Operational Semantics

Course Notes

Steffen van Bakel
Department of Computing
Imperial College of Science, Technology and Medicine

Spring 2002

These notes are based on slides prepared by Chris Hankin, and to be used in the course Operational Semantics 2.12 at the Department of Computing, Imperial College. The notes lean heavily on the book *Semantics with Applications A Formal Introduction* by Hanne Riis Nielson and Flemming Nielson, available at

`http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.html`

that is recommended for further reading.

Contents

1 Why formal semantics?

In dealing with programming, we distinguish two different aspects of programming languages.

(*Syntax*) : Deals with the form of sentences:

$$\underbrace{\textit{The fish}}_{\text{NP}}\ \underbrace{\textit{answered the walk}}_{\text{VP}}$$

Syntax is specified by a *grammar*, normally in BNF.

(*Semantics*) : Expresses the meaning of sentences

$$\texttt{while}\ b\ \texttt{do}\ C$$

"Execute C repeatedly so long as the expression b is true"

Although many programmers could not be bothered with a formal semantics, it plays a major role in everyday programming. When learning a new programming language, normally we only look at the syntax, to know how certain language constructs are written. However, syntax is not concerned with 'soundness', it just deals with correctly formed sentences. This implies that the programmer is lost when it becomes necessary to check (argue) that the specified program actually computes the intended operation.

In this process, normally the syntax suffices. Every language construct, like '**`while`** b **`do`** C' has an intended meaning, formulated in natural language in text books, and it is this informal semantics that the programmer uses when writing an actual program. It is not uncommon, however, for a programmer to find that, although the structure of the program is correct in the sense that the programmer has used all language constructs in the right way and in the right order, the program still does not execute as it should. This is caused by the fact that the intended semantics only speaks in *general terms*, and does not give any means to *actually check* that the program is indeed the desired one.

Programs normally are too big, too complex to understand and check 'by hand.' Frequently programming errors, even the most blatant ones, are overlooked for a long time.

It is important that the semantics is formal, systematic and verifiable to provide:

i) the user with an unambiguous description of the effect of a program. Specifications in natural language may be more easy to read, but they are also highly ambiguous.

ii) a yardstick for implementation. With the formal semantics at hand, more efficient programs can be written, and code generation improved.

iii) a basis for program analysis and synthesis.

- Transformation.
- Optimisation.
- Verification.

It can be that actually showing a program to be correct is more work than writing the program itself, but it is not a task to be neglected. History has many examples of (large) programs that went horribly wrong, and the need for a formal semantics comes from the desire to be able to guarantee that a program does exactly what it is supposed to do, under all circumstances, not just in the testing environment.

1.1 States

Semantics of programming languages deals with the meaning of programs that are supposed to be executing on a computer, i.e. run in memory and use the various resources available. Therefore, to express execution correctly, we need also to consider the status of the memory during execution. Not only because the contents of the memory changes (variables get updated, data bases change), but mostly because the content of a variable *at the moment the program starts* can have a major effect on the end result.

The *state* of the memory s is therefore prominent in all definitions of semantics. However, normally only part of the memory is relevant to the semantics of a program. Moreover, since we will consider only programs that compute through variables[1], we are not interested in the contents of actual physical addresses, but will abstract from the actual memory and focus on the representation of the values stored in those variables that are of interest to us. We will, formally, say that a state is a function that maps variable names to values, and use *State* for the set of all possible states.

We will use the notation

$$[x_1 \mapsto v_1, x_2 \mapsto v_2, \ldots]$$

as a denotation for the state of the memory, indicating the values for those variable that are relevant to the execution of the program.

Globally speaking, there are three approaches to semantics:

(*Operational Semantics*) : This semantics is characterised by the fact that it focuses on how the effect of a computation is produced: it is an abstraction of machine execution in that it expresses the meaning of a program - running on a machine in a specific state - by returning its result, the output.

(*Denotational Semantics*) : This semantics focusses on what is the effect of a computation on the state of the machine. In this approach, the meaning of a program (construct) is a function, that maps the state of the machine before execution to the state after execution.

(*Axiomatic Semantics*) : With this semantics, the properties of the effect of executing the constructs are expressed as assertions.

We will compare these approaches using the (toy) example program

$$z \texttt{ := } x \texttt{;} x \texttt{ := } y \texttt{;} y \texttt{ := } z$$

that swaps the values stored in the variables x and y.

1.2 Operational Semantics

The idea behind Operational Semantics is to express the meaning of a program *starting from a certain state by looking at its end result*, i.e. the state in which the memory is after execution of the program. For example,

- To execute a sequence of statements separated by '`;`', execute the individual statements one after the other from left to right.

[1]It is not difficult to extend this to the more general approach, but this would increase the quantity of definition and notation significantly. We are concentrating on the essentials here.

- To execute 'x := y', determine the value of y and assign it to x.

We use the notation '$\langle p, s\rangle$', that is to be read as 'the semantics of program p in state s'.

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

So, before starting the program the state of the memory is $[x \mapsto 5, y \mapsto 7, z \mapsto 0]$, and after the program has finished it is $[x \mapsto 7, y \mapsto 5, z \mapsto 5]$, which is also the *meaning* of the program

$$z \texttt{ := } x \texttt{ ; } x \texttt{ := } y \texttt{ ; } y \texttt{ := } z$$

in the state $[x \mapsto 5, y \mapsto 7, z \mapsto 0]$. The symbol '$\Rightarrow$' is used for '*evaluates to*' or '*means*'. So, for operational semantics, you can only look at a program with a given input or initial state, you cannot say anything about a program alone.

1.3 Denotational Semantics

The idea behind this semantics is to look at a program as a *mathematical function*, i.e. the effect of a program is a mathematical function in *State* $\rightarrow$ *State*.

- The effect of a sequence of statements separated by '**;**' is the functional composition of the effects of the individual statements:

$$\mathcal{S}_{ds}[\![s_1 \texttt{ ; } s_2]\!] = \mathcal{S}_{ds}[\![s_2]\!] \circ \mathcal{S}_{ds}[\![s_1]\!]$$

(Notice the inversion of s_1 and s_2.)

- The effect of a statement 'x := y' is a function in *State* $\rightarrow$ *State*, such that the new state is identical to the old state except that the (new) value of x is equal to the (old) value of y.

$$\begin{array}{rcll}
\mathcal{S}_{ds}[\![x \texttt{ := } v]\!]\, s\, y & = & s\, y, & \text{if } y \neq x \\
& = & v, & \text{otherwise}
\end{array}$$

For our example program, notice that

$$\mathcal{S}_{ds}[\![z \texttt{ := } x \texttt{ ; } x \texttt{ := } y \texttt{ ; } y \texttt{ := } z]\!] = \mathcal{S}_{ds}[\![y \texttt{ := } z]\!] \circ \mathcal{S}_{ds}[\![x \texttt{ := } y]\!] \circ \mathcal{S}_{ds}[\![z \texttt{ := } x]\!]$$

and, therefore

$$\begin{array}{rl}
\mathcal{S}_{ds}[\![z \texttt{ := } x \texttt{ ; } x \texttt{ := } y \texttt{ ; } y \texttt{ := } z]\!]\,([x \mapsto 5, y \mapsto 7, z \mapsto 0]) & = \\
\mathcal{S}_{ds}[\![y \texttt{ := } z]\!] \circ \mathcal{S}_{ds}[\![x \texttt{ := } y]\!] \circ \mathcal{S}_{ds}[\![z \texttt{ := } x]\!]\,([x \mapsto 5, y \mapsto 7, z \mapsto 0]) & = \\
\mathcal{S}_{ds}[\![y \texttt{ := } z]\!] \circ \mathcal{S}_{ds}[\![x \texttt{ := } y]\!]\,([x \mapsto 5, y \mapsto 7, z \mapsto 5]) & = \\
\mathcal{S}_{ds}[\![y \texttt{ := } z]\!]\,([x \mapsto 7, y \mapsto 7, z \mapsto 5]) & = \\
[x \mapsto 7, y \mapsto 5, z \mapsto 5] &
\end{array}$$

For denotational semantics, the meaning of a program depends only on the pogram itself. No state information is needed to establish a meaning.

1.4 Axiomatic Semantics

Axiomatic Semantics deals with the *partial correctness* (with respect to *Pre-* and *Post-condition*) of a given program.

Take the statement

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

that expresses that *if* $x = n$ and $y = m$, then $x = m$ and $y = n$ *after* the execution of the program '$z \texttt{ := } x \texttt{;} x \texttt{ := } y \texttt{;} y \texttt{ := } z$'.

Let

$(P1)$: $\{x = n \;\&\; y = m\}\; z \texttt{ := } x\; \{z = n \;\&\; y = m\}$

$(P2)$: $\{z = n \;\&\; y = m\}\; x \texttt{ := } y\; \{z = n \;\&\; x = m\}$

$(P3)$: $\{x = n \;\&\; y = m\}\; z \texttt{ := } x \texttt{;} x \texttt{ := } y\; \{z = n \;\&\; x = m\}$

$(P4)$: $\{z = n \;\&\; x = m\}\; y \texttt{ := } z\; \{y = n \;\&\; x = m\}$

$(P5)$: $\{x = n \;\&\; y = m\}\; z \texttt{ := } x \texttt{;} x \texttt{ := } y \texttt{;} y \texttt{ := } z\; \{y = n \;\&\; x = m\}$

Then it is easy to see that we can deduce **P3** from **P1** and **P2**, and **P5** from **P3** and **P4**, so for this kind of program the axiomatic approach works well.

But how to work on

$$\{x = n \;\&\; y = m\}\; \texttt{while true do skip}\; \{y = n \;\&\; x = m\}$$

is not easy to see.

2 Induction

Many of the definitions and properties we are about to study depend heavily on induction. Not only is the majority of our definitions inductive in nature, also most of the proofs are inductive. Since the kind of induction we use is of a more general nature than just induction over numbers, before discussing the real topic of this course, we have a close look at the principle of induction.

2.1 Mathematical Induction

The principle of Mathematical Induction over the set $\mathbb{N}$ of natural numbers states: to prove a property $P(x)$ for all natural numbers if suffices to show:

$(\textit{Base case})$: Prove $P(0)$.

$(\textit{Inductive Case})$: For every k, using the assumption that $P(k)$ holds, prove $P(k+1)$; in other words: prove $P(k) \to P(k+1)$.

These two proofs give you the ‘right’ to say that $P(n)$ holds for all n.

In Logic the principle of induction over $\mathbb{N}$ is formalised as follows:

$$\frac{P(0) \quad \forall k \in \mathbb{N}\, [P(k) \to P(k+1)]}{\forall n \in \mathbb{N}.P(n)}$$

(It should be noted that the Principle of Induction

$$\forall P.[\ (\ P(0)\ \&\ \forall k \in \mathbb{N}\, [P(k) \to P(k+1)]\ ) \to \forall n \in \mathbb{N}.P(n)\ ]$$

is not a theorem, i.e. it does not get a proof in mathematics. If anything, it is an axiom, that is assumed to be true. Stating that it is a *principle* is better, though, because it can be extended to all inductively definable structures, as we will see shortly.)

Theorem 2.1 $\Sigma_{i=0}^{n}\, i^2 = (n \times (n+1) \times (2n+1))/6$.

Proof : By induction on the structure of $\mathbb{N}$.

(*Base case*) : Immediate, since $0 = (0 \times 1 \times 1)/6$.

(*Inductive Case*) :

$$\begin{array}{rl}
\Sigma_{i=0}^{k+1}\, i^2 & = \\
(\Sigma_{i=0}^{k}\, i^2) + (k+1)^2 & = (\mathit{IH}) \\
(k \times (k+1) \times (2k+1))/6 + (k+1)^2 & = \\
(k \times (k+1) \times (2k+1) + 6(k+1)^2)/6 & = \\
(2k^3 + 3k^2 + k + 6k^2 + 12k + 6)/6 & = \\
((k+1) \times (k+2) \times (2k+3))/6 &
\end{array}$$

Notice that in the second part of the proof, we are proving

$$\Sigma_{i=0}^{k+1}\, i^2 = ((k+1) \times (k+2) \times (2k+3))/6$$

and that there we use the hypothesis (i.e., assume that it is true without checking)

$$\Sigma_{i=0}^{k}\, i^2 = (k \times (k+1) \times (2k+1))/6,$$

so that formally we prove that the first is *implied by* the second.

Exercise 2.2 Prove that, for any natural number n, that there are exactly $n!$ permutations of n objects.

2.2 Why does this work?

The principle of induction is an accepted proof method, because you can ‘see’ that it works. If you need to show that $\forall n \in \mathbb{N}.P(n)$ holds, then you could just produce a new proof for each new number given. But, of course, it would be more constructive to have a method that, for every $n \in \mathbb{N}$, shows that $P(n)$ holds. Normally there are various methods, sometimes you can even show $P(n)$ directly (like, for example, for the statement: $n \in \mathbb{N} \to n{+}1 \in \mathbb{N}$). But if you would prove $P(0)$, and, for very $k \in \mathbb{N}$, you have a method to extend a proof for $P(k)$ to one for $P(k{+}1)$ (the inductive step), then, if asked for a proof for $P(n)$, it would suffice to give the proof for $P(0)$ and extend it n times. In a certain sense, if you automate the method that builds the extension of the proof of $P(k)$ to $P(k{+}1)$, then you could build a system, that, given the number n, produces the proof for $P(n)$ from the proof for $P(0)$.

The secret is twofold. First of all, in the inductive step we are only interested in the proof that we can *extend* a proof of $P(k)$ to one for $P(k{+}1)$. If $P(k)$ itself is true is of no concern, the only thing that is of interest is:

$$\underline{\textit{if}}\ P(k),\ \underline{\textit{then}}\ P(k{+}1).$$

This means that we are looking to prove the *implication*, and we are allowed to *assume* that $P(k)$ holds. If from this assumption we show $P(k{+}1)$, we have shown the implication to hold. Normally, inserting the assumption in your proof structure is called 'by induction' (and *not* 'by induction hypothesis'), and highlights the fact that you use the liberty to assume that $P(k)$ holds.

Secret number two lies in the fact that you *can* do it for *every* $n \in \mathbb{N}$, since the set $\mathbb{N}$ of natural numbers satisfies:

- $0 \in \mathbb{N}$
- if $x \in \mathbb{N}$, then $x + 1 \in \mathbb{N}$.

and $\mathbb{N}$ is the *least* set with both of these properties, so the method sketched above does not miss out on certain numbers.

Usually, one would write '*Proof: by induction on n*', but the correct formulation is '*Proof: by induction on the structure of natural numbers*'.

That $\mathbb{N}$ is indeed the smallest set that satisfies these criteria, is shown by the following:

Theorem 2.3 *Suppose X satisfies the properties*
- $0 \in X$
- *if $x \in X$, then $x + 1 \in X$.*

then $\mathbb{N} \subseteq X$.

Proof : We will show: $\forall n \in \mathbb{N}. n \in X$, by mathematical induction over $\mathbb{N}$. Take $k \in \mathbb{N}$. From the definition of $\mathbb{N}$, either $k = 0$, or $k = k' + 1$, with $k' \in \mathbb{N}$. To show: $k \in X$.
$(k = 0)$: By definition of X, we have $0 \in X$.
$(k = k' + 1)$: Since $k' \in \mathbb{N}$, by induction also $k' \in X$. Then, by definition of X, also $k + 1 \in X$.
So, for all $k \in \mathbb{N}$, $k \in X$.

Exercise 2.4 Prove that, for every even number n, $n \times k$ is even for any natural number k.

Not every (correct) statement over numbers is proved by induction:

Theorem 2.5 *There are infinitely many prime numbers.*

Proof : Take $\mathcal{P}$ to be the set of prime numbers, and write $\#\mathcal{P}$ for its size. We will show that $\#\mathcal{P}$ is not finite, by showing that there is no n such that $\#\mathcal{P} = n$.

So, suppose $\#\mathcal{P} = n$, and let $\mathcal{P} = \{p_1, \ldots, p_n\}$. Define $m = (p_1 \times \cdots \times p_n) + 1$. Then either m is prime, and obviously $m \not\in \mathcal{P}$, or there is a prime number p' and number q such that $m = p' \times q$. Now the question is: is $p' \in \mathcal{P}$? Suppose it is, so $p' = p_i$, for some $1 \leq i \leq n$. Since $p'_i \mid m$, also $p_i \mid (p_1 \times \cdots \times p_n) + 1$. Obviously $p_i \mid p_1 \times \cdots \times p_n$, so also $p_i \mid 1$. This is impossible, so $p' \not\in \mathcal{P}$.

