Department of Computer Science
Australian National University

COMP3610

Principles of Programming Languages

The Semantics of Programming Languages

Clem Baker-Finch
October 17, 2007

Contents

This section of the *Principles of Programming Languages* course aims to give a basic introduction to two of the main techniques for specifying the *semantics* (i.e. the 'meaning') of programming languages. The mathematical precision of the specifications admits formal proofs, as will be demonstrated by proving some simple properties of programs and of programming languages.

Chapter 1

Introduction

The essence of formal semantics is the treatment of programs (and more generally, similar things such as electronic circuits and software systems) in a mathematically rigorous way.

There are a number of advantages to such an approach:

- It gives a completely *unambiguous* definition. In comparison, programming language standards and manuals that are written using only descriptive prose are bound to be incomplete and/or ambiguous. A precise definition of the behaviour of programs is of great value to compiler writers (and to programmers if it sufficiently readable).
- Otherwise, formal mathematical analysis is impossible. For example, if we wish to *prove* a compiler correct or verify that a program or system satisfies its specification then we need to be able to treat both the programs and the specifications as mathematical objects. This is increasingly important for safety-critical applications such as heart pace-makers and other life-support systems, fly-by-wire control systems and the like. Among the strongest critics of Reagan's Strategic Defence Initiative were computer scientists who argued that the software controlling the weapons could never be guaranteed reliable. SDI has recently resurfaced as the National Missile Defense Program (NMD) which is currently being promoted by some sections of the U.S. adminstration. There are numerous documented cases where incorrect software has had disastrous results. Recent ones include frequent undetected radiation overdoses on an X-ray machine due to an initialisation error; the Hubble telescope; the lost Venus space shot; The Ariane 5 failure; and the Pentium chip floating point error. A good source of information on such matter is the moderated newsgroup `comp.risks`.
- Program analyses and transformations, such as are used in sophisticated optimising compilers, are often expressed and designed in terms of the semantics of the programming language.

Of course, an *implementation* of a particular programming language on a particular platform can also serve as an unambiguous definition of its behaviour — run a program and it does what it does! On the other hand, there is no way to show that it agrees with the ISO Standard for Modula-2 or C or whatever and it says nothing helpful about porting to other platforms. Most importantly, it is of no use for *proving properties of programs*, the aim outlined in the second dot point above.

There are two aspects to the specification of programming languages:

syntax: what a program *looks like*;

semantics: the *meaning* of a program.

Very crudely, there are several ways to specify the semantics of programming languages:

operational semantics: the *behaviour* of programs;

denotational semantics: a function from initial to final state (e.g. input to output);

axiomatic semantics: a pre-condition, post-condition pair describing some particular properties of interest.

1.1 Abstract syntax

Right now we are not concerned with specifications of the *syntax* of programming languages. Hence we will use a BNF-like notation but we will not be concerned with 'concrete' matters such as ambiguity or deterministic parsing. For example, consider the following (ambiguous) syntax for simple arithmetic expressions:

$$e ::= n \mid id \mid e + e \mid e * e$$

You should compare this with the rather complicated hierarchy used in a typical syntactic definition for a language like Modula-2, C or Ada. For the productions above, there are two different ways to parse $x + y * 3$:

Only the leftmost parse tree properly reflects the priority of operations. To avoid this kind of problem which really has little to do with the semantics of programming languages, we will assume that we are given a *parse tree* rather that some string of the language. Hence the term 'abstract syntax.'

The main example programming language will be a simple language of *while-loops* which we will call **Imp** (for *imp*erative). Its abstract syntax is as follows:

Syntactic Domains

n : Num	numerals
x : Ide	variable identifiers
a : Aexp	arithmetic expressions
b : Bexp	boolean expressions
c : Com	commands (i.e. statements)

Syntax

$$a ::= n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2$$
$$b ::= \textbf{true} \mid \textbf{false} \mid a_1 == a_2 \mid a_1 <= a_2 \mid \textbf{not}\ b \mid b_1\ \textbf{and}\ b_2 \mid b_1\ \textbf{or}\ b_2$$
$$c ::= x := a \mid \textbf{skip} \mid c_1; c_2 \mid \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \mid \textbf{while}\ b\ \textbf{do}\ c$$

We will gloss over irrelevant details by assuming that Num and Ide are defined elsewhere. For example, we could assume that numerals are optionally signed strings of decimal digits and that variable identifiers are represented by strings of letters.

We will often use the term *program phrase* to refer to a fragment of an **Imp** program that is well-formed, in the sense that it belongs to one of the syntactic domains. It is inconvenient and clumsy to draw parse trees, so we will use parentheses to indicate their shape. for example the two parse trees above could be written as $x + (y * 3)$ and $(x + y) * 3$ respectively.

In the following three sections, the operational, denotational and axiomatic approaches to semantics will be introduced through a simple example program which swaps the value in two variable identifiers, x and y:

$$z := x;\ x := y;\ y := z$$

In each case we need the notion of a *State* which is an object relating variables (Ide) to their values ($\mathbb{Z}$ is sufficient for **Imp**). Most generally, states can be modelled by (partial) functions Ide $\to \mathbb{Z}$, but a data structure such as a list of pairs is an acceptable alternative.

Without being too precise, the following notation represents the state where x has value 5, y has value 7, z has value 0 and the value of all other variable identifiers is undefined:

$$[x \mapsto 5, y \mapsto 7, z \mapsto 0]$$

While working through the semantics of particular programs may help you to understand the techniques, it is important to emphasise that it is not a common activity. *The primary application of formal semantic definitions is to prove general properties about programming languages and programs* and that will be our main focus. Nevertheless, going through the details of some some simple examples will often help you to grasp the basic ideas.

1.2 Operational semantics

The essence of operational semantics is that the meaning of a program phrase is specified by its computational behaviour during execution. The central focus is on *how* the result or effect is obtained.

The most common modern technique is *Structural Operational Semantics (SOS)*, developed by Gordon Plotkin about 1981. The word 'structural' is to indicate that the semantic definitions are *syntax-directed*. Essentially, there will be a semantic rule corresponding to each syntactic construct, and the meaning of each construct will be defined in terms of its subcomponents.

There are two main approaches within Structural Operational Semantics:

Computation semantics is expressed as a relation which defines how an expression or command successively rewrites or evolves to another. In other words, it describes the fine-grained steps taken in a computation. Computation semantics is often called *small-step* semantics.

Evaluation semantics is expressed as a relation which describes how the value of an expression or the effect of a command is computed, in terms of the complete computation of its component expressions and commands. Evaluation semantics is often called *big-step* or *natural* semantics.

We will concentrate on *evaluation* semantics in this course. The most important proof technique for this technique is *rule induction* which is essentially *induction over the structure of derivation trees*.

Example 1.2.1
The behaviour (i.e. the operational semantics) of the *swap* program can be specified as follows.

The execution of a sequence of commands is given by the separate execution of the commands one after the other, in order from left to right. The execution of an assignment statement $x := a$ is to evaluate a in the current state, then bind that value to x, creating a new state.

$$\begin{aligned}\langle z := x; x := y; y := z,\ [x \mapsto 5, y \mapsto 7, z \mapsto 0]\rangle \\ &\Rightarrow\ \langle x := y; y := z,\ [x \mapsto 5, y \mapsto 7, z \mapsto 5]\rangle \\ &\Rightarrow\ \langle y := z,\ [x \mapsto 7, y \mapsto 7, z \mapsto 5]\rangle \\ &\Rightarrow\ [x \mapsto 7, y \mapsto 5, z \mapsto 5]\end{aligned}$$

The evaluation semantics approach is to describe an *evaluation relation* $\_ \Downarrow \_$.[1] The rules defining the relation are usually given in a notation like the natural deduction rules in logic. For example:

$$\frac{\langle c_1, \sigma_1\rangle \Downarrow \sigma_2 \qquad \langle c_2, \sigma_2\rangle \Downarrow \sigma_3}{\langle c_1; c_2, \sigma_1\rangle \Downarrow \sigma_3}$$

[1] In fact, there is usually a *family* of such relations but we prefer to overload $\Downarrow$, and allow the context to indicate which is intended.

indicates that if we have already shown that $\langle c_1, \sigma_1 \rangle \Downarrow \sigma_2$ and $\langle c_2, \sigma_2 \rangle \Downarrow \sigma_3$ then we can deduce that $\langle c_1; c_2, \sigma_1 \rangle \Downarrow \sigma_3$.

For assignment:

$$\frac{\langle a, \sigma \rangle \Downarrow n}{\langle x := a, \sigma \rangle \Downarrow \sigma[x \mapsto n]}$$

means that if an arithmetic expression a evaluates in state σ to $\mathbf{n}$, then the assignment statement $x := a$ produces a new state $\sigma[x \mapsto n]$.

1.3 Denotational semantics

This technique was developed by Dana Scott and Christopher Strachey in the late 1960's. Essentially, the meaning of a program phrase is modelled by a *mathematical object*, generally a continuous function. For example, the meaning of a program may be given as a function from *Input* to *Output*.

The focus of denotational semantics is on the *effect* or *result*, not how it is obtained.

The most important proof techniques are *structural induction* and *fixpoint induction*. The mathematical foundations of denotational semantics are beyond the scope of this course, but we will take a brief look.

Example 1.3.1
The denotation of the *swap* program is as follows. Again, don't worry too much about notation for now.

The effect of an **Imp** command is modelled by a partial function State $\rightarrow$ State. The effect of a sequence of commands is given by the *composition* of those functions. The effect of an *assignment* statement $x := a$ is a function which takes a state σ to a new state σ' which is the same as σ except that x is now bound to the value of expression a.

For *swap* we may calculate as follows:

$$\begin{aligned}
\mathcal{C}[\![z := x; x := y; y := z]\!][x \mapsto 5, y \mapsto 7, z \mapsto 0] \\
&= (\mathcal{C}[\![y := z]\!] \circ \mathcal{C}[\![x := y]\!] \circ \mathcal{C}[\![z := x]\!])[x \mapsto 5, y \mapsto 7, z \mapsto 0] \\
&= (\mathcal{C}[\![y := z]\!] \circ \mathcal{C}[\![x := y]\!])[x \mapsto 5, y \mapsto 7, z \mapsto 5] \\
&= \mathcal{C}[\![y := z]\!][x \mapsto 7, y \mapsto 7, z \mapsto 5] \\
&= [x \mapsto 7, y \mapsto 5, z \mapsto 5]
\end{aligned}$$

finally giving:

$$\mathcal{C}[\![z := x; x := y; y := z]\!][x \mapsto 5, y \mapsto 7, z \mapsto 0] = [x \mapsto 7, y \mapsto 5, z \mapsto 5]$$

1.4 Axiomatic semantics

The essence of axiomatic semantics is that *aspects* of the meaning of a program phrase are given as pre-condition/post-condition pairs. While operational and denotational semantics completely determine the behaviour or effect of a program phrase, the axiomatic approach is oriented toward proving particular properties of programs, generally expressed as assertions (about the 'state').

Example 1.4.1
To show that the swap program successfully swaps the values of x and y, we need to prove that:

$$\{x = n \ \wedge \ y = m\} z := x; x := y; y := z \{x = m \ \wedge \ y = n\}$$

note that this expression does not completely capture the effect of the program, since there is no mention of z.

Chapter 2

Operational Semantics

The focus in Operational Semantics is on *how* a program is executed, rather than just the *result* of its execution. We will concentrate on *evaluation semantics*, but there are other operational styles.

Underlying most models of the behaviour of imperative programming languages like **Imp** is the idea of a *state*, giving the current values of the program variables. Expressions are evaluated relative to the state and commands modify the state. A general abstract model of state is provided by a function mapping variables to their values:

$$\sigma \ : \ \text{State} \ = \ \text{Ide} \to \mathbb{Z}$$

State is a new *semantic domain* consisting of partial functions from Ide (a syntactic domain) to $\mathbb{Z}$ (another semantic domain). The definition also indicates that $\sigma, \sigma', \ldots$ will be used as metavariables (i.e. variables used in the semantic definition). Note that we have represented states as *partial* functions, indicating that not all variables need have a value. If **Imp** programs assumed that all variables were initialised (to 0, say) then State could be a total function $\text{Ide} \to \mathbb{Z}$ (initially the constant 0 function).

Definition 2.0.2
If σ is some state, then the value of a variable x in σ is given simply by applying the function σ to x, which is written either as $\sigma(x)$ or simply σx.

Assignment commands will change the bindings of values to variables, so we need some way to 'alter' the state function. To indicate that the new value for x in state σ is n, we write $\sigma[x \mapsto n]$, which is a new state. More precisely, we define:

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

Definition 2.0.3
A pair $\langle c, \sigma \rangle$ is a *Command configuration*, intuitively indicating that command c is to be executed on state σ.

A pair $\langle a, \sigma \rangle$ is an *Aexp configuration*, intuitively indicating that arithmetic expression a is to be evaluated in state σ.

A pair $\langle b, \sigma \rangle$ is an *Bexp configuration*, intuitively indicating that boolean expression b is to be evaluated in state σ.

2.1 Evaluation semantics

Evaluation Semantics is sometimes[1] called *big-step structural operational semantics* because it deals with the relationship between the *initial* and *final* states of an execution, rather than explicit 'atomic' steps. In that aspect, evaluation semantics sits between denotational semantics and *small-step* or *computation* operational semantics.

2.1.1 Evaluation of arithmetic expressions

Write a transition:

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

to represent the evaluation of Aexp a in state σ giving result n. The evaluation mechanism is presented as a *transition relation* which is specified by *rules* with a *conclusion* and zero or more *premises*. The rules correspond to the syntactic constructs of the programming language, **Imp** in this case.

For example, the rule for addition is as follows:

$$\frac{\langle a_0, \sigma \rangle \Downarrow n_0 \qquad \langle a_1, \sigma \rangle \Downarrow n_1}{\langle a_0 + a_1, \sigma \rangle \Downarrow n} \; plus$$

where n is the sum of n_0 and n_1. The rule can be read as follows:

> if a_0 evaluates to n_0 in σ and a_1 evaluates to n_1 in σ (the same state), then $a_0 + a_1$ evaluates to $n_0 + n_1$.

Note 2.1.1
Since this is the first operational semantics rule you have met, it is worth explaining some important points in more detail:

- We will not make any distinction between numerals (which are syntactic entities) and integers (which are semantic entities), using metavariable n in both cases. One justification for this is that we are working with an abstract syntax so we could choose to represent the syntactic domain *Numeral* by $\mathbb{Z}$. An alternative would be to make a distinction between the syntactic symbol n, which is a metavariable representing a numeral occuring as part of the programming language (**Imp**) and the semantic symbol **n** representing an integer which is the value of that numeral.

[1] Most often, in fact.

- Similarly, we didn't write $n_0 + n_1$ in the rule, so as to emphasise the difference between the **Imp** syntactic symbol '+' and the semantic operation of addition, usually written as '+', too. More often, we would write this rule as follows, and expect the reader to distinguish which '+' is intended from its context:

$$\frac{\langle a_0, \sigma\rangle \Downarrow n_0 \qquad \langle a_1, \sigma\rangle \Downarrow n_1}{\langle a_0 + a_1, \sigma\rangle \Downarrow n_0 + n_1}\ \mathit{plus}$$

- a_0 and a_1 are evaluated in the same state. This rule reflects the important fact that arithmetic expressions in **Imp** do not have *side-effects*. Many imperative languages do allow expressions to alter the values of variables, so potentially a_1 may be evaluated in a different state to a_0 in such languages.

- The definition is *syntax directed* — the semantics of $a_0 + a_1$ is defined in terms of the semantics of a_0 and the semantics of a_1.

It was pointed out above that transition rules may have zero premises. If that is the case then the rule is an *axiom*. The rule for Aexps that are simply a numeral is an example of an axiom:

$$\frac{}{\langle n, \sigma\rangle \Downarrow n}\ \mathit{num}$$

The semantics of the other Aexp constructs can be given with similar rules to the two above. The complete set of transition rules is as follows:

$$\frac{}{\langle n, \sigma\rangle \Downarrow n}\ \mathit{num}$$

$$\frac{}{\langle x, \sigma\rangle \Downarrow \sigma(x)}\ \mathit{var}$$

$$\frac{\langle a_0, \sigma\rangle \Downarrow n_0 \qquad \langle a_1, \sigma\rangle \Downarrow n_1}{\langle a_0 + a_1, \sigma\rangle \Downarrow n_0 + n_1}\ \mathit{plus}$$

$$\frac{\langle a_0, \sigma\rangle \Downarrow n_0 \qquad \langle a_1, \sigma\rangle \Downarrow n_1}{\langle a_0 - a_1, \sigma\rangle \Downarrow n_0 - n_1}\ \mathit{minus}$$

$$\frac{\langle a_0, \sigma\rangle \Downarrow n_0 \qquad \langle a_1, \sigma\rangle \Downarrow n_1}{\langle a_0 * a_1, \sigma\rangle \Downarrow n_0 \times n_1}\ \mathit{times}$$

The evaluation rules are written using metavariables n, x, a_0, a_1 ranging over syntactic domains Num, Ide and Aexp respectively, σ ranging over semantic domain State and n over $\mathbb{Z}$. Instantiating those metavariables to particular syntactic and semantic entities gives *rule instances*. For example (let σ_4 denote some state where the variable x is bound to 4):

$$\frac{\langle x + 5, \sigma_4\rangle \Downarrow 9 \qquad \langle 3, \sigma_4\rangle \Downarrow 3}{\langle (x + 5) * 3, \sigma_4\rangle \Downarrow 27}\ \mathit{times}$$

The evaluation of a particular Aexp is given by a *derivation tree*. A derivation tree has a single conclusion (the root), with rule instances as internal nodes and all leaves are axioms. For example:

$$\dfrac{\dfrac{\dfrac{}{\langle x, \sigma_4\rangle \Downarrow 4}\,var \qquad \dfrac{}{\langle 5, \sigma_4\rangle \Downarrow 5}\,num}{\langle x+5, \sigma_4\rangle \Downarrow 9}\,plus \qquad \dfrac{}{\langle 3, \sigma_4\rangle \Downarrow 3}\,num}{\langle (x+5)*3, \sigma_4\rangle \Downarrow 27}\,times$$

2.1.2 Evaluation of boolean expressions

The operational semantics transition rules defining the evaluation of boolean expressions are straighforward and similar to those for arithmetic expressions above. Again there are no side-effects. The semantic domain of truth values is $t : \mathbb{B} = \{\mathbf{tt}, \mathbf{ff}\}$. Note that the semantics of Bexp depends on the semantics of Aexp.

$$\frac{}{\langle \mathbf{true}, \sigma\rangle \Downarrow \mathbf{tt}}\,true$$

$$\frac{}{\langle \mathbf{false}, \sigma\rangle \Downarrow \mathbf{ff}}\,false$$

$$\frac{\langle a_0, \sigma\rangle \Downarrow n_0 \qquad \langle a_1, \sigma\rangle \Downarrow n_1}{\langle a_0 == a_1, \sigma\rangle \Downarrow (n_0 = n_1)}\,equal$$

$$\frac{\langle a_0, \sigma\rangle \Downarrow n_0 \qquad \langle a_1, \sigma\rangle \Downarrow n_1}{\langle a_0 <= a_1, \sigma\rangle \Downarrow (n_0 \leq n_1)}\,leq$$

$$\frac{\langle b, \sigma\rangle \Downarrow t}{\langle \mathbf{not}\ b, \sigma\rangle \Downarrow \neg t}\,not$$

$$\frac{\langle b_0, \sigma\rangle \Downarrow t_0 \qquad \langle b_1, \sigma\rangle \Downarrow t_1}{\langle b_0\ \mathbf{and}\ b_1, \sigma\rangle \Downarrow t_0 \wedge t_1}\,and$$

$$\frac{\langle b_0, \sigma\rangle \Downarrow t_0 \qquad \langle b_1, \sigma\rangle \Downarrow t_1}{\langle b_0\ \mathbf{or}\ b_1, \sigma\rangle \Downarrow t_0 \vee t_1}\,or$$

Exercise 2.1.2
Program the evaluation rules and test them. This is quite easy in a language like Prolog or a functional language, but you can try some other language if you wish. First you need to decide how to represent the syntactic constructs and semantic objects as data structures.

