Chapter 3

The While programming language

Contents

3 The While programming language **1**
3.1 Big-step semantics 2
3.2 Small-step semantics 9
3.3 Properties 14
3.4 Extensions to the language *While* 22
3.4.1 Local declarations 22
3.4.2 Aborting computations 25
3.4.3 Adding parallelism 28

3.1 Big-step semantics

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:

```
L2 := 1;
L3 := 0;
while ¬ (L1 = L2) do
  L2 := L2 + 1;
  L3 := L3 + 1
```

which subsequently we we refer to as C_1.

Exercise: 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 L := E where E is an arithmetic expression and 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 L, K, . . . 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 n as a meta-variable to range over

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

Figure 3.1: The language *While*

the set of numerals, *Nums*, used in previous chapters. Apart from these, we are allowed to use one operator + to construct arithmetic expressions, although others can be easily added.

Thus a typical example of an assignment command is

$$\mathtt{K} := \mathtt{L} + 2$$

Intuitively this refers to the command:

- 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.

- **Tests**, `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.

- **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 the next section.

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 2 using indentation and white space.

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:

$$s : \mathrm{Locs} \rightarrow \mathit{Nums}$$

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

$$s[\mathrm{K} \mapsto n](\mathrm{L}) = \begin{cases} \mathtt{n} & \text{if } \mathrm{K} = \mathrm{L} \\ s(\mathrm{L}) & \text{otherwise} \end{cases}$$

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

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

where C is a command from *Com* and s, s' 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 it eventually terminates in the state s'.

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

- $\mathtt{if}\ \mathrm{L} = \mathrm{K}\ \mathtt{then}\ \mathrm{L}_1 := \mathrm{K} + \mathrm{L}_1\ \mathtt{else}\ \mathrm{L}_2 := \mathrm{L} + (\mathrm{K} + \mathtt{2})$
- $\mathtt{while}\ \neg\,(\mathrm{L}_1 = \mathrm{L}_2)\ \mathtt{do}\ \mathrm{L}_2 := \mathrm{L}_2 + \mathtt{1}\ ;\ \mathrm{L}_3 := \mathrm{L}_3 + \mathtt{1}$

In order to explain these commands we need to know how to evaluate expressions such as $\mathrm{L} + (\mathrm{K} + \mathtt{2})$ and Boolean expressions $\neg\,(\mathrm{L}_1 = \mathrm{L}_2)$. Consequently before embarking on commands we have to first give a formal semantics to the auxiliary languages *Arith*

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

(B-LOC)
$$\frac{}{\langle \mathrm{L}, s\rangle \Downarrow s(\mathrm{L})}$$

(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

and *Bool*, from Figure 3.1. We have already considered arithmetic expressions in detail in the previous to chapters. 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 $\mathrm{L} + (\mathrm{K} + \mathtt{2})$ without knowing what numerals are currently stored in the locations L and K.

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

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

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 the Boolean value $\mathtt{bv}$.

Note that the form of these judgements imply that we do not expect the evaluation of expressions to affect the state.

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: 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 $\langle B, s\rangle \Downarrow \mathtt{true}$ or $\langle B, s\rangle \Downarrow \mathtt{false}$. However to express the evaluation rules it is best to introduce a meta-variable `bv`, to represent either of these Boolean values. 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

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

(B-ASSIGN)
$$\frac{\langle E, s\rangle \Downarrow \texttt{n}}{\langle \text{L} := E, s\rangle \Downarrow s[\text{L} \mapsto \texttt{n}]}$$

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

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

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

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

(B-WHILE.T)
$$\frac{\langle B, s\rangle \Downarrow \texttt{true} \quad \langle C, s\rangle \Downarrow s_1 \quad \langle \texttt{while}\ B\ \texttt{do}\ C, s_1\rangle \Downarrow s'}{\langle \texttt{while}\ B\ \texttt{do}\ C, s\rangle \Downarrow s'}$$

Figure 3.3: Big-step semantics of *While*

rule, or pair of rules, which directly formalises the intuition given above, on pages 2 and 3. We look briefly at each of these in turn.

The command $\text{L} := E$ 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 L, $s[\text{L} \mapsto \texttt{n}]$

This is the import of the rule (B-ASSIGN).

