2 Introduction to operational semantics

This chapter presents the syntax of a programming language, **IMP**, a small language of while programs. **IMP** is called an "imperative" language because program execution involves carrying out a series of explicit commands to change state. Formally, **IMP**'s behaviour is described by rules which specify how its expressions are evaluated and its commands are executed. The rules provide an operational semantics of **IMP** in that they are close to giving an implementation of the language, for example, in the programming language Prolog. It is also shown how they furnish a basis for simple proofs of equivalence between commands.

2.1 IMP—a simple imperative language

Firstly, we list the syntactic sets associated with **IMP**:

- numbers **N**, consisting of positive and negative integers with zero,
- truth values $\mathbf{T} = \{\mathbf{true}, \mathbf{false}\}$,
- locations **Loc**,
- arithmetic expressions **Aexp**,
- boolean expressions **Bexp**,
- commands **Com**.

We assume the syntactic structure of numbers and locations is given. For instance, the set **Loc** might consist of non-empty strings of letters or such strings followed by digits, while **N** might be the set of signed decimal numerals for positive and negative whole numbers—indeed these are the representations we use when considering specific examples. (Locations are often called program variables but we reserve that term for another concept.)

For the other syntactic sets we have to say how their elements are built-up. We'll use a variant of BNF (Backus-Naur form) as a way of writing down the rules of formation of the elements of these syntactic sets. The formation rules will express things like:

$$\text{If } a_0 \text{ and } a_1 \text{ are arithmetic expressions then so is } a_0 + a_1.$$

It's clear that the symbols a_0 and a_1 are being used to stand for any arithmetic expression. In our informal presentation of syntax we'll use such *metavariables* to range over the syntactic sets—the metavariables a_0, a_1 above are understood to range over the set of arithmetic expressions. In presenting the syntax of **IMP** we'll follow the convention that

- n, m range over numbers $\mathbf{N}$,
- X, Y range over locations **Loc**,
- a ranges over arithmetic expressions **Aexp**,
- b ranges over boolean expressions **Bexp**,
- c ranges over commands **Com**.

The metavariables we use to range over the syntactic categories can be primed or subscripted. So, *e.g.*, X, X', X_0, X_1, Y'' stand for locations.

We describe the formation rules for arithmetic expressions **Aexp** by:

$$a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1.$$

The symbol "::=" should be read as "can be" and the symbol "|" as "or". Thus an arithmetic expression a can be a number n or a location X or $a_0 + a_1$ or $a_0 - a_1$ or $a_0 \times a_1$, built from arithmetic expressions a_0 and a_1.

Notice our notation for the formation rules of arithmetic expressions does not tell us how to parse

$$2 + 3 \times 4 - 5,$$

whether as $2 + ((3 \times 4) - 5)$ or as $(2 + 3) \times (4 - 5)$ *etc.*. The notation gives the so-called *abstract syntax* of arithmetic expressions in that it simply says how to build up new arithmetic expressions. For any arithmetic expression we care to write down it leaves us the task of putting in enough parentheses to ensure it has been built-up in a unique way. It is helpful to think of abstract syntax as specifying the parse trees of a language; it is the job of *concrete syntax* to provide enough information through parentheses or orders of precedence between operation symbols for a string to parse uniquely. Our concerns are with the meaning of programming languages and not with the theory of how to write them down. Abstract syntax suffices for our purposes.

Here are the formation rules for the whole of **IMP**:

For **Aexp**:

$$a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1.$$

For **Bexp**:

$$b ::= \mathbf{true} \mid \mathbf{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1$$

For **Com**:

$$c ::= \mathbf{skip} \mid X := a \mid c_0; c_1 \mid \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1 \mid \mathbf{while}\ b\ \mathbf{do}\ c$$

From a set-theory point of view this notation provides an *inductive definition* of the syntactic sets of **IMP**, which are the least sets closed under the formation rules, in a sense we'll make clear in the next two chapters. For the moment, this notation should be viewed as simply telling us how to construct elements of the syntactic sets.

We need some notation to express when two elements e_0, e_1 of the same syntactic set are identical, in the sense of having been built-up in exactly the same way according to the abstract syntax or, equivalently, having the same parse tree. We use $e_0 \equiv e_1$ to mean e_0 is identical to e_1. The arithmetic expression $3 + 5$ built up from the numbers 3 and 5 is not syntactically identical to the expression 8 or $5 + 3$, though of course we expect them to evaluate to the same number. Thus we do *not* have $3 + 5 \equiv 5 + 3$. Note we *do* have $(3 + 5) \equiv 3 + 5$!

