1 Operational Semantics for While

The language *While* of simple *while programs* has a grammar consisting of three syntactic categories: *numeric expressions*, which represent natural numbers; *booleans*, which are similar to expressions but represent truth values rather than numbers; and *commands*, which are imperative statements which affect the store of the computer. [1] (The dots in slide 1 denote that we can add more operations if necessary.)

Slide 1

Syntax of While

$$
\begin{aligned}
B \in \textit{Bool} &::= \texttt{true} \mid \texttt{false} \mid E = E \mid E < E \mid \ldots \\
&\quad \mid B \& B \mid \neg B \mid \ldots \\
E \in \textit{Exp} &::= x \mid n \mid E + E \mid \ldots \\
C \in \textit{Com} &::= x := E \mid \texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C \\
&\quad \mid C; C \mid \texttt{skip} \mid \texttt{while}\ B\ \texttt{do}\ C
\end{aligned}
$$

We use brackets were necessary to disambiguate.

The commands C are:

- *assignment*, which takes a variable and an expression and gives a command, written $x := E$;
- the *conditional*, which takes a boolean and two commands and yields a command, written $\texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C$;
- *sequential composition*, which takes two commands and yields a command, written $C_1; C_2$ (note that the semicolon is an operator joining two commands into one, and not just a piece of punctuation at the end of a command);
- the constant `skip` command, which does nothing;
- the *loop constructor*, which takes a boolean and a command and

[1] We always deal with *abstract syntax*, even though the grammar for the syntax of While looks a bit like the kind of concrete syntax you might type into a computer. So, we're really dealing with *trees* built up out of the term-forming operators of While.

yields a command, written `while` B `do` C.

We will study the operational semantics of *While*. However, we first study the operational semantics of simple expressions (without variables), since it is helpful to introduce the basic concepts of the operational semantics using such simple terms.

1.1 Operational Semantics for *SimpleExp*

We give the operational semantics for simple expressions without variables.

Slide 2

Syntax of Simple Expressions

$$E \in \textit{SimpleExp} ::= n \mid E + E \mid E \times E \mid ...$$

where n ranges over the natural numbers $0, 1, 2, ...$, and $+$ and $\times$ denote operators to form expressions. We can add more operations if we need to.

We work with abstract syntax (trees).

It is quite straightforward to change the interpretation of expression n to range over the integers $... -2, -1, 0, 1, 2, ...$

Slide 3

An operational semantics for *SimpleExp*

An operational semantics for *SimpleExp* will tell us how to evaluate an expression to get a result. This can be done in two ways:

- *small-step*, or *structural*, operational semantics gives a method for evaluating an expression step-by-step;
- *big-step*, or *natural*, operational semantics ignores the intermediate steps and gives the result immediately.

1.1.1 Big-step semantics for *SimpleExp*

Let us consider big-step semantics first. The big-step semantics for *SimpleExp* takes the form of a relation $\Downarrow$ between expressions and *values*, which are those expressions we deem to be a 'final answer'. We relate expressions to numbers, writing

$$E \Downarrow n.$$

Slide 4

Big-step Semantics of *SimpleExp*

$$\text{(B-NUM)}\ \frac{}{n \Downarrow n}$$

$$\text{(B-ADD)}\ \frac{E_1 \Downarrow n_1 \qquad E_2 \Downarrow n_2}{E_1 + E_2 \Downarrow n_3}\ n_3 = n_1 \underline{+} n_2$$

where $\underline{+}$ denotes the normal addition of natural numbers. We can give similar rules for multiplication and other natural operations.

Intuitively, a *rule* such as

$$\frac{E_1 \Downarrow n_1 \qquad E_2 \Downarrow n_2}{E_3 \Downarrow n_3}$$

means that, if it is the case that $E_1 \Downarrow n_1$ and also $E_2 \Downarrow n_2$, then it is the case that $E_3 \Downarrow n_3$. We call $E_1 \Downarrow n_1$ and also $E_2 \Downarrow n_2$ the premises of the rule, and $E_3 \Downarrow n_3$ the conclusion. When there are no entries above the line, the rule is an *axiom*, which is to say that it always holds.

Notice that the side-condition for the rule for addition, (B-ADD), talks about the addition operation $\underline{+}$ on *numbers* in order to define the semantics of the expression addition $+$.