Definition 2.1.3
The evaluation relation determines an equivalence relation on both Aexp and Bexp.

$$a_0 \sim a_1 \quad \text{iff} \quad \forall n : \mathbb{Z}, \forall \sigma : \text{State},\ \langle a_0, \sigma\rangle \Downarrow n \ \Leftrightarrow\ \langle a_1, \sigma\rangle \Downarrow n$$

That is, a_0 and a_1 are $\sim$-equivalent if given any state, they evaluate to the same result in that state. If so, we say that a_0 and a_1 are *semantically equivalent.*

Similarly:

$$b_0 \sim b_1 \quad \text{iff} \quad \forall t : \mathbb{B}, \forall \sigma : \text{State}, \langle b_0, \sigma \rangle \Downarrow t \Leftrightarrow \langle b_1, \sigma \rangle \Downarrow t$$

Note that the semantics for Bexp requires the evaluation of both conjuncts in b_0 **and** b_1 and both disjuncts in b_0 **or** b_1. If we wished to specify *left-sequential* evaluation instead, the following rules would be appropriate:

$$\frac{\langle b_0, \sigma \rangle \Downarrow \mathbf{tt} \quad \langle b_1, \sigma \rangle \Downarrow t_1}{\langle b_0 \textbf{ and } b_1, \sigma \rangle \Downarrow t_1} \, and_{\mathbf{tt}}$$

$$\frac{\langle b_0, \sigma \rangle \Downarrow \mathbf{ff}}{\langle b_0 \textbf{ and } b_1, \sigma \rangle \Downarrow \mathbf{ff}} \, and_{\mathbf{ff}}$$

Similarly for b_0 **or** b_1. (Exercise.)

Exercise 2.1.4
Write down rules that reflect 'parallel' evaluation of b_0 and b_1 in the expressions b_0 **and** b_1 such that b_0 **and** b_1 evaluates to **ff** if either b_0 evaluates to **ff** or b_1 evaluates to **ff**. Otherwise b_0 **and** b_1 evaluates to **tt**.

2.1.3 Executing commands

The effect of executing a command is to change the state, so the transition relation (and hence the premises and conclusions of the operational semantics rules for commands) are of the form:

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

We will go through the rules, discussing them in turn.

Assignment. If an expression a evaluates to n in state σ then the effect of executing $x := a$ on σ is to give a final state which is the same as σ except that x now has value n.

$$\frac{\langle a, \sigma \rangle \Downarrow n}{\langle x := a, \sigma \rangle \Downarrow \sigma[x \mapsto n]} \, asst$$

A simple derivation tree with assignment as the last rule is the following (assume that σ_4 is a state where x has value 4):

$$\dfrac{\dfrac{\dfrac{}{\langle x, \sigma_4 \rangle \Downarrow 4} \, var \qquad \dfrac{}{\langle 1, \sigma_4 \rangle \Downarrow 1} \, num}{\langle x + 1, \sigma_4 \rangle \Downarrow 5} \, plus}{\langle x := x + 1, \sigma_4 \rangle \Downarrow \sigma_4[x \mapsto 5]} \, asst$$

Skip. The **skip** command has no effect on the state. Hence the rule is an axiom:

$$\frac{}{\langle \mathbf{skip}, \sigma \rangle \Downarrow \sigma} \; skip$$

Sequence. To execute $c_0; c_1$ in state σ, first execute c_0 to yield σ'. Then execute c_1 on σ' to yield the final state.

$$\frac{\langle c_0, \sigma \rangle \Downarrow \sigma' \qquad \langle c_1, \sigma' \rangle \Downarrow \sigma''}{\langle c_0; c_1, \sigma \rangle \Downarrow \sigma''} \; seq$$

Conditional. If the condition b evaluates to **tt** then execute the then-part c_0. Note that c_1 is not executed. Similarly, if b evaluates to **ff** then execute c_1. Thus we have two rules:

$$\frac{\langle b, \sigma \rangle \Downarrow \mathbf{tt} \qquad \langle c_0, \sigma \rangle \Downarrow \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \Downarrow \sigma'} \; if_{\mathbf{tt}}$$

$$\frac{\langle b, \sigma \rangle \Downarrow \mathbf{ff} \qquad \langle c_1, \sigma \rangle \Downarrow \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \Downarrow \sigma'} \; if_{\mathbf{ff}}$$

A simple instance follows:

$$\frac{\langle x == 0, \sigma_4 \rangle \Downarrow \mathbf{ff} \qquad \langle x := x + 1, \sigma_4 \rangle \Downarrow \sigma_4[x \mapsto 5]}{\langle \mathbf{if}\ x == 0\ \mathbf{then\ skip\ else}\ x := x + 1, \sigma_4 \rangle \Downarrow \sigma_4[x \mapsto 5]} \; if_{\mathbf{ff}}$$

While. If the condition b evaluates to **tt** the the body c is executed yielding a new state, σ' say. The entire **while**-command is then run on σ'. The $while_{\mathbf{tt}}$ rule corresponds to a single unwinding of the loop. If the condition b evaluates to **ff** then the **while**-command has no effect, thus leaving the state unchanged. The $while_{\mathbf{ff}}$ rule is thus the escape from the loop unwinding.

$$\frac{\langle b, \sigma \rangle \Downarrow \mathbf{tt} \qquad \langle c, \sigma \rangle \Downarrow \sigma' \qquad \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma' \rangle \Downarrow \sigma''}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \Downarrow \sigma''} \; while_{\mathbf{tt}}$$

$$\frac{\langle b, \sigma \rangle \Downarrow \mathbf{ff}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \Downarrow \sigma} \; while_{\mathbf{ff}}$$

2.1.4 Full evaluation semantics of Imp

For easy reference, we repeat the syntax and the entire set of rules defining the operational semantics of **Imp** in the section.

Syntactic Domains

n : Num	numerals
x : Ide	variable identifiers
a : Aexp	arithmetic expressions
b : Bexp	boolean expressions
c : Com	commands (i.e. statements)

Syntax

$$a ::= n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2$$
$$b ::= \textbf{true} \mid \textbf{false} \mid a_1 == a_2 \mid a_1 <= a_2 \mid \textbf{not}\ b \mid b_1\ \textbf{and}\ b_2 \mid b_1\ \textbf{or}\ b_2$$
$$c ::= x := a \mid \textbf{skip} \mid c_1; c_2 \mid \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \mid \textbf{while}\ b\ \textbf{do}\ c$$

Semantics

$$\frac{}{\langle n, \sigma\rangle \Downarrow n}\ num$$

$$\frac{}{\langle x, \sigma\rangle \Downarrow \sigma(x)}\ var$$

$$\frac{\langle a_0, \sigma\rangle \Downarrow n_0 \qquad \langle a_1, \sigma\rangle \Downarrow n_1}{\langle a_0 + a_1, \sigma\rangle \Downarrow n_0 + n_1}\ plus$$

$$\frac{\langle a_0, \sigma\rangle \Downarrow n_0 \qquad \langle a_1, \sigma\rangle \Downarrow n_1}{\langle a_0 - a_1, \sigma\rangle \Downarrow n_0 - n_1}\ minus$$

$$\frac{\langle a_0, \sigma\rangle \Downarrow n_0 \qquad \langle a_1, \sigma\rangle \Downarrow n_1}{\langle a_0 * a_1, \sigma\rangle \Downarrow n_0 \times n_1}\ times$$

$$\frac{}{\langle \textbf{true}, \sigma\rangle \Downarrow \textbf{tt}}\ true$$

$$\frac{}{\langle \textbf{false}, \sigma\rangle \Downarrow \textbf{ff}}\ false$$

$$\frac{\langle a_0, \sigma\rangle \Downarrow n_0 \qquad \langle a_1, \sigma\rangle \Downarrow n_1}{\langle a_0 == a_1, \sigma\rangle \Downarrow (n_0 = n_1)}\ equal$$

$$\frac{\langle a_0, \sigma\rangle \Downarrow n_0 \qquad \langle a_1, \sigma\rangle \Downarrow n_1}{\langle a_0 <= a_1, \sigma\rangle \Downarrow (n_0 \leq n_1)}\ leq$$

$$\frac{\langle b, \sigma\rangle \Downarrow t}{\langle \textbf{not}\ b, \sigma\rangle \Downarrow \neg t}\ not$$

$$\frac{\langle b_0, \sigma\rangle \Downarrow t_0 \qquad \langle b_1, \sigma\rangle \Downarrow t_1}{\langle b_0\ \textbf{and}\ b_1, \sigma\rangle \Downarrow t_0 \wedge t_1}\ and$$

$$\frac{\langle b_0, \sigma\rangle \Downarrow t_0 \qquad \langle b_1, \sigma\rangle \Downarrow t_1}{\langle b_0\ \textbf{or}\ b_1, \sigma\rangle \Downarrow t_0 \vee t_1}\ or$$

$$\frac{\langle a, \sigma\rangle \Downarrow n}{\langle x := a, \sigma\rangle \Downarrow \sigma[x \mapsto n]}\ asst$$

$$\frac{}{\langle \mathbf{skip}, \sigma\rangle \Downarrow \sigma}\ skip$$

$$\frac{\langle c_0, \sigma\rangle \Downarrow \sigma' \qquad \langle c_1, \sigma'\rangle \Downarrow \sigma''}{\langle c_0; c_1, \sigma\rangle \Downarrow \sigma''}\ seq$$

$$\frac{\langle b, \sigma\rangle \Downarrow \mathbf{tt} \qquad \langle c_0, \sigma\rangle \Downarrow \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma\rangle \Downarrow \sigma'}\ if_{\mathbf{tt}}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \mathbf{ff} \qquad \langle c_1, \sigma\rangle \Downarrow \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma\rangle \Downarrow \sigma'}\ if_{\mathbf{ff}}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \mathbf{tt} \qquad \langle c, \sigma\rangle \Downarrow \sigma' \qquad \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma'\rangle \Downarrow \sigma''}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \Downarrow \sigma''}\ while_{\mathbf{tt}}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \mathbf{ff}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \Downarrow \sigma}\ while_{\mathbf{ff}}$$

2.1.5 An example derivation

This section demonstrates some of the rules by constructing a derivation tree for a factorial program:

$$y := 1; \mathbf{while\ not}\ (x == 1)\ \mathbf{do}\ (y := y * x; x := x - 1)$$

Let σ_0 represent a state where all variables have value 0. By running the program on $\sigma_0[x \mapsto 3]$, the final value of y is 3!. To show that this is the case we will construct a derivation tree with root:

$$\langle y := 1; \mathbf{while\ not}\ (x == 1)\ \mathbf{do}\ (y := y * x; x := x - 1), \sigma_0[x \mapsto 3]\rangle \Downarrow \sigma_{61}$$

where $\sigma_{61}(y) = 6$. In one sense, we are solving for σ_{61}.

We work from the root up *guided by the syntactic structure of the program phrase*. For brevity (so it fits on the page) we will write c_{yx} for the loop body $(y := y * x; x := x - 1)$. Since the program is a sequence of commands, the last rule must be *seq*.

$$\frac{\begin{array}{c}T1\\ \vdots\\ \langle y := 1, \sigma_0[x \mapsto 3]\rangle \Downarrow \sigma_{13}\end{array} \qquad \begin{array}{c}T2\\ \vdots\\ \langle \mathbf{while\ not}\ (x == 1)\ \mathbf{do}\ c_{yx}, \sigma_{13}\rangle \Downarrow \sigma_{61}\end{array}}{\langle y := 1; \mathbf{while\ not}\ (x == 1)\ \mathbf{do}\ c_{yx}, \sigma_0[x \mapsto 3]\rangle \Downarrow \sigma_{61}}\ seq$$

Derivation tree $T1$ ends with an assignment, so it must be:

$$\frac{\dfrac{}{\langle 1, \sigma_0[x \mapsto 3]\rangle \Downarrow 1}\ num}{\langle y := 1, \sigma_0[x \mapsto 3]\rangle \Downarrow \sigma_0[y \mapsto 1, x \mapsto 3]}\ asst$$

This shows that $\sigma_{13} = \sigma_0[y \mapsto 1, x \mapsto 3]$, hence we can deduce that $T2$ is:

$$\dfrac{\begin{array}{cc} \overset{T4}{\vdots} & \overset{T5}{\vdots} \\ T3 \quad \langle c_{yx}, \sigma_{13}\rangle \Downarrow \sigma_{32} & \langle \textbf{while not } (x == 1) \textbf{ do } c_{yx}, \sigma_{32}\rangle \Downarrow \sigma_{61} \end{array}}{\langle \textbf{while not } (x == 1) \textbf{ do } c_{yx}, \sigma_{13}\rangle \Downarrow \sigma_{61}}\ \mathit{while}_{\mathbf{tt}}$$

We are justified in using the $\mathit{while}_{\mathbf{tt}}$ rule because $T3$ is:

$$\dfrac{\dfrac{\dfrac{}{\langle x, \sigma_{13}\rangle \Downarrow 3}\ \mathit{var} \qquad \dfrac{}{\langle 1, \sigma_{13}\rangle \Downarrow 1}\ \mathit{num}}{\langle x == 1, \sigma_{13}\rangle \Downarrow \mathbf{ff}}\ \mathit{equal}}{\langle \textbf{not } (x == 1), \sigma_{13}\rangle \Downarrow \mathbf{tt}}\ \mathit{not}$$

By constructing $T4$ it is easy to show that $\sigma_{32} = \sigma_0[y \mapsto 3, x \mapsto 2]$. Following the same process, it is straightforward to construct $T5$ and thus show that $\sigma_{61} = \sigma_0[y \mapsto 6, x \mapsto 1]$. Thus we have $\sigma_{61}(y) = 6$ as expected.

Exercise 2.1.5
By constructing the derivation tree $T5$, show that $\sigma_{61} = \sigma_0[y \mapsto 6, x \mapsto 1]$, as required.

Note 2.1.6
The existence of a transition $\langle c, \sigma\rangle \Downarrow \sigma'$ implies that the execution of c on state σ *terminates*. If there is no such σ' then c *diverges* on state σ. We may indicate this with the notation:

$$\langle c, \sigma\rangle \Uparrow$$

In general, a configuration from which there is no valid transition is called a *stuck configuration*.

Exercise 2.1.7 (Important)
By considering the form of derivations, explain why, for all $\sigma : \mathit{State}$, $\langle \textbf{while true do skip}, \sigma\rangle \Uparrow$. In other words, show that there is no state σ' such that $\langle \textbf{while true do skip}, \sigma\rangle \Downarrow \sigma'$.

Definition 2.1.8
There is a natural equivalence relation on commands, induced by the operational semantics rules for their execution:

$$c_0 \sim c_1 \quad \text{iff} \quad \forall \sigma, \sigma' : \text{State}, \ \langle c_0, \sigma\rangle \Downarrow \sigma' \ \Leftrightarrow \ \langle c_1, \sigma\rangle \Downarrow \sigma'$$

We say that c_0 and c_1 are *semantically equivalent*.

Exercise 2.1.9
Carefully explain the meaning of definition 2.1.8 in the case where c_0 and/or c_1 diverge.

2.1.6 A simple application of evaluation semantics

Proposition 2.1.10
The command

$$\textbf{while } b \textbf{ do } c$$

is semantically equivalent to

$$\textbf{if } b \textbf{ then } c; \textbf{while } b \textbf{ do } c \textbf{ else skip}$$

(Write w for this second command, for brevity.)

That is, $\textbf{while } b \textbf{ do } c \ \sim \ w$.

Proof The proof is in two parts, corresponding the the two directions of '$\Leftrightarrow$'.

part 1: Show $\forall \sigma, \sigma'' : \text{State}, \langle \textbf{while } b \textbf{ do } c, \sigma \rangle \Downarrow \sigma'' \ \Rightarrow \ \langle w, \sigma \rangle \Downarrow \sigma''$.

Assuming $\langle \textbf{while } b \textbf{ do } c, \sigma \rangle \Downarrow \sigma''$, there must be a derivation with that as root, by either the $while_{\textbf{tt}}$ or $while_{\textbf{ff}}$ rules.

i. Suppose the last rule is $while_{\textbf{tt}}$. Then the following derivation exists, for some derivation trees $T0$, $T1$ and $T2$ with respective conclusions $\langle b, \sigma \rangle \Downarrow \textbf{tt}$, $\langle c, \sigma \rangle \Downarrow \sigma'$ and $\langle \textbf{while } b \textbf{ do } c, \sigma' \rangle \Downarrow \sigma''$:

$$\dfrac{\begin{array}{ccc} T0 & T1 & T2 \\ \vdots & \vdots & \vdots \\ \langle b, \sigma \rangle \Downarrow \textbf{tt} & \langle c, \sigma \rangle \Downarrow \sigma' & \langle \textbf{while } b \textbf{ do } c, \sigma' \rangle \Downarrow \sigma'' \end{array}}{\langle \textbf{while } b \textbf{ do } c, \sigma \rangle \Downarrow \sigma''} \ while_{\textbf{tt}}$$

Now using $T0$, and $T1$ and $T2$ as premises to the *seq* rule we can construct the following derivation:

$$\dfrac{\begin{array}{cc} T0 & \dfrac{\begin{array}{cc} T1 & T2 \end{array}}{\langle c; \textbf{while } b \textbf{ do } c, \sigma \rangle \Downarrow \sigma''} \ seq \end{array}}{\langle \textbf{if } b \textbf{ then } c; \textbf{while } b \textbf{ do } c \textbf{ else skip}, \sigma \rangle \Downarrow \sigma''} \ if_{\textbf{tt}}$$

as required.

ii. Alternatively, the derivation tree with root $\langle \textbf{while } b \textbf{ do } c, \sigma \rangle \Downarrow \sigma''$ could have last rule $while_{\textbf{ff}}$. In that case we have $\sigma = \sigma''$ and the derivation tree is:

$$\dfrac{\begin{array}{c} T3 \\ \vdots \\ \langle b, \sigma \rangle \Downarrow \textbf{ff} \end{array}}{\langle \textbf{while } b \textbf{ do } c, \sigma \rangle \Downarrow \sigma} \ while_{\textbf{ff}}$$

Now we can use $T3$ and rules *skip* and $if_{\textbf{ff}}$ to construct the following derivation tree:

$$\dfrac{\begin{array}{cc} T3 & \dfrac{}{\langle \textbf{skip}, \sigma \rangle \Downarrow \sigma} \ skip \end{array}}{\langle \textbf{if } b \textbf{ then } c; \textbf{while } b \textbf{ do } c \textbf{ else skip}, \sigma \rangle \Downarrow \sigma} \ if_{\textbf{ff}}$$

as required.

part 2: Now assume $\langle \textbf{if } b \textbf{ then } c; \textbf{while } b \textbf{ do } c \textbf{ else skip}, \sigma\rangle \Downarrow \sigma''$ and show $\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \Downarrow \sigma''$.

The last rule in a derivation of $\langle w, \sigma\rangle \Downarrow \sigma''$ must be either $if_{\textbf{tt}}$ or $if_{\textbf{ff}}$.

i. Suppose the last rule is $if_{\textbf{tt}}$. Then the derivation tree must look like:

$$\dfrac{\begin{array}{cc}\begin{array}{c}T0\\ \vdots\\ \langle b,\sigma\rangle \Downarrow \textbf{tt}\end{array} & \dfrac{\begin{array}{cc}\begin{array}{c}T1\\ \vdots\\ \langle c,\sigma\rangle \Downarrow \sigma'\end{array} & \begin{array}{c}T2\\ \vdots\\ \langle \textbf{while } b \textbf{ do } c, \sigma'\rangle \Downarrow \sigma''\end{array}\end{array}}{\langle c; \textbf{while } b \textbf{ do } c, \sigma\rangle \Downarrow \sigma''}\, seq\end{array}}{\langle \textbf{if } b \textbf{ then } c; \textbf{while } b \textbf{ do } c \textbf{ else skip}, \sigma\rangle \Downarrow \sigma''}\, if_{\textbf{tt}}$$

Now using $while_{\textbf{tt}}$, we can construct:

$$\dfrac{\begin{array}{ccc}T0 & T1 & T2\end{array}}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \Downarrow \sigma''}\, while_{\textbf{tt}}$$

as required.

ii. Alternatively, if the last rule is $if_{\textbf{ff}}$, the derivation tree must be:

$$\dfrac{\begin{array}{cc}\begin{array}{c}T3\\ \vdots\\ \langle b,\sigma\rangle \Downarrow \textbf{ff}\end{array} & \dfrac{}{\langle \textbf{skip}, \sigma\rangle \Downarrow \sigma}\, skip\end{array}}{\langle \textbf{if } b \textbf{ then } c; \textbf{while } b \textbf{ do } c \textbf{ else skip}, \sigma\rangle \Downarrow \sigma}\, if_{\textbf{ff}}$$

Now we can use $while_{\textbf{ff}}$ to build:

$$\dfrac{T3}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \Downarrow \sigma}\, while_{\textbf{ff}}$$

as required. ■

Notice how the proof was guided by the structure of the derivation tree. This is a general technique.