Exercise 2.1 If you are familiar with the programming language ML (see *e.g.*[101]) or Miranda (see *e.g.*[22]) define the syntactic sets of **IMP** as datatypes. If you are familiar with the programming language Prolog (see *e.g.*[31]) program the formation rules of **IMP** in it. Write a program to check whether or not $e_0 \equiv e_1$ holds of syntactic elements e_0, e_1. □

So much for the syntax of **IMP**. Let's turn to its semantics, how programs behave when we run them.

2.2 The evaluation of arithmetic expressions

Most probably, the reader has an intuitive model with which to understand the behaviours of programs written in **IMP**. Underlying most models is an idea of state determined by what contents are in the locations. With respect to a state, an arithmetic expression evaluates to an integer and a boolean expression evaluates to a truth value. The resulting values can influence the execution of commands which will lead to changes in state. Our formal description of the behaviour of **IMP** will follow this line. First we define *states* and then the *evaluation* of integer and boolean expressions, and finally the *execution* of commands.

The set of *states* Σ consists of functions $\sigma : \mathbf{Loc} \to \mathbf{N}$ from locations to numbers. Thus $\sigma(X)$ is the value, or contents, of location X in state σ.

Consider the evaluation of an arithmetic expression a in a state σ. We can represent the situation of expression a waiting to be evaluated in state σ by the pair $\langle a, \sigma \rangle$. We shall define an evaluation relation between such pairs and numbers

$$\langle a, \sigma \rangle \to n$$

meaning: expression a in state σ evaluates to n. Call pairs $\langle a, \sigma \rangle$, where a is an arithmetic expression and σ is a state, arithmetic-expression *configurations*.

Consider how we might explain to someone how to evaluate an arithmetic expression $(a_0 + a_1)$. We might say something along the lines of:

1. Evaluate a_0 to get a number n_0 as result and
2. Evaluate a_1 to get a number n_1 as result.
3. Then add n_0 and n_1 to get n, say, as the result of evaluating $a_0 + a_1$.

Although informal we can see that this specifies how to evaluate a sum in terms of how to evaluate its summands; the specification is *syntax-directed*. The formal specification of the evaluation relation is given by rules which follow intuitive and informal descriptions like this rather closely.

We specify the evaluation relation in a syntax-directed way, by the following rules:
Evaluation of numbers:

$$\langle n, \sigma \rangle \to n$$

Thus any number is already evaluated with itself as value.
Evaluation of locations:

$$\langle X, \sigma \rangle \to \sigma(X)$$

Thus a location evaluates to its contents in a state.
Evaluation of sums:

$$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 + a_1, \sigma \rangle \to n} \qquad \text{where } n \text{ is the sum of } n_0 \text{ and } n_1.$$

Evaluation of subtractions:

$$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 - a_1, \sigma \rangle \to n} \qquad \text{where } n \text{ is the result of subtracting } n_1 \text{ from } n_0.$$

Evaluation of products:

$$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 \times a_1, \sigma \rangle \to n} \qquad \text{where } n \text{ is the product of } n_0 \text{ and } n_1.$$

How are we to read such rules? The rule for sums can be read as:
If $\langle a_0, \sigma \rangle \to n_0$ and $\langle a_1, \sigma \rangle \to n_1$ then $\langle a_0 + a_1, \sigma \rangle \to n$, where n is the sum of n_0 and n_1. The rule has a *premise* and a *conclusion* and we have followed the common practice of writing the rule with the premise above and the conclusion below a solid line. The rule will be applied in derivations where the facts below the line are derived from facts above.

Some rules like those for evaluating numbers or locations require no premise. Sometimes they are written with a line, for example, as in

$$\frac{}{\langle n, \sigma\rangle \to n}.$$

Rules with empty premises are called *axioms*. Given any arithmetic expression a, state σ and number n, we take a in σ to evaluate to n, *i.e.* $\langle a, \sigma\rangle \to n$, if it can be derived from the rules starting from the axioms, in a way to be made precise soon.

The rule for sums expresses that the sum of two expressions evaluates to the number which is obtained by summing the two numbers which the summands evaluate to. It leaves unexplained the mechanism by which the sum of two numbers is obtained. I have chosen not to analyse in detail how numerals are constructed and the above rules only express how locations and operations $+, -, \times$ can be eliminated from expressions to give the number they evaluate to. If, on the other hand, we chose to describe a particular numeral system, like decimal or roman, further rules would be required to specify operations like multiplication. Such a level of description can be important when considering devices in hardware, for example. Here we want to avoid such details—we all know how to do simple arithmetic!