The rules define a relation $\Downarrow$ which says when an expression evaluates to a final answer. We say that $E \Downarrow n$ is in this relation, or $E \Downarrow n$ *holds*, *only* if it can be established from the axioms and rules. So, if we want to assert that, for example, $3 + (2 + 1) \Downarrow 6$, we need to show this to be the case by applying the axioms and rules given in the definition of $\Downarrow$.

Slide 5

A Proof that $3 + (2 + 1) \Downarrow 6$

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

There are two natural properties of the big-step semantics for *SimpleExp*: *determinacy*, which says that an expression can evaluate to at most one answer; and *totality*, which says that an expression must evaluate to at least one answer.

Slide 6

Some properties of $\Downarrow$

Determinacy

For all E, n_1 and n_2, if $E \Downarrow n_1$ and $E \Downarrow n_2$ then $n_1 = n_2$.

Totality

For all E, there exists a n such that $E \Downarrow n$.

1.1.2 Small-step Semantics of *SimpleExp*

The big-step semantics given above tells us what the final value of an expression is straight away. The rules tell us how to compute the answer, but sometimes it is desirable to be more explicit about exactly how programs are evaluated. A small-step semantics lets us do just this. We shall define a relation

$$E \rightarrow E'$$

which describes *one step* of evaluation of E.

Slide 7

Small-step Semantics of *SimpleExp*

$$\text{(S-LEFT)}\quad \frac{E_1 \to E_1'}{E_1 + E_2 \to E_1' + E_2}$$

$$\text{(S-RIGHT)}\quad \frac{E \to E'}{n + E \to n + E'}$$

$$\text{(S-ADD)}\quad \frac{}{n_1 + n_2 \to n_3}\ n_3 = n_1 \underline{+} n_2$$

These rules say: to evaluate an addition, first evaluate the left-hand argument; when you get to a number, evaluate the right-hand argument; when you get to a number there too, add the two together to get a number. Note that there are *no* rules to evaluate a number, because it has already been fully evaluated.

Consider the expression $3+(2+1)$. By the axiom (S-ADD), we have $2+1 \to 3$ so, by rule (S-RIGHT), we have $3 + (2 + 1) \to 3 + 3$. The axiom also says that $3 + 3 \to 6$, so we have

$$3 + (2 + 1) \to 3 + 3 \to 6.$$

It is important to realise that the order of evaluation is fixed by this semantics. We have

$$(1 + 2) + (3 + 4) \to 3 + (3 + 4),$$

not $(1 + 2) + 7$. The big-step semantics cannot make such a fine-grained stipulation.

1.1.3 Getting the final answer

While the intermediate expressions of a computation are interesting, we are ultimately concerned with the final answer yielded by evaluating an expres-

sion. To capture this mathematically from the small-step semantics, we define the relation which expresses multiple-step evaluations.

Slide 8

Many Steps of Evaluation

Given a relation $\longrightarrow$, we define a new relation $\longrightarrow^*$ by:

$E \longrightarrow^* E'$ holds if and only if either $= E'$ (so no steps of evaluation are needed to get from E to E') or there is a finite sequence

$$E \longrightarrow E_1 \longrightarrow E_2 \ldots \longrightarrow E_k \longrightarrow E'.$$

This relation $\longrightarrow^*$ is called the *reflexive transitive closure* of $\longrightarrow$.

For our expressions, we say that number n is the final answer of E if $E \longrightarrow^* n$.

Slide 9

Normal Form

An expression E is in normal form (and said to be irreducible), if there is no E' such that $E \longrightarrow E'$.

Theorem

The normal forms of expressions are the numbers.

Analogous to the big-step semantics, *determinacy* for the small-step semantics means that each expression can only evaluate (for one step) in at most one way. Note that we do not have totality, since normal forms do not evaluate any further. A more general property of $\longrightarrow$ is *confluence*, which states that if we choose two different evaluation paths for an expression then they can both be extended so that they eventaully converge. Since $\longrightarrow$ is deterministic, confluence follows trivially, but we shall see other semantics where determinacy does not hold, but confluence does. *Normalisation* for $\longrightarrow$ means that every evaluation path must eventually reach a normal form; it cannot go on reducing forever. (There is also a property called *weak normalisation*, which means that every expression has *some* evaluation path that eventually reaches a normal form. This is less interesting for our semantics here, but is interesting when we consider the lambda calculus alter in the course.) Since $\longrightarrow$ is both confluent and normalising, it follows that every expression evaluates to exactly one normal form. As we know, the normal forms are the numbers. These properties are typically proved by induction, as we shall see.