So there is no n such that $\#\mathcal{P} = n$, so $\#\mathcal{P}$ is infinite.

2.3 Complete Induction

An alternative to the rule for induction is the principle of Complete Induction (*course of values*):

To prove a property $P(x)$ for all natural numbers:

(*Base case*) : Prove $P(0)$.

(*Inductive Case*) : For every k, on the assumption that $P(i)$ holds for every i smaller than or equal to k, prove $P(k{+}1)$; in other words: prove $(\forall i{\leq}k \,.\, P(i)) \to P(k{+}1)$.

Also these two proofs give you the 'right' to say that $P(n)$ holds for all n.

In Logic:

$$\frac{P(0) \quad \forall k \,.\, [(\forall i{\leq}k \,.\, P(i)) \to P(k{+}1)]}{\forall n.P(n)}$$

Theorem 2.6 *These two principles of induction coincide, i.e. accepting one principle you can show the other holds, and vice versa.*

Proof : The proof has two parts. First we show that Complete Induction implies Mathematical Induction, and then we show the reverse.

- Let P be a property, and assume

$$P(0) \tag{1}$$
$$\forall k \,.\, P(k) \to P(k{+}1) \tag{2}$$

  We have to show that $\forall n.P(n)$. We can reach this result using Complete Induction, but then we need to show first that

$$\forall k \,.\, [(\forall i{\leq}k \,.\, P(i)) \to P(k{+}1)] \tag{3}$$

  For every k, this is an implication, that is shown as follows: assume $\forall i{\leq}k \,.\, P(i)$, then, in particular, $P(k)$, and, by assumption (2), we have $P(k{+}1)$. So $(\forall i{\leq}k \,.\, P(i)) \to P(k{+}1)$, for every k, so we have $\forall k.[(\forall i{\leq}k \,.\, P(i)) \to P(k{+}1)]$. Since also $P(0)$ by (1), we get $\forall n.P(n)$.

- Let P be such that

$$P(0) \tag{4}$$
$$\forall k \,.\, [(\forall i{\leq}k \,.\, P(i)) \to P(k{+}1)] \tag{5}$$

  We have to show $\forall n.P(n)$. We can reach this result using Mathematical Induction, but then we need to show first that

$$\forall k.[P(k) \to P(k{+}1)] \tag{6}$$

  Let $Q(n)$ be the property defined by $\forall j{\leq}n.P(j)$. We will show $\forall n.Q(n)$, using Mathematical Induction, so we will show $\forall n.[\forall j{\leq}n.P(j)]$, so, in particular, $\forall n.P(n)$.

  (*Base case*) : $Q(0) = \forall j{\leq}0.P(j) = P(0)$, so $Q(0)$ holds by assumption (4).

  (*Inductive case*) : To prove $Q(k) \to Q(k{+}1)$, for all k, first assume $Q(k) = \forall j{\leq}k \,.\, P(j)$.
  Then, of course, also $\forall j{<}k{+}1.P(j)$, so, by assumption (5), $P(k{+}1)$, and, therefore, $\forall j{\leq}k{+}1.P(j) = Q(k{+}1)$.

  So $\forall n.Q(n)$.

2.4 Other induction

In general, we will define many sets, relations, ..., as the *least* ones satisfying a set of conditions or rules.

Example 2.7 *Ev*, the set of even numbers, is the least set such that:

- $0 \in Ev$.
- if $k \in Ev$, then $k+2 \in Ev$.

When looking to prove a property for all $n \in Ev$, we can, as much as for the set $\mathbb{N}$, use induction. The principle of induction for *Ev* gets formulated as follows: to prove $P(n)$ for all $n \in Ev$, you

(*Base case*) : Prove $P(0)$.
(*Inductive Case*) : For every k, using the assumption that $P(k)$ holds, prove $P(k+2)$.

Example 2.8 If $n \in Ev$ and $m \in Ev$, then $n+m \in Ev$.

Proof : Take $P(x)$ to be $\forall m \in Ev.[x+m \in Ev]$. We will show $P(x)$ by induction on the structure of *Ev*.

(*Base case*) : $\forall m \in Ev.[0+m \in Ev]$. Immediate.
(*Inductive Case*) : To show: $\forall m \in Ev.[(x+2)+m \in Ev]$. By induction, $\forall m \in Ev.[x+m \in Ev]$. Then, by the second part of the definition of *Ev*, also $\forall m \in Ev.[(x+m)+2 \in Ev]$. Since $(x+m)+2 = (x+2)+m$ by associativity of $+$, we are done.

So we have shown $\forall x \in Ev.P(x)$, which is $\forall n \in Ev.\forall m \in Ev.[n+m \in Ev]$.

2.5 Structural induction

Our definitions often define sets of objects which are syntactic or have significant structure, in the sense that they are defined *inductively*. A 'proof by induction' over such a set is then known as a 'proof by structural induction.' Take for example the definition of lists over natural numbers, $List(\mathbb{N})$.

- $[\,] \in List(\mathbb{N})$.
- If $n \in \mathbb{N}$, and $l \in List(\mathbb{N})$, then $n : l \in List(\mathbb{N})$.

Notice that this definition is inductive. Implicit in this definition is that ':' is a list-constructor that, given a number and a list of numbers, produces a list of numbers.

A very common presentation of the inductive definition above is to use *rules*:

$$\frac{\textit{premises}}{\textit{conclusions}}$$

with the intended meaning:

if all the *premises* hold, then so do the *conclusions*.

Using rules, the definition of $List(\mathbb{N})$ then becomes

$$\frac{}{[\,] \in List(\mathbb{N})} \qquad \frac{n \in \mathbb{N} \quad l \in List(\mathbb{N})}{n : l \in List(\mathbb{N})}$$

Using the convention that n is always in $\mathbb{N}$, we can omit the premises $n \in \mathbb{N}$.

In this case, an inductive proof that the property $P(l)$ holds, for every $l \in$ *List*$(\mathbb{N})$, would follow:

$($*Base case*$)$: Prove $P([\,])$.

$($*Inductive Case*$)$: Assuming $P(l)$, for every $n \in \mathbb{N}$, prove $P(n : l)$.

Example 2.9 Take the 'Miranda' program

```
maximum [ ]    = 0
maximum (a:x) = a, a >= n
                n
                where n = maximum x

length [ ]     = 0
length (a:x)   = 1 + length x

sum [ ]        = 0
sum (a:x)      = a + sum x
```

then, using the definition of *List*$(\mathbb{N})$ given before, we can prove: let

$$P(l) = \texttt{sum}\, l \leq \texttt{maximum}\, l \times \texttt{length}\, l,$$

then for all $l \in$ *List*$(\mathbb{N})$, $P(l)$.

$($*Base case*$)$: $P([\,])$ is trivial, since **sum** [ ] = 0 .

$($*Inductive Case*$)$:

$$\begin{array}{rl} \texttt{sum}\,(n : l) & = \\ n + \texttt{sum}\, l & \leq \\ n + \texttt{maximum}\, l \times \texttt{length}\, l & \leq \\ \texttt{maximum}\,(n : l) + \texttt{maximum}\,(n : l) \times \texttt{length}\, l & = \\ \texttt{maximum}\,(n : l) \times \texttt{length}\,(n : l) & \end{array}$$

2.6 The general case

As mentioned before, the principle of induction extends to every 'inductive' structure, i.e. to every set X defined in terms of

$($*Base case*$)$: Some *constants* $a_1, \ldots, a_m$ are assumed to be in X.

$($*Inductive Case*$)$: There is a limited number of *constructors* $C_1, \ldots, C_m$, that, given a number of elements of X, produce another element of X:

$$\begin{array}{c} \text{if } t_1, \ldots, t_{n_1} \in X, \text{ then } C_1(t_1, \ldots, t_{n_1}) \in X. \\ \ldots \\ \text{if } t_1, \ldots, t_{n_m} \in X, \text{ then } C_m(t_1, \ldots, t_{n_m}) \in X. \end{array}$$

$($*Closure*$)$: X is defined as the smallest set satisfying the above two rules.

It is common to not separate *constants* from *constructors*, but rather to treat constants as constructors of *arity* zero, and deal with both constants and constructors in *one* go. We will use both expressions, whenever convenient.

Because of the third rule, the general form of structural induction states that to prove $P(x)$ for all elements $x \in X$, it is sufficient to:

$($*Base case*$)$: Prove $P(a_1)$ up to $P(a_m)$.

$($*Inductive Case*$)$: For every constructor C_i, assuming that $P(t_1)$ up to $P(t_{n_i})$, prove that also $P(C_i(t_1,\ldots,t_{n_i}))$.

Example 2.10

$$E \quad ::= \quad \texttt{zero} \mid E_1 \times E_2 \mid (E)$$

or, alternatively

$$\frac{}{\texttt{zero} \in \mathit{Exp}} \qquad \frac{E_1 \in \mathit{Exp} \quad E_2 \in \mathit{Exp}}{E_1 \times E_2 \in \mathit{Exp}} \qquad \frac{E \in \mathit{Exp}}{(E) \in \mathit{Exp}}$$

Constructors in this case are '$\cdot \times \cdot$' and '$(\cdot)$', and '**`zero`**' is a constant.

Property 2.11 All elements of Exp have the same number of left parentheses and right parentheses.

Proof : Induction on the structure of elements of *Exp*.
$($*Base case*$)$: The number of parentheses in **`zero`** is 0.
$($*Inductive Case 1*$)$: $E = E_1 \times E_2$, and by induction we can assume the property to hold for both E_1 and E_2. Let the number of left (and right) parentheses in E_1 be n_1, and that in E_2 be n_2. Then E has $n_1 + n_2$ left and $n_1 + n_2$ right parentheses, so the number of left - and right parentheses in E is equal.
$($*Inductive Case 2*$)$: $E = (E_1)$, and by induction we can assume the property to hold for E_1. Let the number of left (and right) parentheses in E_1 be n, then E has $n + 1$ left (right) parentheses, so the number of left - and right parentheses in E is equal.

We occasionally need to do a proof by structural induction over a number of domains simultaneously, like

$$\begin{array}{lcl} S & ::= & *E* \\ E & ::= & +S \mid ** \end{array}$$

Exercise 2.12 All S-values have an even number of occurrences of the $*$-token.

We have informally used *rules* in our inductive definitions:

$$\frac{\textit{premises}}{\textit{conclusions}}$$

With the intended meaning of:

if *premises*, then *conclusions.*

But since paper is patient, care is needed. What set is defined by:

$$\frac{n \in X}{n+3 \in X}$$

or:

$$\frac{}{0 \in X} \qquad \frac{n \in X}{n+2 \in X} \qquad \frac{n+2 \in X}{n \notin X}$$

We can define many relations using rules.

In fact, every 'object' that is defined in terms of 'if … then … ' can be formulated using rules. They are a very clean and precise notation for definitions that would otherwise take up a lot of words. However, one important thing to note is that, using systems defined by *rules*, the *nature* of the objects we work with changes. When we want to prove properties of a system that is defined only in terms of '*if … then …* ', we would need to follow that structure. When using rules, however, the objects over which you prove become *trees* constructed by applying the rules, also called *derivations*.

2.7 How to use rules

We will now focus on the particular details involved with using rules to define inductive sets.

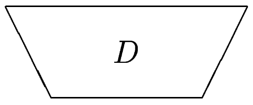
As an example, we take the definition of (Implicative) Logic (IL), where we derive statements of the shape $\Gamma \vdash A$, whose intention is 'Γ *shows* A', or '*from* Γ *we can deduce* A'.

$$(Ax)\ \frac{}{\Gamma \vdash A}\,(A \in \Gamma) \qquad (\to I)\ \frac{\Gamma \cup \{A\} \vdash B}{\Gamma \vdash A \to B} \qquad (\to E)\ \frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

How to read these three rules? They describe how to *build derivations*. The first, *Ax*, states that, for any set Γ of formulae, if A occurs in Γ, then

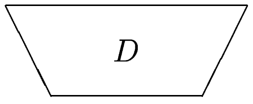
$$\frac{}{\Gamma \vdash A}\,(A \in \Gamma)$$

is a correct derivation. Moreover, suppose we have constructed a derivation D that *ends with* (or *derived*) $\Gamma \cup \{A\} \vdash B$.



$$\Gamma \cup \{A\} \vdash B$$

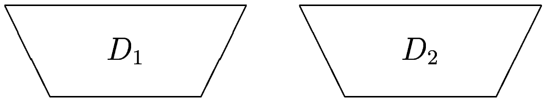
then by the second rule $(\to I)$, we will create a correct derivation when we draw a line underneath it, and put the line $\Gamma \vdash A \to B$ under it all.



$$\frac{\Gamma \cup \{A\} \vdash B}{\Gamma \vdash A \to B}$$

(Notice that A has moved from left to right.)

And, similarly, if we have two separate derivations D_1 and D_2, one with last line $\Gamma \vdash A \to B$, the other with last line $\Gamma \vdash A$, then by the last rule $(\to E)$, when we draw a line underneath them both, and put the line $\Gamma \vdash B$ under it all, we obtain a correct derivation.



$$\frac{\Gamma \vdash A{\to}B \qquad\qquad \Gamma \vdash A}{\Gamma \vdash B}$$

These three steps are the only permitted to construct derivations; in other words: the *set of derivations for* IL is the smallest set closed for the three *derivation constructors*, the rules (Ax), $(\to I)$, and $(\to E)$ given above.

Officially, the correct denotation for objects defined by these rules would be $D :: \Gamma \vdash A$, meaning that D is a derivation with last line $\Gamma \vdash A$. However, it is more common to use *only* the expression $\Gamma \vdash A$ when speaking of objects in IL. This then is meant to say that

There exists a derivation built using the three rules above, that ends with $\Gamma \vdash A$.

since, normally, we are not interested in the actual structure of the derivation showing $\Gamma \vdash A$, but only in the fact that the formula is derivable.

However, when you are aiming to prove *properties* of IL inductively, the actual structure of derivations becomes important. In fact, any (suitable) property over expressions in IL is actually a property over *derivations* and can normally be proven by *induction on the structure of derivations*. Therefore, globally, the structure of the proof would be the same as discussed above, i.e. first prove the property to hold for the base cases, and then for the inductive steps.

We can present the proof in a slightly different way, by giving it the following structure.

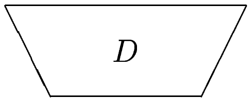
Proof: By induction on the structure of derivations. We focus on the last rule used.

(Ax) : Here the derivation is nothing but an application of rule *Ax*.

$$\frac{}{\Gamma \vdash A}\ (A \in \Gamma)$$

This is the base case of the induction, and you need to show directly that the property to prove holds for this derivation.

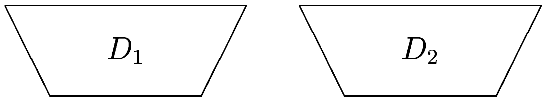
$(\to I)$: In this case, we have a derivation D' of the structure



$$\frac{\Gamma \cup \{A\} \vdash B}{\Gamma \vdash A{\to}B}$$

The derivation D with conclusion $\Gamma \cup \{A\} \vdash B$ is a subderivation of D' for $\Gamma \vdash A{\to}B$, and so is *smaller* in the sense of the inductive structure. Now we can assume that the property to prove holds for D, and use that to prove that the property holds for D'.

$(\to E)$: In this case, we have a derivation D' of the structure



$$\frac{\Gamma \vdash A{\to}B \qquad\qquad \Gamma \vdash A}{\Gamma \vdash B}$$

Again, the derivations D_1 and D_2 are subderivations of D', and we can assume that the property to prove holds for both, and use that to prove that the property holds for D'.

In each case, we can use the fact that we know what is the last rule applied, which gives us not only existence of the subderivations, but also all other properties that are specified in the rule.

Notice that this is, in approach, sufficient for a complete inductive proof. The base case is covered by the first part, since, when the last applied rule is *Ax*, this is exactly the base case of the definition of derivations in IL. The other two cases deal with the constructors, the rules that show how to construct new derivations from those that already exist.

Exercise 2.13
- The *height* of a derivation D, $\textit{height}\,(D)$, is inductively defined by (we focus on the last rule applied):

  $(\textit{Ax})$: Then $D :: \Gamma \vdash A$, where $A \in \Gamma$, and $\textit{height}\,(D) = 1$.