The rules for evaluation are written using metavariables n, X, a_0, a_1 ranging over the appropriate syntactic sets as well as σ ranging over states. A *rule instance* is obtained by instantiating these to particular numbers, locations and expressions and states. For example, when σ_0 is the particular state, with 0 in each location, this is a rule instance:

$$\frac{\langle 2, \sigma_0\rangle \to 2 \quad \langle 3, \sigma_0\rangle \to 3}{\langle 2 \times 3, \sigma_0\rangle \to 6}$$

So is this:

$$\frac{\langle 2, \sigma_0\rangle \to 3 \quad \langle 3, \sigma_0\rangle \to 4}{\langle 2 \times 3, \sigma_0\rangle \to 12},$$

though not one in which the premises, or conclusion, can ever be derived.

To see the structure of derivations, consider the evaluation of $a \equiv (\mathrm{Init} + 5) + (7 + 9)$ in state σ_0, where Init is a location with $\sigma_0(\mathrm{Init}) = 0$. Inspecting the rules we see that this requires the evaluation of $(\mathrm{Init} + 5)$ and $(7 + 9)$ and these in turn may depend on other evaluations. In fact the evaluation of $\langle a, \sigma_0\rangle$ can be seen as depending on a tree of evaluations:

$$\frac{\dfrac{\dfrac{}{\langle \mathrm{Init}, \sigma_0\rangle \to 0} \quad \dfrac{}{\langle 5, \sigma_0\rangle \to 5}}{\langle (\mathrm{Init} + 5), \sigma_0\rangle \to 5} \qquad \dfrac{\dfrac{}{\langle 7, \sigma_0\rangle \to 7} \quad \dfrac{}{\langle 9, \sigma_0\rangle \to 9}}{\langle 7 + 9, \sigma_0\rangle \to 16}}{\langle (\mathrm{Init} + 5) + (7 + 9), \sigma_0\rangle \to 21}$$

We call such a structure a *derivation tree* or simply a *derivation.* It is built out of instances of the rules in such a way that all the premises of instances of rules which occur are conclusions of instances of rules immediately above them, so right at the top come the axioms, marked by the lines with no premises above them. The conclusion of the bottom-most rule is called the conclusion of the derivation. Something is said to be *derived* from the rules precisely when there is a derivation with it as conclusion.

In general, we write $\langle a, \sigma \rangle \to n$, and say a in σ evaluates to n, iff it can be derived from the rules for the evaluation of arithmetic expressions. The particular derivation above concludes with

$$\langle (\mathrm{Init} + 5) + (7 + 9), \sigma_0 \rangle \to 21.$$

It follows that $(\mathrm{Init} + 5) + (7 + 9)$ in state σ evaluates to 21—just what we want.

Consider the problem of evaluating an arithmetic expression a in some state σ. This amounts to finding a derivation in which the left part of the conclusion matches $\langle a, \sigma \rangle$. The search for a derivation is best achieved by trying to build a derivation in an upwards fashion: Start by finding a rule with conclusion matching $\langle a, \sigma \rangle$; if this is an axiom the derivation is complete; otherwise try to build derivations up from the premises, and, if successful, fill in the conclusion of the first rule to complete the derivation with conclusion of the form $\langle a, \sigma \rangle \to n$.

Although it doesn't happen for the evaluation of arithmetic expressions, in general, more than one rule has a left part which matches a given configuration. To guarantee finding a derivation tree with conclusion that matches, when one exists, all of the rules with left part matching the configuration must be considered, to see if they can be the conclusions of derivations. All possible derivations with conclusion of the right form must be constructed "in parallel".

In this way the rules provide an algorithm for the evaluation of arithmetic expressions based on the search for a derivation tree. Because it can be implemented fairly directly the rules specify the meaning, or semantics, of arithmetic expressions in an operational way, and the rules are said to give an *operational semantics* of such expressions. There are other ways to give the meaning of expressions in a way that leads fairly directly to an implementation. The way we have chosen is just one—any detailed description of an implementation is also an operational semantics. The style of semantics we have chosen is one which is becoming prevalent however. It is one which is often called *structural operational semantics* because of the syntax-directed way in which the rules are presented. It is also called *natural semantics* because of the way derivations resemble proofs in natural deduction—a method of constructing formal proofs. We shall see more complicated, and perhaps more convincing, examples of operational semantics later.

The evaluation relation determines a natural equivalence relation on expressions. De-

fine

$$a_0 \sim a_1 \text{ iff } (\forall n \in \mathbf{N} \forall \sigma \in \Sigma.\ \langle a_0, \sigma \rangle \to n \iff \langle a_1, \sigma \rangle \to n),$$

which makes two arithmetic expressions equivalent if they evaluate to the same value in all states.