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

The rule (B-SEQ) is a direct formalisation of this informal description.

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 rules, one which can be applied when $\langle B, s\rangle \Downarrow \texttt{true}$ and the

other when $\langle B, s\rangle \Downarrow$ `false`. These, (B-IF.T) and (B-IF.F), formalise the obvious intuition that executing `if` B `then` C_1 `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 = `while` B `do` C; the intuitive explanation given on page 3 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 `false` immediately in the start state s, then the body C is never executed and the command immediately terminates in the final state s. This is formalised in the rule (B-WHILE.F).

(ii) If B evaluates to `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 a 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 `if` B `then` C_1 `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 $\langle$`while` B `do` $C, s\rangle \Downarrow s'$ we have a premise which still involves the command `while` B `do` C itself.

The final possible command is the ineffective `skip`; its execution has no effect on the state and therefore we have the axiom $\langle$`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. Consdier the command C_1 given on page 2. In order to set out the derivation we use the following abbreviations:

$$\begin{array}{rll} C_{11} & \text{for} & \mathrm{L}_2 := \mathtt{1} \,;\, \mathrm{L}_3 := \mathtt{0} \\ C_{12} & \text{for} & \mathrm{L}_2 := \mathrm{L}_2 + \mathtt{1} \,;\, \mathrm{L}_3 := \mathrm{L}_3 + \mathtt{1} \\ B & \text{for} & \neg\,(\mathrm{L}_1 = \mathrm{L}_2) \\ W & \text{for} & \mathtt{while}\ \neg\,(\mathrm{L}_1 = \mathrm{L}_2)\ \mathtt{do}\ C_{12} \end{array}$$

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 L_1 contains the numeral `m`, L_2 contains `n` and L_3 contains `k`; these are the only locations used by the command C_1. With these abbreviations a formal derivation of the judgement

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

is given in Figure 3.4. So if a compiler is to agree with our formal semantics it must ensure that if C_1 is exectured from the initial state s_{377} it must eventually terminate with s_{322} as the final state.