  $(\rightarrow I)$: Then $D :: \Gamma \vdash A \rightarrow B$, and D has a sub-derivation $D' :: \Gamma \cup \{A\} \vdash B$. Then $\textit{height}\,(D) = \textit{height}\,(D') + 1$.

  $(\rightarrow E)$: Then $D :: \Gamma \vdash B$ has two sub-derivations $D_1 :: \Gamma \vdash A \rightarrow B$ and $D_2 :: \Gamma \vdash A$. Then $\textit{height}\,(D) = \textit{max}\,(\textit{height}\,(D_1), \textit{height}\,(D_2)) + 1$.
- The *complexity* of a derivation D, $\textit{comp}\,(D)$, is inductively defined by (we focus on the last rule applied):

  $(\textit{Ax})$: Then $D :: \Gamma \vdash A$, where $A \in \Gamma$, and $\textit{comp}\,(D) = 2$.

  $(\rightarrow I)$: Then $D :: \Gamma \vdash A \rightarrow B$, and D has a sub-derivation $D' :: \Gamma \cup \{A\} \vdash B$. Then $\textit{comp}\,(D) = \textit{comp}\,(D') + 1$.

  $(\rightarrow E)$: Then $D :: \Gamma \vdash B$ has two sub-derivations $D_1 :: \Gamma \vdash A \rightarrow B$ and $D_2 :: \Gamma \vdash A$. Then $\textit{comp}\,(D) = \textit{comp}\,(D_1) + \textit{comp}\,(D_2)$.

Show that, for all derivations, $\textit{height}(D) < \textit{comp}(D)$.

2.8 (Apparent) Static and Reverse Induction

The observations made above can become crucial when trying to prove properties over systems defined with rules.

Take, for example, the (not so very exciting) system defined by:

$$\frac{}{A \sim B} \qquad \frac{}{N \sim N} \qquad \frac{M \sim N}{N \sim M} \qquad \frac{N \sim M \quad M \sim P}{N \sim P}$$

Again, writing $M \sim N$ means that there exists a derivation that has that formula in the bottom line.

Exercise 2.14 Show: If $M \sim N$, then either $M \equiv N$, or $M \equiv A$ and $N \equiv B$, or $M \equiv B$ and $N \equiv A$.

In this exercise, the only point of difficulty could be the third rule.

Exercise 2.15 Let X be defined by:

$$\frac{}{0 \in X} \qquad \frac{n \in X}{n+3 \in X} \qquad \frac{n \in X}{n-5 \in X}\,(n > 5)$$

Show that $\mathbb{N} \subseteq X$.

2.9 Alternatives

Using the definition of 'height of a derivation' suggested above, the '*proof by induction on the structure of derivations*' could be replaced by a '*proof by induction on the height of a derivation*'. Notice that then you would need the 'course of values' variant of the principle of induction for this technique, since, in general, a sub-derivation does not have a height that is precisely 1 smaller (see, for example, rule $(\to E)$).

Sometimes the structure we deal with has the special property that all rules have exactly *one* premise. Then a '*proof by induction on the structure of derivations*' could become '*proof by induction on the length of the derivation*'.

Because of these two observations, you can argue that *any* kind of induction is essentially induction on $\mathbb{N}$. It is the readability of proof, together with the specific objects you prove over, that decide which appearance the inductive proof will have.

3 Syntax

We will present our semantics in terms of the *Abstract* syntax of a programming language. This in contrast to normal programming languages that are defined in the *concrete* syntax; the point is that details that are important to programmers, like how to exactly write a number, can be ignored for our purposes.

3.1 Concrete syntax

The *Concrete* syntax defines the sequences of symbols allowable in a syntactically correct program:

$$\begin{array}{rcl}
\langle\texttt{exp}\rangle & ::= & \langle\texttt{num}\rangle \mid \langle\texttt{exp}\rangle\langle\texttt{op}\rangle\langle\texttt{exp}\rangle \\
\langle\texttt{op}\rangle & ::= & + \mid - \mid \times \mid / \\
\langle\texttt{num}\rangle & ::= & \langle\texttt{digit}\rangle \mid \langle\texttt{digit}\rangle\langle\texttt{num}\rangle \\
\langle\texttt{digit}\rangle & ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 0
\end{array}$$

This syntax gives a sufficient specification of numbers. Notice, however, that although it allows us to parse all permitted expressions, this particular grammar is not precise enough, since it is *ambiguous:* when *interpreting* the expressions to understand what value they stand for, i.e. to understand their semantics, it is not clear what the intended *precedence* of the operations is in, for example, the expression $4 \times 2 - 1$, and neither the *associativity* of the operations is clear.

Instead, take

$$\begin{array}{rcl}
\langle\texttt{exp}\rangle & ::= & \langle\texttt{num}\rangle \mid (\langle\texttt{exp}\rangle\langle\texttt{op}\rangle\langle\texttt{exp}\rangle) \\
\langle\texttt{op}\rangle & ::= & + \mid - \mid \times \mid / \\
\langle\texttt{num}\rangle & ::= & \langle\texttt{digit}\rangle \mid \langle\texttt{digit}\rangle\langle\texttt{num}\rangle \\
\langle\texttt{digit}\rangle & ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 0
\end{array}$$

The bracketing now forces the computation, and it will be easier to define a correct semantics.

Exercise 3.1 Consider the following Abstract Syntax:

$$\begin{array}{rcl}
n & \in & \textit{Numeral} \\
x & \in & \textit{Variables} \\
a & \in & \textit{Arithmetic Expressions} \\
b & \in & \textit{Boolean Expressions} \\
p & \in & \textit{Procedure Names} \\
D_v & \in & \textit{Declared Variables} \\
D_p & \in & \textit{Declared Procedures} \\
S & \in & \textit{Statements}
\end{array}$$

$$\begin{array}{rcl}
a & ::= & n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \\
b & ::= & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \;\&\; b_2 \\
D_v & ::= & \texttt{var} x \;\texttt{:=}\; a \,\texttt{;}\, D_v \mid \epsilon \\
D_p & ::= & \texttt{proc}\, p \,\texttt{is}\, S \,\texttt{;}\, D_p \mid \epsilon \\
S & ::= & x \;\texttt{:=}\; a \mid \texttt{skip} \mid S_1 \,\texttt{;}\, S_2 \mid \texttt{if}\, b \,\texttt{then}\, S_1 \,\texttt{else}\, S_2 \\
 & & \mid \texttt{while}\, b \,\texttt{do}\, S \mid \texttt{begin}\, D_v\; D_p\; S \,\texttt{end} \mid \texttt{call}\, p
\end{array}$$

Show that any program in the language has the same number of **`begin`** and **`end`** tokens.

3.2 Abstract syntax

The *Abstract* syntax formalises the *Allowable Parse Trees*.

For the previous syntax for numbers, the abstract syntax specifies the syntactic categories

$$\begin{array}{rcl}
e & \in & \textit{exp} \\
\texttt{op} & \in & \textit{Op} \\
n & \in & \textit{Numeral}
\end{array}$$

and the definitions

$$\begin{array}{rcl}
\texttt{op} & ::= & + \mid - \mid \times \mid / \\
e & ::= & n \mid (e_1 \;\texttt{op}\; e_2)
\end{array}$$

(The second line should be read as: ‘An expression is either a number, or composed of two expressions with an operation in between them’.)

These two (syntactic categories and definitions) together are sufficiently precise for our purposes. Notice that the main difference between the concrete and the abstract syntax is that in the latter we leave the syntactic construction of numbers (elements of *Numeral*) unspecified. In this way, we abstract from their actual syntax (hence the name *abstract* syntax). We also assume that our alphabet of symbols is infinite: n is used as a meta-variable, a place holder for *any* number, and we have no rule specifying the syntactic structure of numbers, nor any information on the set *Numeral*.

4 Natural Semantics

We will now start concentrating on the semantics of a (toy) programming language, `While`. The first semantics we will study is called the *Natural Semantics*; it caries that name because it is concentrating on the 'result' of a program, which is the natural thing to study. We will introduce this semantics in stages. First we define Natural Semantics of Arithmetic Expressions, which gives us the possibility to get acquainted with notation and do some elementary proofs, before focusing on a real language.

The syntactic categories used for the definition of arithmetic expressions are:

$$\begin{array}{lcl} a & \in & \textit{Arithmetic Expressions} \\ n & \in & \textit{Numeral} \\ x & \in & \textit{Variables} \end{array}$$

where *Variables* is supposed to be an infinite set of variable names. The definitions are:

$$a \quad ::= \quad n \mid x \mid (a_1 + a_2) \mid (a_1 - a_2) \mid (a_1 \times a_2)$$

4.1 Natural Semantics of Arithmetic Expressions

Natural Semantics of *Arithmetic Expressions* is defined as a transition system over the following two kinds of configurations:

$$\begin{array}{rcl} \langle a, s \rangle & - & \textit{Arithmetic Expressions} \text{ and } \textit{state} \\ v & - & \textit{an integer} \text{ (in } \mathbb{Z}\text{, the final state)} \end{array}$$

Notice that we consider *two* sets of 'numbers' in this course.

- *Numeral*, that stands for the syntactic representation of numbers (without specifying *how* numbers are written).

- $\mathbb{Z}$, the set of integers that are the actual numbers, or *values*, syntactically represented by elements of *Numeral*.

Of course, when writing down an element of $\mathbb{Z}$, we need to use some syntax. For example, the forty-second positive number can be represented in different ways, depending the chosen syntax. One could have $42_{10} = 101010_2 = 222_4 = 52_8 = 2A_{16} = 10_{42}$. We will use the normal decimal notation for both, and to distinguish between *our* syntax and that of elements in *Numeral*, we will underline if necessary. So, $4 \in$ *Numeral* is a syntactic representation of the fourth number, and $\underline{4} \in \mathbb{Z}$ is its actual value. Sometimes, alternatively, we will use $\mathcal{N}[\![\ ]\!]$, so $\mathcal{N}[\![4]\!] = \underline{4}$ is the fourth element of $\mathbb{N}$.

In the Natural Semantics we are concerned with the relation between the *initial* and *final* state, denoted by: $\langle a, s \rangle \to v$ (remember that a state maps variables to values, and that an arithmetic expression should represent a value).

The rules that define the Natural Semantics for Arithmetic Expressions are:

$$(\text{NUM}_{ns}) \quad \frac{}{\langle n, s \rangle \to \underline{n}} \qquad (\text{VAR}_{ns}) \quad \frac{}{\langle x, s \rangle \to s\, x} \qquad (\text{OP}_{ns}) \quad \frac{\langle a_1, s \rangle \to v_1 \quad \langle a_2, s \rangle \to v_2}{\langle a_1 \ \textbf{op}\ a_2, s \rangle \to v_1 \ \underline{\textbf{op}}\ v_2}$$

where ‘**op**’ is any of ‘$+, \times, -$’. We will omit the brackets around $(a_1$ **op** $a_2)$ whenever convenient (like for the outermost pair).

Notice that, in rule (NUM$_{ns}$), both n and $\underline{n}$ are used. Similarly, in rule (OP$_{ns}$), both **op** and $\underline{\textbf{op}}$ are used. The first, **op**, is the syntactic representation of the intended operation, whereas $\underline{\textbf{op}}$ is the actual operation, that lives in the semantic setting. Since we map on to the ‘real’ world, we can immediately – or whenever convenient – perform the actual calculation there. This implies that, normally, $\underline{(3\ \times\ 4)\ +\ 8}$ is not written in our notation; instead, we will write the ‘result’ $\underline{20}$.

Also, notice that the Natural Semantics maps pairs of $\langle$*expression*, *state*$\rangle$ to a *value*, a number in $\mathbb{Z}$. This implies that a state is a mapping from variables to the ‘real world’ of $\mathbb{Z}$.

Example 4.1 Suppose you would want to know the semantics of $(3 \times x) + 8$, in a state s such that $s\,x = 4$. Since the semantics is defined using rules, what you need to do is to build a derivation that gives you the desired result.

You know that the last line in this derivation will be of the shape

$$\langle (3 \times x) + 8, s\rangle \to ?.$$

This leaves no choice for the last rule applied: $(+_{ns})$.

$$\dfrac{\langle 3 \times x, s\rangle \to ? \quad \langle 8, s\rangle \to ?}{\langle (3 \times x) + 8, s\rangle \to ?}$$

which leaves us with the task of finding derivations for both $\langle 3 \times x\rangle \to ?$ and $\langle 8\rangle \to ?$. For the first, again the syntax of the expression involved tells us that the last rule applied must have been $(\times_{ns})$.

$$\dfrac{\dfrac{\langle 3, s\rangle \to ? \quad \langle x, s\rangle \to ?}{\langle 3 \times x, s\rangle \to ?} \qquad \langle 8, s\rangle \to ?}{\langle (3 \times x) + 8, s\rangle \to ?}$$

So we only need to find the derivations for $\langle 3, s\rangle \to ?$, $\langle 8\rangle \to ?$, and $\langle x, s\rangle \to ?$. Of course, by rule (NUM$_{ns}$), this is easy for the first two. We use rule (VAR$_{ns}$) for the third, and use in the derivation that we know that $s\,x = 4$. Notice that this is a mathematical equation, so $s\,x$ and 4 represent exactly the same value, and can be used both there. Since 4 carries more information to us human beings, we prefer to write the actual number over a mysterious expression.

We can for these three (sub)derivations fill in the open place kept by ‘?’, and use these results to fill in the open places below. Thus we obtain:

$$\dfrac{\dfrac{\dfrac{}{\langle 3, s\rangle \to \underline{3}} \qquad \dfrac{}{\langle x, s\rangle \to \underline{4}}}{\langle 3 \times x, s\rangle \to \underline{12}} \qquad \dfrac{}{\langle 8, s\rangle \to \underline{8}}}{\langle (3 \times x) + 8, s\rangle \to \underline{20}}$$

In what follows, we will often omit the underline, if from the immediate context it is clear if we intend the syntactic representation or the semantic value. Then the previous derivation becomes:

$$\dfrac{\dfrac{\overline{\langle 3, s\rangle \to 3} \qquad \overline{\langle x, s\rangle \to 4}}{\langle 3 \times x, s\rangle \to 12} \qquad \overline{\langle 8, s\rangle \to 8}}{\langle (3 \times x) + 8, s\rangle \to 20}$$

Exercise 4.2 Write down a suitable abstract syntax for decimal numerals and define a semantic function $\mathcal{N}$ which maps numerals to integers.

4.2 Determinism

The Natural Semantics for natural numbers is deterministic, i.e., there is only *one* possible derivation given an expression and a state:

Theorem 4.3 *If $\langle a, s\rangle \to v$ and $\langle a, s\rangle \to v'$, then $v = v'$.*

Proof : Assume $\langle a, s\rangle \to v$. Then by definition, there is a derivation that has this at the bottom line. The proof goes by induction on the structure of derivations, where we focus on the last rule applied.

(NUM_{ns}) : Then $a \in$ *Numeral*, say $a = n$, and $v = n$. The only way to derive $\langle n, s\rangle \to v'$ is by a derivation consisting only of rule (NUM_{ns}), so also $v' = n$.

(VAR_{ns}) : Then $a \in$ *Variables*, say $a = x$, and $v = s\,x$. The only way to derive $\langle x, s\rangle \to v'$ is by a derivation consisting only of rule (VAR_{ns}), so also $v' = s\,x$; since s is a function, $v' = v$.

(OP_{ns}) : Then $a \equiv (a_1 \textbf{ op } a_2)$, $v = v_1 \textbf{ op } v_2$, and there are sub-derivations showing both $\langle a_1, s\rangle \to v_1$ and $\langle a_2, s\rangle \to v_2$. Likewise, $\langle a_1 \textbf{ op } a_2, s\rangle \to v'$ can only be obtained via a derivation that ends with rule (OP_{ns}), and again there are sub-derivations showing $\langle a_1, s\rangle \to v_1'$ and $\langle a_2, s\rangle \to v_2'$. Now, by induction, $v_1 = v_1'$ and $v_2 = v_2'$; therefore, also $(v_1 \textbf{ op } v_2) = (v_1' \textbf{ op } v_2')$, so $v = v'$.

Exercise 4.4 The Natural Semantics for Arithmetic Expressions is terminating, i.e., for every $a \in$ *Arithmetic Expressions*, for all states s, there is a v such that $\langle a, s\rangle \to v$.