2.2 Interlude — proof by induction

No doubt you are familiar with the well-known proof technique of *mathematical induction* where, to show that some property P holds for all natural numbers, it is sufficient to show:

i. (basis case) $P(0)$;

ii. (inductive case) for all n, $P(n) \Rightarrow P(n+1)$

More generally the principle can be stated as

$$(\forall n.(\forall k < n.P(k)) \Rightarrow P(n)) \;\Rightarrow\; \forall n.P(n)$$

Mathematical induction is just a special case of *well-founded*(or Noetherian) induction.

If $\prec$ is a *well-founded* relation on some set S (i.e. there are no infinite descending chains $\ldots \prec a_2 \prec a_1 \prec a_0$) then

$$\forall a \in S.P(a) \quad \text{iff} \quad \forall a \in S.((\forall b \prec a.P(b)) \Rightarrow P(a))$$

Thus the general induction proof technique is the following:

i. Show $P(x)$ for all least elements x (i.e. all elements x for which there is no y such that $y \prec x$);

ii. Show $P(a)$ assuming $P(b)$ for all $b \prec a$.

This general proof technique is useful in operational semantics, two particular instances being *rule induction* and *structural induction*.

Structural Induction

This is induction over some syntactic structure. The ordering $s_1 \prec s_2$ is "s_1 is a (proper) syntactic subphrase of s_2". To prove that some property P holds for all elements of a syntactic domain S:

i. Show P holds for all basis elements of S. (That is, the syntactic phrases with no sub-components.)

ii. Show P holds for all composite elements of S, *assuming* that P holds for all sub-components of the element. This assumption is the *inductive hypothesis*.

For example, to show that $P(a)$ is true of all a : Aexp, by structural induction it is sufficient to show that:

i. $P(n)$ holds for all n : Num. (basis case)

ii. $P(x)$ holds for all x : Ide. (basis case)

iii. for all a_0, a_1 : Aexp, if $P(a_0)$ and $P(a_1)$ are true, then $P(a_0 + a_1)$ is true.

iv. for all a_0, a_1 : Aexp, if $P(a_0)$ and $P(a_1)$ are true, then $P(a_0 - a_1)$ is true.

v. for all a_0, a_1 : Aexp, if $P(a_0)$ and $P(a_1)$ are true, then $P(a_0 * a_1)$ is true.

In (iii)–(v) the inductive hypothesis is the phrase "for all a_0, a_1 : Aexp, if $P(a_0)$ and $P(a_1)$ are true" since a_0 and a_1 are the syntactic subphrases of $a_0 + a_1$, $a_0 - a_1$, $a_0 * a_1$.

Rule Induction

This is essentially induction over the *structure of derivation trees*. The ordering $T_1 \prec T_2$ is "T_1 is a (proper) subtree of T_2". The proof technique, to show $P(T)$ for all derivation trees T is as follows:

i. Show P holds for all *axioms*. (These are the least elements, since they have no subtrees.)

ii. For each transition rule, assume P holds for all the rule's *premises* (this is the *inductive hypothesis*) and hence show that it holds for the rule's *conclusion*.

For example, to show that $P(T)$ is true of every derivation tree T with a conclusion of the form $\langle a, \sigma\rangle \Downarrow n$ (constructed from the transition rules for **Imp**), it is sufficient to show, by rule induction that:

i. P holds of the axioms:

$$\frac{}{\langle n, \sigma\rangle \Downarrow n}\ num$$

$$\frac{}{\langle x, \sigma\rangle \Downarrow \sigma(x)}\ var$$

for all n : Num and x : Ide.

ii. Assuming that P holds of $T1$ and $T2$, P is true of the following derivation tree:

$$\frac{\begin{array}{cc} T1 & T2 \\ \vdots & \vdots \\ \langle a_0, \sigma\rangle \Downarrow n_0 & \langle a_1, \sigma\rangle \Downarrow n_1 \end{array}}{\langle a_0 + a_1, \sigma\rangle \Downarrow n_0 + n_1}\ plus$$

iii. Similarly for derivation trees with conclusions $\langle a_0 - a_1, \sigma\rangle \Downarrow n_0 - n_1$ and $\langle a_0 * a_1, \sigma\rangle \Downarrow n_0 \times n_1$.

2.3 Proofs by induction — examples

As a first simple example of proof by structural induction, we will show that the evaluation of arithmetic expressions is *deterministic* in the sense that it always gives the same answer when evaluated on the same state. We certainly expect this to be the case, so this simple proof is as much to do with validating the semantics as demonstrating a feature of **Imp**.

Proposition 2.3.1
The evaluation of arithmetic expressions in **Imp** is deterministic. That is, for all a : Aexp, σ : State, $m, m' : \mathbb{Z}$,

if $\langle a, \sigma\rangle \Downarrow m$ and $\langle a, \sigma\rangle \Downarrow m'$ then $m = m'$

Proof The proof is by structural induction over a : Aexp. First the basis cases:

- $a \equiv n$
  We need to show that the evaluation of a simple numeral n is deterministic for all states σ.
  There is only one rule:
  $$\frac{}{\langle n, \sigma\rangle \Downarrow n}\ num$$
  so $m = m' = n$.

- $a \equiv x$
  We need to show that the evaluation of a simple variable identifier x is deterministic for all states σ.
  There is only one rule:
  $$\frac{}{\langle x, \sigma\rangle \Downarrow \sigma(x)}\ var$$
  so $m = m' = \sigma(x)$.

Now the inductive cases:

- $a \equiv a_0 + a_1$
  We need to show that *if* the evaluations of both a_0 and a_1 are deterministic, *then* the evaluation of $a_0 + a_1$ is deterministic. Inductive hypotheses: if $\langle a_0, \sigma\rangle \Downarrow m$ and $\langle a_0, \sigma\rangle \Downarrow m'$ then $m = m'$ and if $\langle a_1, \sigma\rangle \Downarrow m$ and $\langle a_1, \sigma\rangle \Downarrow m'$ then $m = m'$ (since a_0 and a_1 are proper subcomponents of a in this case).

  Again there is only one applicable rule (*plus*), so a derivation of $\langle a_0 + a_1, \sigma\rangle \Downarrow m$ must have premises $\langle a_0, \sigma\rangle \Downarrow m_0$ and $\langle a_1, \sigma\rangle \Downarrow m_1$ where $m = m_0 + m_1$.

  Similarly, a derivation of $\langle a_0 + a_1, \sigma\rangle \Downarrow m'$ must have premises $\langle a_0, \sigma\rangle \Downarrow m'_0$ and $\langle a_1, \sigma\rangle \Downarrow m'_1$ where $m' = m'_0 + m'_1$.

  But by the inductive hypothesis, $m_0 = m'_0$ and $m_1 = m'_1$ so we have $m = m_0 + m_1 = m'_0 + m'_1 = m'$ as required.

The other inductive cases, $a \equiv a_0 - a_1$ and $a \equiv a_0 * a_1$ are similar. ■

Exercise 2.3.2
Consider how the above proof fails if we add a non-deterministic operator such as McCarthy's '**amb**' with semantics defined by a pair of rules:

$$\frac{\langle a_0, \sigma\rangle \Downarrow n_0}{\langle a_0\ \mathbf{amb}\ a_1, \sigma\rangle \Downarrow n_0}\ amb_L$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow n_1}{\langle a_0\ \mathbf{amb}\ a_1, \sigma\rangle \Downarrow n_1}\ amb_R$$

Note that all the cases in the proof above remain valid because they are *establishing an implication*. It just so happens that the *premise* may be false in the presence of **amb**.

Proposition 2.3.3
Arithmetic expressions in **Imp** have a kind of *extensionality* property. For all $a_1, a_2 : \text{Aexp}, x : \text{Ide}\ \sigma : \text{State},\ n_1, n_2 : \mathbb{Z}$,

$$\text{if } \langle a_1, \sigma\rangle \Downarrow n_1 \text{ and } \langle a_2, \sigma[x \mapsto n_1]\rangle \Downarrow n_2 \text{ then } \langle a_2[a_1/x], \sigma\rangle \Downarrow n_2.$$[2]

Proof Exercise. Proceed by structural induction over a_2. ■

We also expect the evaluation of boolean expressions to be deterministic:

Proposition 2.3.4
The evaluation of boolean expressions in **Imp** is deterministic. That is, for all $b : \text{Bexp},\ \sigma : \text{State},\ t, t' : \mathbb{B}$,

$$\text{if } \langle b, \sigma\rangle \Downarrow t \text{ and } \langle b, \sigma\rangle \Downarrow t' \text{ then } t = t'$$

Proof Exercise. Again the proof should be by structural induction. Note that for relational expressions, e.g. $a_0 <= a_1$, this proof will rely on proposition 2.3.1. ■

Exercise 2.3.5
State and prove an extensionality result for boolean expressions, similar to that for arithmetic expressions given in proposition 2.3.3.

Finally, we also expect the execution of **Imp** commands to be deterministic:

Proposition 2.3.6
The execution of commands in **Imp** is deterministic. That is, for all $c : \text{Com},\ \sigma, \sigma', \sigma'' : \text{State}$,

$$\text{if } \langle c, \sigma\rangle \Downarrow \sigma' \text{ and } \langle c, \sigma\rangle \Downarrow \sigma'' \text{ then } \sigma' = \sigma''$$

Unfortunately, structural induction is not always a sufficient proof technique. To see why, consider the following attempt to prove proposition 2.3.6 by structural induction:

Non-Proof *One of the cases in the attempted proof by structural induction will be* **while** b **do** c. *By proposition 2.3.4 we know that the evaluation of* b *will be deterministic and by the inductive hypothesis, we can assume that the execution of* c *is deterministic. However, the rule* $while_{\mathbf{tt}}$*:*

$$\frac{\langle b, \sigma\rangle \Downarrow \mathbf{tt} \qquad \langle c, \sigma\rangle \Downarrow \sigma' \qquad \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma'\rangle \Downarrow \sigma''}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \Downarrow \sigma''}\ while_{\mathbf{tt}}$$

[2] The notation $a[a'/x]$ indicates textual substitution of a' for x in a.

has a premise containing the whole command **while** b **do** c*, which we cannot assume is deterministic because it is not a* proper *sub-component of* **while** b **do** c*. Hence the proof cannot proceed.*

On the other hand, that premise is the conclusion of a proper subtree of the derivation, so rule induction may be applicable.

Proof The proof is by *rule induction* over the operational semantics transition rules for commands of **Imp**. That is, we will show that if:

$$\begin{array}{c} T1 \\ \vdots \\ \langle c, \sigma\rangle \Downarrow \sigma' \end{array} \qquad \text{and} \qquad \begin{array}{c} T2 \\ \vdots \\ \langle c, \sigma\rangle \Downarrow \sigma'' \end{array}$$

then $\sigma' = \sigma''$. (In fact, combining propositions 2.3.1, 2.3.4, 2.3.6, it is clear that $T1$ and $T2$ must be identical as well.) The inductive hypotheses will be that corresponding subtrees of $T1$ and $T2$ (with commands as conclusions) satisfy this condition. The cases are generated by each rule of the semantics appearing last in the derivation:

- *asst*:
  Suppose we have:
  $$\dfrac{\begin{array}{c} T1 \\ \vdots \\ \langle a, \sigma\rangle \Downarrow m \end{array}}{\langle x := a, \sigma\rangle \Downarrow \sigma[x \mapsto m]}\ \mathit{asst}$$
  and
  $$\dfrac{\begin{array}{c} T2 \\ \vdots \\ \langle a, \sigma\rangle \Downarrow m' \end{array}}{\langle x := a, \sigma\rangle \Downarrow \sigma[x \mapsto m']}\ \mathit{asst}$$
  Then by proposition 2.3.1 we have that $m = m'$. Hence $\sigma[x \mapsto m] = \sigma[x \mapsto m']$ as required.

- *skip*:
  The only possible derivation is
  $$\dfrac{}{\langle \mathbf{skip}, \sigma\rangle \Downarrow \sigma}\ \mathit{skip}$$
  so $\sigma' = \sigma'' = \sigma$.

- *seq*:
  Suppose we have:
  $$\dfrac{\begin{array}{c} T1 \\ \vdots \\ \langle c_0, \sigma\rangle \Downarrow \sigma_0 \end{array} \qquad \begin{array}{c} T3 \\ \vdots \\ \langle c_1, \sigma_0\rangle \Downarrow \sigma' \end{array}}{\langle c_0; c_1, \sigma\rangle \Downarrow \sigma'}\ \mathit{seq}$$

and

$$\dfrac{\begin{array}{cc} T2 & T4 \\ \vdots & \vdots \\ \langle c_0, \sigma\rangle \Downarrow \sigma_1 & \langle c_1, \sigma_1\rangle \Downarrow \sigma'' \end{array}}{\langle c_0; c_1, \sigma\rangle \Downarrow \sigma''}\ seq$$

Then by the inductive hypothesis for $T1$ and $T2$ we have $\sigma_0 = \sigma_1$. Thus, the inductive hypothesis also applies to $T3$ and $T4$, giving $\sigma' = \sigma''$ as required.

- $if_{\mathbf{tt}}$:
  By proposition 2.3.4, $\langle b, \sigma\rangle \Downarrow \mathbf{tt}$ or $\langle b, \sigma\rangle \Downarrow \mathbf{ff}$ but not both.

  Suppose we have:

$$\dfrac{\begin{array}{cc} T0 & T1 \\ \vdots & \vdots \\ \langle b, \sigma\rangle \Downarrow \mathbf{tt} & \langle c_0, \sigma\rangle \Downarrow \sigma' \end{array}}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma\rangle \Downarrow \sigma'}\ if_{\mathbf{tt}}$$

  and

$$\dfrac{\begin{array}{cc} T0 & T2 \\ \vdots & \vdots \\ \langle b, \sigma\rangle \Downarrow \mathbf{tt} & \langle c_0, \sigma\rangle \Downarrow \sigma'' \end{array}}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma\rangle \Downarrow \sigma''}\ if_{\mathbf{tt}}$$

  Immediately from the inductive hypothesis on $T1$ and $T2$ we have $\sigma' = \sigma''$.

- $if_{\mathbf{ff}}$:
  Similar to the $if_{\mathbf{tt}}$ case.

- $while_{\mathbf{ff}}$:
  By proposition 2.3.4, all b : Bexp deterministically evaluate to $\mathbf{tt}$ or $\mathbf{ff}$. Therefore, for some $T0$:

$$\dfrac{\begin{array}{c} T0 \\ \vdots \\ \langle b, \sigma\rangle \Downarrow \mathbf{ff} \end{array}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \Downarrow \sigma}\ while_{\mathbf{ff}}$$

  is the only possible derivation concluding with $while_{\mathbf{ff}}$, so $\sigma' = \sigma'' = \sigma$.

- $while_{\mathbf{tt}}$:
  Suppose we have:

$$\dfrac{\begin{array}{ccc} T0 & T1 & T3 \\ \vdots & \vdots & \vdots \\ \langle b, \sigma\rangle \Downarrow \mathbf{tt} & \langle c, \sigma\rangle \Downarrow \sigma_0 & \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma_0\rangle \Downarrow \sigma' \end{array}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \Downarrow \sigma'}\ while_{\mathbf{tt}}$$

  and

$$\dfrac{\begin{array}{ccc} T0 & T2 & T4 \\ \vdots & \vdots & \vdots \\ \langle b, \sigma\rangle \Downarrow \mathbf{tt} & \langle c, \sigma\rangle \Downarrow \sigma_1 & \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma_1\rangle \Downarrow \sigma'' \end{array}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \Downarrow \sigma''}\ while_{\mathbf{tt}}$$

As in the $c_0; c_1$ case, the inductive hypothesis on $T1$ and $T2$ give us $\sigma_0 = \sigma_1$. Hence the inductive hypothesis applies to $T3$ and $T4$, thus giving $\sigma' = \sigma''$, as required.

Hence by rule induction, the proposition is proven. ■

Note 2.3.7
Remember that rule induction is over the structure of derivation trees. The cases are generated by the *concluding rule* of the derivation. For the operational semantics of **Imp** the rules are in partial correspondence to the syntactic constructs of the commands of **Imp**. It is important that you remain clear about the difference between the two proof techniques. In structural induction, the inductive hypothesis applies to *proper syntactic subphrases*, while for rule induction the inductive hypothesis applies to the *proper subtrees of the derivation*.

2.4 Operational semantics of blocks and procedures

Most imperative languages have procedure and variable declarations and blocks with some kind of associated *scope*. In this section we will extend **Imp** with these features and discuss their operational semantics.

To this point, all identifiers have been *variable* identifiers. In this section the names of procedures, functions, parameters and so on, will be introduced. Generally these are all identifiers, indistinguishable from variables.

2.4.1 Blocks and variable declarations

Extend **Imp** with a new command representing a block:

$$c \ ::= \ \ldots \ | \ \textbf{begin}\ d; c\ \textbf{end}$$

where d is a metavariable of syntactic set Decl defined as follows:

$$d \ ::= \ \textbf{var}\, x := a \ | \ d_0; d_1$$

The idea is that variables declared in a block are local to that block (i.e. the *scope* of the variables is that block). For example, consider the following program:

```
begin
    var y := 1;
    var x := 1;
    begin
    var x := 2;
        y := x + 1;
        x := y + 2;
```

```
    end;
    x := y + x
end
```

The final value of x is 4 (not 8) since the assignment $x := y + 2$ alters the value of the x declared in the inner block.

To deal with this idea of several different occurences of the same variable existing at the same time. we 'factor' the state into two parts: an *Environment* mapping variable identifiers to their associated *Location*, and a *Store* mapping locations to their contents (i.e. integer values, in the case of **Imp**).

The new semantic domains are specified as follows. Environments are modelled as functions from variable identifiers to the location to which they are currently bound:

$$\rho \ : \ \text{Env} = \text{Ide} \to \text{Loc}$$

We will use the same notation to indicate the update of environments as was introduced for states in definition 2.0.2, i.e. $\rho[x \mapsto l]$.

Stores record the current contents of each location:

$$\sigma \ : \ \text{Store} = \text{Loc} \to [\mathbb{Z} + \{\mathit{unused}\}]$$

The State mappings are now represented as the composition of an environment and a store. We have left the Loc semantic domain unspecified, but it could be represented by $\mathbb{N}$ (reminiscent of addresses) if you wish to be concrete.

We need some way of binding new locations to variables when they are declared and this is where the special value *unused* comes in. To define a function:

$$\mathit{new} \ : \ \text{Store} \to \text{Loc}$$

which returns a new unused location each time it is called, we can proceed by looking at the locations one at a time in order until we find one whose contents is *unused*. The first such unused location is returned. In more detail, assume that the 'first' location is l_0 and that there is a function $\mathit{next} : \text{Loc} \to \text{Loc}$ (for example, $l_0 = 0$ and *next* is the successor function if we choose the concrete representation of Loc as $\mathbb{N}$):

$$\mathit{new}(\sigma) = \mathit{find}(\sigma, l_0)$$

where

$$\mathit{find}(\sigma, l) = \textbf{if } \sigma(l) = \mathit{unused} \textbf{ then } l \textbf{ else } \mathit{find}(\sigma, \mathit{next}(l))$$

Note that locations must be initialised *immediately* they are allocated if the *new* function is to be well-behaved. In fact, this is a rather poor way to deal with storage allocation but at least it is conceptually and notationally simple. More general alternatives are given in Nielson & Nielson, *Semantics with Applications*, and Schmidt, *Denotational Semantics*, among others.