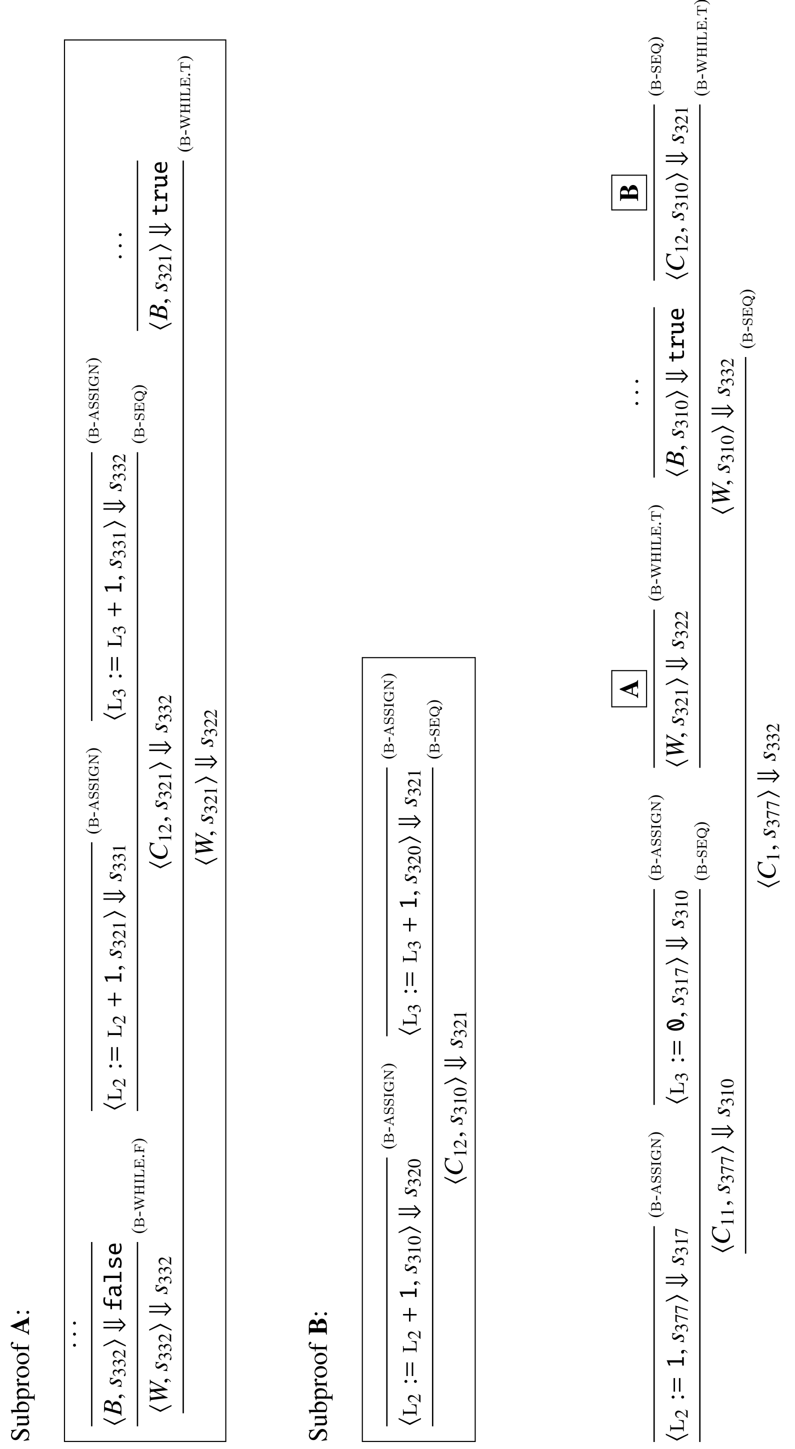

Figure 3.4: An example derivation

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

$$\texttt{while}\,(\neg\,\mathrm{L}1 = \mathbb{0})\,\texttt{do}\;\mathrm{L} := \mathrm{L} + 1 \tag{3.1}$$

which we denote by *LP* and let s be any state such that $s(\mathrm{L}) > 0$. 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

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

for some state s'; 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'.

So how do we know that (3.2) is not true for any state s' ? We can prove it by contradiction. Suppose a judgement $\langle LP, s\rangle \Downarrow s'$ could be derived for some state s such that $s(\mathrm{L}) > \mathbb{0}$ and some arbitrary state s'. If so there is such a judgement which has a shortest derivation; that is there is no other such judgement which has a shorter proof. Suppose this particular judgement is actually $\langle LP, s_1\rangle \Downarrow s_2$ for a some states $s_1,\ s_2$ such that $s_1(\mathrm{L}) > 0$.

How can this judgement be derived? Because $\langle \neg\,\mathrm{L}1 = \mathbb{0}, s_1\rangle \Downarrow \texttt{true}$ every derivation, including the shortest one, must involve an application of the rule (B-WHILE.T). Specifically the structure of the shortest derivation must take the form

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

Now because $\vdash_{big} \langle \mathrm{L} := \mathrm{L} + 1, s_1\rangle \Downarrow s_3$ we know that $s_3(\mathrm{L}) > \mathbb{0}$. And in the above derivation the ...... 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 $\langle LP, s_1\rangle \Downarrow s_2$. But this is a contradiction since we assumed that this derivation of $\langle LP, s_1\rangle \Downarrow s_2$ was shortest.

Exercise: 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 $\langle LP1, s\rangle \Downarrow s'$ for any state s'. □

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

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

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 3.5, are given in terms of 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

$$\langle \mathtt{skip}, s\rangle \to \langle C, s'\rangle$$

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

Let us now briefly look at the rules in Figure 3.5. Executing the command $\text{L} := E$ involves performing one basic operation, namely updating the numeral stored in L to be whatever the expression E evaluates to. Thus in (S-ASS)

- E is evaluated to the numeral n, that is $\langle E, s\rangle \Downarrow \mathtt{n}$
- the state s changes to the modified store $s[\text{L} \mapsto \mathtt{n}]$
- 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 $\langle C_1, s\rangle \to \langle C_1', s'\rangle$, representing the execution of this basic operation. Then the execution of the first step of the compound command is given by the judgement $\langle C_1\,; C_2, s\rangle \to \langle C_1'\,; C_2, s'\rangle$; 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

(S-ASS)
$$\frac{\langle E, s\rangle \Downarrow \mathtt{n}}{\langle \mathrm{L} := E, s\rangle \to \langle \mathtt{skip}, s[\mathrm{L} \mapsto \mathtt{n}]\rangle}$$

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

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

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

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

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

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

Figure 3.5: Small-step semantics of *While*

the small-step semantics which do not correspond to either (a) or (b) above; these may be considered to be *housekeeping steps*.