Slide 10

Some Properties of $\longrightarrow$

Determinacy

For all E, E_1, E_2, if $E \longrightarrow E_1$ and $E \longrightarrow E_2$ then $E_1 = E_2$.

Confluence For all E, E_1, E_2, if $E \longrightarrow^* E_1$ and $E \longrightarrow^* E_2$ then there exists E' such that $E_1 \longrightarrow^* E'$ and $E_2 \longrightarrow^* E'$

(Strong) Normalisation There are no infinite sequences of expressions $E_1, E_2, E_3, \ldots$ such that, for all i, $E_i \longrightarrow E_{i+1}$. This means that every evaluation path eventually reaches a normal form.

Theorem

For all E, n_1, n_2, if $E \longrightarrow^* n_1$ and $E \longrightarrow^* n_2$ then $n_1 = n_2$.

Both $\Downarrow$ and $\longrightarrow$ give semantics to expressions in terms of the numbers to which they evaluate. Although they were defined differently, we can prove a theorem which shows that they agree.

Slide 11

Theorem

Theorem

For all E and n, $E \Downarrow n$ if and only if $E \rightarrow^* n$.

1.2 Operational Semantics of *While*

We give a *small-step* operational semantics to the programming language *While*, whose syntax was introduced in slide 1 and is repeated in slide 12. The first immediate issue is what it means to evaluate an assignment:

$$x := E \rightarrow ?$$

where E might now contain variables. We need more information about the state of the machine's *memory*. Slide 13 gives the definition of a *state* suitable for modelling *While*. Intuitively, a state s tells us what, if anything, is stored in the memory location corresponding to each variable of the language, and $s[x \mapsto n]$ is the state s updated so that the location corresponding to x contains n. For example, consider the state $s[x \mapsto 4][y \mapsto 5][z \mapsto 6]$. We have

$$\begin{aligned} s[x \mapsto 4][y \mapsto 5][z \mapsto 6](y) &= 5 \\ s[x \mapsto 4][y \mapsto 5][z \mapsto 6](z) &= 6 \\ s[x \mapsto 4][y \mapsto 5][z \mapsto 6](x) &= 4 \end{aligned}$$

Our small-step semantics will therefore be concerned with programs together with their store.

Slide 12

Syntax of While

$$\begin{aligned} B \in \textit{Bool} &::= \texttt{true} \mid \texttt{false} \mid E = E \mid E < E \mid \ldots \\ &\quad \mid B \& B \mid \neg B \mid \ldots \\ E \in \textit{Exp} &::= x \mid n \mid E + E \mid \ldots \\ C \in \textit{Com} &::= x := E \mid \texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C \\ &\quad \mid C; C \mid \texttt{skip} \mid \texttt{while}\ B\ \texttt{do}\ C \end{aligned}$$

where x is variable and n is a number.

Slide 13

States

A **state** is a partial function from variables to numbers such that $s(x)$ is defined for finitely many x.

We have

$$\begin{aligned} s[x \mapsto n](y) &= n && \text{if } y = x \\ &= s(y) && \text{otherwise} \end{aligned}$$

Our small-step semantics for *While* will be defined using **configurations** of the form $\langle E, s\rangle$, $\langle B, s\rangle$ and $\langle C, s\rangle$. The idea is that we evaluate E, B and C with respect to state s.

Expressions and Booleans Expressions and booleans to do not present any difficulty. The only new kind of expression is the variable. Its semantics involves fetching the appropriate value of the variable from the state (the variable store). The expression evaluation relation $\rightarrow_e$ is described using the rules in slide 14. The rules describing the the boolean evaluation relation $\rightarrow_b$ are left as an exercise. We usually omit the subscripts e and b, since it is clear from the context which evaluation relation is appropriate. We use them in the formal definitions, to clarify the interactions between these relations and also the execution relation $\rightarrow_c$ for our language *While*.