The operational semantics rules for **Imp** must now be changed to take account of both the environment and the store, since the value of a variable is dependent

on the location to which it is bound and the current value in that location. We extend the transition system so that environments appear as 'contexts' to the transitions. For example:

$$\rho \vdash \langle a, \sigma \rangle \Downarrow n$$

indicates that a is being evaluated in store σ, relative to environment ρ. It is important to note that the environment is unaffected by the transitions. Some of the rules for arithmetic expressions are as follows:

$$\frac{}{\rho \vdash \langle n, \sigma \rangle \Downarrow n}\ num$$

$$\frac{}{\rho \vdash \langle x, \sigma \rangle \Downarrow \sigma(\rho(x))}\ var$$

$$\frac{\rho \vdash \langle a_0, \sigma \rangle \Downarrow n_0 \qquad \rho \vdash \langle a_1, \sigma \rangle \Downarrow n_1}{\rho \vdash \langle a_0 + a_1, \sigma \rangle \Downarrow n_0 + n_1}\ plus$$

Now the rules for the declarations themselves. Each declaration will effect both the environment (associating a new location with each variable declared) and the store (initialising that location). Hence the evaluation relations are of the form:

$$\langle d, \rho, \sigma \rangle \Downarrow (\rho', \sigma')$$

The effect of a single declaration **var** $x := a$ is to evaluate a (in the current store and environment), allocate space for the newly declared variable x and modify the store so that location contains the value of a:

$$\frac{\rho \vdash \langle a, \sigma \rangle \Downarrow n}{\langle \mathbf{var}\ x := a, \rho, \sigma \rangle \Downarrow (\rho[x \mapsto l], \sigma[l \mapsto n])}\ vdec$$

where $l = new(\sigma)$.

The effect of a sequence of variable declarations is simply the composition of the effects:

$$\frac{\langle d_0, \rho, \sigma \rangle \Downarrow (\rho', \sigma') \qquad \langle d_1, \rho', \sigma' \rangle \Downarrow (\rho'', \sigma'')}{\langle d_0; d_1, \rho, \sigma \rangle \Downarrow (\rho'', \sigma'')}\ vseq$$

Now for the semantics of blocks. The effect of **begin** $d; c$ **end** is to execute c, but in a modified environment appropriate to the block. The environment existing when the block is entered is extended by the declarations d and the program must revert to the pre-existing environment at block exit. The rule for blocks is as follows:

$$\frac{\langle d, \rho, \sigma \rangle \Downarrow (\rho', \sigma') \qquad \rho' \vdash \langle c, \sigma' \rangle \Downarrow \sigma''}{\rho \vdash \langle \mathbf{begin}\ d; c\ \mathbf{end}, \sigma \rangle \Downarrow \sigma''}\ block$$

Exercise 2.4.1
In the *block* rule above there is no explicit action of resetting the environment at block exit. Explain. In particular, if some variable x is in scope at block entry and a new declaration of x occurs inside the block then the environment is modified to only include the binding for that new x. How is the old x binding retrieved at block exit?

We will leave the definition of the other rules for the operational semantics of **Imp** with blocks and variable declarations as an exercise and move on to further extend the language by the introduction of procedures.

Exercise 2.4.2
Give the full operational semantics definition for **Imp** extended with blocks and variable declarations.

2.4.2 Procedures

This section considers extending the language to include simple procedure declarations and calls. We will call the new language **Proc**. The syntax is the same as **Imp** with the addition of blocks, variable declarations, procedure declarations and procedure calls:

Syntactic Domains

n : Num	numerals
x : Ide	variable identifiers
a : Aexp	arithmetic expressions
b : Bexp	boolean expressions
c : Com	commands (i.e. statements)
d : Decl	variable declarations
p : Pdecl	procedure declarations

Syntax

$$
\begin{aligned}
a &::= n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \\
b &::= \textbf{true} \mid \textbf{false} \mid a_1 == a_2 \mid a_1 <= a_2 \mid \textbf{not}\ b \mid b_1\ \textbf{and}\ b_2 \mid b_1\ \textbf{or}\ b_2 \\
c &::= x := a \mid \textbf{skip} \mid c_1; c_2 \mid \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \mid \textbf{while}\ b\ \textbf{do}\ c \mid \\
&\quad\ \textbf{begin}\ d; p; c\ \textbf{end} \mid \textbf{call}\ x \\
d &::= \textbf{var}\ x := a \mid d_0; d_1 \\
p &::= \textbf{proc}\ x\ \textbf{is}\ c \mid p_0; p_1
\end{aligned}
$$

Before we can give the semantics we must determine whether **Proc** will have *static* or *dynamic* scope rules. Essentially, static scope implies the block within which entities are *declared* while dynamic scope refers to the block within which the entities are *used*. Consider the following program:

```
begin
    var x := 0;
    proc q is x:= x * 2;
    proc r is call q;
    begin
    var x := 5;
```

```
        proc q is x:= x + 1;
        call r;
        y := x;
    end;
end
```

If the scope of variables and procedures is *dynamic* then the final value of y will be 6 because the call of r will call the *local* q which will effect the *local* x.

If the scope of procedures and variables is *static* then the final value of y will be 5 because the call of r will call the *global* q which will alter the *global* x (since that is the one existing when q is declared).

Static scope is familiar from common imperative languages and is the approach we will choose for **Proc**.

As with variable declarations, we need some notion of an environment which records the information we need about the procedures as they are declared, so it is available when the procedure is called. The main piece of information we need to bind to procedure names is the command which is its body. Further, since we are describing static scope, it is also necessary to keep some information about where it was declared. (If we chose to describe dynamic scope, this would not be necessary.) Thus we need to keep both the procedure body and the environment that exists at its point of declaration. This combination is called a *closure*:

$$\text{Clo} = \text{Com} \times \text{Env}$$

Now with procedure *and* variable declarations, the semantic domain of environments will have two different kinds of information bound to identifiers, so the denotable values are either locations or closures. We will deal with this as a *discriminated union* of Loc and Clo, written $\text{Loc} + \text{Clo}$. The basic idea of a discriminated union $A + B$ of two sets A and B is that we 'tag' the elements of A and B differently and then take the union of the tagged sets. Then it is possible to tell whether an element of $A + B$ is from A or B simply by inspecting its tag. For example, we could tag every element a of A with a 0 by constructing a pair $(0, a)$ and similarly tag elements of B with a 1. In other words, $A + B = (\{0\} \times A) \cup (\{1\} \times B)$. We can tell whether (x, y) is from A or B by asking whether $x = 0$ or 1.

The semantic domain of environments is thus as follows:

$$\rho \,:\, \text{Env} = \text{Ide} \to (\text{Loc} + \text{Clo})$$

The environment inside a closure will contain all the information about the variables and procedures that are in scope at the point of declaration of the procedure.

The operational semantics rules for procedure declarations is as follows:

$$\frac{}{\langle \textbf{proc}\ x\ \textbf{is}\ c, \rho\rangle \Downarrow \rho[x \mapsto (c, \rho)]}\ pdec$$

$$\frac{\langle p_0, \rho\rangle \Downarrow \rho' \qquad \langle p_1, \rho'\rangle \Downarrow \rho''}{\langle p_0; p_1, \rho\rangle \Downarrow \rho''}\ pseq$$

The rule for blocks becomes:

$$\frac{\langle d, \rho, \sigma\rangle \Downarrow (\rho', \sigma') \qquad \langle p, \rho'\rangle \Downarrow \rho'' \qquad \rho'' \vdash \langle c, \sigma'\rangle \Downarrow \sigma''}{\rho \vdash \langle \textbf{begin}\ d; p; c\ \textbf{end}, \sigma\rangle \Downarrow \sigma''}\ block$$

It is important to note from these rules that the environment in the closure for each procedure only contains bindings for the procedures that have been declared lexically above it. Similarly, the initialising expression in a variable declaration (§2.4.1) can only use variables occurring in an outer scope or lexically above it in that block. This is familiar from many common imperative languages.

The collection of rules for the operational semantics of **Proc** is as follows. In many cases they are simply derived from the rules for **Imp** by the addition of the environment as a context.

$$\frac{}{\rho \vdash \langle n, \sigma\rangle \Downarrow n}\ num$$

$$\frac{}{\rho \vdash \langle x, \sigma\rangle \Downarrow \sigma \circ \rho(x)}\ var$$

$$\frac{\rho \vdash \langle a_0, \sigma\rangle \Downarrow n_0 \qquad \rho \vdash \langle a_1, \sigma\rangle \Downarrow n_1}{\rho \vdash \langle a_0 + a_1, \sigma\rangle \Downarrow n_0 + n_1}\ plus$$

$$\frac{\rho \vdash \langle a_0, \sigma\rangle \Downarrow n_0 \qquad \rho \vdash \langle a_1, \sigma\rangle \Downarrow n_1}{\rho \vdash \langle a_0 - a_1, \sigma\rangle \Downarrow n_0 - n_1}\ minus$$

$$\frac{\rho \vdash \langle a_0, \sigma\rangle \Downarrow n_0 \qquad \rho \vdash \langle a_1, \sigma\rangle \Downarrow n_1}{\rho \vdash \langle a_0 * a_1, \sigma\rangle \Downarrow n_0 \times n_1}\ times$$

$$\frac{}{\rho \vdash \langle \textbf{true}, \sigma\rangle \Downarrow \textbf{tt}}\ true$$

$$\frac{}{\rho \vdash \langle \textbf{false}, \sigma\rangle \Downarrow \textbf{ff}}\ false$$

$$\frac{\rho \vdash \langle a_0, \sigma\rangle \Downarrow n_0 \qquad \rho \vdash \langle a_1, \sigma\rangle \Downarrow n_1}{\rho \vdash \langle a_0 == a_1, \sigma\rangle \Downarrow (n_0 = n_1)}\ equal$$

$$\frac{\rho \vdash \langle a_0, \sigma\rangle \Downarrow n_0 \qquad \rho \vdash \langle a_1, \sigma\rangle \Downarrow n_1}{\rho \vdash \langle a_0 <= a_1, \sigma\rangle \Downarrow (n_0 \leq n_1)}\ leq$$

$$\frac{\rho \vdash \langle b, \sigma\rangle \Downarrow t}{\rho \vdash \langle \textbf{not}\ b, \sigma\rangle \Downarrow \neg t}\ not$$

$$\frac{\rho \vdash \langle b_0, \sigma\rangle \Downarrow t_0 \qquad \rho \vdash \langle b_1, \sigma\rangle \Downarrow t_1}{\rho \vdash \langle b_0\ \textbf{and}\ b_1, \sigma\rangle \Downarrow t_0 \wedge t_1}\ and$$

$$\frac{\rho \vdash \langle b_0, \sigma\rangle \Downarrow t_0 \qquad \rho \vdash \langle b_1, \sigma\rangle \Downarrow t_1}{\rho \vdash \langle b_0\ \textbf{or}\ b_1, \sigma\rangle \Downarrow t_0 \vee t_1}\ or$$

$$\frac{\rho \vdash \langle a, \sigma\rangle \Downarrow n \qquad l = new(\sigma)}{\langle \mathbf{var}\ x := a, \rho, \sigma\rangle \Downarrow (\rho[x \mapsto l], \sigma[l \mapsto n])}\ vdec$$

$$\frac{\langle d_0, \rho, \sigma\rangle \Downarrow (\rho', \sigma') \qquad \langle d_1, \rho', \sigma'\rangle \Downarrow (\rho'', \sigma'')}{\langle d_0; d_1, \rho, \sigma\rangle \Downarrow (\rho'', \sigma'')}\ vseq$$

$$\frac{}{\langle \mathbf{proc}\ x\ \mathbf{is}\ c, \rho\rangle \Downarrow \rho[x \mapsto (c, \rho)]}\ pdec$$

$$\frac{\langle p_0, \rho\rangle \Downarrow \rho' \qquad \langle p_1, \rho'\rangle \Downarrow \rho''}{\langle p_0; p_1, \rho\rangle \Downarrow \rho''}\ pseq$$

$$\frac{\rho \vdash \langle a, \sigma\rangle \Downarrow n}{\rho \vdash \langle x := a, \sigma\rangle \Downarrow \sigma[\rho(x) \mapsto n]}\ asst$$

$$\frac{}{\rho \vdash \langle \mathbf{skip}, \sigma\rangle \Downarrow \sigma}\ skip$$

$$\frac{\rho \vdash \langle c_0, \sigma\rangle \Downarrow \sigma' \qquad \rho \vdash \langle c_1, \sigma'\rangle \Downarrow \sigma''}{\rho \vdash \langle c_0; c_1, \sigma\rangle \Downarrow \sigma''}\ seq$$

$$\frac{\rho \vdash \langle b, \sigma\rangle \Downarrow \mathbf{tt} \qquad \rho \vdash \langle c_0, \sigma\rangle \Downarrow \sigma'}{\rho \vdash \langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma\rangle \Downarrow \sigma'}\ if_{\mathbf{tt}}$$

$$\frac{\rho \vdash \langle b, \sigma\rangle \Downarrow \mathbf{ff} \qquad \rho \vdash \langle c_1, \sigma\rangle \Downarrow \sigma'}{\rho \vdash \langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma\rangle \Downarrow \sigma'}\ if_{\mathbf{ff}}$$

$$\frac{\rho \vdash \langle b, \sigma\rangle \Downarrow \mathbf{tt} \quad \rho \vdash \langle c, \sigma\rangle \Downarrow \sigma' \quad \rho \vdash \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma'\rangle \Downarrow \sigma''}{\rho \vdash \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \Downarrow \sigma''}\ while_{\mathbf{tt}}$$

$$\frac{\rho \vdash \langle b, \sigma\rangle \Downarrow \mathbf{ff}}{\rho \vdash \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \Downarrow \sigma}\ while_{\mathbf{ff}}$$

$$\frac{\langle d, \rho, \sigma\rangle \Downarrow (\rho', \sigma') \qquad \langle p, \rho'\rangle \Downarrow \rho'' \qquad \rho'' \vdash \langle c, \sigma'\rangle \Downarrow \sigma''}{\rho \vdash \langle \mathbf{begin}\ d; p; c\ \mathbf{end}, \sigma\rangle \Downarrow \sigma''}\ block$$

Finally we come to the rule for procedure calls. Informally, the execution of the command **call** x is begun by looking up x in the environment ρ. If it is not bound to a closure then the program is in error. Remember that it is easy to test whether $\rho(x)$ is a location or a closure by inspecting the tags associated with the discriminated union Loc + Clo. The closure bound to x will be a command c (the body of the procedure) and an environment ρ' (from the point of declaration of the procedure). The effect of the call is to execute c on the current store, but in the context of ρ'. Hence the following rule:

$$\frac{\rho' \vdash \langle c, \sigma\rangle \Downarrow \sigma'}{\rho \vdash \langle \mathbf{call}\ x, \sigma\rangle \Downarrow \sigma'}\ call$$

where $\rho(x)$ is a closure and $\rho(x) = (c, \rho')$.

Careful inspection of this rule reveals that it does not permit recursive procedure calls since ρ' does not contain the binding of x to its closure. (See rules *pdec*

and *pseq.*) One solution is to add that binding to ρ' when x is called, as in the following rule:

$$\frac{\rho'[x \mapsto \rho(x)] \ \vdash \ \langle c, \sigma \rangle \Downarrow \sigma'}{\rho \ \vdash \ \langle \mathbf{call}\, x, \sigma \rangle \Downarrow \sigma'} \ \mathit{call}$$

where $\rho(x)$ is a closure and $\rho(x) = (c, \rho')$.

Now the body c is executed in an environment which does contain the binding of x to its closure, so if c contains a recursive call of x, then it will be found in the environment.

Exercise 2.4.3
Trace the behaviour of the program used to demonstrate the difference between static and dynamic binding at the beginning of the section. That is, build its derivation tree, paying particular attention to the construction of environments.

Exercise 2.4.4
Show that the semantics of **Proc** (with blocks, declarations and procedure calls removed) are equivalent to the original semantics of **Imp**. Of course, you should begin by stating the equivalence relation precisely.

Chapter 3

Denotational Semantics

Denotational semantics gives the essence of the meaning of a program: its *effect*. In general terms this is an association (i.e. a function) between input and output or, more generally, between initial and final states.

The denotational semantics of a programming language is defined by giving *semantic functions* mapping each syntactic domain into a corresponding semantic domain. The semantic domains are mathematical structures of some interest in their own right.

A very important requirement on a denotational definition is that it is *strictly compositional*. In other words, the meaning of a syntactic construct is defined only in terms of the meanings of the proper sub-components of that construct. Thus the semantic functions are homomorphisms in a certain sense. For example, the meaning of $c_0; c_1$ is defined in terms of the meanings of c_0 and c_1; the meaning of **while** b **do** c is defined in terms of the meanings of b and c.

If you look carefully at §2 you will notice that natural semantics is not compositional in the same sense. For example, in the $\mathit{while}_{\mathbf{tt}}$ rule, the whole syntactic phrase appears in one of the premises. In contrast, the denotational semantic clause for **while** b **do** c must only be in terms of b and c. Similarly, consider the natural semantics rule for procedure calls:

$$\frac{\rho'[x \mapsto \rho(x)] \ \vdash \ \langle c, \sigma\rangle \Downarrow \sigma'}{\rho \ \vdash \ \langle \mathbf{call}\, x, \sigma\rangle \Downarrow \sigma'}\ \mathit{call}$$

where $\rho(x)$ is a closure and $\rho(x) = (c, \rho')$.

A command c appears in the premise yet it does not appear in the conclusion. Again this is not compositional.

3.1 Example — Binary numerals

The following very simple example will illustrate the basic ideas and introduce the notation of denotational semantics.

Example 3.1.1
We can define the meaning of *binary numerals* by their corresponding integer values. First, the syntax of binary numerals:

$$b : \text{Binary}$$

$$b ::= 0 \mid 1 \mid b0 \mid b1$$

The denotational semantics is given by defining a semantic function from the syntactic domain Binary to the semantic domain of natural numbers:

$$\mathcal{B} : \text{Binary} \to \mathbb{N}$$

Now $\mathcal{B}$ is defined case-by-case on the structure of the syntactic domain Binary:

$$\begin{aligned}
\mathcal{B}[\![0]\!] &= 0 \\
\mathcal{B}[\![1]\!] &= 1 \\
\mathcal{B}[\![b0]\!] &= 2 \times \mathcal{B}[\![b]\!] \\
\mathcal{B}[\![b1]\!] &= 2 \times \mathcal{B}[\![b]\!] + 1
\end{aligned}$$

A point about notation: The special 'syntactic' brackets $[\![\ ]\!]$ always surround syntactic phrases, so that they stand out in the clauses. (This makes it rather easy to check for compositionality.) In this case, the same symbol has been used for the syntactic entities '0' and '1' and the semantic entities 0 and 1. The brackets $[\![\ ]\!]$ resolve any ambiguity.

The clauses are rather easy to interpret. Consider the following example, 'evaluating' the binary numeral 1011:

$$\begin{aligned}
\mathcal{B}[\![1011]\!] &= 2 \times \mathcal{B}[\![101]\!] + 1 \\
&= 2 \times (2 \times \mathcal{B}[\![10]\!] + 1) + 1 \\
&= 2 \times (2 \times (2 \times \mathcal{B}[\![1]\!]) + 1) + 1 \\
&= 2 \times (2 \times (2 \times 1) + 1) + 1 \\
&= 11
\end{aligned}$$

Exercise 3.1.2
Extend the example of binary numerals to *signed* binary numerals.

Finally, we introduce some notation that will appear from time to time in this section.

Notation 3.1.3 (Lambda Notation)
At times, we will want to refer to a function *anonymously* (i.e. without giving it a name). Usually we would be forced to write something like "f where $f(x) = x^3 + x - 1$" thus giving the function a name. Alternatively, we could specify the arguments (*bound variables*) with the defining expression by using λ-*notation*. For example, the function f above could be written anonymously as $\lambda x.x^3 + x - 1$. We need to be a little careful about parentheses to disambiguate

applications of anonymous functions, for example, write $(\lambda x.x^3 + x - 1)7$ to represent $f(7)$.

On an historical note, this notation derives from Alonzo Church's *lambda calculus* which (among other things) can serve as a model of computation of power equivalent to Turing machines and the like.

3.2 Denotational semantics of Imp

The denotational semantics of **Imp** will be given, following the same semantic ideas as the natural semantics in the previous section. (See Winskell, Ch5, for more on this correspondence.)

3.2.1 Arithmetic expressions

The basic idea is that the meaning of an arithmetic expression is a function from states to integers, since the value of any variables appearing in an expression is determined by the state.