Example Consider the execution of the compound command $\mathrm{L}_2 := \mathtt{1} \,;\, \mathrm{L}_3 := \mathtt{0}$ from the initial stare 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:

$$\cfrac{\cfrac{}{\langle \mathrm{L}_2 := \mathtt{1}, s_{377}\rangle \to \langle \mathtt{skip}, s_{317}\rangle}\ (\text{S-ASS})}{\langle \mathrm{L}_2 := \mathtt{1} \,;\, \mathrm{L}_3 := \mathtt{0}, s_{377}\rangle \to \langle \mathtt{skip} \,;\, \mathrm{L}_3 := \mathtt{0}, s_{317}\rangle}\ (\text{S-SEQ.LEFT})$$

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

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

$$\frac{}{\langle \mathtt{skip} \,;\, \mathrm{L}_3 := \mathtt{0}, s_{317}\rangle \to \langle \mathrm{L}_3 := \mathtt{0}, s_{317}\rangle}\ (\text{S-SEQ.SKIP})$$

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

$$\frac{}{\langle \text{L}_3 := \mathtt{0}, s_{317}\rangle \rightarrow \langle \mathtt{skip}, s_{310}\rangle}\ \text{(S-SEQ.SKIP)}$$

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. Borrowing the notation from Chapter 1 we have

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

Returning to our discussion of the inference rules in Figure 3.5, the treatment of the test, $\mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2$ 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 $\mathtt{while}\ B\ \mathtt{do}\ C$. The behaviour depends naturally on the value of the guard B in the current state. Intuitively if this evaluates to $\mathtt{false}$ 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 $\mathtt{while}\ B\ \mathtt{do}\ C$ to the command $C ; \mathtt{while}\ B\ \mathtt{do}\ C$.

Let us revisit the command C_1 on page 2, which re-using the abbreviations on page 7 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 L_2 with the numeral 1, and this is borne out formally by the following derivation:

$$\cfrac{\cfrac{\cfrac{}{\langle \text{L}_2 := \mathtt{1}, s_{377}\rangle \rightarrow \langle \mathtt{skip}, s_{317}\rangle}\ \text{(S-ASS)}}{\langle C_{11}, s_{377}\rangle \rightarrow \langle (\mathtt{skip} ; \text{L}_3 := \mathtt{1}), s_{317}\rangle}\ \text{(S-SEQ.L)}}{\langle C_1, s_{377}\rangle \rightarrow \langle (\mathtt{skip} ; \text{L}_3 := \mathtt{0}) ; W, s_{317}\rangle}\ \text{(S-SEQ.L)}$$

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

The second step is the rather uninteresting housekeeping move

$$\vdash_{sm} \langle (\mathtt{skip} ; \text{L}_3 := \mathtt{0}) ; W, s_{317}\rangle \rightarrow \langle \text{L}_3 := \mathtt{0} ; W, s_{317}\rangle$$

justified by the formal derivation

$$\cfrac{\cfrac{}{\langle \mathtt{skip} ; \text{L}_3 := \mathtt{0}, s_{377}\rangle \rightarrow \langle \text{L}_3 := \mathtt{1}, s_{317}\rangle}\ \text{(S-SEQ.S)}}{\langle (\mathtt{skip} ; \text{L}_3 := \mathtt{0}) ; W, s_{317}\rangle \rightarrow \langle \text{L}_3 := \mathtt{0} ; W, s_{317}\rangle}\ \text{(S-SEQ.L)}$$

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

$$\vdash_{sm} \langle \mathrm{L}_3 := \mathtt{0} \,;\, W, s_{317}\rangle \rightarrow \langle \mathtt{skip} \,;\, W, s_{311}\rangle \qquad \vdash_{sm} \langle \mathtt{skip} \,;\, W, s_{310}\rangle \rightarrow \langle W, s_{310}\rangle$$

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

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

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 $\langle \neg\,(\mathrm{L}_1 = \mathrm{L}_2), s_{310}\rangle \Downarrow \mathtt{true}$ can be derived. Then the next step

