\Gamma \vdash y : \rho_1$ and $\Gamma \vdash y : \rho_2$ can be inferred then then types $\rho_1$ $\rho_2$ must coincide. Referring this to (5.15) above means that the proposed simple types $\sigma_3$ and $\sigma_2 \to \sigma_3$ must coincide. But this is impossible for any simple types $\sigma_2$, $\sigma_3$, as given by the grammar in Figure 5.3. This should be obvious; for example the number of symbols used to construct the type $\sigma_2 \to \sigma_3$ is always larger than the number used to construct $\sigma_3$.

We have just shown that the term $\lambda x\,.xx$ can not be assigned a type. This is perfectly acceptable but similar reasoning shows that the fixpoint combinators, such as $\Theta$ and $Y$ above, can also not be assigned types. Unfortunately it is only with these combinators can we define recursive functions in *Lambda*. So there appears to be a clash between the use of (simple) types and recursion. We will see one way of overcoming this in the next section.

## 5.3 Simply typed *Lambda*

The terms of an *explicitly* typed version of *Lambda*, which we call *Lambda*$^T$, is given in Figure 5.5. The only difference is that here in function abstractions the type of the expected argument must be explicitly mentioned, $\lambda x : \sigma\,.M$. For convenience in the same figure we recall the grammar for this simple types, ranged over by $\sigma$.

The presence of these types have no impact on the structure of allowed expressions, except that abstractions have to be annotated by types. Neither is there any impact on the behaviour of expressions. Therefore all of the concepts required to define the semantics of *Lambda*$^T$ are directly inherited from Chapter 5.1. These include $fv(M)$, the definition of *programs*, and the definition of the small-step relations $P \to_{\text{E}} Q$ and $P \to_{\text{L}} Q$. Moreover all of the properties of these relations developed for *Lambda* may be assumed to be true also for *Lambda*$^T$.

$$\begin{array}{rcl}
M \in \mathit{Lambda}^T & ::= & v \\
 & & \mid (M_1 + M_2) \mid (M_1 \times M_2) \\
 & & \mid M_1 \;\mathsf{and}\; M_2 \mid M_1 \;\mathsf{or}\; M_2 \mid M_1 = M_2 \mid M_1 < M_2 \\
 & & \mid \mathtt{if}\; M_1 \;\mathtt{then}\; M_2 \;\mathtt{else}\; M_3 \\
 & & \mid M_1 \; M_2 \\
v \in \mathit{Val} & ::= & x \in \mathit{Vars} \mid \mathtt{n} \in \mathit{Nums} \mid \mathtt{tt} \mid \mathtt{ff} \mid \lambda x : \sigma . M
\end{array}$$

$$\begin{array}{rrcl}
\textbf{Simple types :} & \sigma & ::= & \mathsf{base} \mid (\sigma_1 \to \sigma_2) \\
 & \mathsf{base} & ::= & \mathsf{int} \mid \mathsf{bool}
\end{array}$$

Figure 5.5: The explicitly typed language

Most of the development of type inference for terms in *Lambda* also applies to *Lambda*$^T$. The only change is in the rule (TY-ABS) for function abstractions given on page 111. This should now read

$$\text{(TY-ABS.T)}\quad \frac{\Gamma, x : \sigma \vdash M : \sigma_2}{\Gamma \vdash \lambda x{:}\sigma . M : \sigma \to \sigma_2}$$

This is where the explicit presence of types in terms has an impact; it reduces considerably the possible types which can be inferred for functions. Nevertheless all of the results on type inference for *Lambda* from Chapter 5.2, such as *Progress*, apply equally well to *Lambda*$^T$, with this modified inference rule. But there is a new property:

**Proposition 46 (Uniqueness of types)** *In Lambda$^T$, if $\Gamma \vdash M : \sigma_1$ and $\Gamma \vdash M : \sigma_2$. Then $\sigma_1$ is identical to $\sigma_2$.*

**Proof:** The proof proceeds by induction on the derivation of the judgement $\Gamma \vdash M : \sigma_1$ and then an analysis of $M$. The only interesting case is when $M$ has the form $\lambda x{:}\rho . N$ for some type $\rho$. □

**Exercise 50** *Find simple types $\sigma_1$, $\sigma_2$, $\sigma_3$ such that a typing judgement*

$$\varepsilon \vdash \lambda f{:}\sigma_1 . \lambda x{:}\sigma_2 . (\lambda g{:}\sigma_3 . g\ 2)\ f(x) : \sigma$$

*can be derived, for some simple type $\sigma$.* □

Theorem 45 can also be replicated for the language *Lambda*$^T$; indeed the proofs of *Progress* and *Preservation* are virtually the same for the two languages *Lambda* and *Lambda*$^T$. But the most impressive property of the typed language *Lambda*$^T$ is that well-typed programs are guaranteed to terminate:

**Theorem 47 (Normalisation)** *In* $Lambda^T$ *suppose* $\vdash P : \sigma$. *Then there exists some value* $v$ *such that* $P \to^* v$.