As in the binary numerals example, we give the semantic functions script letter names:

$$\mathcal{A} : \text{Aexp} \to \text{State} \to \mathbb{Z}$$

We assume that $\to$ and $\to$ are right associative, so Aexp $\to$ State $\to \mathbb{Z}$ is to be read as Aexp $\to$ (State $\to \mathbb{Z}$). Correspondingly, function application is assumed to be left associative, so $\mathcal{A}[\![a]\!]\sigma$ means $(\mathcal{A}[\![a]\!])\sigma$. This bracketing may be different to your usual practice but is unambiguous and clean and sits well with lambda notation.

The semantic domain of states is the same as for the natural semantics of **Imp**:

$$\sigma : \text{State} = \text{Ide} \to \mathbb{Z}$$

and the notation for altering bindings is also the same.

We assume the existence of a semantic function $\mathcal{N} : \text{Num} \to \mathbb{Z}$. This could be trivially defined if necessary, following the binary numerals example.

The semantic functions ($\mathcal{A}$ in this case), are each defined by a set of clauses, one for each syntactic construct:

$$\mathcal{A}[\![n]\!]\sigma = \mathcal{N}[\![n]\!]$$

The value of a numeral n is determined irrespective of the state σ. An alternative way to write this clause would be using lambda notation, more clearly reflecting that the meaning of an arithmetic expression is a function from states to $\mathbb{Z}$:

$$\mathcal{A}[\![n]\!] = \lambda\sigma.\mathcal{N}[\![n]\!]$$

We will continue to write clauses without lambda notation but this equivalence should be borne in mind.

$$\mathcal{A}[\![x]\!]\sigma = \sigma[\![x]\!]$$

The value of a variable is given by its value in the state.

$$\mathcal{A}[\![a_0 + a_1]\!]\sigma = \mathcal{A}[\![a_0]\!]\sigma + \mathcal{A}[\![a_1]\!]\sigma$$

This clause (as are the other two below) is obvious, but note the compositionality.

$$\mathcal{A}[\![a_0 - a_1]\!]\sigma = \mathcal{A}[\![a_0]\!]\sigma - \mathcal{A}[\![a_1]\!]\sigma$$
$$\mathcal{A}[\![a_0 * a_1]\!]\sigma = \mathcal{A}[\![a_0]\!]\sigma \times \mathcal{A}[\![a_1]\!]\sigma$$

3.2.2 Boolean expressions

The meaning of a boolean expression is a function from states to truth values so the corresponding semantic function is the following:

$$\mathcal{B} : \text{Bexp} \to \text{State} \to \mathbb{B}$$

The defining clauses for boolean expressions are straightforward:

$$\mathcal{B}[\![\textbf{true}]\!]\sigma = \textbf{tt}$$
$$\mathcal{B}[\![\textbf{false}]\!]\sigma = \textbf{ff}$$
$$\mathcal{B}[\![a_0 == a_1]\!]\sigma = (\mathcal{A}[\![a_0]\!]\sigma = \mathcal{A}[\![a_1]\!]\sigma)$$
$$\mathcal{B}[\![a_0 <= a_1]\!]\sigma = \mathcal{A}[\![a_0]\!]\sigma \leq \mathcal{A}[\![a_1]\!]\sigma$$
$$\mathcal{B}[\![\textbf{not } b]\!]\sigma = \neg\mathcal{B}[\![b]\!]\sigma$$
$$\mathcal{B}[\![b_0 \textbf{ or } b_1]\!]\sigma = \mathcal{B}[\![b_0]\!]\sigma \vee \mathcal{B}[\![b_1]\!]\sigma$$
$$\mathcal{B}[\![b_0 \textbf{ and } b_1]\!]\sigma = \mathcal{B}[\![b_0]\!]\sigma \wedge \mathcal{B}[\![b_1]\!]\sigma$$

3.2.3 Commands

Finally, the meaning of commands is a partial function from states to states:

$$\mathcal{C} : \text{Com} \to \text{State} \to \text{State}$$

Of course, the simplicity of the semantic domains and functions results from the simplicity of the language **Imp**. Of course, "real-world" languages are much more complex.

Modelling the meaning of commands as *partial* functions on States reflects the fact that commands may not terminate, in which case a result state is not

reached. As in another part of the course[1] we handle partiality reflecting non-termination by *lifting*, so the semantic domain of States has a least value $\perp$.

But first, assignment is obvious:

$$\mathcal{C}[\![x := a]\!]\sigma = \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]$$

which says that the effect of an assignment command is to evaluate the expression and bind its value to the variable, thus modifying the state.

The **skip** command has no effect on the state:

$$\mathcal{C}[\![\mathbf{skip}]\!]\sigma = \sigma$$

or alternatively:

$$\mathcal{C}[\![\mathbf{skip}]\!] = \mathbf{I}$$

Where $\mathbf{I}$ is the identity function (on State).

The semantics of a sequence of commands is given by the composition of their semantics (which are functions: State $\rightarrow$ State):

$$\mathcal{C}[\![c_0; c_1]\!]\sigma = \mathcal{C}[\![c_1]\!](\mathcal{C}[\![c_0]\!]\sigma)$$

or alternatively:

$$\mathcal{C}[\![c_0; c_1]\!] = \mathcal{C}[\![c_1]\!] \circ \mathcal{C}[\![c_0]\!]$$

For conditional commands we introduce a semantic operator which will be formally defined later in definition 3.2.3. It is written 'mixfix' as $(\_ \rightarrow \_ ; \_)$ and has (polymorphic) type $(\mathbb{B} \times D \times D) \rightarrow D$ for any domain D. As you might expect, $t \rightarrow d_0 ; d_1$ equals d_0 if $t = \mathbf{tt}$ and d_1 if $t = \mathbf{ff}$. The semantic clause for conditionals is thus:

$$\mathcal{C}[\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!]\sigma = \mathcal{B}[\![b]\!]\sigma \rightarrow \mathcal{C}[\![c_0]\!]\sigma ; \mathcal{C}[\![c_1]\!]\sigma$$

As we would wish, if b is true c_0 is executed and if b is false then c_1 is executed.

Non-termination

Before going on to while loops, it is useful to consider how the semantics might deal with *non-termination* of commands. First, notice that the semantic domain for the meanings of commands is a partial function from states to states, reflecting the possibility that a non-terminating command will not produce a result state.

The approach usually taken has already been discussed in another section of the course dealing with non-termination: add a special element $\perp$ which is less defined than or equal to all other values in the semantic domain. When we come to consider repetitive structures in **Imp**, it will turn out that the development

[1] *Fixpoint Theory of Recursive Function Definitions*

in terms of complete partial orders and monotonic functions naturally carries over.

Let's concentrate on the case where c_0 does not terminate in the sequence $c_0; c_1$. In the natural semantics, this is reflected by $\langle c_0, \sigma\rangle$ being a stuck configuration. Now since $\langle c_0, \sigma\rangle \Downarrow \sigma'$ is a premise of the *seq* rule, there can be no derivation of $\langle c_0; c_1, \sigma\rangle \Downarrow \sigma''$ so $\langle c_0; c_1, \sigma\rangle$ is also a stuck configuration. What we want in the denotational semantics is some similar sense of being well-behaved with respect to non-termination. For example, if $\mathcal{C}[\![c_0]\!]\sigma$ is undefined, then clearly we would wish $\mathcal{C}[\![c_1]\!](\mathcal{C}[\![c_0]\!]\sigma)$ to be undefined too.

To make this more concrete, suppose we extend **Imp** with a command, **reset**, whoes effect is to set the value of all variables in the current state to 0. The following (slightly informal) clause describes such an action:

$$\mathcal{C}[\![\mathbf{reset}]\!]\sigma = \sigma_0$$

Now, if a non-terminating command c is followed by a **reset** what do we expect? Using the clauses we can calculate:

$$\begin{aligned}\mathcal{C}[\![c; \mathbf{reset}]\!]\sigma &= \mathcal{C}[\![\mathbf{reset}]\!](\mathcal{C}[\![c]\!]\sigma)\\ &= \mathcal{C}[\![\mathbf{reset}]\!]\bot\end{aligned}$$

Should this evaluate to σ_0 or $\bot$? Computationally, it is impossible for **reset** to detect and "recover" from all non-terminating commands, so the intuitive choice (the *only* choice) is $\bot$.

Based on these observations, we could require all semantic functions to be *strict*, which means that if its argument is undefined then so is its result.

Definition 3.2.1
A function f is *strict* iff $f\bot = \bot$.

Exercise 3.2.2
Is the function f defined as $f(x) = 3$ strict? A core feature of Haskell is that it is non-strict, but can you write a function in C (or some other such language) for f that is *not* strict?

Strictness seems to satisfy our requirement on sequential composition since if $\mathcal{C}[\![c_0]\!]\sigma = \bot$ then $\mathcal{C}[\![c_1]\!](\mathcal{C}[\![c_0]\!]\sigma) = \mathcal{C}[\![c_1]\!]\bot = \bot$ if $\mathcal{C}[\![c_1]\!]$ is strict. However, it turns out to be too strong a restriction in general. Consider the conditional command. If we made $(\_ \to \_\,;\,\_)$ strict in all its arguments then we would have, for example, $\mathbf{tt} \to \sigma\,;\,\bot = \bot$. In terms of the programming language, this would mean (operationally) that **if** b **then** c_0 **else** c_1 would not terminate if either c_0 or c_1 did not terminate, irrespective of the value of b. This is certainly not the usual behaviour of conditional commands.

Definition 3.2.3
The semantic conditional operator is defined as follows:

$$t \to d_0\,;\,d_1 = \begin{cases} d_0 & \text{if } t = \mathbf{tt}\\ d_1 & \text{if } t = \mathbf{ff}\\ \bot & \text{if } t = \bot\end{cases}$$

Therefore this function is strict on its first argument but not on its second or third arguments. Actually, this definition is more general than necessary for **Imp** since $\mathcal{B}[\![b]\!]$ is always total. (That is, the case where $t = \bot$ is redundant for **Imp**.)

Of course, based on our experience in a previous section of the course, what we require for the functions in a denotational semantics definition to be well-behaved is that they are *monotonic*.

Repetitive commands

We now move on to defining the denotational semantics of while loops. From the natural semantics, and the result (Proposition 2.1.10) that **while** b **do** c is semantically equivalent to **if** b **then** c; **while** b **do** c **else skip**, the following clause looks reasonable:

$$\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma = \mathcal{B}[\![b]\!]\sigma \ \rightarrow \ \mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!](\mathcal{C}[\![c]\!]\sigma) \ ; \ \sigma \qquad (3.1)$$

This clause may seem to capture our intentions, but it is not acceptable because it is *not compositional*.

Why is compositionality so important? First, recall that arbitrary recursive equations may be satisfied by none, one or many functions. For example, consider the command **while true do skip**. We intuitively expect its semantics to be the constant $\bot$ function (i.e. undefined for any state) to reflect the fact that it is a non-terminating loop. Substituting into clause 3.1 above we get:

$$\begin{aligned}\mathcal{C}[\![\textbf{while true do skip}]\!]\sigma = \\ \mathcal{B}[\![\textbf{true}]\!]\sigma \ \rightarrow \ \mathcal{C}[\![\textbf{while true do skip}]\!](\mathcal{C}[\![\textbf{skip}]\!]\sigma) \ ; \ \sigma\end{aligned}$$

According to the definition of $(\_ \ \rightarrow \ \_ \ ; \ \_)$ and the semantic clauses for **skip** and **true**, this reduces to:

$$\mathcal{C}[\![\textbf{while true do skip}]\!]\sigma = \mathcal{C}[\![\textbf{while true do skip}]\!]\sigma$$

which is satisfied by *any* function substituted for $\mathcal{C}[\![\textbf{while true do skip}]\!]$. This is obviously quite unsatisfactory.

The solution is to bring the least fixpoint function **fix** into play. Abstracting the recursive component from the right hand side of equation 3.1 gives:

$$\lambda\phi.\lambda\sigma.\mathcal{B}[\![b]\!]\sigma \ \rightarrow \ \phi(\mathcal{C}[\![c]\!]\sigma) \ ; \ \sigma$$

The semantics of while commands is its least fixpoint:

$$\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] = \textbf{fix}(\lambda\phi.\lambda\sigma.\mathcal{B}[\![b]\!]\sigma \ \rightarrow \ \phi(\mathcal{C}[\![c]\!]\sigma) \ ; \ \sigma)$$

Let's see how this copes with the problem of non-terminating loops that first motivated our diversion into fixpoint theory. We want to calculate:

$$\mathcal{C}[\![\textbf{while true do skip}]\!]$$

to see if it matches our intuitions by being the totally undefined function. By the clause above we have:

$$\begin{aligned}\mathcal{C}[\![\textbf{while true do skip}]\!] &= \textbf{fix}(\lambda\phi.\lambda\sigma.\mathcal{B}[\![\textbf{true}]\!]\sigma \ \rightarrow \ \phi(\mathcal{C}[\![\textbf{skip}]\!]\sigma) \ ; \ \sigma) \\ &= \textbf{fix}(\lambda\phi.\lambda\sigma.\textbf{tt} \ \rightarrow \ \phi\sigma \ ; \ \sigma) \\ &= \textbf{fix}(\lambda\phi.\lambda\sigma.\phi\sigma)\end{aligned}$$

For notational brevity, let $F = \lambda\phi.\lambda\sigma.\phi\sigma$. Then $\textbf{fix}\, F = \bigsqcup\{F^k(\bot)|k \geq 0\}$. The sequence of finite approximations is as follows:

$$\begin{aligned}F^0(\bot) &= \lambda\sigma.\bot \\ F^1(\bot) &= (\lambda\phi.\lambda\sigma.\phi\sigma)(\lambda\sigma.\bot) = \lambda\sigma.\bot\end{aligned}$$

The sequence has already converged, so we have shown that:

$$\mathcal{C}[\![\textbf{while true do skip}]\!] = \lambda\sigma.\bot$$

as we would expect.

Exercise 3.2.4 (Worked)
what is the type of ϕ in clauses such as (3.1) in the denotational semantics definition of **Imp**?

First, recall that the (polymorphic) type of **fix** is

$$\textbf{fix} : (a \rightarrow a) \rightarrow a \tag{3.2}$$

It's easy to deduce from the type of $\mathcal{C}$

$$\mathcal{C} : \text{Com} \rightarrow \text{State} \rightarrow \text{State}$$

that the type of the left hand side of (3.1) must be

$$\text{State} \rightarrow \text{State}$$

and of course that must also be the type of the right hand side of (3.1). Hence the result type a for this instance of (3.2) must be State $\rightarrow$ State so the type instance for **fix** in (3.1) is

$$\textbf{fix} \ : \ ((\text{State} \rightarrow \text{State}) \rightarrow (\text{State} \rightarrow \text{State})) \rightarrow (\text{State} \rightarrow \text{State})$$

and therefore the type of the function being provided to **fix** must have type

$$(\lambda\phi.\lambda\sigma.\mathcal{B}[\![b]\!]\sigma \ \rightarrow \ \phi(\mathcal{C}[\![c]\!]\sigma) \ ; \ \sigma) \ : \ ((\text{State} \rightarrow \text{State}) \rightarrow (\text{State} \rightarrow \text{State}))$$

In that case, the bound variable ϕ must have type

$$\phi \ : \ \text{State} \rightarrow \text{State} \tag{3.3}$$

and the type of the body must be

$$(\lambda\sigma.\mathcal{B}[\![b]\!]\sigma \ \rightarrow \ \phi(\mathcal{C}[\![c]\!]\sigma) \ ; \ \sigma) \ : \ \text{State} \rightarrow \text{State}$$

which we can observe to be the case from the type of the bound variable

$$\sigma : \text{State}$$

and the type of the conditional expression

$$(\mathcal{B}[\![b]\!]\sigma \ \rightarrow \ \phi(\mathcal{C}[\![c]\!]\sigma) \ ; \ \sigma) \ : \ \text{State} \tag{3.4}$$

Notice that since $\mathcal{C}[\![c]\!]\sigma : \text{State}$, observation (3.4) relies on $\phi : \text{State} \rightarrow \text{State}$ in the subterm $\phi(\mathcal{C}[\![c]\!]\sigma)$, which is consistent with our earlier deduction (3.3). ■

The next example uses the following result which is obvious by the fact that $\mathbf{fix}\, F$ is a fixpoint of F:

Proposition 3.2.5
$\mathbf{fix}\, F = F(\mathbf{fix}\, F)$

Example 3.2.6
Show that if the command **while** $x <= 2$ **do** $x := x + 1$ is run on a state σ_0 with all variables initially zero, the resultant state has x bound to 3.

Substituting into the clause for while loops, we get:

$$\mathcal{C}[\![\textbf{while } x <= 2 \textbf{ do } x := x + 1]\!]\sigma_0 = \\ \mathbf{fix}(\lambda\phi.\lambda\sigma.\mathcal{B}[\![x <= 2]\!]\sigma \ \rightarrow \ \phi(\mathcal{C}[\![x := x + 1]\!]\sigma) \ ; \ \sigma)\sigma_0$$

Writing F for $\lambda\phi.\lambda\sigma.\mathcal{B}[\![x <= 2]\!]\sigma \ \rightarrow \ \phi(\mathcal{C}[\![x := x + 1]\!]\sigma) \ ; \ \sigma$, we can use proposition 3.2.5 to calculate as follows:

$$\begin{aligned}
&(\mathbf{fix}\, F)\sigma_0 \\
&\quad = F(\mathbf{fix}\, F)\sigma_0 \\
&\quad = (\lambda\sigma.\mathcal{B}[\![x <= 2]\!]\sigma \ \rightarrow \ \mathbf{fix}\, F(\mathcal{C}[\![x := x + 1]\!]\sigma) \ ; \ \sigma)\sigma_0 \\
&\quad = \mathcal{B}[\![x <= 2]\!]\sigma_0 \ \rightarrow \ \mathbf{fix}\, F(\mathcal{C}[\![x := x + 1]\!]\sigma_0) \ ; \ \sigma_0 \\
&\quad = \mathbf{fix}\, F(\sigma_0[x \mapsto 1]) \\
&\quad = (\lambda\sigma.\mathcal{B}[\![x <= 2]\!]\sigma \ \rightarrow \ \mathbf{fix}\, F(\mathcal{C}[\![x := x + 1]\!]\sigma) \ ; \ \sigma)(\sigma_0[x \mapsto 1]) \\
&\quad = \mathcal{B}[\![x <= 2]\!]\sigma_0[x \mapsto 1] \ \rightarrow \ \mathbf{fix}\, F(\mathcal{C}[\![x := x + 1]\!]\sigma_0[x \mapsto 1]) \ ; \ \sigma_0[x \mapsto 1] \\
&\quad = \mathbf{fix}\, F(\sigma_0[x \mapsto 2]) \\
&\quad = \ldots
\end{aligned}$$

The important point of this example is that by proposition 3.2.5, it is quite acceptable to calculate by a sequence of unfold and reduce steps.

Exercise 3.2.7
What happens with **while true do skip** if we apply the approach suggested by proposition 3.2.5?

Exercise 3.2.8
Show that the following (iterative) factorial function run on an initial state with $y = 1$ and $x = 3$ results in a state where $y = 6$.

$$\textbf{while } 1 <= x \textbf{ do } (y := y * x; x := x - 1)$$

3.2.4 Full denotational semantics of Imp

For easy reference, we repeat the syntax and the entire set of clauses defining the denotational semantics of **Imp**.