4.3 Denotational Semantics for Arithmetic Expressions

We model the memory (or store) of the machine on which we are to run a program by functions of type

$$\textit{State} \quad = \quad \textit{Variables} \to \mathbb{Z}.$$

The Denotational Semantics on *Arithmetic Expressions* is a *total* function

$$\mathcal{A} \quad : \quad \textit{Arithmetic Expressions} \to \textit{State} \to \mathbb{Z}.$$

In defining $\mathcal{A}$, we will (as above), when needed, underline to give meaning to the syntactic representation of the number and operations.

$$\begin{array}{rcl} \mathcal{A}[\![n]\!]\,s & = & \underline{n} \\ \mathcal{A}[\![x]\!]\,s & = & s\,x \\ \mathcal{A}[\![a_1 \textbf{ op } a_1]\!]\,s & = & \mathcal{A}[\![a_1]\!]\,s \ \underline{\textbf{op}}\ \mathcal{A}[\![a_1]\!]\,s \end{array}$$

Example 4.5 Suppose $s\,x = 3$. Then:

$$\begin{array}{rcl}\mathcal{A}[\![x+1]\!]\,s & = & \mathcal{A}[\![x]\!]\,s \,\underline{+}\, \mathcal{A}[\![1]\!]\,s \\ & = & s\,x \,\underline{+}\, \underline{1} \\ & = & \underline{3+1} \\ & = & \underline{4}\end{array}$$

Exercise 4.6 Show that the Natural Semantics and the Denotational Semantics of Arithmetic Expressions are equivalent.

4.4 Free variables

The free variables of an expression $a \in$ *Arithmetic Expressions*, $FV(a)$, is the set of variables occurring in a and is inductively defined by:

$$\begin{array}{rcl}FV(n) & = & \emptyset \\ FV(x) & = & \{x\} \\ FV(a_1 \ \textbf{op}\ a_2) & = & FV(a_1) \cup FV(a_2)\end{array}$$

Theorem 4.7 *Let s and s' be two states satisfying $s\,x = s'x$, for all $x \in FV(a)$. Then $\mathcal{A}[\![a]\!]\,s = \mathcal{A}[\![a]\!]\,s'$.*

Proof : By induction on the structure of terms in *Arithmetic Expressions*.
$(a \equiv n)$: $\mathcal{A}[\![n]\!]\,s = n = \mathcal{A}[\![n]\!]\,s'$.
$(a \equiv x)$: $\mathcal{A}[\![x]\!]\,s = s\,x$ and $\mathcal{A}[\![x]\!]\,s' = s'x$, by definition of $\mathcal{A}$. Since $FV(x) = \{x\}$, we get $s\,x = s'x$, and therefore also A $[\![x]\!]$ s = A $[\![x]\!]$ s’.
$(a \equiv (a_1 \ \textbf{op}\ a_1))$: By definition of $\mathcal{A}$, we have $\mathcal{A}[\![a_1 \ \textbf{op}\ a_1]\!]\,s = \mathcal{A}[\![a_1]\!]\,s \ \textbf{op}\ \mathcal{A}[\![a_1]\!]\,s$, as well as $\mathcal{A}[\![a_1 \ \textbf{op}\ a_1]\!]\,s' = \mathcal{A}[\![a_1]\!]\,s' \ \textbf{op}\ \mathcal{A}[\![a_1]\!]\,s'$. Since $FV(a_i) \subseteq FV(a_1 \ \textbf{op}\ a_1)$, the states s and s' agree on both a_1 and a_2, so, by induction, we have $\mathcal{A}[\![a_1]\!]\,s = \mathcal{A}[\![a_1]\!]\,s'$ and $\mathcal{A}[\![a_2]\!]\,s = \mathcal{A}[\![a_2]\!]\,s'$. But then $\mathcal{A}[\![a_1]\!]\,s \ \textbf{op}\ \mathcal{A}[\![a_1]\!]\,s = \mathcal{A}[\![a_1]\!]\,s' \ \textbf{op}\ \mathcal{A}[\![a_1]\!]\,s'$, so also $\mathcal{A}[\![a_1 \ \textbf{op}\ a_1]\!]\,s = \mathcal{A}[\![a_1 \ \textbf{op}\ a_1]\!]\,s'$.

4.5 The language `While`

We will develop various semantics for a basic programming language, called `While`. Its abstract syntax is given by:

$$\begin{array}{rcl}a & \in & \textit{Arithmetic Expressions} \\ n & \in & \textit{Numeral} \\ x & \in & \textit{Variables} \\ b & \in & \textit{Boolean Expressions} \\ S & \in & \textit{Statements}\end{array}$$

$$\begin{array}{rcl}a & ::= & n \mid x \mid (a_1 + a_2) \mid (a_1 - a_2) \mid (a_1 \times a_2) \\ b & ::= & \texttt{true} \mid \texttt{false} \mid (a_1 = a_2) \mid (a_1 \leq a_2) \mid (\neg b) \mid (b_1 \ \&\ b_2) \\ S & ::= & x \texttt{ := } a \mid \texttt{skip} \mid (S_1 \texttt{ ; } S_2) \mid (\texttt{while}\ b\ \texttt{do}\ S) \mid (\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2)\end{array}$$

We will normally only write those brackets that are necessary to avoid confusion.

Example 4.8

$$y \texttt{ := } 1 \texttt{;} \textbf{while} \, \neg(x = 1) \, \textbf{do} \, (y \texttt{ := } y \times x \texttt{;} x \texttt{ := } x - 1)$$

4.6 Semantics of Boolean expressions

Below, we will define the Natural Semantics of the language `While`. Before we come to that, we need to define the Denotational Semantics of Boolean Expressions as a function of the following type:

$$\mathcal{B} \quad : \quad \textit{Boolean Expressions} \to \textit{State} \to \mathbb{T}$$

where $\mathbb{T} = \{\mathbf{tt}, \mathbf{ff}\}$, the set of (semantic) truth values – as follows:

$$\begin{array}{rcl}
\mathcal{B}[\![\texttt{true}]\!]\,s & = & \mathbf{tt} \\
\mathcal{B}[\![\texttt{false}]\!]\,s & = & \mathbf{ff} \\
\mathcal{B}[\![a_1 = a_2]\!]\,s & = & \mathbf{tt}, \text{ if } \mathcal{A}[\![a_1]\!]\,s = \mathcal{A}[\![a_2]\!]\,s \\
 & = & \mathbf{ff}, \text{ if } \mathcal{A}[\![a_1]\!]\,s \neq \mathcal{A}[\![a_2]\!]\,s \\
\mathcal{B}[\![a_1 \leq a_2]\!]\,s & = & \mathbf{tt}, \text{ if } \mathcal{A}[\![a_1]\!]\,s \leq \mathcal{A}[\![a_2]\!]\,s \\
 & = & \mathbf{ff}, \text{ if } \mathcal{A}[\![a_1]\!]\,s > \mathcal{A}[\![a_2]\!]\,s \\
\mathcal{B}[\![\neg b]\!]\,s & = & \mathbf{tt}, \text{ if } \mathcal{B}[\![b]\!]\,s = \mathbf{ff} \\
 & = & \mathbf{ff}, \text{ if } \mathcal{B}[\![b]\!]\,s = \mathbf{tt} \\
\mathcal{B}[\![b_1 \,\&\, b_2]\!]\,s & = & \mathbf{tt}, \text{ if } \mathcal{B}[\![b_1]\!]\,s = \mathbf{tt} \;\&\; \mathcal{B}[\![b_2]\!]\,s = \mathbf{tt} \\
 & = & \mathbf{ff}, \text{ otherwise}
\end{array}$$

Alternatively, since $\mathcal{A}[\![a_1]\!]\,s = \mathcal{A}[\![a_2]\!]\,s$ lives in the semantic world, and has the same value as $\mathbf{tt}$ if both values are equal, we can define the semantics like this:

$$\begin{array}{rcl}
\mathcal{B}[\![\texttt{true}]\!]\,s & = & \mathbf{tt} \\
\mathcal{B}[\![\texttt{false}]\!]\,s & = & \mathbf{ff} \\
\mathcal{B}[\![a_1 = a_2]\!]\,s & = & (\mathcal{A}[\![a_1]\!]\,s = \mathcal{A}[\![a_2]\!]\,s) \\
\mathcal{B}[\![a_1 \leq a_2]\!]\,s & = & (\mathcal{A}[\![a_1]\!]\,s \leq \mathcal{A}[\![a_2]\!]\,s) \\
\mathcal{B}[\![\neg b]\!]\,s & = & \neg\,(\mathcal{B}[\![b]\!]\,s) \\
\mathcal{B}[\![b_1 \,\&\, b_2]\!]\,s & = & (\mathcal{B}[\![b_1]\!]\,s \;\&\; \mathcal{B}[\![b_2]\!]\,s)
\end{array}$$

4.7 Natural Semantics of `While`

We will now present the Natural Semantics of `While`. This semantics is defined as a binary relation between configurations, that are defined being either:

$$\begin{array}{rcl}
\langle S, s\rangle & – & S \text{ is to be executed from state } s, \\
s & – & \text{a terminal state, or value.}
\end{array}$$

Transitions from the initial pair to the terminal state are denoted by $\langle S, s\rangle \rightarrow s'$, and the Natural Semantics is defined by the set of (derivation) rules below:

$$(\textsc{num}_{ns}) \quad \frac{}{\langle n, s\rangle \rightarrow \underline{n}} \qquad (\textsc{op}_{ns}) \quad \frac{\langle a_1, s\rangle \rightarrow v_1 \quad \langle a_2, s\rangle \rightarrow v_2}{\langle a_1 \textbf{ op } a_2, s\rangle \rightarrow v_1 \underline{\textbf{op}}\, v_2}$$

$$(\textsc{var}_{ns}) \quad \frac{}{\langle x, s\rangle \rightarrow s\,x} \qquad (\textsc{skip}_{ns}) \quad \frac{}{\langle \texttt{skip}, s\rangle \rightarrow s}$$

$$(\textsc{ass}_{ns}) \quad \frac{\langle a, s\rangle \rightarrow v}{\langle x := a, s\rangle \rightarrow s[x \mapsto v]} \qquad (\textsc{comp}_{ns}) \quad \frac{\langle S_1, s_1\rangle \rightarrow s_2 \quad \langle S_2, s_2\rangle \rightarrow s_3}{\langle S_1 \,\texttt{;}\, S_2, s_1\rangle \rightarrow s_3}$$

$$(\textsc{while}^{\mathrm{F}}_{ns}) \quad \frac{}{\langle \texttt{while}\ b\ \textbf{do}\ S, s\rangle \rightarrow s} \ (\mathcal{B}[\![b]\!]\, s_1 = \mathbf{ff})$$

$$(\textsc{while}^{\mathrm{T}}_{ns}) \quad \frac{\langle S, s_1\rangle \rightarrow s_2 \quad \langle \texttt{while}\ b\ \textbf{do}\ S, s_2\rangle \rightarrow s_3}{\langle \texttt{while}\ b\ \textbf{do}\ S, s_1\rangle \rightarrow s_3} \ (\mathcal{B}[\![b]\!]\, s_1 = \mathbf{tt})$$

$$(\textsc{cond}^{\mathrm{T}}_{ns}) \quad \frac{\langle S_1, s_1\rangle \rightarrow s_2}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s_1\rangle \rightarrow s_2} \ (\mathcal{B}[\![b]\!]\, s_1 = \mathbf{tt})$$

$$(\textsc{cond}^{\mathrm{F}}_{ns}) \quad \frac{\langle S_2, s_1\rangle \rightarrow s_2}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s_1\rangle \rightarrow s_2} \ (\mathcal{B}[\![b]\!]\, s_1 = \mathbf{ff})$$

In rule $(\textsc{ass}_{ns})$, the notation $s[x \mapsto v]$ stands for the state (function) defined by

$$\begin{array}{rcll} s[x \mapsto v]\,y & = & v, & \text{if } x = y \\ & = & s\,y, & \text{if } x \neq y \end{array}$$

Notice that rule $(\textsc{num}_{ns})$ is actually a rule-*scheme*: there is an instance of this rule for every $n \in$ *Numeral*.

The present form of the rules needs some comments. Notice that, of course, it is possible to define the Natural Semantics of Boolean expressions, and use that above when we define the rules for the `while` -loop and the conditional. However, since a boolean expression as such is never a statement in *Statements*, and the rules are intended to specify the semantics for `While`, we will never actually want to derive the semantics for a boolean expression. To have slightly less complicated derivations in the system, we have used the semantics of a boolean expression *in a side-condition to the rule*, rather than build a separate derivation for it.

Remark that the same kind of reasoning also applies to *Arithmetic Expressions*. We could, therefore, have used the Denotational Semantics for *Arithmetic Expressions*, $\mathcal{A}[\![a]\!]\, s$, and have presented the rule

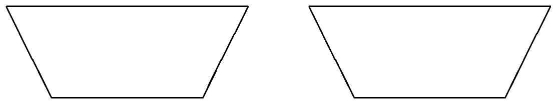
$$(\textsc{ass}'_{ns}) \quad \frac{}{\langle x \texttt{ := } a, s\rangle \rightarrow s[x \mapsto \mathcal{A}[\![a]\!]\, s]}$$

instead of the rules $(\textsc{num}_{ns})$, $(\textsc{op}_{ns})$, $(\textsc{var}_{ns})$, and $(\textsc{ass}_{ns})$ above.

As before, the rules are used to construct derivations. In general, when looking for the semantics of a certain statement S in a specific state s, we would first need to construct the derivation, as done above when looking for the semantics for $(3 \times x) + 8$.

$$\langle S, s\rangle \to ?$$

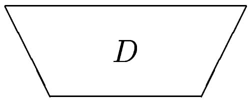
In general, the syntax of S guides the construction of the derivation; looking at its ‘top-level’ syntax construct, we know what the last rule used in the derivation must have been.



$$\frac{\langle S_1, s_1\rangle \to ? \qquad \langle S_2, s_2\rangle \to ?}{\langle S, s\rangle \to ?}$$

This enables us to build the derivation, filled with ‘?’, until we will, normally, end up looking for derivations for $\langle n, s'\rangle \to ?$ or $\langle \texttt{skip}, s'\rangle \to ?$. These are easily found, using the rules $(\textsc{ass}_{ns})$ and $(\textsc{skip}_{ns})$.

The only problem in this approach is the presence of rule $(\textsc{while}^{\mathrm{T}}_{ns})$.

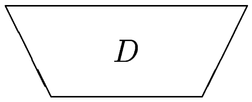


$$\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \to ?$$

The structure of D depends on the value of $\mathcal{B}[\![b]\!]\,s$, since that decides which of the rules $(\textsc{while}^{\mathrm{T}}_{ns})$ or $(\textsc{while}^{\mathrm{F}}_{ns})$ has been used. Suppose $\mathcal{B}[\![b]\!]\,s = \mathbf{tt}$, then the structure of the derivation is as follows:

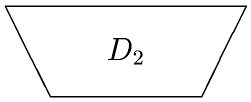
$$\frac{\langle S, s\rangle \to ?_1 \qquad \langle \texttt{while}\ b\ \texttt{do}\ S, ?_1\rangle \to ?_2}{\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \to ?_2}$$

To find D_2, we find ourselves with a problem similar to the one we started with; the only difference between the two is the use of $?_1$ in one and $?_2$ in the other. And in fact, for some statements this approach does not terminate. Take



$$\langle \texttt{while true do skip}, s\rangle \to ?$$

Since $\mathcal{B}[\![\texttt{true}]\!]\,s = \mathbf{tt}$ for all s, after the first step we obtain



$$\frac{\dfrac{}{\langle \texttt{skip}, s\rangle \to s} \qquad \langle \texttt{while true do skip}, s\rangle \to ?}{\langle \texttt{while true do skip}, s\rangle \to ?}$$

which brings us on an infinite search for the derivation D. In fact, since the bottom line of D, $\langle \texttt{while true do skip}, s\rangle \to ?$, is *exactly* the same as the one for D_2, we get that the derivations D and D_2 must be identical. Since D_2 is a real sub-derivation of D, it is impossible for D to exist.

The execution of a statement S on a state s

- *terminates* if and only if there is a state s' such that $\langle S, s\rangle \to s'$, and
- *loops* if and only if there is no state s' such that $\langle S, s\rangle \to s'$.