Slide 14

Expressions

(W-EXP.LEFT) $$\frac{\langle E_1, s\rangle \to_e \langle E_1', s'\rangle}{\langle E_1 + E_2, s\rangle \to_e \langle E_1' + E_2, s'\rangle}$$

(W-EXP.RIGHT) $$\frac{\langle E, s\rangle \to_e \langle E', s'\rangle}{\langle n + E, s\rangle \to_e \langle n + E', s'\rangle}$$

(W-EXP.NUM) $$\frac{}{\langle x, s\rangle \to_e \langle n, s\rangle}\ s(x) = n$$

(W-EMP.ADD) $$\frac{}{\langle n_1 + n_2, s\rangle \to_e \langle n_3, s\rangle}\ n_3 = n_1 \underline{+} n_2$$

Exercise: Now write down the rules for booleans.

Notice that the rules for evaluating expressions do not affect the state. Can you explain in words why this is the case? How might you prove it (difficult at the moment)?

Commands The command execution relation has the form $\langle C, s\rangle \to_c \langle C', s'\rangle$. The rules for $\to_c$ (again, we often omit the subscript c) are different from the rules for $\to_e$ and $\to_b$, in that they will directly alter the state. Intuitively, we want our rules to show how the commands update the state, and we will know that a command has finished its work when it reduces to `skip`. We shall now consider each command in turn and write down the appropriate rules. For assignment, $x := E$, we first want to evaluate E to some number n, and then update the state so that x contains the number n. For sequential composition, $C_1; C_2$, we first allow C_1 to run to completion, changing the state as it does so, and then compute C_2. For conditionals, we first evaluate the boolean guard: if it returns `true` we take the first branch; if it returns `false`, we take the second branch.

Slide 15

Assignment

(W-ASS.EXP) $$\frac{\langle E, s\rangle \to_e \langle E', s'\rangle}{\langle x := E, s\rangle \to_c \langle x := E', s'\rangle}$$

(W-ASS.NUM) $$\frac{}{\langle x := n, s\rangle \to_c \langle \texttt{skip}, s[x \mapsto n]\rangle}$$

Slide 16

Sequential Composition

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

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

Slide 17

Conditional

(W-COND.TRUE)
$$\frac{}{\langle \texttt{if true then } C_1 \texttt{ else } C_2, s\rangle \rightarrow_c \langle C_1, s\rangle}$$

(W-COND.FALSE)
$$\frac{}{\langle \texttt{if false then } C_1 \texttt{ else } C_2, s\rangle \rightarrow_c \langle C_2, s\rangle}$$

(W-COND.TRUE)
$$\frac{\langle B, s\rangle \rightarrow_b \langle B', s'\rangle}{\langle \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, s\rangle \rightarrow_c \langle \texttt{if } B' \texttt{ then } C_1 \texttt{ else } C_2, s'\rangle}$$

What about `while`? Intuitively, we want to evaluate the boolean guard, and, if true, run the command then go back to the beginning and start again. Consider as a first try slide 18. The problem with this approach is that the only rule we've got which is capable of entering the loop body is the one for `while true do` C, which ought to be an infinite loop. By evaluating the boolean guard in place, as in the first rule of slide 18, we have made a serious error. The point is that we do not want to evaluate that boolean once and use that value for ever more, but rather we want to evaluate that boolean every time we go through the loop. So, when we evaluate it the first time, it is vital that we don't throw away the 'old' B, which this rule does. The solution is to make a *copy* of B to evaluate each time. In fact, we are able to give a single rule for `while` using the conditional command, as shown in slide 19.

Slide 18

Incorrect Semantics for `while`

(W-WHILE?) $$\frac{\langle B, s\rangle \rightarrow_b \langle B', s'\rangle}{\langle \texttt{while}\ B\ \texttt{do}\ C, s\rangle \rightarrow_c \langle \texttt{while}\ B'\ \texttt{do}\ C, s'\rangle}$$

(W-WHILE?) $$\frac{}{\langle \texttt{while false do}\ C, s\rangle \rightarrow_c \langle \texttt{skip}, s\rangle}$$

(W-WHILE?) $$\frac{}{\langle \texttt{while true do}\ C, s\rangle \rightarrow_c \langle\, ?\,\rangle}$$

Slide 19

Correct Semantics for `while`

(W-WHILE) $$\frac{}{\langle \texttt{while}\ B\ \texttt{do}\ C, s\rangle \rightarrow_c \langle \texttt{if}\ B\ \texttt{then}\ (C; \texttt{while}\ B\ \texttt{do}\ C)\ \text{else}\ \texttt{skip}, s\rangle}$$

All this rule does is ‘unfold’ the while loop once. If we could write down the infinite unfolding, there would be no need for the `while` syntax.