**Proof:** This is true for both the eager and late semantics, but is very difficult result to prove. See Chapter 12 of [2] for a proof for a language very similar to $Lambda^T$ with an eager semantics; this also contains further references to proofs in the lazy case.

Note that one can continue to write fixpoint combinators in $Lambda^T$ but it is not possible to assign types to them. For example let the combinator $\lambda x{:}\mathsf{int}\,.\lambda y{:}(\mathsf{int} \to \mathsf{int})\,.y(xxy)$ be abbreviated by $A_i$ and let $\Theta_i$ the combinator $A_i\, A_i$. Then $\Theta_i$, under the lazy reduction rules, acts exactly like the fixpoint combinator $\Theta$ of the previous section, and can be used to define recursive functions. But it is imposible to infer a judgement $\Gamma \vdash \Theta_i : \sigma$ for any simple type $\sigma$.

**Exercise 51** *Show that in* $Lambda^T$ *it is impossible to derive a judgement* $\Gamma \vdash \lambda x{:}\sigma_1\,.x\,x : \sigma$ *for any simple types* $\sigma_1$, $\sigma$.

*Show that the same is true for* $\Theta_a = A_a\, A_a$ *where* $A$ *is any combinator of the form* $\lambda x{:}\sigma_1\,.\lambda y{:}\sigma_2\,.y(xxy)$. □

## 5.4 Fixpoint operators

A priori the language $Lambda^T$ may be viewed as a good basis for a functional programming language; if a program can be typed then it is guaranteed to return a value. But without some form of recursion, or iteration, it would be difficult to use as a programming tool. Here we view one simple way of augmenting it with such a mechanism. We confine our discussion to the language with a lazy semantics, which is somewhat easier than the strict case.

We add to the language a new constructor FIX which takes a simple type $\sigma$, a variable $y$ and a term $M$, and returns a new term $\text{FIX}\, y{:}\sigma\,.M$. We use $Lambda^{\text{fix}}$ to denote the resulting language; this is essentially the language studied in Chapter 4 of [1]. These new terms of the form $\text{FIX}\, y{:}\sigma\,.M$ are called *fixpoints*. Essentially they play exactly the same role as the combinators $\text{REC} f.B$ in the language *Lambda* in Chapter 5.1. Recall that $\text{REC} f.B$ is simply an abbreviation for a particular program in untyped language *Lambda* whose form depends on the term $B$. In contrast in $Lambda^T$ we introduce a new construct FIX with which we can build new terms; these terms will be well-typed, and the reduction semantics for terms using this new construct will ensure that terms of the form $\text{FIX}\, y{:}\sigma\,.M$ will behave very much like the untyped terms $\text{REC} f.B$ in the untyped language *Lambda*.

Intuitively such a term should be considered as the *solution* to the recursive equation

$$y \quad = \quad M$$

and can be used as a term of type $\sigma$. As an example let $G$ denote the term

$$\lambda x{:}\mathsf{int}\,.\mathtt{if}\ x = \mathtt{0}\ \mathtt{then}\ 1\ \mathtt{else}\ x \times y\,(x \mathbin{\dot{-}} 1), \tag{5.16}$$

the untyped version of which was used in (5.13) above to define the factorial function in *Lambda*; note that $fv(G) = \{y\}$. Intuitively the new program $\textsc{fix}\, y{:}(\mathsf{int} \to \mathsf{int})\,.G$ in *Lambda*$^{\text{fix}}$ can be considered as an object satisfying the equation

$$y = \lambda x\,.\mathtt{if}\ x = \mathtt{0}\ \mathtt{then}\ 1\ \mathtt{else}\ x \times y\,(x \dot{-} 1)$$

and, as we shall see, will have the same behaviour as a program in *Lambda*$^{\text{fix}}$, as the term $\Theta(\lambda f\,.\lambda x\,.F)$ in *Lambda*, which on page 106 we abbreviated to $\textsc{rec} f.F$. For convenience we now use the notation $Fac_{\text{ty}}$ for this term, $\textsc{fix}\, f{:}(\mathsf{int} \to \mathsf{int})\,.G$.