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

is justified by an application of the rule (S-WHILE.T), in the nearly trivial derivation:

$$\frac{\overline{\langle \neg\,(\mathrm{L}_1 = \mathrm{L}_2), s_{310}\rangle \Downarrow \mathtt{true}}}{\langle \mathtt{while}\ \neg\,(\mathrm{L}_1 = \mathrm{L}_2)\ \mathtt{do}\ C_{12}, s_{310}\rangle \rightarrow \langle C_{12} \,;\, W, s_{310}\rangle}\ \text{(S-WHILE.T)}$$

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

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

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

Another round of five derivations gives

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

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

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

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

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

Exercise: Let C be any command different from `skip`. Prove that for every state s there is a derivation of the form $\langle C, s\rangle \rightarrow \langle C', s'\rangle$ for some configuration $\langle C', s'\rangle$. □

To end this section let us revisit the non-terminating command $LP = \mathtt{while}\ \neg\,\mathrm{L1} = \mathtt{0}\ \mathtt{do}\ \mathrm{L} := \mathrm{L} + 1$ discussed in the previous section. Again let s be any state satisfying

$s(\text{L}) > 0$. Assuming $\langle \neg\ \text{L1} = \mathtt{0}, s\rangle \Downarrow \mathtt{true}$ is derivable, an application of the rule (S-WHILE.T) will justify the judgement

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

We then have

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

where s_1 is some state which also satisfies $s_1(\text{L}) > \mathtt{0}$. 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:

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

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

Exercise: Use your small-step semantics of arithmetics and Boolean expressions to rewrite the semantics of commands in Figure 3.5, 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 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 *seeds* or starting points, and three kinds of constructors:

- **Base cases**:
  - the constant `skip` is a command
  - For every location name 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 $P(\texttt{skip})$ is true.
  - Prove that $P(\textsc{l} := E)$ is true for every location name 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.

Proposition 1 *For every expression $E \in$ Arith and every state s*

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

(ii) ***(Determinacy)*** *if* $\vdash_{big} \langle E, s\rangle \Downarrow \texttt{n}_1$ *and* $\vdash_{big} \langle E, s\rangle \Downarrow \texttt{n}_2$ *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: 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 $\langle C, s\rangle \rightarrow \langle C', s'\rangle$ for some configuration $\langle C', s'\rangle$. □

Proposition 2 *For every command $C \in$ Com and every state s, if $\vdash_{sm} \langle C, s\rangle \rightarrow \langle C_1, s_1\rangle$ and $\vdash_{sm} \langle C, s\rangle \rightarrow \langle C_1, s_1\rangle$ 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 $\vdash_{sm} \langle C, s\rangle \rightarrow \langle C_1, s_1\rangle$ and $\vdash_{sm} \langle C, s\rangle \rightarrow \langle C_1, s_1\rangle$ 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 $P(\texttt{skip})$ is vacuously true, as it is not possible to derive any judgement of the form $\langle \texttt{skip}, s\rangle \rightarrow \langle D, s'\rangle$, for any pair $\langle D, s'\rangle$.

(ii) Another base case, when C is $\textsc{l} := E$. From Proposition 1 we know that, for a given state s, there is exactly one number n such that $\vdash_{big} \langle E, s\rangle \Downarrow \texttt{n}$. Looking at the collection of rules in Figure 3.5, there is only one possible rule to apply to the pair $\langle \textsc{l} := E, s\rangle$, namely (S-ASS). Consequently, if $\vdash_{sm} \langle \textsc{l} := E, s\rangle \rightarrow \langle C_1, s_1\rangle$ and $\vdash_{sm} \langle \textsc{l} := E, s\rangle \rightarrow \langle C_1, s_1\rangle$ then both C_1 and C_2 must be `skip`, and both s_1 and s_2 must be the same state, $s[\textsc{l} \mapsto \texttt{n}]$.