1.2.1 An Example

Slide 20 shows a program for computing the factorial of x and storing the answer in variable a. Let s be the state $(x \mapsto 3, y \mapsto 2, a \mapsto 9)$, using an obvious notation for states. It should be the case that

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

where $s'(a) = 6$. (Can you predict the final values of x and y?) Let's check that this is correct. First some abbreviations: we write C' for the sub-program

$$\begin{array}{l}\texttt{while}\ y > 0\ \texttt{do}\\ \quad (a := a \times y;\\ \quad\ y := y - 1)\end{array}$$

and $s_{i,j,k}$ for the state $(x \mapsto i, y \mapsto j, a \mapsto k)$. Thus, for our factorial example, the initial state s can be written $s_{3,2,9}$. Now let's do the evaluation. Each line should really be justified by reference to one of the rules of the operational semantics.

Slide 20

A factorial program!

$$\begin{array}{ll} C = & y := x; a := 1;\\ & \texttt{while}\ y > 0\ \texttt{do}\\ & (a := a \times y;\\ & y := y - 1)\end{array}$$

$$
\begin{array}{ll}
 & \langle y := x; a := 1; C', s_{3,2,9} \rangle \\
\rightarrow & \langle y := 3; a := 1; C', s_{3,2,9} \rangle \\
\rightarrow & \langle \texttt{skip}; a := 1; C', s_{3,3,9} \rangle \\
\rightarrow & \langle a := 1; C', s_{3,3,9} \rangle \\
\rightarrow & \langle \texttt{skip}; C', s_{3,3,1} \rangle \\
\rightarrow & \langle C', s_{3,3,1} \rangle \\
\rightarrow & \langle \texttt{if } y > 0 \texttt{ then } (a := a \times y; y := y - 1; C') \texttt{ else skip}, s_{3,3,1} \rangle \\
\rightarrow & \langle \texttt{if } 3 > 0 \texttt{ then } (a := a \times y; y := y - 1; C') \texttt{ else skip}, s_{3,3,1} \rangle \\
\rightarrow & \langle \texttt{if true then } (a := a \times y; y := y - 1; C') \texttt{ else skip}, s_{3,3,1} \rangle \\
\rightarrow & \langle a := a \times y; y := y - 1; C', s_{3,3,1} \rangle \\
\rightarrow & \langle a := 1 \times y; y := y - 1; C', s_{3,3,1} \rangle \\
\rightarrow & \langle a := 1 \times 3; y := y - 1; C', s_{3,3,1} \rangle \\
\rightarrow & \langle a := 3; y := y - 1; C', s_{3,3,1} \rangle \\
\rightarrow & \langle \texttt{skip}; y := y - 1; C', s_{3,3,3} \rangle \\
\rightarrow & \langle y := y - 1; C', s_{3,3,3} \rangle \\
\rightarrow & \langle y := 3 - 1; C', s_{3,3,3} \rangle \\
\rightarrow & \langle y := 2; C', s_{3,3,3} \rangle \\
\rightarrow & \langle \texttt{skip}; C', s_{3,2,3} \rangle \\
\rightarrow & \langle C', s_{3,2,3} \rangle \\
\rightarrow & \langle \texttt{if } y > 0 \texttt{ then } (a := a \times y; y := y - 1; C') \texttt{ else skip}, s_{3,2,3} \rangle \\
\rightarrow & \langle \texttt{if } 2 > 0 \texttt{ then } (a := a \times y; y := y - 1; C') \texttt{ else skip}, s_{3,2,3} \rangle \\
\rightarrow & \langle \texttt{if true then } (a := a \times y; y := y - 1; C') \texttt{ else skip}, s_{3,2,3} \rangle \\
\rightarrow & \langle a := a \times y; y := y - 1; C', s_{3,2,3} \rangle \\
\rightarrow & \langle a := 3 \times y; y := y - 1; C', s_{3,2,3} \rangle \\
\rightarrow & \langle a := 3 \times 2; y := y - 1; C', s_{3,2,3} \rangle \\
\rightarrow & \langle a := 6; y := y - 1; C', s_{3,2,3} \rangle \\
\rightarrow & \langle \texttt{skip}; y := y - 1; C', s_{3,2,6} \rangle \\