Notice that these notions correspond exactly to what you would expect in real life, since the program **`while true do skip`** loops forever.

Exercise 4.9 Construct a derivation tree for the following `While` program.

$$z \texttt{ := } 0 \,\texttt{;}\, \textbf{while}\ y \leq x\ \textbf{do}\ (z \texttt{ := } z+1 \,\texttt{;}\, x \texttt{ := } x-y)$$

in a state where x has value 17 and y has value 5.

4.8 Semantic Equivalence

We can use the semantics to determine whether two statements S_1 and S_2 are *semantically equivalent*, i.e. whether for all states s and s':

$$\langle S_1, s\rangle \to s' \text{ if and only if } \langle S_2, s\rangle \to s'.$$

Notice that this definition is actually over *derivations*: a derivation exists for $\langle S_1, s\rangle \to s'$ if and only if a derivation exists for $\langle S_2, s\rangle \to s'$; notice that these derivations need not be similar in shape.

Lemma 4.10 The statements

$$\textbf{while}\ b\ \textbf{do}\ S$$
and
$$\textbf{if}\ b\ \textbf{then}\ (S\,\texttt{;}\,\textbf{while}\ b\ \textbf{do}\ S)\ \textbf{else skip}$$

are semantically equivalent.

Proof : We will show that, *given* a derivation for one result, we can *construct* a derivation for the other, and vice versa. This implies that *if* a derivation exists for the one, *then* a derivation exists for the other, and vice versa, as requested.

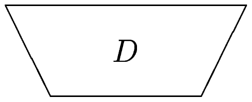
The proof comes in two parts. We will show that, for all states s and s':

If $\langle \textbf{while}\ b\ \textbf{do}\ S, s\rangle \to s'$, then $\langle \textbf{if}\ b\ \textbf{then}\ (S\,\texttt{;}\,\textbf{while}\ b\ \textbf{do}\ S)\ \textbf{else skip}, s\rangle \to s'$,

and

If $\langle \textbf{if}\ b\ \textbf{then}\ (S\,\texttt{;}\,\textbf{while}\ b\ \textbf{do}\ S)\ \textbf{else skip}, s\rangle \to s'$, then $\langle \textbf{while}\ b\ \textbf{do}\ S, s\rangle \to s'$.

i) If $\langle \textbf{while}\ b\ \textbf{do}\ S, s\rangle \to s'$, there is a derivation that has this as its conclusion.



$$\langle \textbf{while}\ b\ \textbf{do}\ S, s\rangle \to s'$$

The structure of D depends on the value of $\mathcal{B}[\![b]\!]\, s$, since that decides which of the rules $(\text{WHILE}^{\text{F}}_{ns})$ or $(\text{WHILE}^{\text{T}}_{ns})$ has been used. We have two choices:

$(\mathcal{B}[\![b]\!]\,s = \mathbf{ff})$: Then the structure of the derivation D is as follows:

$$\frac{}{\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \to s}\ (\mathcal{B}[\![b]\!]\,s = \mathbf{ff})$$

so, in particular, $s = s'$. The following is easy to construct:

$$\frac{\dfrac{}{\langle \texttt{skip}, s\rangle \to s}}{\langle \texttt{if}\ b\ \texttt{then}\ (S\,\texttt{;}\,\texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else}\ \texttt{skip}, s\rangle \to s}\ (\mathcal{B}[\![b]\!]\,s = \mathbf{ff})$$

$(\mathcal{B}[\![b]\!]\,s = \mathbf{tt})$: Then the structure of the derivation is as follows:

$$\frac{\begin{array}{c} D_1 \\ \langle S, s\rangle \to s_1 \end{array} \qquad \begin{array}{c} D_2 \\ \langle \texttt{while}\ b\ \texttt{do}\ S, s_1\rangle \to s' \end{array}}{\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \to s'}\ (\mathcal{B}[\![b]\!]\,s = \mathbf{tt})$$

We can now construct:

$$\frac{\dfrac{\begin{array}{c} D_1 \\ \langle S, s\rangle \to s_1 \end{array} \qquad \begin{array}{c} D_2 \\ \langle \texttt{while}\ b\ \texttt{do}\ S, s_1\rangle \to s' \end{array}}{\langle S\,\texttt{;}\,\texttt{while}\ b\ \texttt{do}\ S, s\rangle \to s'}}{\langle \texttt{if}\ b\ \texttt{then}\ (S\,\texttt{;}\,\texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else}\ \texttt{skip}, s\rangle \to s'}\ (\mathcal{B}[\![b]\!]\,s = \mathbf{tt})$$

ii) If $\langle \texttt{if}\ b\ \texttt{then}\ (S\,\texttt{;}\,\texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else}\ \texttt{skip}, s\rangle \to s'$, there is a derivation that has this as its conclusion.

$$\begin{array}{c} D \\ \langle \texttt{if}\ b\ \texttt{then}\ (S\,\texttt{;}\,\texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else}\ \texttt{skip}, s\rangle \to s' \end{array}$$

Again, the structure of D depends on the value of B $[\![b]\!]$ s, since that decides which of the rules $(\text{COND}^{\text{F}}_{ns})$ or $(\text{COND}^{\text{T}}_{ns})$ has been used as the last rule. We have two choices:
$(\mathcal{B}[\![b]\!]\,s = \mathbf{ff})$: Then the structure of D is as follows:

$$\frac{\dfrac{}{\langle \texttt{skip}, s\rangle \to s}}{\langle \texttt{if}\ b\ \texttt{then}\ (S\,\texttt{;}\,\texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else}\ \texttt{skip}, s\rangle \to s}\ (\mathcal{B}[\![b]\!]\,s = \mathbf{ff})$$

so, in particular, $s = s'$. The following is immediate:

$$\frac{}{\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \to s}\ (\mathcal{B}[\![b]\!]\,s = \mathbf{ff})$$

$(\mathcal{B}[\![b]\!]\,s = \mathbf{tt})$: Then the structure of D is as follows:

$$\cfrac{\cfrac{\begin{array}{cc} D_1 & D_2 \\ \langle S, s\rangle \to s_1 & \langle \texttt{while}\, b\, \textbf{do}\, S, s_1\rangle \to s' \end{array}}{\langle S\,\texttt{;}\,\texttt{while}\, b\, \textbf{do}\, S, s\rangle \to s'}}{\langle \texttt{if}\, b\, \textbf{then}\, (S\,\texttt{;}\,\texttt{while}\, b\, \textbf{do}\, S)\, \texttt{else}\, \texttt{skip}, s\rangle \to s'}\ (\mathcal{B}[\![b]\!]\, s = \mathbf{tt})$$

We can now construct:

$$\cfrac{\begin{array}{cc} D_1 & D_2 \\ \langle S, s\rangle \to s_1 & \langle \texttt{while}\, b\, \textbf{do}\, S, s_1\rangle \to s' \end{array}}{\langle \texttt{while}\, b\, \textbf{do}\, S, s\rangle \to s'}\ (\mathcal{B}[\![b]\!]\, s = \mathbf{tt})$$

Which completes the proof.

(Notice that this has *not* been a proof by induction on the structure of derivations.)

Theorem 4.11 *The Natural Semantics is deterministic. i.e. if* $\langle S, s\rangle \to s^1$ *and* $\langle S, s\rangle \to s^2$*, then* $s^1 = s^2$.

Proof : Let D^1 be the derivation that shows $\langle S, s\rangle \to s^1$. The proof is by induction on the structure of derivations, by focusing on the last rule used. We only show some of the cases.

(SKIP_{ns}) : Then $S = \texttt{skip}$ and D^1 is structured like

$$\cfrac{}{\langle \texttt{skip}, s\rangle \to s}$$

so also $s = s^1$. Likewise, the derivation that shows $\langle \texttt{skip}, s\rangle \to s^2$ can only be composed of (SKIP_{ns}), so also $s = s^2$. So, in particular, $s^1 = s^2$.

(COMP_{ns}) : Then $S = (S_1\,\texttt{;}\,S_2)$, and D^1 is structured like

$$\cfrac{\begin{array}{cc} D_1 & D_2 \\ \langle S_1, s\rangle \to s_1 & \langle S_2, s_1\rangle \to s^1 \end{array}}{\langle S_1\,\texttt{;}\,S_2, s\rangle \to s^1}$$

Let D^2 be the derivation that shows $\langle S_1\,\texttt{;}\,S_2, s\rangle \to s^2$. The only rule that could have been applied last in D^2 is (COMP_{ns}), so the structure of D^2 is similar to that of D^1, and in particular there are sub-derivations that show $\langle S_1, s\rangle \to s_2$ and $\langle S_2, s_2\rangle \to s^2$. Then, by induction, $s_1 = s_2$, and therefore, again by induction, $s^1 = s^2$.

$(\text{WHILE}^{\text{T}}_{ns})$: Then $S = \texttt{while}\, b\, \textbf{do}\, S_1$, and D is structured like

$$\cfrac{\begin{array}{cc} D_1 & D_2 \\ \langle S_1, s\rangle \to s_1 & \langle \texttt{while}\, b\, \textbf{do}\, S_1, s_1\rangle \to s^1 \end{array}}{\langle \texttt{while}\, b\, \textbf{do}\, S_1, s\rangle \to s^1}\ (\mathcal{B}[\![b]\!]\, s = \mathbf{tt})$$

Let D^2 be the derivation that shows $\langle \texttt{while}\, b\, \textbf{do}\, S_1, s\rangle \to s^2$. Again, the only rule that could have been applied last in D^2 is ($\textsc{while}^{\mathrm{T}}_{ns}$), so the structure of D^2 is similar to that of D^1, and in particular there are sub-derivations that show $\langle S_1, s\rangle \rightarrow s_2$ and $\langle \texttt{while}\ b\ \texttt{do}\ S_1, s_2\rangle \rightarrow s^2$. Then, by induction, $s_1 = s_2$, and therefore, again by induction, $s^1 = s^2$.

Exercise 4.12 Complete this proof, i.e. write out the missing parts.

The meaning of statements can now be summarised as a (partial) function from *State* to *State*.

$$\mathcal{S}_{ns} \;:\; \mathit{Statements} \rightarrow \mathit{State} \hookrightarrow \mathit{State}$$

$$\begin{array}{lcll} \mathcal{S}_{ns}\,[\![S]\!]\,s & = & s', & \text{if } \langle S, s\rangle \rightarrow s' \\ & = & \mathit{undefined}, & \text{otherwise} \end{array}$$

This notion is well-defined, because the Natural Semantics, $\langle\cdot,\cdot\rangle \rightarrow \cdot$, is deterministic.

Note: the above defined function $\mathcal{S}_{ns}$ is truly partial, because $\mathcal{S}_{ns}\,[\![\texttt{while true do skip}]\!]\,s$ is *undefined*.

5 Structural Operational Semantics

In Structural Operational Semantics, the emphasis lies on the individual steps of the execution (evaluation of operators). It is defined, as Natural Semantics, by means of transitions, but of the shape: $\langle S, s\rangle \Rightarrow \gamma$. A difference between Natural and Structural Operational Semantics lies in the fact that now the right-hand side of the transition, γ, need not be the final state, but can be an intermediate result. In other words: γ is of the form $\langle S', s'\rangle$ or s'. The configurations are the same as for the Natural Semantics. We say that a configuration $\langle S, s\rangle$ is stuck if there is no γ such that $\langle S, s\rangle \Rightarrow \gamma$.

We give the Structural Operational Semantics for `While`.

$$(\textsc{ass}_{sos})\quad \frac{}{\langle x := a, s\rangle \Rightarrow s[x \mapsto \mathcal{A}\,[\![a]\!]\,s]} \qquad (\textsc{skip}_{sos})\quad \frac{}{\langle \texttt{skip}, s\rangle \Rightarrow s}$$

$$(\textsc{comp}^{\mathrm{T}}_{sos})\quad \frac{\langle S_1, s_1\rangle \Rightarrow s_2}{\langle S_1\,\texttt{;}\,S_2, s_1\rangle \Rightarrow \langle S_2, s_2\rangle} \qquad (\textsc{comp}^{\mathrm{I}}_{sos})\quad \frac{\langle S_1, s_1\rangle \Rightarrow \langle S_1', s_2\rangle}{\langle S_1\,\texttt{;}\,S_2, s_1\rangle \Rightarrow \langle S_1'\,\texttt{;}\,S_2, s_2\rangle}$$

$$(\textsc{cond}^{\mathrm{T}}_{sos})\quad \frac{}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \Rightarrow \langle S_1, s\rangle}\ (\mathcal{B}\,[\![b]\!]\,s = \mathbf{tt})$$

$$(\textsc{cond}^{\mathrm{F}}_{sos})\quad \frac{}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \Rightarrow \langle S_2, s\rangle}\ (\mathcal{B}\,[\![b]\!]\,s = \mathbf{ff})$$

$$(\textsc{while}_{sos})\quad \frac{}{\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \Rightarrow \langle \texttt{if}\ b\ \texttt{then}\ (S\,\texttt{;}\,\texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else skip}, s\rangle}$$

Example 5.1

$$\dfrac{\dfrac{\overline{\langle x \texttt{ := } 1, s\rangle \;\Rightarrow\; s[x \mapsto 1]}}{\langle x \texttt{ := } 1 \texttt{;} y \texttt{ := } 2, s\rangle \;\Rightarrow\; \langle y \texttt{ := } 2, s[x \mapsto 1]\rangle}}{\langle x \texttt{ := } 1 \texttt{;} y \texttt{ := } 2 \texttt{;} z \texttt{ := } 3, s\rangle \;\Rightarrow\; \langle y \texttt{ := } 2 \texttt{;} z \texttt{ := } 3, s[x \mapsto 1]\rangle}$$

Using the above rules, it is also possible to derive this result with an even shorter derivation.

$$\dfrac{\overline{\langle x \texttt{ := } 1, s\rangle \;\Rightarrow\; s[x \mapsto 1]}}{\langle x \texttt{ := } 1 \texttt{;} y \texttt{ := } 2 \texttt{;} z \texttt{ := } 3, s\rangle \;\Rightarrow\; \langle y \texttt{ := } 2 \texttt{;} z \texttt{ := } 3, s[x \mapsto 1]\rangle}$$

Exercise 5.2 Extend the `While`-language with the statement

$$\texttt{for } x := a_1 \texttt{ to } a_2 \texttt{ do } S$$

and define both the Natural Semantics and the Structural Operational Semantics of this new construct. You may need to assume that you have an 'inverse' to the semantic function for numerals, so that there is a numeral for each number that may arise during the computation. The sematics for the **`for`** -loop should not rely on the semantics of the **`while`** -loop.

Exercise 5.3 Prove that the SOS-style semantics for `While` are deterministic.

Exercise 5.4 Use the Natural Semantics of `While` to show that

$$S_1 \texttt{ ; if } b \texttt{ then } S \texttt{ else } S'$$

is semantically equivalent to

$$\texttt{if } b \texttt{ then } S_1 \texttt{ ; } S \texttt{ else } S_1 \texttt{ ; } S'$$

providing that b does not depend on any of the variables modified by S_1.

Exercise 5.5 Prove that, for the `While`-language,

$$\langle S_1, s\rangle \;\Rightarrow^k\; s' \text{ implies } \langle S_1 \texttt{ ; } S_2, s\rangle \;\Rightarrow^k\; \langle S_2, s'\rangle.$$

5.1 Sequences

A (possibly infinite) number of configurations $\gamma_0, \gamma_1, \gamma_2, \ldots$, such that $\gamma_0 = \langle S, s\rangle$, $\gamma_i \;\Rightarrow\; \gamma_{i+1}$ for every $0 \leq i$, is called a *sequence* of a statement S in state s.

A *derivation sequence* of a statement S in state s is a sequence of S in s such that either

- the sequence is finite (of length $n \geq 0$), and γ_n is terminal or stuck, or
- the sequence is infinite.

We write $\gamma_1 \;\Rightarrow^i\; \gamma_2$ if there is a sequence of length i form γ_1 to γ_2, and $\gamma_1 \;\Rightarrow^*\; \gamma_2$ if there is a sequence of finite length. Notice that $\gamma_1 \;\Rightarrow^i\; \gamma_2$ (or $\gamma_1 \;\Rightarrow^*\; \gamma_2$) not necessarily represents a *derivation* sequence, since that would require γ_2 to be terminal or stuck. Also, for each step in the sequence there is a derivation tree, as defined by the rules above.