Exercise 2.2 Program the rules for the evaluation of arithmetic expressions in Prolog and/or ML (or another language of your choice). This, of course, requires a representation of the abstract syntax of such expressions in Prolog and/or ML. □

2.3 The evaluation of boolean expressions

We show how to evaluate boolean expressions to truth values (**true**, **false**) with the following rules:

$$\langle \mathbf{true}, \sigma \rangle \to \mathbf{true}$$

$$\langle \mathbf{false}, \sigma \rangle \to \mathbf{false}$$

$$\frac{\langle a_0, \sigma \rangle \to n \quad \langle a_1, \sigma \rangle \to m}{\langle a_0 = a_1, \sigma \rangle \to \mathbf{true}} \quad \text{if } n \text{ and } m \text{ are equal}$$

$$\frac{\langle a_0, \sigma \rangle \to n \quad \langle a_1, \sigma \rangle \to m}{\langle a_0 = a_1, \sigma \rangle \to \mathbf{false}} \quad \text{if } n \text{ and } m \text{ are unequal}$$

$$\frac{\langle a_0, \sigma \rangle \to n \quad \langle a_1, \sigma \rangle \to m}{\langle a_0 \leq a_1, \sigma \rangle \to \mathbf{true}} \quad \text{if } n \text{ is less than or equal to } m$$

$$\frac{\langle a_0, \sigma \rangle \to n \quad \langle a_1, \sigma \rangle \to m}{\langle a_0 \leq a_1, \sigma \rangle \to \mathbf{false}} \quad \text{if } n \text{ is not less than or equal to } m$$

$$\frac{\langle b, \sigma \rangle \to \mathbf{true}}{\langle \neg b, \sigma \rangle \to \mathbf{false}} \qquad \frac{\langle b, \sigma \rangle \to \mathbf{false}}{\langle \neg b, \sigma \rangle \to \mathbf{true}}$$

$$\frac{\langle b_0, \sigma\rangle \to t_0 \quad \langle b_1, \sigma\rangle \to t_1}{\langle b_0 \wedge b_1, \sigma\rangle \to t}$$

where t is **true** if $t_0 \equiv$ **true** and $t_1 \equiv$ **true**, and is **false** otherwise.

$$\frac{\langle b_0, \sigma\rangle \to t_0 \quad \langle b_1, \sigma\rangle \to t_1}{\langle b_0 \vee b_1, \sigma\rangle \to t}$$

where t is **true** if $t_0 \equiv$ **true** or $t_1 \equiv$ **true**, and is **false** otherwise.

This time the rules tell us how to eliminate all boolean operators and connectives and so reduce a boolean expression to a truth value.

Again, there is a natural equivalence relation on boolean expressions. Two expressions are equivalent if they evaluate to the same truth value in all states. Define

$$b_0 \sim b_1 \text{ iff } \forall t \forall \sigma \in \Sigma.\ \langle b_0, \sigma\rangle \to t \iff \langle b_1, \sigma\rangle \to t.$$

It may be a concern that our method of evaluating expressions is not the most efficient. For example, according to the present rules, to evaluate a conjunction $b_0 \wedge b_1$ we must evaluate both b_0 and b_1 which is clearly unnecessary if b_0 evaluates to **false** before b_1 is fully evaluated. A more efficient evaluation strategy is to first evaluate b_0 and then only in the case where its evaluation yields **true** to proceed with the evaluation of b_1. We can call this strategy *left-first-sequential* evaluation. Its evaluation rules are:

$$\frac{\langle b_0, \sigma\rangle \to \textbf{false}}{\langle b_0 \wedge b_1, \sigma\rangle \to \textbf{false}}$$

$$\frac{\langle b_0, \sigma\rangle \to \textbf{true} \quad \langle b_1, \sigma\rangle \to \textbf{false}}{\langle b_0 \wedge b_1, \sigma\rangle \to \textbf{false}}$$

$$\frac{\langle b_0, \sigma\rangle \to \textbf{true} \quad \langle b_1, \sigma\rangle \to \textbf{true}}{\langle b_0 \wedge b_1, \sigma\rangle \to \textbf{true}}$$

Exercise 2.3 Write down rules to evaluate boolean expressions of the form $b_0 \vee b_1$, which take advantage of the fact that there is no need to evaluate b in **true** $\vee\ b$ as the result will be true independent of the result of evaluating b. The rules written down should describe a method of left-sequential evaluation. Of course, by symmetry, there is a method of right-sequential evaluation. □

Exercise 2.4 Write down rules which express the "parallel" evaluation of b_0 and b_1 in $b_0 \vee b_1$ so that $b_0 \vee b_1$ evaluates to **true** if either b_0 evaluates to **true**, and b_1 is unevaluated, or b_1 evaluates to **true**, and b_0 is unevaluated. □

It may have been felt that we side-stepped too many issues by assuming we were given mechanisms to perform addition or conjunction of truth values for example. If so try:

Exercise 2.5 Give a semantics in the same style but for expressions which evaluate to strings (or lists) instead of integers and truth-values. Choose your own basic operations on strings, define expressions based on them, define the evaluation of expressions in the style used above. Can you see how to use your language to implement the expression part of **IMP** by representing integers as strings and operations on integers as operations on strings? (Proving that you have implemented the operations on integers correctly is quite hard.) □

2.4 The execution of commands

The role of expressions is to evaluate to values in a particular state. The role of a program, and so commands, is to execute to change the state. When we execute an **IMP** program we shall assume that initially the state is such that all locations are set to zero. So the *initial state* σ_0 has the property that $\sigma_0(X) = 0$ for all locations X. As we all know the execution may *terminate* in a final state, or may *diverge* and never yield a final state. A pair $\langle c, \sigma \rangle$ represents the *(command) configuration* from which it remains to execute command c from state σ. We shall define a relation

$$\langle c, \sigma \rangle \to \sigma'$$

which means the (full) execution of command c in state σ terminates in final state σ'. For example,

$$\langle X := 5, \sigma \rangle \to \sigma'$$

where σ' is the state σ updated to have 5 in location X. We shall use this notation:

Notation: Let σ be a state. Let $m \in \mathbf{N}$. Let $X \in \mathbf{Loc}$. We write $\sigma[m/X]$ for the state obtained from σ by replacing its contents in X by m, i.e. define

$$\sigma[m/X](Y) = \begin{cases} m & \text{if } Y = X, \\ \sigma(Y) & \text{if } Y \neq X. \end{cases}$$

Now we can instead write

$$\langle X := 5, \sigma \rangle \to \sigma[5/X].$$

The execution relation for arbitrary commands and states is given by the following rules.

Rules for commands

Atomic commands:

$$\langle \mathbf{skip}, \sigma \rangle \to \sigma$$

$$\frac{\langle a, \sigma \rangle \to m}{\langle X := a, \sigma \rangle \to \sigma[m/X]}$$

Sequencing:

$$\frac{\langle c_0, \sigma \rangle \to \sigma'' \quad \langle c_1, \sigma'' \rangle \to \sigma'}{\langle c_0; c_1, \sigma \rangle \to \sigma'}$$

Conditionals:

$$\frac{\langle b, \sigma \rangle \to \mathbf{true} \quad \langle c_0, \sigma \rangle \to \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \to \sigma'}$$

$$\frac{\langle b, \sigma \rangle \to \mathbf{false} \quad \langle c_1, \sigma \rangle \to \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \to \sigma'}$$

While-loops:

$$\frac{\langle b, \sigma \rangle \to \mathbf{false}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \to \sigma}$$

$$\frac{\langle b, \sigma \rangle \to \mathbf{true} \quad \langle c, \sigma \rangle \to \sigma'' \quad \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma'' \rangle \to \sigma'}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \to \sigma'}$$

Again there is a natural equivalence relation on commands. Define

$$c_0 \sim c_1 \text{ iff } \forall \sigma, \sigma' \in \Sigma.\ \langle c_0, \sigma \rangle \to \sigma' \iff \langle c_1, \sigma \rangle \to \sigma'.$$

Exercise 2.6 Complete Exercise 2.2 of Section 2.2, by coding the rules for the evaluation of boolean expressions and execution of commands in Prolog and/or ML. □

Exercise 2.7 Let $w \equiv$ **while true do skip**. By considering the form of derivations, explain why, for any state σ, there is no state σ' such that $\langle w, \sigma \rangle \to \sigma'$. □

2.5 A simple proof

The operational semantics of the syntactic sets **Aexp**, **Bexp** and **Com** has been given using the same method. By means of rules we have specified the evaluation relations of

both types of expressions and the execution relation of commands. All three relations are examples of the general notion of *transition relations*, or *transition systems*, in which the configurations are thought of as some kind of state and the relations as expressing possible transitions, or changes, between states. For instance, we can consider each of

$$\langle 3, \sigma\rangle \to 3, \quad \langle \mathbf{true}, \sigma\rangle \to \mathbf{true}, \quad \langle X := 2, \sigma\rangle \to \sigma[2/X].$$

to be transitions.

Because the transition systems for **IMP** are given by rules, we have an elementary, but very useful, proof technique for proving properties of the operational semantics **IMP**.

As an illustration, consider the execution of a while-command $w \equiv \mathbf{while}\ b\ \mathbf{do}\ c$, with $b \in \mathbf{Bexp}, c \in \mathbf{Com}$, in a state σ. We expect that if b evaluates to **true** in σ, then w executes as c followed by w again, and otherwise, in the case where b evaluates to **false**, that the execution of w terminates immediately with the state unchanged. This informal explanation of the execution of commands leads us to expect that for all states σ, σ'

$$\langle w, \sigma\rangle \to \sigma' \text{ iff } \langle \mathbf{if}\ b\ \mathbf{then}\ c; w\ \mathbf{else}\ \mathbf{skip}, \sigma\rangle \to \sigma',$$

i.e., that the following proposition holds.