\rightarrow & \langle y := y - 1; C', s_{3,2,6} \rangle \\
\rightarrow & \langle y := 2 - 1; C', s_{3,2,6} \rangle \\
\rightarrow & \langle y := 1; C', s_{3,2,6} \rangle \\
\rightarrow & \langle \texttt{skip}; C', s_{3,1,6} \rangle \\
\rightarrow & \langle C', s_{3,1,6} \rangle \\
\rightarrow & \langle \texttt{if } y > 0 \texttt{ then } (a := a \times y; y := y - 1; C') \texttt{ else skip}, s_{3,1,6} \rangle \\
\rightarrow & \langle \texttt{if } 1 > 0 \texttt{ then } (a := a \times y; y := y - 1; C') \texttt{ else skip}, s_{3,1,6} \rangle \\
\rightarrow & \langle \texttt{if true then } (a := a \times y; y := y - 1; C') \texttt{ else skip}, s_{3,1,6} \rangle \\
\rightarrow & \langle a := a \times y; y := y - 1; C', s_{3,1,6} \rangle \\
\rightarrow & \langle a := 6 \times y; y := y - 1; C', s_{3,1,6} \rangle \\
\rightarrow & \langle a := 6 \times 1; y := y - 1; C', s_{3,1,6} \rangle \\
\rightarrow & \langle a := 6; y := y - 1; C', s_{3,1,6} \rangle \\
\rightarrow & \langle \texttt{skip}; y := y - 1; C', s_{3,1,6} \rangle \\
\rightarrow & \langle y := y - 1; C', s_{3,1,6} \rangle \\
\rightarrow & \langle y := 1 - 1; C', s_{3,1,6} \rangle \\
\rightarrow & \langle y := 0; C', s_{3,1,6} \rangle \\
\rightarrow & \langle \texttt{skip}; C', s_{3,0,6} \rangle \\
\rightarrow & \langle C', s_{3,0,6} \rangle \\
\rightarrow & \langle \texttt{if } y > 0 \texttt{ then } (a := a \times y; y := y - 1; C') \texttt{ else skip}, s_{3,0,6} \rangle \\
\rightarrow & \langle \texttt{if } 0 > 0 \texttt{ then } (a := a \times y; y := y - 1; C') \texttt{ else skip}, s_{3,0,6} \rangle \\
\rightarrow & \langle \texttt{if false then } (a := a \times y; y := y - 1; C') \texttt{ else skip}, s_{3,0,6} \rangle \\
\rightarrow & \langle \texttt{skip}, s_{3,0,6} \rangle
\end{array}
$$

As you can see, this kind of calculation is horrible to do by hand. It can, however, be automated to give a simple *interpreter* for the language, based directly on the semantics. It is also formal and precise, with no argument about what should happen at any given point. Finally, it did compute the right answer!

1.2.2 Some Facts

Recall that the small-step semantics for expressions satisfies determinacy, confluence and normalisation, slide 10. Determinacy and confluence hold for the execution relation $\rightarrow_c$ for *While*, but normalisation does not hold.

Slide 21

Determinacy and Confluence for $\rightarrow_c$

The execution relation $\rightarrow_c$ for *While* is **deterministic**: that is, for all C, s, C_1, s_1, C_2, s_2,

if $\langle C, s\rangle \rightarrow_c \langle C_1, s_1\rangle$ and $\langle C, s\rangle \rightarrow_c \langle C_2, s_2\rangle$ then $\langle C_1, s_1\rangle = \langle C_2, s_2\rangle$.

Thus, the relation $\rightarrow_c$ is **confluent**: that is, for all C, s, C_1, s_1, C_2,

if $\langle C, s\rangle \rightarrow_c^* \langle C_1, s_1\rangle$ and $\langle C, s\rangle \rightarrow_c^* \langle C_2, s_2\rangle$ then there exists $\langle C', s'\rangle$ such that $\langle C_1, s_1\rangle \rightarrow_c^* \langle C', s'\rangle$ and $\langle C_2, s_2\rangle \rightarrow_c^* \langle C', s'\rangle$

Analogous results hold for the relations $\rightarrow_e$ and $\rightarrow_b$.