(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

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

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 3.5 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, $\langle \texttt{skip}, s\rangle \rightarrow \langle C', s'\rangle$ can not be found. So both of the above derivations must have exactly the same the form, namely:

$$\frac{}{\langle \texttt{skip} ; D_2, s\rangle \rightarrow \langle D_2, s_1\rangle}\ \text{(S-SEQ.SKIP)}$$

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 3.5 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

$$\frac{\dfrac{\cdots}{\langle D_1, s\rangle \rightarrow \langle D_1', s'\rangle}\ ?}{\langle D_1 ; D_2, s\rangle \rightarrow \langle D_1' ; D_2, s'\rangle}\ \text{(S-SEQ.SKIP)} \qquad \text{and} \qquad \frac{\dfrac{\cdots}{\langle D_1, s\rangle \rightarrow \langle D_1'', s''\rangle}\ ??}{\langle D_1 ; D_2, s\rangle \rightarrow \langle D_1'' ; D_2, s''\rangle}\ \text{(S-SEQ.SKIP)}$$

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

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

using the rules from Figure 3.5. 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 1, 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:

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

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, 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:

$$\frac{\langle B, s\rangle \Downarrow \texttt{true}}{\langle \texttt{while}\ B\ \texttt{do}\ D, s\rangle \to \langle D \,;\, \texttt{while}\ B\ \texttt{do}\ D, s\rangle}\ \text{(S-WHILE.F)}$$

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 ; `while` B `do` D; also s_1 and s_2 are the same state s.

There is one more possibility for C, that it is of the form `if` B `then` C_1 `else` C_2; this we leave to the reader to verify. □

Corollary 3 *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: Write out the proof of Corollary 3 in detail.

Theorem 4 (Determinacy) *For every command $C \in Com$, every state s, if $\langle C, s\rangle \rightarrow^* \langle \mathtt{skip}, s_1\rangle$ and $\langle C, s\rangle \rightarrow^* \langle \mathtt{skip}, s_2\rangle$ then $s_1 = s_2$.*

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

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

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

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

for some $\langle C', s_2'\rangle$, where k_3 is the difference between k_1 and k_2. But by Corollary 3 this must mean that the intermediate command C' must actually be `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 `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 5 $\vdash_{big} \langle C, s\rangle \Downarrow s'$ *implies* $\langle C, s\rangle \rightarrow^* \langle \mathtt{skip}, s'\rangle$

Proof: Similar in style to that of Proposition 4 of the previous chapter. 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 $\langle C, s\rangle \rightarrow^* \langle \mathtt{skip}, s'\rangle$ is a shorthand notation for *there is some natural number k such that* $\langle C, s\rangle \rightarrow^k \langle \mathtt{skip}, s'\rangle$. 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 $\langle C, s\rangle \rightarrow^k \langle \mathtt{skip}, s'\rangle$.

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 $\langle \texttt{skip}, s\rangle \rightarrow^0 \langle \texttt{skip}, s\rangle$.

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 $\langle E, s\rangle \Downarrow \texttt{n}$; we know this is unique from Proposition 1. Then it is easy to use the rule (s-ASS) from Figure 3.5 to show that $\langle C, s\rangle \rightarrow^1 \langle \texttt{skip}, s'\rangle$.

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

$$\frac{\dfrac{\cdots}{\langle C_1, s\rangle \Downarrow s_1}\,\scriptstyle{(\text{B-?})} \qquad \dfrac{\cdots}{\langle C_2, s_1\rangle \Downarrow s'}\,\scriptstyle{(\text{B-?})}}{\langle C_1 \,;\, C_2, s\rangle \Downarrow s'}\,\scriptstyle{(\text{B-SEQ})} \tag{3.10}$$

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 $\langle C_1, s\rangle \rightarrow^{k_1} \langle \texttt{skip}, s_1\rangle$.

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

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 $\langle C_1, s_1\rangle \rightarrow^{k_2} \langle \texttt{skip}, s'\rangle$.

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

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 1, 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 $\langle C, s\rangle \rightarrow^1 \langle \texttt{skip}, s\rangle$ 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

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

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 $\langle D, s\rangle \rightarrow^{k_1} \langle \mathtt{skip}, s_1\rangle$ for some k_1.

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

We can now combine these two sequences, using part (i) in the exercise below, to obtain the required $\langle \mathtt{while}\ B\ \mathtt{do}\ D, s\rangle \rightarrow^{k} \langle \mathtt{skip}, s'\rangle$ for $k = k_1 + k_2 + 2$:

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

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

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

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.