We extend the small-step semantics of *Lambda*$^{T}$ to *Lambda*$^{\text{fix}}$, by explaining how the new fixpoint terms are handled. First the definition of *free variables in a term* is extended by defining

$$fv(\textsc{fix}\, y{:}\sigma\,.M) = fv(M) - \{y\}$$

So $\textsc{fix}\, y{:}\sigma\,.M$ is a program if $fv(M) \subseteq \{y\}$. For example the term $Fac_{\text{ty}}$ which we constructed above is a program in *Lambda*$^{\text{fix}}$ because $fv(G) \subseteq \{y\}$.

The (lazy) small-step semantics for programs in *Lambda*$^{\text{fix}}$ is now defined by inheriting all the rules used for *Lambda*$^{T}$, together with one new rule for fixpoint programs, which simply *unwinds* the definitions:

(s-fix)
$$\frac{}{\textsc{fix}\, y{:}\sigma\,.M \ \to_{\text{L}}\ M\{^{\textsc{fix}\, y{:}\sigma\,.M}/_{y}\}}$$

One application of this new rule gives the single evaluation step

$$Fac_{\text{ty}} \ \to_{\text{L}}\ \lambda x\,.\mathtt{if}\ x = \mathtt{0}\ \mathtt{then}\ 1\ \mathtt{else}\ x \times\ Fac_{\text{ty}}(x - 1)$$

With this step it should now be obvious that $Fac_{\text{ty}}$ is a combinator in *Lambda*$^{\text{fix}}$ which implements the factorial function. For example the evaluation of the *Lambda* program $\textsc{rec} f.F\ 3$ to the value 6, given in (5.14) above, can be immediately be duplicated as an evaluation of the program $Fac_{\text{ty}}\ 3$ in *Lambda*$^{\text{fix}}$.

**Exercise 52** *Show that all the functions given in Exercise 44 can be implemented using programs in* Lambda$^{\text{fix}}$. □

To typecheck terms in *Lambda*$^{\text{fix}}$ it is sufficient to add to the set of rules already used for *Lambda*$^{T}$ one extra rule for the new fixpoint terms:

(ty-fix)
$$\frac{\Gamma, x : \sigma \vdash M : \sigma}{\Gamma \vdash \textsc{fix}\, x{:}\sigma\,.M : \sigma}$$

As an example one can derive the typing judgement

$$\vdash G : \mathsf{int} \to \mathsf{int}$$

where $G$ is the function given in (5.16) above; the last rule used in this this derivation is (TY-ABS.T). Then an application of the new rule (TY-FIX) then gives the judgement

$$\vdash \mathit{Fac}_{\mathrm{ty}} : \mathsf{int} \to \mathsf{int}$$

So our new version of factorial is a well-typed program in $\mathit{Lambda}^{\mathrm{fix}}$.

**Theorem 48 (Well-typed programs don't go wrong)** *In* $\mathsf{Lambda}^{fix}$*, suppose* $\vdash P : \sigma$*, where* $P$ *is a program and* $\sigma$ *is a simple type. Then there exists some value* $v$ *of type* $\sigma$ *such that* $P \to^* v$*.*

**Proof:** The proof requires the development of many auxiliary results, all given in detail for the language $\mathit{Exp}_B$ in Chapter 4.3. There is no major difficulty in extending these to $\mathit{Lambda}^{\mathrm{fix}}$. □

Thus $\mathit{Lambda}^{\mathrm{fix}}$ is a language which supports static typing for simple types and which also supports programming via the definition of recursive functions. However Normalisation, as in Theorem 47 for the language $\mathit{Lambda}^{T}$, no longer holds; well-typed programs may loop forever.

**Exercise 53** *Give an example of a well-typed program* $P$ *in* $\mathsf{Lambda}^{fix}$ *whose evaluation never leads to a value.* □

**Exercise 54** *Is it possible to add a facility to* $\mathsf{Lambda}^{T}$ *for defining recursive functions, is such a way Normalisation remains true in the extended language?* □

# Bibliography

[1] Carl A. Gunter. *Semanics of Programming Languages*. MIT Press, 1992.

[2] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.