However, normalisation does not hold, as illustrated in slide 24. It is possible for a computation to be non-terminating: that is, a computation to result in an infinite loop. Let us prove that (`while true do skip`) never reaches a result, using the fact that the small-step semantics of *While* is deterministic.

Slide 22

Answer Configurations

A configuration $\langle \texttt{skip}, s \rangle$ is said to be an **answer configuration**.

Since there is no rule for executing `skip`, answer configuations are **normal forms**: that is, there is no $\langle C', s' \rangle$ with $\langle \texttt{skip}, s \rangle \rightarrow_c \langle P', s' \rangle$.

For expressions, the answer configurations are $\langle n, s \rangle$ for number n. For booleans, the answer configurations are $\langle \texttt{true}, s \rangle$ and $\langle \texttt{false}, s \rangle$.

Slide 23

Stuck Configurations

There are other normal forms for configurations, called **stuck configurations** .

For example, $\langle y, (x \mapsto 3) \rangle$ is a normal expression configuration, since y cannot be evaluated in the state $(x \mapsto 3)$.This configuration is **stuck**.

Another example is the stuck command configuration $\langle x := y + 7; y := y - 1, (x \mapsto 3) \rangle$.

A further example is the stuck boolean configuration $\langle 5 < y, (x \mapsto 2) \rangle$. Note that the configuration $\langle x < y, (x \mapsto 2) \rangle$ is not stuck, but reduces to a stuck state.

Slide 24

Normalisation

The evaluation relations $\longrightarrow_e$ and $\longrightarrow_b$ are normalising.

The execution relation $\longrightarrow_c$ is not normalising.

Specifically, we can have infinite loops. For example, the program $(\texttt{while true do skip})$ loops forever.

Let s be any state. We have the execution path

$$\langle \texttt{while true do skip}, s\rangle \longrightarrow_c^3 \langle \texttt{while true do skip}, s\rangle$$

where $\longrightarrow_c^3$ means three steps. Hence, we have an infinite execution path.

Theorem For any state s, there is no s' such that

$$\langle \texttt{while true do skip}, s\rangle \longrightarrow^* \langle \texttt{skip}, s'\rangle$$

Proof Let us first calculate a few steps of the evaluation of this program:

$$\begin{aligned}
& \langle \texttt{while true do skip}, s\rangle \\
\longrightarrow\ & \langle \texttt{if true then } (\texttt{skip}; \texttt{while true do skip}) \texttt{ else skip}, s\rangle \\
\longrightarrow\ & \langle \texttt{skip}; \texttt{while true do skip}, s\rangle \\
\longrightarrow\ & \langle \texttt{while true do skip}, s\rangle
\end{aligned}$$

As you can see, it seems unlikely that this will ever get anywhere. But we need to prove this! Suppose to the contrary that it is possible for

$$\langle \texttt{while true do skip}, s\rangle \longrightarrow^* \langle \texttt{skip}, s'\rangle,$$

and let n be the number of steps taken for this evaluation. Note that, since the semantics is deterministic, this number n is well-defined.
Again, determinacy tells us that the first three steps of the evaluation must be the steps we calculated above, and then the remaining $n-3$ steps of the

evaluation show that

$$\langle \texttt{while true do skip}, s\rangle \to^* \langle \texttt{skip}, s'\rangle,$$

which is not possible, since this takes n steps! This is a contradiction, so we deduce that no such evaluation exists. ■

This result demonstrates that inherent in our language is the fact that some computations do not yield final answers. It fits our intuition about how programs work!