Proposition 6 *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 the previous chapter; 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 $\mathtt{while}\ B\ \mathtt{do}\ D$.

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

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 $\langle C, s\rangle \to \langle \texttt{skip}, s\rangle$. 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 $\langle C, s\rangle \to \langle C', s'\rangle$ must look like $\langle \texttt{while}\ B\ \texttt{do}\ D, s\rangle \to \langle D; \texttt{while}\ B\ \texttt{do}\ D, s\rangle$, 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 constructing 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: Fill in the remaining four cases in the proof of the previous theorem. □

Theorem 7 $\vdash_{sm} \langle C, s\rangle \to^* \langle \texttt{skip}, s_t\rangle$ *implies* $\vdash_{big} \langle C, s\rangle \Downarrow s_t$.

Proof: The previous result can be generalised to:

> For any natural number $k \geq 0$, $\langle C, s\rangle \to^k \langle C', s'\rangle$ 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 $\langle C, s\rangle \to^* \langle \texttt{skip}, s'\rangle$. Recall that this means there is some natural number k such that $\langle C, s\rangle \to^k \langle \texttt{skip}, s_t\rangle$. 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 5 and Theorem 7, that they coincide on the behaviour they prescribe for commands. Specifically the following statements are equivalent:

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

Moreover we have shown that the small-step semantics is *consistent* in the sense of Determinacy, Theorem 4: for every configuration $\langle C, s\rangle$ there is at most one terminal state s' such that $\langle C, s\rangle \rightarrow^* \langle \texttt{skip}, s'\rangle$. Incidently the equivalence above also ensures that the big-step semantics is also consistent in this sense.

On page 4 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 ($States \rightharpoonup States$) 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:

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

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

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

For example $[\![LP]\!]$, given in (3.1) above, is the partial function which is only defined for states s satisfying $s(\mathrm{L}) = \mathtt{0}$. 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 $s(\mathrm{L}) = \mathtt{0}$. On the other hand $[\![LP1]\!]$, where $LP1$ is the command defined on page 9, is the totally undefined function; it has the empty domain.

Exercise: Describe, using standard mathematical notation, the partial function $[\![C_1]\!]$, where C_1 is the command given on page 2.

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.

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

$$\texttt{begin loc}\ \mathrm{L} := E\ ;\ C\ \texttt{end}$$

is that

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

Figure 3.6: The language While^{block}, an extension to *While*

- 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

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

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

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

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

(i) $s'(\textsc{k}) = 2$

(ii) $s'(\textsc{l}) = 1$

The first expectation, (i), is because the local execution of the command K := L is relative to the local declaration loc L := 2. The second (ii) is because we expect the original value of L to be restated when the local execution is finished. This is important for executing commands such as C_2:

$$\textsc{l}_1 := 1\,;\textsc{l}_2 := 2\,;\texttt{begin loc}\ \textsc{l}_1 := 7\,;\ \textsc{l}_2 := \textsc{l}_1\ \texttt{end}\,;\textsc{k} := (\textsc{l}_1 + \textsc{l}_2)$$

Here we would expect the judgement

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

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

(B-BLOCK)

$$\frac{\langle E, s\rangle \Downarrow v \qquad \langle C, s[\text{L} \mapsto v]\rangle \Downarrow s'}{\langle \texttt{begin loc } \text{L} := E \; ; \; C \texttt{ end}, s\rangle \Downarrow s'[\text{L} \mapsto s(\text{L})]}$$

Figure 3.7: Big-step rule for blocks

A big-step semantic rule for blocks, (B-BLOCK), is given in Figure 3.7. It says that the only judgements to be made for block commands take the form

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

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 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, $s_t = s'[\text{L} \mapsto s(\text{L})]$.

Exercises:

(1) Use this new rule, together with the existing ones for *While*, to find a state s' such that $\langle C_1, s\rangle \Downarrow s'$ 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.

(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

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

what are the numerals $s'(\text{L})$, $s'(\text{K})$ and $s'(\text{M})$?

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

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

Figure 3.8: Another extension to *While*, called $While^{abort}$

3.4.2 Aborting computations

Another extension to *While* is given in Figure 3.8. 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:

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

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 will have been changed. For example running C_1 will result in an aborted computation in which the resulting state s' satisfies $s'(\mathrm{L}) = 1$.