Syntactic Domains

n : Num	numerals
x : Ide	variable identifiers
a : Aexp	arithmetic expressions
b : Bexp	boolean expressions
c : Com	commands (i.e. statements)

Syntax

$$a ::= n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \star a_2$$

$$b ::= \textbf{true} \mid \textbf{false} \mid a_1 == a_2 \mid a_1 <= a_2 \mid \textbf{not}\ b \mid b_1\ \textbf{and}\ b_2 \mid b_1\ \textbf{or}\ b_2$$

$$c ::= x := a \mid \textbf{skip} \mid c_1; c_2 \mid \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \mid \textbf{while}\ b\ \textbf{do}\ c$$

Semantic Domains

$$\sigma : \text{State} = \text{Ide} \to \mathbb{Z}$$

Semantic Functions

$$\mathcal{A} : \text{Aexp} \to \text{State} \to \mathbb{Z}$$

$$\mathcal{B} : \text{Bexp} \to \text{State} \to \mathbb{B}$$

$$\mathcal{C} : \text{Com} \to \text{State} \to \text{State}$$

Semantic Clauses

$$\mathcal{A}[\![n]\!]\sigma = \mathcal{N}[\![n]\!]$$

$$\mathcal{A}[\![x]\!]\sigma = \sigma[\![x]\!]$$

$$\mathcal{A}[\![a_0 + a_1]\!]\sigma = \mathcal{A}[\![a_0]\!]\sigma + \mathcal{A}[\![a_1]\!]\sigma$$

$$\mathcal{A}[\![a_0 - a_1]\!]\sigma = \mathcal{A}[\![a_0]\!]\sigma - \mathcal{A}[\![a_1]\!]\sigma$$

$$\mathcal{A}[\![a_0 * a_1]\!]\sigma = \mathcal{A}[\![a_0]\!]\sigma \times \mathcal{A}[\![a_1]\!]\sigma$$

$$\mathcal{B}[\![\textbf{true}]\!]\sigma = \textbf{tt}$$

$$\mathcal{B}[\![\textbf{false}]\!]\sigma = \textbf{ff}$$

$$\mathcal{B}[\![a_0 == a_1]\!]\sigma = (\mathcal{A}[\![a_0]\!]\sigma = \mathcal{A}[\![a_1]\!]\sigma)$$

$$\mathcal{B}[\![a_0 <= a_1]\!]\sigma = \mathcal{A}[\![a_0]\!]\sigma \leq \mathcal{A}[\![a_1]\!]\sigma$$

$$\mathcal{B}[\![\textbf{not } b]\!]\sigma = \neg\mathcal{B}[\![b]\!]\sigma$$

$$\mathcal{B}[\![b_0 \textbf{ or } b_1]\!]\sigma = \mathcal{B}[\![b_0]\!]\sigma \vee \mathcal{B}[\![b_1]\!]\sigma$$

$$\mathcal{B}[\![b_0 \textbf{ and } b_1]\!]\sigma = \mathcal{B}[\![b_0]\!]\sigma \wedge \mathcal{B}[\![b_1]\!]\sigma$$

$$\mathcal{C}[\![x := a]\!]\sigma = \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]$$

$$\mathcal{C}[\![\textbf{skip}]\!] = \mathbf{I}$$

$$\mathcal{C}[\![c_0; c_1]\!] = \mathcal{C}[\![c_1]\!] \circ \mathcal{C}[\![c_0]\!]$$

$$\mathcal{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma = \mathcal{B}[\![b]\!]\sigma \ \rightarrow\ \mathcal{C}[\![c_0]\!]\sigma \ ; \ \mathcal{C}[\![c_1]\!]\sigma$$

$$\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] = \textbf{fix}(\lambda\phi.\lambda\sigma.\mathcal{B}[\![b]\!]\sigma \ \rightarrow\ \phi(\mathcal{C}[\![c]\!]\sigma) \ ; \ \sigma)$$

Auxiliary Functions

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

$$t \ \rightarrow\ d_0 \ ; \ d_1 = \begin{cases} d_0 & \text{if } t = \mathbf{tt} \\ d_1 & \text{if } t = \mathbf{ff} \\ \bot & \text{if } t = \bot \end{cases}$$

$$\textbf{fix } f = \bigsqcup\{f^k(\bot) | k \geq 0\}$$

3.3 Semantic congruence — *not examinable*

Naturally, we should expect that the denotational semantics of **Imp** should correspond to the natural semantics developed in §2.1 in some precise way. We say that the two semantic definitions are *congruent* if the behaviour of every **Imp** program (according to the natural semantics) is 'the same as' the effect of that program (according to the denotational semantics). If this were not the case then then we would have defined two different languages with the same syntax. A possible application of this notion of congruence is proving the correctness of compilers.

This section will develop a precise statement of the congruence of the natural and denotational semantics of **Imp**, and prove that it does indeed hold. In fact, this particular case is comparatively straightforward because the two definitions are based on similar semantic foundations. In particular, both are defined in terms of *states* which in both cases are functions Ide $\rightarrow \mathbb{Z}$. In more realistic cases the relation between the semantic definitions is much less clear.

We will say that the natural semantics of **Imp** is congruent to the denotational semantics of **Imp**, provided that, for all $a : \text{Aexp}, b : \text{Bexp}, c : \text{Com}, n : \mathbb{Z}, t :$

$\mathbb{B}, \sigma, \sigma'$: State:[2]

- $\langle a, \sigma \rangle \Downarrow n \quad \Leftrightarrow \quad \mathcal{A}[\![a]\!]\sigma = n$
- $\langle b, \sigma \rangle \Downarrow t \quad \Leftrightarrow \quad \mathcal{B}[\![b]\!]\sigma = t$
- $\langle c, \sigma \rangle \Downarrow \sigma' \quad \Leftrightarrow \quad \mathcal{C}[\![c]\!]\sigma = \sigma'$

There is a slight subtlety here. Note that this implies that *definedness* also corresponds. In particular, an inherent requirement of the definition is that there is a derivation of $\langle c, \sigma \rangle \Downarrow \sigma'$ iff $\mathcal{C}[\![c]\!]\sigma$ is defined. Turning it around, $\langle c, \sigma \rangle$ is a stuck configuration iff $\mathcal{C}[\![c]\!]\sigma = \bot$.

It will be convenient later to work with semantic functions derived from the natural semantics. To distinguish them from the denotational semantic functions, they will be written with an overbar, e.g. $\overline{\mathcal{B}}$.

Definition 3.3.1
Define natural semantic functions:

$$\overline{\mathcal{A}} : \text{Aexp} \to \text{State} \to \mathbb{Z}$$

$$\overline{\mathcal{B}} : \text{Bexp} \to \text{State} \to \mathbb{B}$$

$$\overline{\mathcal{C}} : \text{Com} \to \text{State} \to \text{State}$$

as follows:

$$\overline{\mathcal{A}}[\![a]\!]\sigma = n \Leftrightarrow \langle a, \sigma \rangle \Downarrow n$$

$$\overline{\mathcal{B}}[\![b]\!]\sigma = t \Leftrightarrow \langle b, \sigma \rangle \Downarrow t$$

$$\overline{\mathcal{C}}[\![c]\!]\sigma = \begin{cases} \sigma' & \text{iff } \langle c, \sigma \rangle \Downarrow \sigma' \\ \bot & \text{otherwise, (i.e. } \langle c, \sigma \rangle \text{ is a stuck configuration)} \end{cases}$$

Note that the definitions of $\overline{\mathcal{A}}$ and $\overline{\mathcal{B}}$ are simpler than $\overline{\mathcal{C}}$ because there are no stuck configurations for arithmetic or boolean expressions in **Imp**. (Thus, $\overline{\mathcal{A}}[\![a]\!]$ and $\overline{\mathcal{B}}[\![b]\!]$ are total functions for all a, b.)

The congruence conditions can be neatly re-stated in terms of these natural semantic functions. The natural semantics of **Imp** is congruent to the denotational semantics of **Imp**, provided that:

$$\mathcal{A} = \overline{\mathcal{A}} \text{ and } \mathcal{B} = \overline{\mathcal{B}} \text{ and } \mathcal{C} = \overline{\mathcal{C}}$$

The congruence can be proven by establishing each condition separately.

[2] As a minor simplification, we are no longer distinguishing between numerals and numbers in the denotational semantics. That is, assume the clause for n is now written $\mathcal{A}[\![n]\!]\sigma = n$. This is the approach taken in the natural semantics in previous sections.

Lemma 3.3.2
$\mathcal{A} = \overline{\mathcal{A}}$. That is, for all $a : \text{Aexp}, n : \mathbb{Z}, \sigma : \text{State}$, $\langle a, \sigma \rangle \Downarrow n$ iff $\mathcal{A}[\![a]\!]\sigma = n$.

Proof The proof is easy by structural induction over a : Aexp. First the basis cases:

- $a \equiv n$
  Obvious. The natural semantics rule is:

  $$\frac{}{\langle n, \sigma \rangle \Downarrow n}\ num$$

  The denotational semantics clause is:

  $$\mathcal{A}[\![n]\!]\sigma = n$$

  (See footnote 2.) Hence $\mathcal{A}[\![n]\!]\sigma = \overline{\mathcal{A}}[\![n]\!]\sigma$ as required.

- $a \equiv x$
  Similarly straightforward. The natural semantics rule is:

  $$\frac{}{\langle x, \sigma \rangle \Downarrow \sigma(x)}\ var$$

  The denotational clause is $\mathcal{A}[\![x]\!]\sigma = \sigma[\![x]\!]$ so $\mathcal{A}[\![x]\!]\sigma = \overline{\mathcal{A}}[\![x]\!]\sigma$.

Now the inductive cases:

- $a \equiv a_0 + a_1$
  The inductive hypotheses are:
  $\langle a_0, \sigma \rangle \Downarrow n_0 \Leftrightarrow \mathcal{A}[\![a_0]\!]\sigma = n_0$ and $\langle a_1, \sigma \rangle \Downarrow n_1 \Leftrightarrow \mathcal{A}[\![a_1]\!]\sigma = n_1$.
  The natural semantics rule is:

  $$\frac{\langle a_0, \sigma \rangle \Downarrow n_0 \qquad \langle a_1, \sigma \rangle \Downarrow n_1}{\langle a_0 + a_1, \sigma \rangle \Downarrow n_0 + n_1}\ plus$$

  The denotational clause is:

  $$\mathcal{A}[\![a_0 + a_1]\!]\sigma = \mathcal{A}[\![a_0]\!]\sigma + \mathcal{A}[\![a_1]\!]\sigma$$

  So under the inductive hypothesis $\mathcal{A}[\![a_0 + a_1]\!]\sigma = n_0 + n_1 = \overline{\mathcal{A}}[\![a_0 + a_1]\!]\sigma$ as required.

The other inductive cases, $a \equiv a_0 - a_1$ and $a \equiv a_0 * a_1$ are similar. ■

Lemma 3.3.3
$\mathcal{B} = \overline{\mathcal{B}}$. That is, for all $b : \text{Bexp}, t : \mathbb{B}, \sigma : \text{State}$, $\langle b, \sigma \rangle \Downarrow t$ iff $\mathcal{B}[\![b]\!]\sigma = t$.

Proof Exercise. Again the proof should be by structural induction. Note that for relational expressions, e.g. $a_0 <= a_1$, this proof relies on lemma 3.3.2. ■

Proving the congruence condition for commands is not quite so straightforward, largely due to the recursive clause for while loops in the denotational semantics and the fact that the meanings of commands are partial functions. For this reason we will demonstrate the result in two parts: $\overline{\mathcal{C}}[\![c]\!] \sqsubseteq \mathcal{C}[\![c]\!]$ for all $c : \mathrm{Com}$, and $\mathcal{C}[\![c]\!] \sqsubseteq \overline{\mathcal{C}}[\![c]\!]$ for all $c : \mathrm{Com}$, which together give $\overline{\mathcal{C}} = \mathcal{C}$.

Lemma 3.3.4
For all $c : \mathrm{Com}, \overline{\mathcal{C}}[\![c]\!] \sqsubseteq \mathcal{C}[\![c]\!]$.

Proof By the definition of $\overline{\mathcal{C}}$, it is sufficient to show that $\langle c, \sigma\rangle \Downarrow \sigma' \Rightarrow \mathcal{C}[\![c]\!]\sigma = \sigma'$ for all $c : \mathrm{Com}, \sigma, \sigma' : \mathrm{State}$. The proof is by rule induction, that is, induction over the derivation tree of $\langle c, \sigma\rangle \Downarrow \sigma'$.

- $x := a$
  Suppose there is a derivation concluding that $\langle x := a, \sigma\rangle \Downarrow \sigma'$. The last rule must have been:

  $$\frac{\langle a, \sigma\rangle \Downarrow n}{\langle x := a, \sigma\rangle \Downarrow \sigma[x \mapsto n]}\ asst$$

  so $\sigma' = \sigma[x \mapsto n]$ where $\langle a, \sigma\rangle \Downarrow n$. By lemma 3.3.2, if $\langle a, \sigma\rangle \Downarrow n$ then $\mathcal{A}[\![a]\!]\sigma = n$.

  Hence $\mathcal{C}[\![x := a]\!]\sigma = \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma] = \sigma[x \mapsto n] = \sigma'$, as required.

- **skip**
  By the axiom, $\langle \mathbf{skip}, \sigma\rangle \Downarrow \sigma$. Similarly, $\mathcal{C}[\![\mathbf{skip}]\!]\sigma = \sigma$.

- $c_0; c_1$
  If there is a derivation concluding that $\langle c_0; c_1, \sigma\rangle \Downarrow \sigma'$ then by the rule *seq*, there must be derivations of $\langle c_0, \sigma\rangle \Downarrow \sigma''$ and $\langle c_1, \sigma''\rangle \Downarrow \sigma'$ for some state σ''. By the inductive hypothesis we assume that $\mathcal{C}[\![c_0]\!]\sigma = \sigma''$ and $\mathcal{C}[\![c_1]\!]\sigma'' = \sigma'$.

  Hence:

  $$\begin{aligned}\mathcal{C}[\![c_0; c1]\!]\sigma &= \mathcal{C}[\![c_1]\!](\mathcal{C}[\![c_0]\!]\sigma)\\ &= \mathcal{C}[\![c_1]\!]\sigma''\\ &= \sigma'\end{aligned}$$

  as required.

- **if** b **then** c_0 **else** c_1
  The last rule in the derivation must be either $if_{\mathbf{tt}}$ or $if_{\mathbf{ff}}$.

  - Suppose $\langle b, \sigma\rangle \Downarrow \mathbf{tt}$
    If there is a derivation of $\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma\rangle \Downarrow \sigma'$ then it is by the $if_{\mathbf{tt}}$ rule and there must be a derivation of $\langle c_0, \sigma\rangle \Downarrow \sigma'$. By the inductive hypothesis we assume that $\mathcal{C}[\![c_0]\!]\sigma = \sigma'$. By lemma 3.3.3, $\mathcal{B}[\![b]\!]\sigma = \mathbf{tt}$.

Hence:

$$\begin{aligned}\mathcal{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma &= \mathcal{B}[\![b]\!]\sigma \;\rightarrow\; \mathcal{C}[\![c_0]\!]\sigma \;;\; \mathcal{C}[\![c_1]\!]\sigma \\ &= \mathcal{C}[\![c_0]\!]\sigma \\ &= \sigma'\end{aligned}$$

as required.

– Similarly if $\langle b, \sigma\rangle \Downarrow \textbf{ff}$

- **while** b **do** c
  The last rule in the derivation must be either $\mathit{while}_{\textbf{tt}}$ or $\mathit{while}_{\textbf{ff}}$.

  – Suppose $\langle b, \sigma\rangle \Downarrow \textbf{ff}$
  If there is a derivation of $\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \Downarrow \sigma'$ then it is by the $\mathit{while}_{\textbf{ff}}$ rule and $\sigma' = \sigma$.
  The denotational clause for while loops is $\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] = \textbf{fix}\, W$ where $W = \lambda\phi.\lambda\sigma.\mathcal{B}[\![b]\!]\sigma \;\rightarrow\; \phi(\mathcal{C}[\![c]\!]\sigma) \;;\; \sigma$.
  Unfolding once (see proposition 3.3.1), we get:

$$\begin{aligned}\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma &= \textbf{fix}\, W\sigma \\ &= \mathcal{B}[\![b]\!]\sigma \;\rightarrow\; \textbf{fix}\, W(\mathcal{C}[\![c]\!]\sigma) \;;\; \sigma \\ &= \sigma\end{aligned}$$

  since by lemma 3.3.3, $\langle b, \sigma\rangle \Downarrow \textbf{ff} \Rightarrow \mathcal{B}[\![b]\!]\sigma = \textbf{ff}$.

  – Suppose $\langle b, \sigma\rangle \Downarrow \textbf{tt}$
  If there is a derivation of $\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \Downarrow \sigma'$ then it is by the $\mathit{while}_{\textbf{tt}}$ rule and there must be derivations of $\langle c, \sigma\rangle \Downarrow \sigma''$ and $\langle \textbf{while } b \textbf{ do } c, \sigma''\rangle \Downarrow \sigma'$ for some state σ''. By the inductive hypothesis we can assume that $\mathcal{C}[\![c]\!]\sigma = \sigma''$ and $\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma'' = \sigma'$.[3]
  Hence we have:

$$\begin{aligned}\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma &= \textbf{fix}\, W\sigma \\ &= \mathcal{B}[\![b]\!]\sigma \;\rightarrow\; \textbf{fix}\, W(\mathcal{C}[\![c]\!]\sigma) \;;\; \sigma \\ &= \textbf{fix}\, W(\mathcal{C}[\![c]\!]\sigma) \\ &= \textbf{fix}\, W\sigma'' \\ &= \mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma'' \\ &= \sigma'\end{aligned}$$

  as required. ■

[3] Note that if we had attempted the proof by structural induction, we could not have made this assumption and would have met the same problems that were discussed in relation to proposition 2.3.3.

Lemma 3.3.5
For all $c : \mathrm{Com}, \mathcal{C}[\![c]\!] \sqsubseteq \overline{\mathcal{C}}[\![c]\!]$.

Proof The proof is by structural induction.

- $x := a$
  By the definition of $\overline{\mathcal{C}}$ and the rule *asst* we have $\overline{\mathcal{C}}[\![x := a]\!]\sigma = \sigma[x \mapsto n]$ where $\langle a, \sigma\rangle \Downarrow n$. By lemma 3.3.2 we have $\mathcal{A}[\![a]\!]\sigma = n$.

  Hence:

$$\begin{aligned}\mathcal{C}[\![x := a]\!]\sigma =& \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma] \\ =& \sigma[x \mapsto n] \\ =& \overline{\mathcal{C}}[\![x := a]\!]\sigma\end{aligned}$$

- **skip**
  Straightforward. By the definition of $\overline{\mathcal{C}}$ and the denotational clause for **skip** we have $\mathcal{C}[\![\mathbf{skip}]\!]\sigma = \sigma = \overline{\mathcal{C}}[\![\mathbf{skip}]\!]\sigma$.

- $c_0; c_1$
  The denotational clause is

$$\mathcal{C}[\![c_0; c_1]\!]\sigma = \mathcal{C}[\![c_1]\!](\mathcal{C}[\![c_0]\!]\sigma)$$

  We assume that $\mathcal{C}[\![c_0]\!] \sqsubseteq \overline{\mathcal{C}}[\![c_0]\!]$ and $\mathcal{C}[\![c_1]\!] \sqsubseteq \overline{\mathcal{C}}[\![c_1]\!]$ by the inductive hypothesis, so:

$$\mathcal{C}[\![c_0; c_1]\!]\sigma \sqsubseteq \overline{\mathcal{C}}[\![c_1]\!](\overline{\mathcal{C}}[\![c_0]\!]\sigma)$$

  By the *seq* rule, $\langle c_0; c_1, \sigma\rangle$ is stuck if either $\langle c_0, \sigma\rangle$ is stuck or $\langle c_1, \sigma'\rangle$ is stuck, so by the definition of $\overline{\mathcal{C}}$ (definition 3.3.1):

$$\overline{\mathcal{C}}[\![c_1]\!](\overline{\mathcal{C}}[\![c_0]\!]\sigma) \sqsubseteq \overline{\mathcal{C}}[\![c_0; c_1]\!]\sigma$$

  Hence:

$$\mathcal{C}[\![c_0; c_1]\!]\sigma \sqsubseteq \overline{\mathcal{C}}[\![c_1]\!](\overline{\mathcal{C}}[\![c_0]\!]\sigma) \sqsubseteq \overline{\mathcal{C}}[\![c_0; c_1]\!]\sigma$$

  as required.