1.2.3 Discussion: Side-effects and Evaluation Order

Something worth noticing about our language is that the only phrases which affect the state directly are the assignment commands. Assignments are always found inside commands rather than inside expressions or booleans. Furthermore, commands are strictly sequenced by the ; operator. This means that there is never any confusion about what should be in the state at a given time, and none of the decisions we have made about the order of evaluation affect the final answer configurations.
In more sophisticated languages, this situation can be compromised. For example, commands can often creep into the language of expressions via constructs like a `return` command. For example, we think of

$$\texttt{do } x := x + 1 \texttt{ return } x,$$

as an expression because it returns a numerical result. However, it does have a side-effect on the state. We can also write composite expressions such as

$$(\texttt{do } x := x + 1 \texttt{ return } x) + (\texttt{do } x := x \times 2 \texttt{ return } x)$$

It is now *vital* that we pay close attention to the semantics of addition. Does $+$ evaluate its argument left-to-right or right-to-left?

Exercise Write down the sets of rules corresponding to each evaluation strategy for $+$, and evaluate the above in the state $(x \mapsto 0)$ under each set of rules.

Slide 25

Side-effecting expressions

If we allow expressions like

$$\texttt{do}\ x := x + 1\ \texttt{return}\ x$$

then the result of evaluating

$$(\texttt{do}\ x := x + 1\ \texttt{return}\ x) + (\texttt{do}\ x := x \times 2\ \texttt{return}\ x)$$

depends on the evaluation order.

Strictness In the case of addition, the only reasonable choices for evaluation are left-to-right or right-to-left, although other choices do exists, such as evaluating both arguments twice! In any case, it is clear that both arguments must be evaluated at least once before the result of the addition can be calculated. Sometimes, this is not the case. For example, the logical 'and' operator, written $\&$ in our syntax, when applied to `false` and any other boolean, must return `false`. It is therefore possible to write a semantics for $\&$ as on slide 26. In this case, the programmer really must know what the semantics is. For example, in any state

$$\texttt{false}\,\&\,(\texttt{while true do skip};\texttt{return}(\texttt{true})) \to \texttt{false}$$

while $(\texttt{while true do skip};\texttt{return}(\texttt{true}))\,\&\,\texttt{false}$ gets into an infinite loop.

Slide 26

Short-circuit Semantics of $\&$

$$\frac{B_1 \to B_1'}{B_1 \,\&\, B_2 \to B_1' \,\&\, B_2}$$

$$\frac{}{\texttt{false} \,\&\, B_2 \to \texttt{false}}$$

$$\frac{}{\texttt{true} \,\&\, B_2 \to B_2}$$

Slide 27

Strictness

An operation is called **strict** in one of its arguments if it always needs to evaluate that argument.

Addition is strict in both arguments.

The semantics of $\&$ given in slide 26 makes $\&$ a **left-strict** operator. It is **non-strict** in its right argument.

Procedure and Method Calls Though we will not formally consider languages with procedures or method calls, we can informally apply the ideas given in this course to obtain a deeper understanding of such languages. The issues of strictness and evaluation order crop up again and again. For example, in a method like

$$\begin{array}{l}\texttt{void aMethod(int } x\texttt{)\{}\\ \quad\texttt{return;}\\ \texttt{\}}\end{array}$$

the argument x is never used. So in a call such as

$$\texttt{aMethod(do } y := y + 1 \texttt{ return } y\texttt{)}$$

do we need to evaluate the argument? Clearly, the outcome of a program containing a call like this will depend on the semantic decision we make here. In a method call such as

$$\texttt{anotherMethod(exp1, exp2)}$$

in what order should we evaluate the arguments, if they are used? If an argument is used twice in the body of the method, should it be evaluated twice? There are plenty of different semantic decisions to be made here. Here are some popular choices.

- Evaluate all the arguments, left to right, and use the *result* of this evaluation each time the argument is used. This is called *call-by-value*, and is roughly what Java and ML do.
- Replace each use of an argument in the method body by the text of the actual parameter the programmer has supplied, so that each time the argument is called, it is re-evaluated. This is called *call-by-name*, and is what Algol 60 did (does?).
- Evaluate an argument only when it is actually used in the body of the method, but then remember the result so that if the argument is used again, it is not re-evaluated. The is called *call-by-need*, and is what Haskell does.

Exercise Write the code of a method `myMethod(exp1, exp2)` and a particular call to this method, such that the three evaluation strategies above all give different results.

The purpose of this discussion is to alert you to the fact that there may be several reasonable, but nonetheless crucially different, choices to be made about the semantics of various language constructs. One role of a formal study of semantics is to help discover where such choices and ambiguities lie, and to resolve them in a fixed, clear and well-documented way.