Proposition 2.8 *Let* $w \equiv \mathbf{while}\ b\ \mathbf{do}\ c$ *with* $b \in \mathbf{Bexp}, c \in \mathbf{Com}$. *Then*

$$w \sim \mathbf{if}\ b\ \mathbf{then}\ c; w\ \mathbf{else}\ \mathbf{skip}.$$

Proof: We want to show

$$\langle w, \sigma\rangle \to \sigma' \text{ iff } \langle \mathbf{if}\ b\ \mathbf{then}\ c; w\ \mathbf{else}\ \mathbf{skip}, \sigma\rangle \to \sigma',$$

for all states σ, σ'.

"$\Rightarrow$": Suppose $\langle w, \sigma\rangle \to \sigma'$, for states σ, σ'. Then there must be a derivation of $\langle w, \sigma\rangle \to \sigma'$. Consider the possible forms such a derivation can take. Inspecting the rules for commands we see the final rule of the derivation is either

$$\frac{\langle b, \sigma\rangle \to \mathbf{false}}{\langle w, \sigma\rangle \to \sigma} \qquad (1 \Rightarrow)$$

or

$$\frac{\langle b, \sigma\rangle \to \mathbf{true} \quad \langle c, \sigma\rangle \to \sigma'' \quad \langle w, \sigma''\rangle \to \sigma'}{\langle w, \sigma\rangle \to \sigma'} \qquad (2 \Rightarrow)$$

In case $(1 \Rightarrow)$, the derivation of $\langle w, \sigma\rangle \to \sigma'$ must have the form

$$\dfrac{\dfrac{\vdots}{\langle b, \sigma\rangle \to \mathbf{false}}}{\langle w, \sigma\rangle \to \sigma}$$

which includes a derivation of $\langle b, \sigma\rangle \to \textbf{false}$. Using this derivation we can build the following derivation of $\langle \textbf{if } b \textbf{ then } c; w \textbf{ else skip}, \sigma\rangle \to \sigma$:

$$\frac{\dfrac{\vdots}{\langle b, \sigma\rangle \to \textbf{false}} \quad \dfrac{}{\langle \textbf{skip}, \sigma\rangle \to \sigma}}{\langle \textbf{if } b \textbf{ then } c; w \textbf{ else skip}, \sigma\rangle \to \sigma}$$

In case $(2 \Rightarrow)$, the derivation of $\langle w, \sigma\rangle \to \sigma'$ must take the form

$$\frac{\dfrac{\vdots}{\langle b, \sigma\rangle \to \textbf{true}} \quad \dfrac{\vdots}{\langle c, \sigma\rangle \to \sigma''} \quad \dfrac{\vdots}{\langle w, \sigma''\rangle \to \sigma'}}{\langle w, \sigma\rangle \to \sigma'}$$

which includes derivations of $\langle b, \sigma\rangle \to \textbf{true}$, $\langle c, \sigma\rangle \to \sigma''$ and $\langle w, \sigma''\rangle \to \sigma'$. From these we can obtain a derivation of $\langle c; w, \sigma\rangle \to \sigma'$, *viz.*

$$\frac{\dfrac{\vdots}{\langle c, \sigma\rangle \to \sigma''} \quad \dfrac{\vdots}{\langle w, \sigma''\rangle \to \sigma'}}{\langle c; w, \sigma\rangle \to \sigma'}$$

We can incorporate this into a derivation:

$$\frac{\dfrac{\vdots}{\langle b, \sigma\rangle \to \textbf{true}} \quad \dfrac{\dfrac{\vdots}{\langle c, \sigma\rangle \to \sigma''} \quad \dfrac{\vdots}{\langle w, \sigma''\rangle \to \sigma'}}{\langle c; w, \sigma\rangle \to \sigma'}}{\langle \textbf{if } b \textbf{ then } c; w \textbf{ else skip}, \sigma\rangle \to \sigma'}$$

In either case, $(1 \Rightarrow)$ or $(2 \Rightarrow)$, we obtain a derivation of

$$\langle \textbf{if } b \textbf{ then } c; w \textbf{ else skip}, \sigma\rangle \to \sigma'$$

from a derivation of

$$\langle w, \sigma\rangle \to \sigma'.$$

Thus

$$\langle w, \sigma\rangle \to \sigma' \text{ implies } \langle \textbf{if } b \textbf{ then } c; w \textbf{ else skip}, \sigma\rangle \to \sigma',$$

for any states σ, σ'.