For a given S and s it is always possible to find 1 derivation sequence from $\langle S, s\rangle$: apply the axioms and rules forever until a configuration is either a terminal or stuck. By close inspection of the abstract syntax and the rules, it becomes clear that, for the language `While`, there are no stuck configurations for the Structural Operational Semantics. Since also the Structural Operational Semantics for `While` is deterministic (will be shown later), it becomes clear that there is only 1 derivation sequence for each configuration.

5.2 Some properties

The execution of a statement S in s

- *terminates* if and only if there is a finite derivation sequence of S in s.
- *loops* if and only if there is an infinite derivation sequence of S in s.
- *terminates successfully* if there is a s' such that $\langle S, s\rangle \Rightarrow^* s'$.

NB: any terminating execution is also successful in `While` - this does not hold for some of the extensions to that language we will discuss later.

A statement S *always terminates* (*loops*) if it terminates (loops) on all states.

Theorem 5.6 *If $\langle S_1\,;S_2, s_1\rangle \Rightarrow^k s_2$, then there exists a state s_0 and natural numbers k_1 and k_2 such that $\langle S_1, s_1\rangle \Rightarrow^{k_1} s_0$ and $\langle S_2, s_0\rangle \Rightarrow^{k_2} s_2$, and $k = k_1 + k_2$.*

Proof : By induction on the length (called n) of derivation sequences.

$(n = 1)$: If $\langle S_1\,;S_2, s_1\rangle \Rightarrow^1 s_2$, then $\langle S_1\,;S_2, s_1\rangle \Rightarrow s_2$, so there should be a derivation showing this result. By inspecting the rules, it becomes clear that this is not possible; of all the rules, only $(\textsc{ass}_{sos})$ and $(\textsc{skip}_{sos})$ end with a single state, and neither is of the shape $S_1\,;S_2$. So the result holds vacuously.

$(n > 1)$: If $\langle S_1\,;S_2, s_1\rangle \Rightarrow^n s_2$, then $\langle S_1\,;S_2, s_1\rangle \Rightarrow \langle S_3, s_3\rangle \Rightarrow^{n-1} s_2$, for some S_3 and s_3. By inspecting the rules, there are just two possibilities for the last rule applied in the derivation for the first step.

$(\textsc{comp}^{\text{I}}_{sos})$: So the bottom-end of the derivation looks like:

$$\frac{\langle S_1, s_1\rangle \Rightarrow \langle S_1', s_3\rangle}{\langle S_1\,;S_2, s_1\rangle \Rightarrow \langle S_1'\,;S_2, s_3\rangle}$$

and $S_3 = S_1'\,;S_2$, for some S_1'. In particular, we have $\langle S_1'\,;S_2, s_3\rangle \Rightarrow^{n-1} s_2$. So, by induction, there are s_0, n_1 and n_2 such that

$$\langle S_1', s_3\rangle \Rightarrow^{n_1} s_0,\ \langle S_2, s_0\rangle \Rightarrow^{n_2} s_2, \text{ and } n{-}1 = n_1 + n_2.$$

Then also $\langle S_1, s_1\rangle \Rightarrow \langle S_1', s_3\rangle \Rightarrow^{n_1} s_0$, so $\langle S_1, s_1\rangle \Rightarrow^{n_1+1} s_0$. And we are done: notice that $n_1 + 1 + n_2 = n$.

$(\textsc{comp}^{\text{T}}_{sos})$: Then the derivation ends like:

$$\frac{\langle S_1, s_1\rangle \Rightarrow s_3}{\langle S_1\,;S_2, s_1\rangle \Rightarrow \langle S_2, s_3\rangle}$$

and we are done: take $n_1 = 1$, and $n_2 = n{-}1$.

We say that statements S_1 and S_2 are *semantically equivalent* if for all states s:

- $\langle S_1, s\rangle \Rightarrow^* \gamma$ if and only if $\langle S_2, s\rangle \Rightarrow^* \gamma$ whenever γ is a configuration that is either stuck or terminal, and

- there is an infinite derivation sequence for S_1 in s if and only if there is one for S_2 in s.

Note that, in the first case, the number of steps in the derivation sequence may be different.

As before, the meaning of statements can now be summarised as a (partial) function from *State* to *State*.

$$\begin{array}{rcll} \mathcal{S}_{sos} & : & \mathit{Statements} \to \mathit{State} \hookrightarrow \mathit{State} & \\ \\ \mathcal{S}_{sos}\,[\![S]\!]\,s & = & s', & \text{if } \langle S, s\rangle \Rightarrow^* s' \\ & = & \mathit{undefined}, & \text{otherwise} \end{array}$$

We can now express the equivalence of the Natural and Structural Operational semantics.

Theorem 5.7 *For every statement S of* `While`*, $\mathcal{S}_{ns}\,[\![S]\!] = \mathcal{S}_{sos}\,[\![S]\!]$, i.e., they are equal as functions, so, for every state s, $\mathcal{S}_{ns}\,[\![S]\!]\,s = \mathcal{S}_{sos}\,[\![S]\!]\,s$.*

In other words: both terminate in the same final state, or both loop. This result follows from the following lemmae.

Lemma 5.8 If $\langle S_1, s_1\rangle \Rightarrow^k s_2$, then $\langle S_1\,\mathtt{;}\,S_2, s_1\rangle \Rightarrow^k \langle S_2, s_2\rangle$.

Proof : Induction over the structure of the derivation for $\langle S_1, s_1\rangle \Rightarrow^k s_2$.

Lemma 5.9 For every statement S and states s_1 and s_2, $\langle S, s_1\rangle \to s_2$ implies $\langle S, s_1\rangle \Rightarrow^* s_2$.

So, if the execution of S starting from state s terminates in state s' in the Natural Semantics, it will terminate in the same state in the Structural Operational Semantics.

Proof : By induction over the shape of the derivation tree for $\langle S, s_1\rangle \to s_2$. We only show one case, the others follow by straightforward induction.

(COMP_{ns}) : $\langle S_1\,\mathtt{;}\,S_2, s_1\rangle \to s_2$ because $\langle S_1, s_1\rangle \to s_3$ and $\langle S_2, s_3\rangle \to s_2$. Then, by induction (twice), we obtain $\langle S_1, s_1\rangle \Rightarrow^* s_3$ and $\langle S_2, s_3\rangle \Rightarrow^* s_2$, so $\langle S_1\,\mathtt{;}\,S_2, s_1\rangle \Rightarrow^* \langle S_2, s_3\rangle$ by Lemma 5.8, and the result $\langle S_1\,\mathtt{;}\,S_2, s_1\rangle \Rightarrow^* s_2$ then follows from the definition of $\Rightarrow^*$.

Exercise 5.10 Finish this proof.

Lemma 5.11 For every statement S, and states s_1 and s_2, and natural number k, $\langle S, s_1\rangle \Rightarrow^k s_2$ implies $\langle S, s_1\rangle \to s_2$.

So, if the execution of S from s_1 terminates in the structural operational semantics, it will terminate in the same state in the natural semantics.

Proof : By induction on the length ($\geq$ 1) of the derivation sequence $\langle S, s_1\rangle \Rightarrow^* s_2$. We focus on the first step of $\langle S, s_1\rangle \Rightarrow^{k+1} s_2$. Let S_3, s_3 be such that $\langle S, s_1\rangle \Rightarrow \langle S_3, s_3\rangle$ and $\langle S_3, s_3\rangle \Rightarrow^k s_2$. Then, by induction, we can assume that $\langle S_3, s_3\rangle \to s_2$.

Since $\langle S, s_1\rangle \Rightarrow \langle S_3, s_3\rangle$, by definition of $\langle\cdot,\cdot\rangle \Rightarrow \cdot$, there is a derivation D that shows this result. We will complete the proof by cases on the structure of S, analysing, if necessary, what D must be.

$(k = 1)$: There are only two cases to consider here.

$(S = (x \mathrel{\mathtt{:=}} a))$: Then D consists of rule (ASS_{sos}), and $s_2 = s_1[x \mapsto \mathcal{A}\,[\![a]\!]\,s]$. Then we are done, since $\langle x \mathrel{\mathtt{:=}} a, s_1\rangle \to s_1[x \mapsto \mathcal{A}\,[\![a]\!]\,s]$.

$(S = \texttt{skip})$: Then D consists of rule (SKIP_{sos}), and $s_2 = s_1$. Notice that $\langle \texttt{skip}, s_1 \rangle \to s_1$.

$(k > 1)$: The remaining cases are:

$(S = (S^1\,\texttt{;}\,S^2))$: By Theorem 5.6 $\langle S^1\,\texttt{;}\,S^2, s_1 \rangle \Rightarrow^k s_2$ implies that there exists a state s_0 and natural numbers k_1 and k_2 (both ≥ 1) such that

$$\langle S^1, s_1 \rangle \Rightarrow^{k_1} s_0 \text{ and } \langle S^2, s_0 \rangle \Rightarrow^{k_2} s_2, \text{ and } k = k_1 + k_2.$$

Then, by induction, $\langle S^1, s_1 \rangle \to s_0$ and $\langle S^2, s_0 \rangle \to s_2$, so also $\langle S^1\,\texttt{;}\,S^2, s_1 \rangle \to s_2$, by rule (COMP_{ns}).

$(S = (\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2))$: We have to distinguish two cases: $\mathcal{B}[\![b]\!]\,s = \mathbf{tt}$, or $\mathcal{B}[\![b]\!]\,s = \mathbf{ff}$. In the first case, D finishes with rule $(\text{COND}^{\text{T}}_{sos})$, so $\langle S_1, s_1 \rangle \Rightarrow^k s_2$, $S_3 = S_1$, and $s_3 = s_1$. Since, by induction, $\langle S_1, s_1 \rangle \to s_2$, by rule $(\text{COND}^{\text{T}}_{ns})$, also

$$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s_1 \rangle \to s_2.$$

The second case, $\mathcal{B}[\![b]\!]\,s = \mathbf{ff}$, is similar.

$(S = (\texttt{while}\ b\ \texttt{do}\ S_1))$: Then D consists of nothing but rule (WHILE_{sos}), and

$$\langle S_3, s_3 \rangle \quad = \quad \langle \texttt{if}\ b\ \texttt{then}\ (S_1\,\texttt{;}\,\texttt{while}\ b\ \texttt{do}\ S_1)\ \texttt{else}\ \texttt{skip}, s_3 \rangle \Rightarrow^k s_2,$$

and, by induction,

$$\langle \texttt{if}\ b\ \texttt{then}\ (S_1\,\texttt{;}\,\texttt{while}\ b\ \texttt{do}\ S_1)\ \texttt{else}\ \texttt{skip}, s_1 \rangle \to s_2.$$

We have seen before that the statements '$\texttt{if}\ b\ \texttt{then}\ (S_1\,\texttt{;}\,\texttt{while}\ b\ \texttt{do}\ S_1)\ \texttt{else}\ \texttt{skip}$' and '$\texttt{while}\ b\ \texttt{do}\ S_1$' are equivalent for Natural Semantics. So also

$$\langle \texttt{while}\ b\ \texttt{do}\ S_1, s_1 \rangle \to s_2.$$

Now we can come back to the proof of the theorem: For an arbitrary statement S and state s it follows from Lemma 5.9 and 5.11 that if $\mathcal{S}_{ns}[\![S]\!]\,s_1 = s_2$, then $\mathcal{S}_{sos}[\![S]\!]\,s_1 = s_2$, and vice versa. Thus $\mathcal{S}_{ns}[\![S]\!] = \mathcal{S}_{sos}[\![S]\!]$. So, if one is defined on a state, then so is the other, and, therefore, if one is not defined on a state, then neither is the other.

NB: This proof approach would need modification for a non-deterministic language.

6 Provably correct implementation

We will now focus on the definition of an abstract machine (language), and the definition of a translation (compilation) of programs in `While` to programs for this abstract machine. The goal of this new approach will be to define a new semantics for `While` *via* this translation, and prove that this translation respects both the Natural and Structural Operational Semantics we have seen before. In this way, we will prove the 'compiler' defined here to be correct.

$$\begin{tikzcd}
\texttt{While} \arrow[d, "\text{Compilation}"] \\
\text{Assembly language}
\end{tikzcd}$$

The steps we will focus on are:

- Define the meaning of the abstract machine instructions.
- Define the translation functions.

In this context, the problem of correctness is formulated by: if we translate a program written in `While` into a program written in the abstract machine language and execute that code on the abstract machine, we get the same result - a state - as was specified by $\mathcal{S}_{ns}$ or $\mathcal{S}_{sos}$.

The abstract machine **AM** has configurations $\langle c, e, s\rangle$ where

- c is the *code* (sequence of instructions) to be executed,
- e is the *evaluation stack*, and
- s is the *storage*.

The evaluation stack is a new feature that can be seen as an infinite list; its elements are either in $\mathbb{Z}$ or in $\mathbb{T}$, and therefore the stack is a semantic entity. It is used to evaluate arithmetic and boolean expressions:

$$e \quad \in \quad \mathit{Stack} \quad \equiv \quad (\mathbb{Z} \cup \mathbb{T})^*.$$

The idea is that, while establishing the semantics of expressions, you put the semantics of subexpressions onto the stack; it is used to store intermediate results. We also have $s \in \mathit{State}$, as before.

6.1 The abstract machine language

The definition of the abstract machine programming language is very much like the definition of `While` above. The difference lies in the fact that the statements themselves are more 'primitive', e.g. they explicitly deal with stack handling. The syntactic categories are:

$$\begin{array}{rcll} \mathit{inst} & \in & \mathit{Machine\ Instructions} & \\ c & \in & \mathit{Code} & \text{sequence of instructions.} \end{array}$$

The abstract syntax is defined by:

$$\begin{array}{rcl} \mathit{inst} & ::= & \textbf{PUSH}{-}n \mid \textbf{ADD} \mid \textbf{MULT} \mid \textbf{SUB} \mid \textbf{TRUE} \mid \textbf{FALSE} \mid \textbf{EQ} \mid \textbf{LE} \mid \textbf{AND} \mid \textbf{NEG} \mid \\ & & \textbf{FETCH}{-}x \mid \textbf{STORE}{-}x \mid \textbf{NOOP} \mid \textbf{BRANCH}\,(c,c) \mid \textbf{LOOP}\,(c,c) \\ c & ::= & \epsilon \mid \mathit{inst} : c \end{array}$$

Notice that, for every n there is an instruction **PUSH**$-n$, and, likewise, there are many **FETCH**$-x$ and **STORE**$-x$ instructions.

As mentioned above, we will define our machine semantics (translation from programs in `While` to programs in *Code*) using the following configurations:

$$\langle c, e, s\rangle \quad \in \quad \mathit{Code} \times \mathit{Stack} \times \mathit{State}$$

We call $\langle c, e, s\rangle$ terminal if $c \equiv \epsilon$, i.e. if there is no more instruction to execute.