- **if** b **then** c_0 **else** c_1
  Clearly by the rules $if_{\mathbf{tt}}$ and $if_{\mathbf{ff}}$ and definition 3.3.1:

$$\overline{\mathcal{C}}[\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!]\sigma = \begin{cases}\overline{\mathcal{C}}[\![c_0]\!]\sigma & \text{if } \overline{\mathcal{B}}[\![b]\!]\sigma = \mathbf{tt} \\ \overline{\mathcal{C}}[\![c_1]\!]\sigma & \text{if } \overline{\mathcal{B}}[\![b]\!]\sigma = \mathbf{ff}\end{cases}$$

  Further, by lemma 3.3.3:

$$\overline{\mathcal{C}}[\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!]\sigma = \begin{cases}\overline{\mathcal{C}}[\![c_0]\!]\sigma & \text{if } \mathcal{B}[\![b]\!]\sigma = \mathbf{tt} \\ \overline{\mathcal{C}}[\![c_1]\!]\sigma & \text{if } \mathcal{B}[\![b]\!]\sigma = \mathbf{ff}\end{cases}$$

  So $\overline{\mathcal{C}}[\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!]\sigma = \mathcal{B}[\![b]\!]\sigma \ \rightarrow\ \overline{\mathcal{C}}[\![c_0]\!]\sigma\ ;\ \overline{\mathcal{C}}[\![c_1]\!]\sigma$

We may assume the inductive hypothesis for c_0 and c_1, so:

$$\begin{aligned}\mathcal{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma &= \mathcal{B}[\![b]\!]\sigma \ \rightarrow \ \mathcal{C}[\![c_0]\!]\sigma \ ; \ \mathcal{C}[\![c_1]\!]\sigma \\ &\sqsubseteq \mathcal{B}[\![b]\!]\sigma \ \rightarrow \ \overline{\mathcal{C}}[\![c_0]\!]\sigma \ ; \ \overline{\mathcal{C}}[\![c_1]\!]\sigma \\ &= \overline{\mathcal{C}}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma\end{aligned}$$

as required.

- **while** b **do** c
  The denotational semantic clause is:

  $$\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] = \textbf{fix}(\lambda\phi.\lambda\sigma.\mathcal{B}[\![b]\!]\sigma \ \rightarrow \ \phi(\mathcal{C}[\![c]\!]\sigma) \ ; \ \sigma)$$

  We will write $W = \lambda\phi.\lambda\sigma.\mathcal{B}[\![b]\!]\sigma \ \rightarrow \ \phi(\mathcal{C}[\![c]\!]\sigma) \ ; \ \sigma$ for brevity.

  If it can be shown that:

  $$W(\overline{\mathcal{C}}[\![\textbf{while } b \textbf{ do } c]\!]) \sqsubseteq \overline{\mathcal{C}}[\![\textbf{while } b \textbf{ do } c]\!]$$

  or, in other words:

  $$\mathcal{B}[\![b]\!]\sigma \ \rightarrow \ \overline{\mathcal{C}}[\![\textbf{while } b \textbf{ do } c]\!](\mathcal{C}[\![c]\!]\sigma) \ ; \ \sigma \sqsubseteq \overline{\mathcal{C}}[\![\textbf{while } b \textbf{ do } c]\!]\sigma$$

  then we are done, because $\overline{\mathcal{C}}[\![\textbf{while } b \textbf{ do } c]\!]$ is therefore a 'prefixpoint' of W and since $\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]$ is the *least* fixpoint of W it must follow that:[4]

  $$\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] \sqsubseteq \overline{\mathcal{C}}[\![\textbf{while } b \textbf{ do } c]\!]$$

  As in the previous case, by rules $\mathit{while}_{\mathbf{tt}}$ and $\mathit{while}_{\mathbf{ff}}$ and lemma 3.3.3, we have:

  $$\overline{\mathcal{C}}[\![\textbf{while } b \textbf{ do } c]\!]\sigma = \begin{cases} \overline{\mathcal{C}}[\![\textbf{while } b \textbf{ do } c]\!](\overline{\mathcal{C}}[\![c]\!]\sigma) & \text{if } \mathcal{B}[\![b]\!]\sigma = \mathbf{tt} \\ \sigma & \text{if } \mathcal{B}[\![b]\!]\sigma = \mathbf{ff} \end{cases}$$

  So $\overline{\mathcal{C}}[\![\textbf{while } b \textbf{ do } c]\!]\sigma = \mathcal{B}[\![b]\!]\sigma \ \rightarrow \ \overline{\mathcal{C}}[\![\textbf{while } b \textbf{ do } c]\!](\overline{\mathcal{C}}[\![c]\!]\sigma) \ ; \ \sigma$.

  By the inductive hypothesis we may assume $\mathcal{C}[\![c]\!] \sqsubseteq \overline{\mathcal{C}}[\![c]\!]$ so substituting into the equation we get:

  $$\mathcal{B}[\![b]\!]\sigma \ \rightarrow \ \overline{\mathcal{C}}[\![\textbf{while } b \textbf{ do } c]\!](\mathcal{C}[\![c]\!]\sigma) \ ; \ \sigma \sqsubseteq \overline{\mathcal{C}}[\![\textbf{while } b \textbf{ do } c]\!]\sigma$$

  as required. ■

[4] See Winskell, theorem 5.11 for more on prefixpoints and their relation to least fixpoints.

3.4 Denotational semantics of blocks and procedures

As we did for natural semantics in the preceding section, we will extend our language to include declarations of variables and procedures. We will assume static scope as the norm.

3.4.1 Blocks and variable declarations

We will treat variable declarations in isolation before going on to include procedures. Extend **Imp** with a new command representing a block:

$$c ::= \ldots \mid \textbf{begin } d; c \textbf{ end}$$

where d is a metavariable of syntactic set Decl defined as follows:

$$d ::= \textbf{var } x := a \mid d_0; d_1$$

The idea is that the scope of the variables declared in a block is limited to that block.

Again we follow the approach taken in the natural semantics by introducing domains of locations, environments and stores:

$$\rho : \text{Env} = \text{Ide} \to \text{Loc}$$

and:

$$\sigma : \text{Store} = \text{Loc} \to [\mathbb{Z} + \{\mathit{unused}\}]$$

Assume the same function $\mathit{new} : \text{Store} \to \text{Loc}$ which returns a new unused location each time it is called. (See §2.4.1.)

The semantic functions must take environments into account, so the state is 'factored' into an environment and a store, giving:

$$\mathcal{A} : \text{Aexp} \to \text{Env} \to \text{Store} \to \mathbb{Z}$$

$$\mathcal{B} : \text{Bexp} \to \text{Env} \to \text{Store} \to \mathbb{B}$$

$$\mathcal{C} : \text{Com} \to \text{Env} \to \text{Store} \to \text{Store}$$

The clauses for arithmetic expressions and boolean expressions are extended in a straightforward fashion to account for their new semantics in terms of environments. A few clauses will be given to demonstrate the basic idea. The full denotational semantics can be found at the end of this section. First some of the arithmetic expressions:

$$\mathcal{A}[\![n]\!]\rho\sigma = \mathcal{N}[\![n]\!]$$

$$\mathcal{A}[\![x]\!]\rho\sigma = \sigma \circ \rho[\![x]\!]$$

$$\mathcal{A}[\![a_0 + a_1]\!]\rho\sigma = \mathcal{A}[\![a_0]\!]\rho\sigma + \mathcal{A}[\![a_1]\!]\rho\sigma$$

The boolean expressions are equally straightforward:

$$\mathcal{B}[\![\mathbf{true}]\!]\rho\sigma = \mathbf{tt}$$

$$\mathcal{B}[\![a_0 <= a_1]\!]\rho\sigma = \mathcal{A}[\![a_0]\!]\rho\sigma \leq \mathcal{A}[\![a_1]\!]\rho\sigma$$

$$\mathcal{B}[\![b_0 \ \mathbf{or}\ b_1]\!]\rho\sigma = \mathcal{B}[\![b_0]\!]\rho\sigma \vee \mathcal{B}[\![b_1]\!]\rho\sigma$$

The meanings of commands are also in terms of environments and stores:

$$\mathcal{C}[\![x := a]\!]\rho\sigma = \sigma[\rho[\![x]\!] \mapsto \mathcal{A}[\![a]\!]\rho\sigma]$$

$$\mathcal{C}[\![c_0; c_1]\!]\rho = (\mathcal{C}[\![c_1]\!]\rho) \circ (\mathcal{C}[\![c_0]\!]\rho)$$

$$\mathcal{C}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!]\rho = \mathbf{fix}(\lambda\phi.\lambda\sigma.\mathcal{B}[\![b]\!]\rho\sigma \ \rightarrow\ \phi(\mathcal{C}[\![c]\!]\rho\sigma)\ ;\ \sigma)$$

and so on.

Our interest at the moment is in blocks and the declaration of variables. As in the natural semantics, the declarations set up the environment and initialise variables in the store, prior to executing the command that is the body of the block. The block is just another command so it will be handled by the semantic function $\mathcal{C}$. On the other hand, declarations are a new syntactic domain so we will require a new semantic function. The effect of a variable declaration is to modify both the environment and store. Hence the meaning of a declaration will be a function from and environment and a store to an environment and a store:

$$\mathcal{D} : \text{Decl} \rightarrow (\text{Env} \times \text{Store}) \rightarrow (\text{Env} \times \text{Store})$$

For each declared variable $\mathbf{var}\, x := a$, a new location is allocated and bound to x in the environment. In the store, that location is set to the value of a:

$$\mathcal{D}[\![\mathbf{var}\ x := a]\!](\rho, \sigma) = (\rho[x \mapsto l], \sigma[l \mapsto \mathcal{A}[\![a]\!]\rho\sigma])$$

where $l = new(\sigma)$.

The semantics of a sequence of variable declarations is naturally given by the composition of the semantics of the individual declarations:

$$\mathcal{D}[\![d_0; d_1]\!](\rho, \sigma) = \mathcal{D}[\![d_1]\!](\mathcal{D}[\![d_0]\!](\rho, \sigma))$$

In that case, the denotational semantic clause for blocks is as follows:

$$\mathcal{C}[\![\mathbf{begin}\ d; c\ \mathbf{end}]\!]\rho\sigma = \mathcal{C}[\![c]\!]\rho'\sigma'$$

where $\mathcal{D}[\![d]\!](\rho, \sigma) = (\rho', \sigma')$.

That is, construct a new environment ρ' and a new store σ' from the declarations d and then run the command c on ρ' and σ'.

Exercise 3.4.1
Use the semantic clauses above to show that the final value of y in the program below is 6.

```
begin
    var x := 1;
    var y := 0;
    begin
    var x := 5;
        y := y + x;
    end;
    y := y + x;
end
```

3.4.2 Procedures

Now we extend the language to include simple procedure declarations and calls. The syntax of **Proc** is an extension of **Imp** to include these features.

$$c ::= \ldots \mid \textbf{begin } d; p; c \textbf{ end}$$

where p is a metavariable of the syntactic set Pdecl defined as follows:

$$p ::= \textbf{proc } x \textbf{ is } c \mid p_0; p_1$$

As in the natural semantics treatment of this language, we will assume static scoping (see §2.4.2).

Procedures — the wrong way

It is tempting to simply follow the approach to procedures that was employed in the natural semantics by constructing closures from the body of the procedure and the static environment. In fact, this is not acceptable as will be seen if we follow through this plan. The semantic domains would be as in the natural semantics:

$$\text{Clo} = \text{Com} \times \text{Env}$$

$$\text{Env} = \text{Ide} \to (\text{Loc} + \text{Clo})$$

Introduce a semantic function $\mathcal{P}$ for procedure declarations:

$$\mathcal{P} : \text{Pdecl} \to \text{Env} \to \text{Env}$$

The clause for procedure declarations would directly correspond to the natural semantics rule:

$$\mathcal{P}[\![\textbf{proc } x \textbf{ is } c]\!]\rho = \rho[x \mapsto (c, \rho)]$$

The clause for procedure calls (not permitting recursion) could then be as follows:

$$\mathcal{C}[\![\textbf{call } x]\!]\rho\sigma = \mathcal{C}[\![c]\!]\rho'\sigma$$

where $\rho[\![x]\!] = (c, \rho')$.

On the face of it, these clauses seem reasonable. However, they are *not compositional* — the meaning of **call**x is defined in terms of some command c which is not a subphrase of **call** x. Since compositionality is a fundamental requirement of denotational definitions, we must reject the idea of basing the semantics of procedures on closures or any other such 'syntactic' constructs.

Procedures — the right way

Rather than keeping a closure in the environment, the solution is to keep the *meaning* of the procedure in the environment. Intuitively, a call to a function has the same effect as executing the command which is its body. Now the meaning of a command is a function Env $\to$ Store $\to$ Store so the meaning of a procedure should be something similar. However, since **Proc** has static scoping rules, the environment that occurs at the point of procedure declaration is the one in which the body will be executed when the procedure is called (that's precisely why the closures in the natural semantics of **Proc** contain the environment from the point of declaration). Hence, the meaning (or *denotation*) of a procedure is simply a map:

$$\text{Store} \to \text{Store}$$

When some procedure x is called, look x up in the environment to get the store to store map that it represents and then supply the current store (i.e. from the point of call). Now for the details.

The semantic domain of environments is extended as follows:

$$\rho : \text{Env} = \text{Ide} \to (\text{Loc} + (\text{Store} \to \text{Store}))$$

A new semantic function for procedure declarations is introduced:

$$\mathcal{P} : \text{Pdecl} \to \text{Env} \to \text{Env}$$

Procedure declarations are handled by the following two clauses:

$$\mathcal{P}[\![\textbf{proc } x \textbf{ is } c]\!]\rho = \rho[x \mapsto \mathcal{C}[\![c]\!]\rho]$$

$$\mathcal{P}[\![p_0; p_1]\!]\rho = \mathcal{P}[\![p_1]\!](\mathcal{P}[\![p_0]\!]\rho)$$

The second clause is simply a composition, as you may expect. In fact we could have written $\mathcal{P}[\![p_0; p_1]\!] = (\mathcal{P}[\![p_1]\!]) \circ (\mathcal{P}[\![p_0]\!])$. The first clause binds $\mathcal{C}[\![c]\!]\rho$ to x where c is the body of the procedure and ρ is the static environment. Notice that the store argument has not yet been supplied so $\mathcal{C}[\![c]\!]\rho$, the value bound to x, is a function Store $\to$ Store. As you expect, the store argument is supplied at the point of the procedure call:

$$\mathcal{C}[\![\textbf{call } x]\!]\rho\sigma = \rho[\![x]\!]\sigma$$

The clauses as presented above have a shortcoming in that they do not permit recursive procedure calls. In attempting to deal with this, it is clear that the

approach taken in natural semantics, where the problem is solved in the *call* rule by adding the procedure binding to the environment before executing the procedure body, will not work here. After all, the environment is 'abstracted' in the sense that it has already been supplied as an argument to $\mathcal{C}[\![c]\!]$ when processing the declaration.

Clearly, recursion must be dealt with in the clauses for procedure declarations. Rather than binding $\mathcal{C}[\![c]\!]\rho$ to x we want $\mathcal{C}[\![c]\!]\rho'$ where ρ' is ρ extended with the binding for x. If this looks circular, you are right! More precisely, what we want is:

$$\mathcal{P}[\![\textbf{proc } x \textbf{ is } c]\!]\rho = \rho'$$

where

$$\rho' = \rho[x \mapsto \mathcal{C}[\![c]\!]\rho']$$

Notice that this second equation is recursive, so it it more correctly written using **fix**. the clause becomes:

$$\mathcal{P}[\![\textbf{proc } x \textbf{ is } c]\!]\rho = \textbf{fix}(\lambda\rho'.\rho[x \mapsto \mathcal{C}[\![c]\!]\rho'])$$

Exercise 3.4.2
The folowing clause is suggested to permit recursive procedure calls in **Proc**:

$$\mathcal{P}[\![\textbf{proc } x \textbf{ is } c]\!]\rho = \rho[x \mapsto \mathcal{C}[\![c]\!]\rho[x \mapsto \mathcal{C}[\![c]\!]\rho]]$$

Clearly explain, with the help of an example, why this not satisfactory.

How about mutual recursion? This was quite difficult in natural semantics, but the validity of fixpoint solutions make it reasonably straightforward in denotational semantics. Though strictly unnecessary, it is simpler if we work with a different abstract syntax for procedure declarations:

$$p ::= \textbf{proc } x \textbf{ is } c; p \mid \epsilon$$

Wher ϵ represents an empty sequence of procedure declarations. Now the environment supplied to each procedure declaration should be the environment that contains all the bindings for the whole block. That is:

$$\mathcal{P}[\![\textbf{proc } x \textbf{ is } c; p]\!]\rho = \rho'$$

where

$$\rho' = \mathcal{P}[\![p]\!](\rho[x \mapsto \mathcal{C}[\![c]\!]\rho']$$

More correctly, using **fix**:

$$\mathcal{P}[\![\textbf{proc } x \textbf{ is } c; p]\!]\rho = \textbf{fix}(\lambda\rho'.\mathcal{P}[\![p]\!](\rho[x \mapsto \mathcal{C}[\![c]\!]\rho'])$$

$$\mathcal{P}[\![\epsilon]\!]\rho = \rho$$

3.4.3 Denotational semantics of Proc

For convenient reference, this section brings together the entire denotational semantic specification of **Proc**.

Syntactic Domains

n : Num	numerals
x : Ide	variable identifiers
a : Aexp	arithmetic expressions
b : Bexp	boolean expressions
c : Com	commands (i.e. statements)
d: Decl	variable declarations
p: Pdecl	procedure declarations

Syntax

$$
\begin{aligned}
a &::= n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \\
b &::= \textbf{true} \mid \textbf{false} \mid a_1 == a_2 \mid a_1 <= a_2 \mid \textbf{not}\ b \mid b_1\ \textbf{and}\ b_2 \mid b_1\ \textbf{or}\ b_2 \\
c &::= x := a \mid \textbf{skip} \mid c_1; c_2 \mid \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \mid \textbf{while}\ b\ \textbf{do}\ c \\
&\quad \mid \textbf{begin}\ d; p; c\ \textbf{end} \mid \textbf{call}\ x \\
d &::= \textbf{var}\ x := a \mid d_0; d_1 \\
p &::= \textbf{proc}\ x\ \textbf{is}\ c \mid p_0; p_1
\end{aligned}
$$

Semantic Domains

$$\rho : \text{Env} = \text{Ide} \to (\text{Loc} + (\text{Store} \to \text{Store}))$$

$$\sigma : \text{Store} = \text{Loc} \to (\mathbb{Z} + \{\mathit{unused}\})$$

Semantic Functions

$$\mathcal{A} : \text{Aexp} \to \text{Env} \to \text{Store} \to \mathbb{Z}$$

$$\mathcal{B} : \text{Bexp} \to \text{Env} \to \text{Store} \to \mathbb{B}$$

$$\mathcal{C} : \text{Com} \to \text{Env} \to \text{Store} \to \text{Store}$$

$$\mathcal{D} : \text{Decl} \to (\text{Env} \times \text{Store}) \to (\text{Env} \times \text{Store})$$

$$\mathcal{P} : \text{Pdecl} \to \text{Env} \to \text{Env}$$

Semantic Clauses

$$
\begin{aligned}
\mathcal{A}[\![n]\!]\rho\sigma &= \mathcal{N}[\![n]\!] \\
\mathcal{A}[\![x]\!]\rho\sigma &= \sigma \circ \rho[\![x]\!] \\
\mathcal{A}[\![a_0 + a_1]\!]\rho\sigma &= \mathcal{A}[\![a_0]\!]\rho\sigma + \mathcal{A}[\![a_1]\!]\rho\sigma \\
\mathcal{A}[\![a_0 - a_1]\!]\rho\sigma &= \mathcal{A}[\![a_0]\!]\rho\sigma - \mathcal{A}[\![a_1]\!]\rho\sigma \\
\mathcal{A}[\![a_0 * a_1]\!]\rho\sigma &= \mathcal{A}[\![a_0]\!]\rho\sigma \times \mathcal{A}[\![a_1]\!]\rho\sigma
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{B}[\![\mathbf{true}]\!]\rho\sigma &= \mathbf{tt} \\
\mathcal{B}[\![\mathbf{false}]\!]\rho\sigma &= \mathbf{ff} \\
\mathcal{B}[\![a_0 == a_1]\!]\rho\sigma &= (\mathcal{A}[\![a_0]\!]\rho\sigma = \mathcal{A}[\![a_1]\!]\rho\sigma) \\
\mathcal{B}[\![a_0 <= a_1]\!]\rho\sigma &= \mathcal{A}[\![a_0]\!]\rho\sigma \leq \mathcal{A}[\![a_1]\!]\rho\sigma \\
\mathcal{B}[\![\mathbf{not}\ b]\!]\rho\sigma &= \neg\mathcal{B}[\![b]\!]\rho\sigma \\
\mathcal{B}[\![b_0\ \mathbf{or}\ b_1]\!]\rho\sigma &= \mathcal{B}[\![b_0]\!]\rho\sigma \vee \mathcal{B}[\![b_1]\!]\rho\sigma \\
\mathcal{B}[\![b_0\ \mathbf{and}\ b_1]\!]\rho\sigma &= \mathcal{B}[\![b_0]\!]\rho\sigma \wedge \mathcal{B}[\![b_1]\!]\rho\sigma \\
\mathcal{D}[\![\mathbf{var}\ x := a]\!](\rho,\sigma) &= (\rho[x \mapsto l], \sigma[l \mapsto \mathcal{A}[\![a]\!]\rho\sigma]) \qquad \text{where } l = new(\sigma). \\
\mathcal{D}[\![d_0; d_1]\!](\rho,\sigma) &= \mathcal{D}[\![d_1]\!](\mathcal{D}[\![d_1]\!](\rho,\sigma)) \\
\mathcal{P}[\![\mathbf{proc}\ x\ \mathbf{is}\ c]\!]\rho &= \rho[x \mapsto \mathcal{C}[\![c]\!]\rho]
\end{aligned}
$$