"$\Leftarrow$": We also want to show the converse, that $\langle \textbf{if } b \textbf{ then } c; w \textbf{ else skip}, \sigma\rangle \to \sigma'$ implies $\langle w, \sigma\rangle \to \sigma'$, for all states σ, σ'.

Suppose $\langle \textbf{if } b \textbf{ then } c; w \textbf{ else skip}, \sigma\rangle \to \sigma'$, for states σ, σ'. Then there is a derivation with one of two possible forms:

$$\dfrac{\dfrac{\vdots}{\langle b, \sigma\rangle \to \textbf{false}} \quad \overline{\langle \textbf{skip}, \sigma\rangle \to \sigma}}{\langle \textbf{if } b \textbf{ then } c; w \textbf{ else skip}, \sigma\rangle \to \sigma} \qquad (1 \Leftarrow)$$

$$\dfrac{\dfrac{\vdots}{\langle b, \sigma\rangle \to \textbf{true}} \quad \dfrac{\vdots}{\langle c; w, \sigma\rangle \to \sigma'}}{\langle \textbf{if } b \textbf{ then } c; w \textbf{ else skip}, \sigma\rangle \to \sigma'} \qquad (2 \Leftarrow)$$

where in the first case, we also have $\sigma' = \sigma$, got by noting the fact that

$$\overline{\langle \textbf{skip}, \sigma\rangle \to \sigma}$$

is the only possible derivation associated with **skip**.

From either derivation, $(1 \Leftarrow)$ or $(2 \Leftarrow)$, we can construct a derivation of $\langle w, \sigma\rangle \to \sigma'$. The second case, $(2 \Leftarrow)$, is the more complicated. Derivation $(2 \Leftarrow)$ includes a derivation of $\langle c; w, \sigma\rangle \to \sigma'$ which has to have the form

$$\dfrac{\dfrac{\vdots}{\langle c, \sigma\rangle \to \sigma''} \quad \dfrac{\vdots}{\langle w, \sigma''\rangle \to \sigma'}}{\langle c; w, \sigma\rangle \to \sigma'}$$

for some state σ''. Using the derivations of $\langle c, \sigma\rangle \to \sigma''$ and $\langle w, \sigma''\rangle \to \sigma'$ with that for $\langle b, \sigma\rangle \to \textbf{true}$, we can produce the derivation

$$\dfrac{\dfrac{\vdots}{\langle b, \sigma\rangle \to \textbf{true}} \quad \dfrac{\vdots}{\langle c, \sigma\rangle \to \sigma''} \quad \dfrac{\vdots}{\langle w, \sigma''\rangle \to \sigma'}}{\langle w, \sigma\rangle \to \sigma'}$$

More directly, from the derivation $(1 \Leftarrow)$, we can construct a derivation of $\langle w, \sigma\rangle \to \sigma'$ (How?).

Thus if $\langle \textbf{if } b \textbf{ then } c; w \textbf{ else skip}, \sigma\rangle \to \sigma'$ then $\langle w, \sigma\rangle \to \sigma'$ for any states σ, σ'.

We can now conclude that

$$\langle w, \sigma\rangle \to \sigma' \text{ iff } \langle \textbf{if } b \textbf{ then } c; w \textbf{ else skip}, \sigma\rangle \to \sigma',$$

for all states σ, σ', and hence

$$w \sim \textbf{if } b \textbf{ then } c; w \textbf{ else skip}$$

as required. □

This simple proof of the equivalence of while-command and its conditional unfolding exhibits an important technique: in order to prove a property of an operational semantics it is helpful to consider the various possible forms of derivations. This idea will be used again and again, though never again in such laborious detail. Later we shall meet other techniques, like "rule induction" which, in principle, can supplant the technique used here. The other techniques are more abstract however, and sometimes more confusing to apply. So keep in mind the technique of considering the forms of derivations when reasoning about operational semantics.

2.6 Alternative semantics

The evaluation relations

$$\langle a, \sigma \rangle \to n \text{ and } \langle b, \sigma \rangle \to t$$

specify the evaluation of expressions in rather large steps; given an expression and a state they yield a value directly. It is possible to give rules for evaluation which capture single steps in the evaluation of expressions. We could instead have defined an evaluation relation between pairs of configurations, taking *e.g.*

$$\langle a, \sigma \rangle \to_1 \langle a', \sigma' \rangle$$

to mean one step in the evaluation of a in state σ yields a' in state σ'. This intended meaning is formalised by taking rules such as the following to specify single steps in the left-to-right evaluation of sum.

$$\frac{\langle a_0, \sigma \rangle \to_1 \langle a_0', \sigma \rangle}{\langle a_0 + a_1, \sigma \rangle \to_1 \langle a_0' + a_1, \sigma \rangle}$$

$$\frac{\langle a_1, \sigma \rangle \to_1 \langle a_1', \sigma \rangle}{\langle n + a_1, \sigma \rangle \to_1 \langle n + a_1', \sigma \rangle}$$

$$\langle n + m, \sigma \rangle \to_1 \langle p, \sigma \rangle$$

where p is the sum of m and n.