The transition relation $\triangleright$ is defined over pairs of configurations

$$\langle c, e, s\rangle \quad \triangleright \quad \langle c', e', s'\rangle$$

and specifies how to execute instructions. (Notice that, in these configurations, only the first component c is syntactic: e is a list in $\mathbb{Z} \cup \mathbb{T}$, which is semantic, and s is a mapping from syntactic to semantic entities.) It is formally defined by:

$$\begin{array}{rcll}
\langle \mathbf{PUSH}{-}n : c, e, s\rangle & \triangleright & \langle c, \mathcal{N}[\![n]\!] : e, s\rangle & \\
\langle \mathbf{ADD} : c, z_1 : z_2 : e, s\rangle & \triangleright & \langle c, z_1 + z_2 : e, s\rangle & \\
\langle \mathbf{SUB} : c, z_1 : z_2 : e, s\rangle & \triangleright & \langle c, z_1 - z_2 : e, s\rangle & \\
\langle \mathbf{MULT} : c, z_1 : z_2 : e, s\rangle & \triangleright & \langle c, z_1 \times z_2 : e, s\rangle & \\
\langle \mathbf{TRUE} : c, e, s\rangle & \triangleright & \langle c, \mathbf{tt} : e, s\rangle & \\
\langle \mathbf{FALSE} : c, e, s\rangle & \triangleright & \langle c, \mathbf{ff} : e, s\rangle & \\
\langle \mathbf{EQ} : c, z_1 : z_2 : e, s\rangle & \triangleright & \langle c, (z_1 = z_2) : e, s\rangle & \text{if } z_1, z_2 \in \mathbb{Z} \\
\langle \mathbf{LE} : c, z_1 : z_2 : e, s\rangle & \triangleright & \langle c, (z_1 \leq z_2) : e, s\rangle & \text{if } z_1, z_2 \in \mathbb{Z} \\
\langle \mathbf{AND} : c, b_1 : b_2 : e, s\rangle & \triangleright & \langle c, (b_1 \;\&\; b_2) : e, s\rangle & \text{if } b_1, b_2 \in \mathbb{T} \\
\langle \mathbf{NEG} : c, b : e, s\rangle & \triangleright & \langle c, \neg\, b : e, s\rangle & \text{if } b \in \mathbb{T} \\
\langle \mathbf{FETCH}{-}x : c, e, s\rangle & \triangleright & \langle c, (s\,x) : e, s\rangle & \\
\langle \mathbf{STORE}{-}x : c, z : e, s\rangle & \triangleright & \langle c, e, s[x \mapsto z]\rangle & \\
\langle \mathbf{NOOP} : c, e, s\rangle & \triangleright & \langle c, e, s\rangle & \\
\langle \mathbf{BRANCH}\,(c_1, c_2) : c, b : e, s\rangle & \triangleright & \langle c_1 : c, e, s\rangle & \text{if } b = \mathbf{tt} \\
\langle \mathbf{BRANCH}\,(c_1, c_2) : c, b : e, s\rangle & \triangleright & \langle c_2 : c, e, s\rangle & \text{if } b = \mathbf{ff} \\
\langle \mathbf{LOOP}\,(c_1, c_2) : c, e, s\rangle & \triangleright & \langle c_1 : \mathbf{BRANCH}\,(c_2, \mathbf{LOOP}\,(c_1, c_2), \mathbf{NOOP}) : c, e, s\rangle &
\end{array}$$

The last rule needs some commments. The **LOOP**-instruction has two operands, that intentionally are: a boolean c_1 and a statement c_2 (notice that the syntax is not specific here; in fact, also $\mathbf{LOOP}\,(1, \mathbf{TRUE})$ is a correct program, although it will not execute much). As can be expected, the boolean decides on whether or not the statement will be executed. However, the boolean c_1 needs to be evaluated first, before, using **BRANCH**, we can decide to execute c_2 or not. So we put c_1 at the head of the code, which forces its evaluation on the stack. By the time that is finished, **BRANCH** will find the semantic representation of c_1 (either $\mathbf{tt}$ or $\mathbf{ff}$) on top of the stack, and will act accordingly.

Since the definition of $\triangleright$ follows the (intended) execution of the programs written in the abstract machine language, the definition of $\triangleright$ gives an SOS-style semantics for programs in **AM**. We can define a *computation sequence* by analogy to a derivation sequence.

We will use the convention that initial configurations always have an empty evaluation stack.

Example 6.1 Take the code

$$\mathbf{PUSH}{-}1 : \mathbf{FETCH}{-}x : \mathbf{ADD} : \mathbf{STORE}{-}x$$

and s such that $s\,x = 3$, then

$$\begin{array}{rcl}
\langle \textbf{PUSH}{-}1 : \textbf{FETCH}{-}x : \textbf{ADD} : \textbf{STORE}{-}x, \epsilon, s\rangle & \triangleright & \\
\langle \textbf{FETCH}{-}x : \textbf{ADD} : \textbf{STORE}{-}x, 1, s\rangle & \triangleright & \\
\langle \textbf{ADD} : \textbf{STORE}{-}x, 3 : 1, s\rangle & \triangleright & \\
\langle \textbf{STORE}{-}x, 4, s\rangle & \triangleright & \langle \epsilon, \epsilon, s[x \mapsto 4]\rangle
\end{array}$$

Notice that, as for the Natural and the Structural Operational Semantics, some configuration sequences are infinite: **LOOP**(**TRUE**, **NOOP**) is non-terminating. On the other hand, it is now possible for a configuration to not fit any of the rules: for example, $\langle \textbf{ADD}, \epsilon, s\rangle$ is stuck. But then again, we will only be interested in code obtained by translating programs, so this particular problem never appears.

Exercise 6.2 Specify how to generate code for a **for**-loop for the abstract machine, **AM**. You may have to extend the instruction set; you should specify the semantics of any new instructions.

Exercise 6.3 **AM** refers to variables by their names rather than by their address. The abstract machine **AM**$'$ differs from **AM** in that

i) The configurations have the form $\langle c, e, m\rangle$ where m, the memory, is a finite list of values;

ii) The instructions **FETCH**$-x$ and **STORE**$-x$ are replaced by instructions **GET**$-n$ and **PUT**$-n$ where n is a natural number (an address).

Specify the operational semantics of the machine. You may write $m[n]$ to select the n-th value in the list m (where n is positive but less than or equal to the length of m). What happens if we reference an address that is outside the memory?

By analogy with the SOS-style semantics of `While` we can prove properties of **AM** by induction on the length of computation sequences.

Lemma 6.4 If $\langle c_1, e_1, s\rangle \triangleright^k \langle c', e', s'\rangle$, *then* $\langle c_1 : c_2, e_1 : e_2, s\rangle \triangleright^k \langle c' : c_2, e' : e_2, s'\rangle$

Proof : By cases, using the definition of $\triangleright$. Notice that none of the cases in the definition of $\triangleright$ specify anything but the first instruction in c_1.

(**PUSH**$-n$) : Since

$$\langle \textbf{PUSH}{-}n : c_1, e_1, s\rangle \triangleright \langle c_1, \mathcal{N}[\![n]\!] : e_1, s_1\rangle$$

by the first line of the definition, by that same line also

$$\langle \textbf{PUSH}{-}n : c_1 : c_2, e_1 : e_2, s\rangle \triangleright \langle c_1 : c_2, \mathcal{N}[\![n]\!] : e_1 : e_2, s_1\rangle.$$

The other cases are similar.

Lemma 6.5 If $\langle c_1 : c_2, e_1, s_1\rangle \triangleright^k \langle \epsilon, e_3, s_3\rangle$, *then there are* s_2, e_2, k_1 *and* k_2 *such that* $k_1 + k_2 = k$, *and* $\langle c_1, e_1, s_1\rangle \triangleright^{k_1} \langle \epsilon, e_2, s_2\rangle$ *and* $\langle c_2, e_2, s_2\rangle \triangleright^{k_2} \langle \epsilon, e_3, s_3\rangle$.

Proof : (Notice that we have proven a similar result for Natural Semantics.) By induction on the length of computation sequences.

$(k = 2)$: Then both c_1 and c_2 are elementary instructions, and the proof follows from the definition of $\triangleright$, and $k_1 = k_2 = 1$.

$(k > 1)$: If $\langle c_1 : c_2, e_1, s_1\rangle \rhd^k \langle \epsilon, e_3, s_3\rangle$, then there are c_1', e_1', s_1' such that

$$\langle c_1 : c_2, e_1, s_1\rangle \rhd \langle c_1' : c_2, e_1', s_1'\rangle, \text{ and } \langle c_1' : c_2, e_1', s_1'\rangle \rhd^{k-1} \langle \epsilon, e_3, s_3\rangle.$$

Then, by induction, there are s_2', e_2', k_1', k_2' such that $k_1' + k_2' = k-1$, and

$$\langle c_1', e_1', s_1'\rangle \rhd^{k_1'} \langle \epsilon, e_2', s_2'\rangle \text{ and } \langle c_2, e_2', s_2'\rangle \rhd^{k_2'} \langle \epsilon, e_3, s_3\rangle.$$

Then, by definition of $\rhd$, we also have:

$$\langle c_1, e_1, s_1\rangle \rhd \langle c_1', e_1', s_1'\rangle, \text{ so also } \langle c_1, e_1, s_1\rangle \rhd^{\cdot} \langle \epsilon, e_2', s_2'\rangle$$

So take $k_1 = k_1'+1$, and $k_2 = k_2'$, and $e_2 = e_2', s_2 = s_2'$.

Theorem 6.6 *The machine semantics is deterministic:*
For all $\gamma_1, \gamma_2, \gamma_3$, *if* $\gamma_1 \rhd \gamma_2$ *and* $\gamma_1 \rhd \gamma_3$, *then* $\gamma_2 = \gamma_3$.

Proof : Exercise.

Using the definitions and results above, we can now approach the main result for the **AM**. To this end, we will first define a semantics for elements of *Code*, define a suitable translation of programs in `While` to instructions in *Code*, and show that the translation is correct, in the sense that both a program in `While` and its translation in *Code* are mapped onto the same state. The meaning of a sequence of instructions can be expressed as a partial function from *State* to *State*.

$$\mathcal{M}_{sos} \quad : \quad \textit{Code} \to \textit{State} \hookrightarrow \textit{State}$$

$$\begin{array}{rcll} \mathcal{M}\,[\![c]\!]\,s & = & s', & \text{if } \langle c, \epsilon, s\rangle \rhd \langle \epsilon, \epsilon, s'\rangle \\ & = & \textit{undefined}, & \text{otherwise} \end{array}$$

6.2 Translation of expressions

To define our translation, we start by defining the translation of *Arithmetic Expressions* into *Code*:

$$\mathcal{CA} \quad : \quad \textit{Arithmetic Expressions} \to \textit{Code}$$

$$\begin{array}{rcl} \mathcal{CA}\,[\![n]\!] & = & \textbf{PUSH}{-}n \\ \mathcal{CA}\,[\![x]\!] & = & \textbf{FETCH}{-}x \\ \mathcal{CA}\,[\![a_1 + a_2]\!] & = & \mathcal{CA}\,[\![a_2]\!] : \mathcal{CA}\,[\![a_1]\!] : \textbf{ADD} \\ \mathcal{CA}\,[\![a_1 - a_2]\!] & = & \mathcal{CA}\,[\![a_2]\!] : \mathcal{CA}\,[\![a_1]\!] : \textbf{SUB} \\ \mathcal{CA}\,[\![a_1 \times a_2]\!] & = & \mathcal{CA}\,[\![a_2]\!] : \mathcal{CA}\,[\![a_1]\!] : \textbf{MULT} \end{array}$$

Notice the inversion of a_1 and a_2.

Next, we define the translation of *Boolean Expressions* into *Code*:

$$\mathcal{CB} \quad : \quad \textit{Boolean Expressions} \to \textit{Code}$$

$$\begin{array}{rcl}
\mathcal{CB}[\![\texttt{true}]\!] & = & \texttt{TRUE} \\
\mathcal{CB}[\![\texttt{false}]\!] & = & \texttt{FALSE} \\
\mathcal{CB}[\![a_1 = a_2]\!] & = & \mathcal{CA}[\![a_2]\!] : \mathcal{CA}[\![a_1]\!] : \texttt{EQ} \\
\mathcal{CB}[\![a_1 \leq a_2]\!] & = & \mathcal{CA}[\![a_2]\!] : \mathcal{CA}[\![a_1]\!] : \texttt{LE} \\
\mathcal{CB}[\![\neg b]\!] & = & \mathcal{CB}[\![b]\!] : \texttt{NEG} \\
\mathcal{CB}[\![b_1 \ \& \ b_2]\!] & = & \mathcal{CB}[\![b_2]\!] : \mathcal{CB}[\![b_1]\!] : \texttt{AND}
\end{array}$$

6.3 Translation of statements

We finish by defining the translation of *Statements* into *Code*:

$$\mathcal{CS} \quad : \quad \textit{Statements} \rightarrow \textit{Code}$$

$$\begin{array}{rcl}
\mathcal{CS}[\![x \texttt{ := } a]\!] & = & \mathcal{CA}[\![a]\!] : \texttt{STORE}-x \\
\mathcal{CS}[\![\texttt{skip}]\!] & = & \texttt{NOOP} \\
\mathcal{CS}[\![S_1 \texttt{;} S_2]\!] & = & \mathcal{CS}[\![S_1]\!] : \mathcal{CS}[\![S_2]\!] \\
\mathcal{CS}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!] & = & \mathcal{CB}[\![b]\!] : \texttt{BRANCH}(\mathcal{CS}[\![S_1]\!], \mathcal{CS}[\![S_2]\!]) \\
\mathcal{CS}[\![\texttt{while } b \texttt{ do } S]\!] & = & \texttt{LOOP}(\mathcal{CB}[\![b]\!] : \mathcal{CS}[\![S]\!])
\end{array}$$

These definitions open the way to define a new semantics for elements of *Statements*. The meaning of a statement S can now be obtained by first translating it into code for **AM** and next executing the code on the abstract machine:

$$\mathcal{S}_{am} \quad : \quad \textit{Statements} \rightarrow \textit{State} \hookrightarrow \textit{State}$$

$$\mathcal{S}_{am} \quad = \quad \mathcal{M} \circ \mathcal{CS}$$

so $\mathcal{S}_{am}[\![S]\!] = \mathcal{M} \circ (\mathcal{CS}[\![S]\!])$.

Exercise 6.7 Use $\mathcal{CS}$ to generate code for the statement:

$$z \texttt{ := } 0 \texttt{;} \texttt{while } y \leq x \texttt{ do } (z \texttt{ := } z + 1 \texttt{;} x \texttt{ := } x - y)$$

Trace the computation of the code starting from a storage where x has value 17 and y has value 5.

Exercise 6.8 Let **AM**′ be as in Exercise 6.3. Modify the code generation so as to translate `While` into code for the abstract machine **AM**′. You may assume the existence of a function *env* : *Variables* $\rightarrow \mathbb{N}$ that maps variables to their addresses.

Exercise 6.9 Prove that the code generated for **AM**′ by the previous exercise is correct. What assumptions do you need to make about *env*?

6.4 Correctness of translation

Correctness for the machine semantics of elements of *Statements* is expressed by:

$$\mathcal{S}_{ns} \quad = \quad \mathcal{S}_{am}$$

or

$$\mathcal{S}_{sos} \quad = \quad \mathcal{S}_{am}$$

These two results need proof, and express that, if we first translate a statement into code for **AM** and the execute that code, we must obtain the same result as specified by the operational semantics for `While`.

In proving this result, we will deal with expressions and statements separately: we will first show

$$\begin{array}{ccc} \langle \mathcal{CA}[\![a]\!], \epsilon, s\rangle & \rhd^* & \langle \epsilon, \mathcal{A}[\![a]\!]\,s, s\rangle \\ & \text{and} & \\ \langle \mathcal{CB}[\![b]\!], \epsilon, s\rangle & \rhd^* & \langle \epsilon, \mathcal{B}[\![b]\!]\,s, s\rangle \end{array}$$

and use these results to show

$$\mathcal{S}_{ns}[\![S]\!] \quad = \quad \mathcal{S}_{am}[\![S]\!]$$

6.5 Correctness for expressions

Correctness of the translation from `While` into the Abstract Machine Language, restricted to the set of *Arithmetic Expressions* is formulated by:

Theorem 6.10 $\langle \mathcal{CA}[\![a]\!], \epsilon, s\rangle \quad \rhd^* \quad \langle \epsilon, \mathcal{A}[\![a]\!]\,s, s\rangle$

Proof : By induction on the structure of arithmetic expressions. We illustrate three cases:

i) $a \equiv n$. Then $\mathcal{CA}[\![n]\!] = \textbf{PUSH}{-}n$. Notice that

$$\langle \textbf{PUSH}{-}n, \epsilon, s\rangle \quad \rhd \quad \langle \epsilon, \mathcal{N}[\![n]\!], s\rangle$$

and that $\mathcal{A}[\![n]\!]\,s = \mathcal{N}[\![n]\!]$. So we're done.

ii) $a \equiv x$. Then $\mathcal{CA}[\![x]\!] = \textbf{FETCH}{-}x$. Notice that

$$\langle \textbf{FETCH}{-}x, \epsilon, s\rangle \quad \rhd \quad \langle \epsilon, s\,x, s\rangle$$

and that $\mathcal{A}[\![x]\!]\,s = s\,x$. So we're done.

iii) $a \equiv a_1 + a_2$. Then $\mathcal{CA}[\![a_1 + a_2]\!] = \mathcal{CA}[\![a_2]\!] : \mathcal{CA}[\![a_1]\!] : \textbf{ADD}$. By induction we can assume:

$$\begin{array}{ccl} \langle \mathcal{CA}[\![a_1]\!], \epsilon, s\rangle & \rhd^* & \langle \epsilon, \mathcal{A}[\![a_1]\!]\,s, s\rangle \text{ and} \\ \langle \mathcal{CA}[\![a_2]\!], \epsilon, s\rangle & \rhd^* & \langle \epsilon, \mathcal{A}[\![a_2]\!]\,s, s\rangle \end{array}$$

Using Lemma 6.4, we get:

$$\begin{array}{rcl} \langle \mathcal{CA}[\![a_1 + a_2]\!], \epsilon, s\rangle & = & \\ \langle \mathcal{CA}[\![a_2]\!] : \mathcal{CA}[\![a_1]\!] : \textbf{ADD}, \epsilon, s\rangle & \rhd^* & \textit{By Lemma 6.4 and induction} \\ \langle \mathcal{CA}[\![a_1]\!] : \textbf{ADD}, \mathcal{A}[\![a_2]\!]\,s, s\rangle & \rhd^* & \textit{By Lemma 6.4 and induction} \\ \langle \textbf{ADD}, \mathcal{A}[\![a_1]\!]\,s : \mathcal{A}[\![a_2]\!]\,s, s\rangle & \rhd & \\ \langle \epsilon, \mathcal{A}[\![a_1]\!]\,s + \mathcal{A}[\![a_2]\!]\,s, s\rangle & = & \langle \epsilon, \mathcal{A}[\![a_1 + a_2]\!]\,s, s\rangle \end{array}$$

Booleans are dealt with in a similar way.