To differentiate between successful computations and unsuccessful ones we design

(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 3.9: Extra rules for arithmetic expressions in While^{abort}

two judgements

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

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 3.5 . 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

- $\langle E, s\rangle \Downarrow \mathtt{n}$ successful evaluation of E to value $\mathtt{n}$
- $\langle E, s\rangle \Downarrow \mathtt{abort}$ unsuccessful attempt at evaluating E
- $\langle B, s\rangle \Downarrow \mathsf{bv}$ successful evaluation of B to the Boolean value bv
- $\langle B, s\rangle \Downarrow \mathtt{abort}$ unsuccessful attempt at evaluating B

The inference rules for arithmetic expressions are given in Figure 3.9, 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

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

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

(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)

$$\frac{\langle E, s\rangle \Downarrow \texttt{n}}{\langle \mathrm{L} := E, s\rangle \Downarrow \langle \texttt{skip}, s[\mathrm{L} \mapsto \texttt{n}]\rangle}$$

(B-ASSIGN.A)

$$\frac{\langle E, s\rangle \Downarrow \texttt{abort}}{\langle \mathrm{L} := E, s\rangle \Downarrow \langle \texttt{abort}, s\rangle}$$

(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 3.10: Big-step inference rules for commands in *While*^{abort}

Exercise: Give the inference rules for Boolean expressions in *While*^{abort}. □

The rules for commands are given in Figure 3.10. The execution of the one-instruction command ($\mathrm{L} := E$), 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

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

Figure 3.11: *While*^{par}: adding parallelism to *While*

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: Use the inference in Figure 3.3 to execute the four commands C_i given on page 25 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 command C:

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

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

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

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

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

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

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

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

Figure 3.12: Rules for parallelism

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

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

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 3.12. 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: Use the rules in Figure 3.12, together with those in Figure 3.5 to find all states s' such that $\langle C, s\rangle \to^* \langle \texttt{skip}, s'\rangle$, where C is given in (3.12) above. This should not depend on the initial state s.

Exercise: Do the same for the command C_2:

$$(\mathrm{L} := \mathrm{K} + \mathtt{1}) \; \texttt{par} \; (\mathrm{K} := \mathrm{L} + \mathtt{1} \,;\, \mathrm{K} := \mathrm{K} + \mathrm{L})$$

relative to an initial state s satisfying $s(\mathrm{L}) = s(\mathrm{K}) = 0$.

In *While*^{par}communication between parallel commands is via the state; information passes between concurrent commands occurs 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 problems; constructs such as *semaphores*, *locks*, *critical regions*, etc.. Here we briefly examine one such construct, **Conditional critical regions**.

We add to *While*^{par} 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:

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

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 $s[x \mapsto \mathtt{2}]$.

As another example consider D_2 defined by

$$\begin{array}{l} \quad(\mathtt{await}\ \mathtt{true}\ \mathtt{protect}\ \mathrm{L} := \mathtt{1}\ ;\ \mathrm{L} := \mathrm{K} + \mathtt{1}\ \mathtt{end}) \\ \mathtt{par} \\ \quad(\mathtt{await}\ \mathtt{true}\ \mathtt{protect}\ \mathrm{K} := \mathtt{2}\ ;\ \mathrm{K} := \mathrm{L} + \mathtt{1}\ \mathtt{end}) \end{array} \tag{3.14}$$

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 $s(\mathrm{L}) = s(\mathrm{K}) = \mathtt{0}$ then there are only two possible terminal states; the first has L, K containing $\mathtt{1}, \mathtt{2}$ respectively, while the second has $\mathtt{2}, \mathtt{1}$.

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 $\rightarrow$ in the hypothesis:

$$\text{(B-AWAIT)}\ \frac{\langle B, s\rangle \Downarrow \langle \mathtt{tt}, s_1\rangle \qquad \langle C, s_1\rangle \rightarrow^* \langle \mathtt{skip}, s'\rangle}{\langle \mathtt{await}\ B\ \mathtt{protect}\ C\ \mathtt{end}, s\rangle \rightarrow \langle \mathtt{skip}, s'\rangle}$$

Exercise: Use this rule, together with those from Figure 3.5, to give formal derivations confirming the expected behaviour of the two commands D_1, D_2 in (3.13) and (3.14) above.