Note how the rules formalise the intention to evaluate sums in a left-to-right sequential fashion. To spell out the meaning of the first sum rule above, it says: if one step in the evaluation of a_0 in state σ leads to a_0' in state σ then one step in the evaluation of $a_0 + a_1$ in state σ leads to $a_0' + a_1$ in state σ. So to evaluate a sum first evaluate the component

expression of the sum and when this leads to a number evaluate the second component of the sum, and finally add the corresponding numerals (and we assume a mechanism to do this is given).

Exercise 2.9 Complete the task, begun above, of writing down the rules for $\rightarrow_1$, one step in the evaluation of integer and boolean expressions. What evaluation strategy have you adopted (left-to-right sequential or $\cdots$) ? □

We have chosen to define full execution of commands in particular states through a relation

$$\langle c, \sigma \rangle \rightarrow \sigma'$$

between command configurations. We could instead have based our explanation of the execution of commands on a relation expressing single steps in the execution. A single step relation between two command configurations

$$\langle c, \sigma \rangle \rightarrow_1 \langle c', \sigma' \rangle$$

means the execution of one instruction in c from state σ leads to the configuration in which it remains to execute c' in state σ'. For example,

$$\langle X := 5; Y := 1, \sigma \rangle \rightarrow_1 \langle Y := 1, \sigma[5/X] \rangle.$$

Of course, as this example makes clear, if we consider continuing the execution, we need some way to represent the fact that the command is empty. A configuration with no command left to execute can be represented by a state standing alone. So continuing the execution above we obtain

$$\langle X := 5; Y := 1, \sigma \rangle \rightarrow_1 \langle Y := 1, \sigma[5/X] \rangle \rightarrow_1 \sigma[5/X][1/Y].$$

We leave the detailed presentation of rules for the definition of this one-step execution relation to an exercise. But note there is some choice in what is regarded as a single step. If

$$\langle b, \sigma \rangle \rightarrow_1 \langle \mathbf{true}, \sigma \rangle$$

do we wish

$$\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \rightarrow_1 \langle c_0, \sigma \rangle$$

or

$$\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \rightarrow_1 \langle \mathbf{if\ true\ then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle$$

to be a single step? For the language **IMP** these issues are not critical, but they become so in languages where commands can be executed in parallel; then different choices can effect the final states of execution sequences.

Exercise 2.10 Write down a full set of rules for $\rightarrow_1$ on command configurations, so $\rightarrow_1$ stands for a single step in the execution of a command from a particular state, as discussed above. Use command configurations of the form $\langle c, \sigma \rangle$ and σ when there is no more command left to execute. Point out where you have made a choice in the rules between alternative understandings of what constitutes a single step in the execution. (Showing $\langle c, \sigma \rangle \rightarrow_1^* \sigma'$ iff $\langle c, \sigma \rangle \rightarrow \sigma'$ is hard and requires the application of induction principles introduced in the next two chapters.) □

Exercise 2.11 In our language, the evaluation of expressions has no side effects—their evaluation does not change the state. If we were to model side-effects it would be natural to consider instead an evaluation relation of the form

$$\langle a, \sigma \rangle \rightarrow \langle n, \sigma' \rangle$$

where σ' is the state that results from the evaluation of a in original state σ. To introduce side effects into the evaluation of arithmetic expressions of **IMP**, extend them by adding a construct

$$c \textbf{ resultis } a$$

where c is a command and a is an arithmetic expression. To evaluate such an expression, the command c is first executed and then a evaluated in the changed state. Formalise this idea by first giving the full syntax of the language and then giving it an operational semantics. □

2.7 Further reading

A convincing demonstration of the wide applicability of "structural operational semantics", of which this chapter has given a taste, was first set out by Gordon Plotkin in his lecture notes for a course at Aarhus University, Denmark, in 1981 [81]. A research group under the direction Gilles Kahn at INRIA in Sophia Antipolis, France are currently working on mechanical tools to support semantics in this style; they have focussed on evaluation or execution to a final value or state, so following their lead this particular kind of structural operational semantics is sometimes called "natural semantics" [26, 28, 29]. We shall take up the operational semantics of functional languages, and nondeterminism and parallelism in later chapters, where further references will be presented. More on abstract syntax can be found in Wikström's book [101], Mosses' chapter in [68] and Tennent's book [97].