The clause above is the non-recursive version. Alternatively, to allow direct recursive function calls, the following clause suffices:

$$
\begin{aligned}
\mathcal{P}[\![\mathbf{proc}\ x\ \mathbf{is}\ c]\!]\rho &= \mathbf{fix}(\lambda\rho'.\rho[x \mapsto \mathcal{C}[\![c]\!]\rho']) \\
\mathcal{P}[\![p_0; p_1]\!]\rho &= \mathcal{P}[\![p_1]\!](\mathcal{P}[\![p_0]\!]\rho) \\
\mathcal{C}[\![x := a]\!]\rho\sigma &= \sigma[\rho[\![x]\!] \mapsto \mathcal{A}[\![a]\!]\rho\sigma] \\
\mathcal{C}[\![\mathbf{skip}]\!] &= \mathbf{I} \\
\mathcal{C}[\![c_0; c_1]\!] &= (\mathcal{C}[\![c_1]\!]) \circ (\mathcal{C}[\![c_0]\!]) \\
\mathcal{C}[\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!]\rho\sigma &= \mathcal{B}[\![b]\!]\rho\sigma\ \rightarrow\ \mathcal{C}[\![c_0]\!]\rho\sigma\ ;\ \mathcal{C}[\![c_1]\!]\rho\sigma \\
\mathcal{C}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!]\rho &= \mathbf{fix}(\lambda\phi.\lambda\sigma.\mathcal{B}[\![b]\!]\rho\sigma\ \rightarrow\ \phi(\mathcal{C}[\![c]\!]\rho\sigma)\ ;\ \sigma) \\
\mathcal{C}[\![\mathbf{begin}\ d; p; c\ \mathbf{end}]\!]\rho\sigma &= \mathcal{C}[\![c]\!](\mathcal{P}[\![p]\!]\rho')\sigma' \qquad \text{where } \mathcal{D}[\![d]\!](\rho,\sigma) = (\rho',\sigma') \\
\mathcal{C}[\![\mathbf{call}\ x]\!]\rho &= \rho[\![x]\!]
\end{aligned}
$$

Auxiliary Functions

$$
\sigma[l \mapsto n](l') = \begin{cases} n & \text{if } l = l' \\ \sigma(l') & \text{otherwise} \end{cases}
$$

$$
\rho[x \mapsto l](y) = \begin{cases} l & \text{if } x = y \\ \rho(y) & \text{otherwise} \end{cases}
$$

$$
t\ \rightarrow\ d_0\ ;\ d_1 = \begin{cases} d_0 & \text{if } t = \mathbf{tt} \\ d_1 & \text{if } t = \mathbf{ff} \\ \perp & \text{if } t = \perp \end{cases}
$$

$$
\mathbf{fix}\ f = \bigsqcup\{f^k(\perp) | k \geq 0\}
$$

3.5 Continuations and control

So far, the semantic clause for a sequence of commands, say $c_0; c_1$ has specified that c_1 will certainly be executed after c_0. But in many languages, this is not the case, e.g. `break` and `continue` in **Java** and **C**, exception throw, function return, and the classic `goto` command.

Clearly, if c_0 transfers control to another point in the program then the $\mathcal{C}[\![c_0; c_1]\!]\sigma$ cannot be the composition $\mathcal{C}[\![c_1]\!](\mathcal{C}[\![c_0]\!]\sigma)$ because we don't want c_1 to be executed.

This issue can be addressed by including another argument to the semantic function, to make control explicit. This argument, known as a *continuation*, represents "the rest of the program". For commands in **Imp** continuations may naturally be modelled as just a State to State mapping. Note that there is no "rest" for a whole program, so the initial continuation in that case is the identity, $\lambda\sigma.\sigma$.

Denotational semantic definitions using continuations are often referred to as *continuation style*, while the definitions we have been looking at so in previous sections are known as *direct style*. Continuations have also been found very useful in practical situations, such as structuring compilers, and to provide "sequencing" control in functional languages. For example, before the adoption of monads, Haskell's I/O was controlled with continuations. There is a style of programming popular in some functional languages, especially Scheme and Lisp, known as *continuation passing style* (CPS), and some may include an expression to call the current continuation.

3.5.1 Command continuations

To give a basic demonstration of the concept and application of continuations we will add a command to **Imp** that alters the execution sequence. One of the simplest such control commands is a Fortran-style **stop** command. As its name suggests, the effect of the **stop** command is to terminate the program, in its current state. Note that it is not intended as an exception, abort or failure — it is a graceful termination.

Clearly, the semantic clause for sequences in **Imp** is now inadequate. The execution of the command $\mathbf{stop}; c_1$ is not intended to run c_1, but the clause instance would be $\mathcal{C}[\![c_1]\!](\mathcal{C}[\![\mathbf{stop}]\!]\sigma)$, specifying that c_1 affects the state that exists after the **stop** command.

Continuations can look a little strange at first. Here's the basic idea: Suppose we have a sequence of commands $c_0; c_1$ and a continuation, say θ, which represents the *rest of the program* following $c_0; c_1$. We must execute c_0 first, but what is its continuation? It is the rest of the program — that is, c_1 followed by θ. By brutally abusing notation and mixing syntactic and semantic values, the idea is something like:

$$(c_0; c_1); \theta \quad \Longrightarrow \quad c_0; (c_1; \theta)$$

Here's the key point: *if c_0 is a **stop** command then it discards the rest of the program — its continuation $(c_1; \theta)$ — so c_1 is not executed. Magic!*

Here are the details. The meaning of the "rest of the program" is a value from the same domain as the meaning of whole programs, so the domain of *command continuations* is:

$$\theta : \text{Cont} = \text{State} \to \text{State}$$

A value from the domain of continuations is now an argument to the semantic function:

$$\mathcal{C} : \text{Com} \to \text{Cont} \to \text{State} \to \text{State}$$

(Or, equivalently: $\mathcal{C} : \text{Com} \to \text{Cont} \to \text{Cont}$.)

Now we can make the sequencing of commands explicit:

$$\mathcal{C}[\![c_0; c_1]\!]\theta\sigma = \mathcal{C}[\![c_0]\!](\mathcal{C}[\![c_1]\!]\theta)\sigma$$

As discussed, the clause for **stop** will ignore the continuation argument, so the state existing at that point becomes the final state:

$$\mathcal{C}[\![\textbf{stop}]\!]\theta\,\sigma = \sigma$$

Example 3.5.1
To demonstrate that continuations really do work, consider the command:

$$x := 1;\ \textbf{stop};\ x := 99$$

Treating it as a whole program, the initial continuation is the identity, so:

$$\begin{aligned}
&\mathcal{C}[\![x := 1; \textbf{stop}; x := 99]\!]\mathbf{I} \\
&= \mathcal{C}[\![x := 1; \textbf{stop}]\!](\mathcal{C}[\![x := 99]\!]\mathbf{I}) \\
&\ldots \\
&= \mathcal{C}[\![x := 1; \textbf{stop}]\!](\lambda\sigma.\sigma[x \mapsto 99]) \\
&= \mathcal{C}[\![x := 1]\!](\mathcal{C}[\![\textbf{stop}]\!](\lambda\sigma.\sigma[x \mapsto 99])) \\
&= \mathcal{C}[\![x := 1]\!]\mathbf{I} \\
&= \lambda\sigma.\sigma[x \mapsto 1]
\end{aligned}$$

Notice how the continuation which binds 99 to x is discarded by the clause for **stop** and hence the overall effect of the program is to bind 1 to x, as desired.

Exercise 3.5.2
Add a **break** command to the language **Proc**. Within a **while** loop, the effect of a **break** command is to directly exit the loop, passing control to the point in the program immediately following the loop. Outside a loop, the effect of **break** is identical to **stop**.

*Hint: In terms of continuations, the meaning of **break** is the continuation of the closest containing **while**. At loop entry, we therefore need to record the loop's continuation, say as a special component of the environment.*

3.5.2 Expression continuations

Sequencing control isn't restricted to commands — it can occur in expressions too, and even in declarations. For example, if commands can be embedded in expressions (e.g. typical function calls in imperative languages) then the sequencing issue leaks into the semantics of expressions.

Exercise 3.5.3
In many (most?) programming languages, $a_0 + a_1$ is not semantically equivalent to $a_1 + a_0$. Name at least one such language and give an example demonstrating the difference.

Do you know of any programming languages where such expressions *are* semantically equivalent? How would you demonstrate such an assertion?

An easy way to introduce side-effects to the expressions of **Imp** is by the addition of an expression of the form:

do c **result** a

The intention is that command c is executed (thus possibly modifying the state) before expression a is evaluated and returned as result.

It is easy to see that expressions:

(**do** x:=1 **result** 2) + x and x + (**do** x:=1 **result** 2)

are not semantically equivalent. (For example, if x is bound to 0 in the initial state.)

In that case, the semantics needs specify the order of evaluation of sub-expressions, as well as handling the possibility of exceptional behaviour of c.

So the situation is slightly different from command continuations. For expressions, "the rest of the program" depends on a value being supplied from the expression, before execution can proceed.

You will see that the sequence of expression evaluation is explicit in the continuation semantics.

Let's get down to the details. First, arithmetic expressions can appear in boolean expressions so both $\mathcal{A}$ and $\mathcal{B}$ will be in terms of continuations. It is notationally convenient to define a semantic domain of *expressible values*:

$$\varepsilon : \mathrm{Val} = \mathbb{Z} + \mathbb{B}$$

Now *expression continuations* are defined as follows:

$$\kappa : \mathrm{ECont} = \mathrm{Val} \to \mathrm{State} \to \mathrm{State}$$

Essentially, expression continuations are functions waiting for a value as input to the state change which represents the rest of the program.

The continuation-style semantic functions are:

$$\mathcal{A} : \text{Aexp} \to \text{ECont} \to \text{State} \to \text{State}$$
$$\mathcal{B} : \text{Bexp} \to \text{ECont} \to \text{State} \to \text{State}$$

Consider the clause for addition expressions:

$$\mathcal{A}[\![a_0 + a_1]\!]\kappa = \mathcal{A}[\![a_0]\!](\lambda n_0.\mathcal{A}[\![a_1]\!](\lambda n_1.\kappa(n_0 + n_1)))$$

Notice how the sequence of steps in evaluating the addition expression are explicit in the "nested" continuation: a_0 is evaluated and bound to n_0; a_1 is evaluated and bound to n_1; and κ, the "rest of the program" (which is waiting for the result), carries on with $n_0 + n_1$.

Some clauses need to make explicit reference to the current state, either for a lookup or an update. In that case, we can specify the state as an explicit argument in the clause, as well as the continuation (look again at the types of $\mathcal{A}$ and $\mathcal{C}$.) Here is an example:

$$\mathcal{A}[\![x]\!]\kappa\,\sigma = \kappa\,(\sigma[\![x]\!])\,\sigma$$

The expression continuation κ is passed the value of x found in the current state σ, and then σ is passed as the second argument to κ, to yield the final state of the program.

The clause for assignments demonstrates how the sequence of actions for a simple command is also made explicit:

$$\mathcal{C}[\![x := a]\!]\theta = \mathcal{A}[\![a]\!](\lambda n.\lambda\sigma.\theta(\sigma[x \mapsto n]))$$

The expression a is evaluated and bound to n and the command continuation is modified to make the update before proceeding with the rest of the program.

Finally, the clause for the new construct indicates that c is executed and its continuation commences with the evaluation of a, as we would wish.

$$\mathcal{A}[\![\textbf{do } c \textbf{ result } a]\!]\kappa = \mathcal{C}[\![c]\!](\mathcal{A}[\![a]\!]\kappa)$$

Example 3.5.4
Let's see how the semantics handles a control transfer inside an expression. An obvious example is a **stop** inside a **result** expression:

$$\begin{aligned} & \mathcal{A}[\![\textbf{do stop result 42}]\!]\kappa\,\sigma \\ = {} & \mathcal{C}[\![\textbf{stop}]\!](\mathcal{A}[\![42]\!]\kappa)\,\sigma \\ = {} & \mathbf{I}\,\sigma \\ = {} & \sigma \end{aligned}$$

Note how the continuation κ is discarded and the current state is returned as the outcome of the program.

Example 3.5.5
The expressions given above to show that addition in the extended **Imp** is not commutative take a little effort to work through:

$$\begin{aligned}
&\mathcal{A}[\![(\mathbf{do}\, x := 1\, \mathbf{result}\, 2) + x]\!]\kappa\, \sigma \\
&= \mathcal{A}[\![\mathbf{do}\, x := 1\, \mathbf{result}\, 2]\!](\lambda n_0.\mathcal{A}[\![x]\!](\lambda n_1.\kappa(n_0 + n_1)))\, \sigma \\
&= \mathcal{C}[\![x := 1]\!](\mathcal{A}[\![2]\!](\lambda n_0.\mathcal{A}[\![x]\!](\lambda n_1.\kappa(n_0 + n_1))))\, \sigma \\
&= \mathcal{A}[\![1]\!](\lambda n.\lambda\sigma'.\mathcal{A}[\![2]\!](\lambda n_0.\mathcal{A}[\![x]\!](\lambda n_1.\kappa(n_0 + n_1)))\, \sigma'[x \mapsto n])\, \sigma \\
&= \mathcal{A}[\![2]\!](\lambda n_0.\mathcal{A}[\![x]\!](\lambda n_1.\kappa(n_0 + n_1)))\, (\sigma[x \mapsto 1]) \\
&= \mathcal{A}[\![x]\!](\lambda n_1.\kappa(2 + n_1))\, (\sigma[x \mapsto 1]) \\
&= (\lambda n_1.\kappa(2 + n_1))(\sigma[x \mapsto 1][\![x]\!])\, (\sigma[x \mapsto 1]) \\
&= (\lambda n_1.\kappa(2 + n_1))\, 1\, (\sigma[x \mapsto 1]) \\
&= \kappa\, 3\, (\sigma[x \mapsto 1])
\end{aligned}$$

So the effect is to pass 3 to the continuation and to bind x to 1 in the state then passed as next argument to κ.

You should attempt the same calculation for $x + (\mathbf{do}\, x := 1\, \mathbf{result}\, 2)$ and note the different outcome.

Exercise 3.5.6
BCPL (the forerunner of C) had constructs similar to those discussed here, but the command and result expression were separated. That is, there was an expression **valof** c, where c was expected to include a command **resultis** e. Add such a pair of constructs to the language and define their semantics.

What would you consider to be a sensible meaning for a **valof** expression that doesn't contain a **resultis** command? What you consider to be a sensible meaning for a **resultis** that occurs outside a **valof**? Make sure your semantic definition expresses your decision.

3.5.3 Continuation semantics of Imp+

Here is the complete continuation semantics of **Imp** extended to include the ***stop*** command and the ***result*** expression discussed above.

Syntactic Domains

n : Num	numerals
x : Ide	variable identifiers
a : Aexp	arithmetic expressions
b : Bexp	boolean expressions
c : Com	commands

Syntax

$$
\begin{array}{rcl}
a & ::= & n \mid x \mid a_1 + a_2 \mid \mathbf{do}\, c\, \mathbf{result}\, a \\
b & ::= & \mathbf{true} \mid \mathbf{false} \mid a_1 == a_2 \mid \mathbf{not}\, b \mid b_1\, \mathbf{and}\, b_2 \\
c & ::= & x := a \mid \mathbf{skip} \mid c_1; c_2 \mid \mathbf{if}\, b\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2 \mid \mathbf{while}\, b\, \mathbf{do}\, c \mid \mathbf{stop}
\end{array}
$$

Semantic Domains

$\varepsilon : \mathrm{Val} = \mathbb{Z} + \mathbb{B}$	Expressible values
$\sigma : \mathrm{State} = \mathrm{Ide} \to \mathbb{Z}$	States
$\theta : \mathrm{Cont} = \mathrm{State} \to \mathrm{State}$	Command continuations
$\kappa : \mathrm{ECont} = \mathrm{Val} \to \mathrm{Cont}$	Expression continuations

Semantic Functions

$$
\begin{array}{c}
\mathcal{A} : \mathrm{Aexp} \to \mathrm{ECont} \to \mathrm{Cont} \\
\mathcal{B} : \mathrm{Bexp} \to \mathrm{ECont} \to \mathrm{Cont} \\
\mathcal{C} : \mathrm{Com} \to \mathrm{Cont} \to \mathrm{Cont}
\end{array}
$$

Semantic Clauses

$$
\begin{array}{rcl}
\mathcal{A}[\![n]\!]\kappa & = & \kappa(\mathcal{N}[\![n]\!]) \\
\mathcal{A}[\![x]\!]\kappa & = & \lambda\sigma.\kappa(\sigma[\![x]\!])\sigma \\
\mathcal{A}[\![a_0 + a_1]\!]\kappa & = & \mathcal{A}[\![a_0]\!](\lambda n_0.\mathcal{A}[\![a_1]\!](\lambda n_1.\kappa(n_0 + n_1))) \\
\mathcal{A}[\![\mathbf{do}\, c\, \mathbf{result}\, a]\!]\kappa & = & \mathcal{C}[\![c]\!](\mathcal{A}[\![a]\!]\kappa) \\
\mathcal{B}[\![\mathbf{true}]\!]\kappa & = & \kappa\, \mathbf{tt} \\
\mathcal{B}[\![\mathbf{false}]\!]\kappa & = & \kappa\, \mathbf{ff} \\
\mathcal{B}[\![a_0 == a_1]\!]\kappa & = & \mathcal{A}[\![a_0]\!](\lambda n_0.\mathcal{A}[\![a_1]\!](\lambda n_1.\kappa(n_0 = n_1))) \\
\mathcal{B}[\![\mathbf{not}\, b]\!]\kappa & = & \mathcal{B}[\![b]\!](\lambda t.\kappa(\neg t)) \\
\mathcal{B}[\![b_0\, \mathbf{and}\, b_1]\!]\kappa & = & \mathcal{B}[\![b_0]\!](\lambda t_0.\mathcal{B}[\![b_1]\!](\lambda t_1.\kappa(t_0 \wedge t_1))) \\
\mathcal{C}[\![x := a]\!]\theta & = & \mathcal{A}[\![a]\!](\lambda n.\lambda\sigma.\theta\, \sigma[x \mapsto n]) \\
\mathcal{C}[\![\mathbf{skip}]\!]\theta & = & \theta \\
\mathcal{C}[\![\mathbf{stop}]\!]\theta & = & \mathbf{I} \\
\mathcal{C}[\![c_0; c_1]\!]\theta & = & \mathcal{C}[\![c_0]\!](\mathcal{C}[\![c_1]\!]\theta) \\
\mathcal{C}[\![\mathbf{if}\, b\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1]\!]\theta & = & \mathcal{B}[\![b]\!](\lambda t.t \;\to\; \mathcal{C}[\![c_0]\!]\theta \;;\; \mathcal{C}[\![c_1]\!]\theta) \\
\mathcal{C}[\![\mathbf{while}\, b\, \mathbf{do}\, c]\!]\theta & = & \mathbf{fix}(\lambda\theta'.\mathcal{B}[\![b]\!](\lambda t.t \;\to\; \mathcal{C}[\![c]\!]\theta' \;;\; \theta))
\end{array}
$$