6.6 Correctness for Statements

For every statement S of `While` we have to show:

$$\mathcal{S}_{ns} \quad = \quad \mathcal{S}_{am}.$$

This equality expresses two properties:

- If the execution of S from some state s terminates in one of the semantics, then it also terminates in the other semantics and the resulting states will be equal.

- Furthermore, if the execution of S from some state s loops in one of the semantics, if will also loop in the other.

We will start our proof by showing that, if the Natural Semantics for $\langle S, s_1\rangle$ is s_2, then also the Machine Semantics gives (i.e. after translation, also the obtained code evaluates to) a configuration with state s_2.

Lemma 6.11 For every statement S and states s_1 and s_2,

$$\text{if } \langle S, s_1\rangle \to s_2, \text{ then } \langle \mathcal{CS}\,[\![S]\!], \epsilon, s_1\rangle \rhd^* \langle \epsilon, \epsilon, s_2\rangle.$$

Proof : By induction on the structure of derivation trees in Natural Semantics. We cover three of the cases:
$(\textsc{ass}_{ns})$: Then $S = x \texttt{ := } a$, and $s_2 = s_1[x \mapsto \mathcal{A}\,[\![a]\!]\,s_1]$. By definition of $\mathcal{CS}$,

$$\mathcal{CS}\,[\![x \texttt{ := } a]\!] = \mathcal{CA}\,[\![a]\!] : \texttt{STORE}{-}x,$$

and by correctness of the translation for *Arithmetic Expressions,* Theorem 6.10 we have

$$\langle \mathcal{CA}\,[\![a]\!], \epsilon, s_1\rangle \rhd^* \langle \epsilon, \mathcal{A}\,[\![a]\!]\,s_1, s_1\rangle$$

Thus (using Lemma 6.4)

$$\begin{array}{rcl} \langle \mathcal{CA}\,[\![a]\!] : \texttt{STORE}{-}x, \epsilon, s_1\rangle & \rhd^* & \langle \texttt{STORE}{-}x, \mathcal{A}\,[\![a]\!]\,s_1, s_1\rangle \\ & \rhd & \langle \epsilon, \epsilon, s_1[x \mapsto \mathcal{A}\,[\![a]\!]\,s_1]\rangle \end{array}$$

and we're done.
$(\textsc{comp}_{ns})$: Then $S = S_1 \texttt{;} S_2$, $\langle S_1, s_1\rangle \to s_3$ and $\langle S_2, s_3\rangle \to s_2$, for some S_1, S_2, and s_3.
Now

$$\mathcal{CS}\,[\![S_1 \texttt{;} S_2]\!] = \mathcal{CS}\,[\![S_1]\!] : \mathcal{CS}\,[\![S_2]\!]$$

By induction, we can assume both

$$\langle \mathcal{CS}\,[\![S_1]\!], \epsilon, s_1\rangle \quad \rhd^* \quad \langle \epsilon, \epsilon, s_3\rangle$$

and

$$\langle \mathcal{CS}\,[\![S_2]\!], \epsilon, s_3\rangle \quad \rhd^* \quad \langle \epsilon, \epsilon, s_2\rangle$$

Thus (using Lemma 6.4)

$$\begin{array}{rcl} \langle \mathcal{CS}\,[\![S_1]\!] : \mathcal{CS}\,[\![S_2]\!], \epsilon, s_1\rangle & \rhd^* & \langle \mathcal{CS}\,[\![S_2]\!], \epsilon, s_3\rangle \\ & \rhd^* & \langle \epsilon, \epsilon, s_2\rangle \end{array}$$

and we're done.

($\text{WHILE}^{\text{T}}_{ns}$) : Then $S = \texttt{while}\ b\ \texttt{do}\ S_1$ and there exists an s_3 such that $\langle S_1, s_1\rangle \to s_3$ and $\langle \texttt{while}\ b\ \texttt{do}\ S_1, s_3\rangle \to s_2$. Since

$$\mathcal{CS}[\![\texttt{while}\ b\ \texttt{do}\ S_1]\!] = \texttt{LOOP}\,(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S_1]\!])$$

we can assume, using induction, that

$$\begin{array}{rcl} \langle \mathcal{CS}[\![S_1]\!], \epsilon, s_1\rangle & \rhd^* & \langle \epsilon, \epsilon, s_3\rangle \\ & \text{and} & \\ \langle \texttt{LOOP}\,(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S_1]\!]), \epsilon, s_3\rangle & \rhd^* & \langle \epsilon, \epsilon, s_2\rangle \end{array}$$

Using these two results, we aim to show that $\langle \texttt{LOOP}\,(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S_1]\!]), \epsilon, s_1\rangle \rhd^* \langle \epsilon, \epsilon, s_2\rangle$. Now

$$\begin{array}{ll} \multicolumn{2}{l}{\langle \texttt{LOOP}\,(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S_1]\!]), \epsilon, s_1\rangle} \\ \rhd & \langle \mathcal{CB}[\![b]\!] : \texttt{BRANCH}\,(\mathcal{CS}[\![S_1]\!] : \texttt{LOOP}\,(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S_1]\!]), \texttt{NOOP}), \epsilon, s_1\rangle \\ \rhd^* & \langle \texttt{BRANCH}\,(\mathcal{CS}[\![S_1]\!] : \texttt{LOOP}\,(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S_1]\!]), \texttt{NOOP}), \mathcal{B}[\![b]\!]\,s_1, s_1\rangle \\ \rhd & \langle \mathcal{CS}[\![S_1]\!] : \texttt{LOOP}\,(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S_1]\!]), \epsilon, s_1\rangle \end{array}$$

Now, using the results we can assume by induction, and Lemma 6.4, we get

$$\begin{array}{rcl} \langle \mathcal{CS}[\![S_1]\!] : \texttt{LOOP}\,(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S_1]\!]), \epsilon, s_1\rangle & \rhd^* & \\ \langle \texttt{LOOP}\,(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S_1]\!]), \epsilon, s_3\rangle & \rhd & \langle \epsilon, \epsilon, s_2\rangle. \end{array}$$

Exercise 6.12 Finish this proof.

In the second part of our proof we will show that, if the Machine Semantics of a configuration with a translation for statement S and state s_1 returns a configuration with state s_2, then also the Natural Semantics for $\langle S, s_1\rangle$ is s_2.

Lemma 6.13 For every statement S, states s_1, s_2: if $\langle \mathcal{CS}[\![S]\!], \epsilon, s_1\rangle \rhd^k \langle \epsilon, e, s_2\rangle$, then $\langle S, s_1\rangle \to s_2$.

Proof : By Complete Induction on the number of steps; we only show part of the proof.
(*Base Case*) : $(x \texttt{ := } a)$: Remember that $\mathcal{CS}[\![x \texttt{ := } a]\!] = \mathcal{CA}[\![a]\!] : \texttt{STORE}{-}x$, so

$$\langle \mathcal{CA}[\![a]\!] : \texttt{STORE}{-}x, \epsilon, s_1\rangle \rhd^k \langle \epsilon, e, s_2\rangle;$$

then, by Lemma 6.5, there are e_3, k_1, and k_2 such that

$$\begin{array}{rcl} \langle \mathcal{CA}[\![a]\!], \epsilon, s_1\rangle & \rhd^{k_1} & \langle \epsilon, e_3, s_3\rangle \\ \langle \texttt{STORE}{-}x, e_3, s_3\rangle & \rhd^{k_2} & \langle \epsilon, e, s_2\rangle \\ k & = & k_1 + k_2. \end{array}$$

But, by correctness of expressions, we have $e_3 = \mathcal{A}[\![a]\!]\,s_1$ and $s_3 = s_1$. Therefore, $s_2 = s_1[x \mapsto \mathcal{A}[\![a]\!]\,s_1]$ and $e = \epsilon$ by the definition of $\rhd$, case $\texttt{STORE}{-}x$. To conclude, notice that $\langle x \texttt{ := } a, s_1\rangle \to s_1[x \mapsto \mathcal{A}[\![a]\!]\,s_1]$.
(*Inductive Case*) : We distinguish the following cases for S:

$(\texttt{while}\ b\ \texttt{do}\ S)$: Notice that $\mathcal{CS}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] = \texttt{LOOP}\,(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S]\!])$, so

$$\langle \texttt{LOOP}\,(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S]\!]), \epsilon, s_1\rangle \rhd^k \langle \epsilon, e, s_2\rangle;$$

Then, the definition of $\rhd$, case **LOOP**:

$$\begin{array}{ll} \multicolumn{2}{l}{\langle \texttt{LOOP}(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S]\!]), \epsilon, s_1\rangle} \\ \rhd & \langle \mathcal{CB}[\![b]\!] : \texttt{BRANCH}\,(\mathcal{CS}[\![S]\!] : \texttt{LOOP}(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S]\!]), \texttt{NOOP}), e, s_1\rangle \\ \rhd^{k-1} & \langle \epsilon, e, s_2\rangle \end{array}$$

Again by Lemma 6.5, there are e_3, s_3, k_1, k_2 such that

$$\begin{array}{rcl} \langle \mathcal{CB}[\![b]\!], \epsilon, s_1\rangle & \rhd^{k_1} & \langle \epsilon, e_3, s_3\rangle \\ \langle \texttt{BRANCH}\,(\mathcal{CS}[\![S]\!] : \texttt{LOOP}(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S]\!]), \texttt{NOOP}), e_3, s_3\rangle & \rhd^{k_2} & \langle \epsilon, e, s_2\rangle \\ k-1 & = & k_1 + k_2 \end{array}$$

Thus $e_3 = \mathcal{B}[\![b]\!]\,s_1$, $s_3 = s_1$ and there are two cases to consider:
$(\mathcal{B}[\![b]\!]\,s_1 = \mathbf{ff})$: Notice that

$$\langle \texttt{NOOP}, \epsilon, s_1\rangle \quad \rhd \quad \langle \epsilon, \epsilon, s_1\rangle$$

So $e = \epsilon$ and $s_1 = s_2$. Using rule $(\textsc{while}^{\text{F}}_{ns})$, and that $\mathcal{B}[\![b]\!]\,s_1 = \mathbf{ff}$, we get the desired result: $\langle \texttt{while}\ b\ \texttt{do}\ S, s_1\rangle \to s_1$.
$(\mathcal{B}[\![b]\!]\,s = \mathbf{tt})$: Then

$$\begin{array}{ll} \multicolumn{2}{l}{\langle \texttt{BRANCH}\,(\mathcal{CS}[\![S]\!] : \texttt{LOOP}(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S]\!]), \texttt{NOOP}), \mathbf{tt}, s_1\rangle} \\ \rhd & \langle \mathcal{CS}[\![S]\!] : \texttt{LOOP}(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S]\!]), \epsilon, s_1\rangle \\ \rhd^{k_2-1} & \langle \epsilon, \epsilon, s_2\rangle \end{array}$$

Now, again by Lemma 6.5, we have e_3, s_3, k_3, and k_4 such that

$$\begin{array}{rcl} \langle \mathcal{CS}[\![S]\!], \epsilon, s_1\rangle & \rhd^{k_3} & \langle \epsilon, e_3, s_3\rangle \\ \langle \texttt{LOOP}(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S]\!]), e_3, s_3\rangle & \rhd^{k_2} & \langle \epsilon, e, s_2\rangle \\ k_2 & = & k_3 + k_4 \end{array}$$

Notice that both instructions are obtained by translation (the first from S, the second from $\texttt{while}\ b\ \texttt{do}\ S$), and that the computation sequences involved are of length smaller than k, so by induction (finally), we can assume

$$\langle S, s_1\rangle \to s_3 \text{ and } \langle \texttt{while}\ b\ \texttt{do}\ S, s_3\rangle \to s_2$$

The desired result follows from these two by applying rule $(\textsc{while}^{\text{T}}_{ns})$.

Exercise 6.14 Finish this proof, i.e. add the missing cases.

We could have used $\mathcal{S}_{sos}$ instead - the proof for equivalence might have been easier because both $\mathcal{S}_{sos}$ and $\mathcal{S}_{am}$ focus on single steps.

7 Denotational Semantics

In the operational approach we were interested in *how* a program is executed. In the denotational approach we are interested in *the effect*, in terms of functions of type ($\mathit{State} \hookrightarrow \mathit{State}$), of executing a program.

The basic idea is:

- Define a semantic function for each syntactic category - it maps each syntactic construct to a mathematical object (which describes the effect of executing the construct).

In Denotational Semantics, the semantic functions are defined *compositionally*:

- there is a semantic clause for each of the basic elements of the syntactic category.
- for each method of constructing a composite element there is a semantic clause defined in terms of the semantic function applied to the immediate constituents of the composite element.

We have seen examples of denotational semantics before: $\mathcal{A}$, $\mathcal{B}$, and non-examples: $\mathcal{S}_{ns}$, $\mathcal{S}_{sos}$.

When defining semantics for `While` as a function from *State* to *State*, certain cases are easy to deal with, and we can define the meaning of a statement S as a (partial) function from *State* to *State*:

$$\mathcal{S}_{ds} \ : \ \mathit{Statements} \to \mathit{State} \hookrightarrow \mathit{State}$$

$$\begin{array}{rcl}
\mathcal{S}_{ds}[\![x \texttt{ := } a]\!]\, s & = & s[x \mapsto \mathcal{A}[\![a]\!]\, s] \\
\mathcal{S}_{ds}[\![\texttt{skip}]\!] & = & \mathit{id} \\
\mathcal{S}_{ds}[\![S_1 \texttt{;} S_2]\!] & = & \mathcal{S}_{ds}[\![S_2]\!] \circ \mathcal{S}_{ds}[\![S_1]\!] \\
\mathcal{S}_{ds}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!] & = & \mathit{cond}\,(\mathcal{B}[\![b]\!]\,, \mathcal{S}_{ds}[\![S_1]\!], \mathcal{S}_{ds}[\![S_2]\!]) \\
\mathcal{S}_{ds}[\![\texttt{while } b \texttt{ do } S]\!] & = & \mathit{Problematic} \ldots
\end{array}$$

Before coming to formulating the problem behind the '$\mathcal{S}_{ds}[\![\texttt{while } b \texttt{ do } S]\!]$', we focus on some of the notation introduced. In the above definition, *id* is the identity function on states. Notice that $\mathcal{S}_{ds}$ is partial for the same reasons as hold for the partiality of $\mathcal{S}_{ns}$ and $\mathcal{S}_{sos}$; for this reason, the part above that deals with composition of statements has to deal with the fact that perhaps one of the two functions involved is not defined. This implies that, formally

$$\begin{array}{rcl}
\mathcal{S}_{ds}[\![S_1 \texttt{;} S_2]\!]\, s & = & (\mathcal{S}_{ds}[\![S_2]\!] \circ \mathcal{S}_{ds}[\![S_1]\!])\, s \\
& = & \mathcal{S}_{ds}[\![S_2]\!]\,(\mathcal{S}_{ds}[\![S_1]\!]\, s) \\
& = & \left\{ \begin{array}{ll}
s_2, & \text{if there exists an } s_1 \text{ such that } \mathcal{S}_{ds}[\![S_1]\!]\, s = s_1, \\
 & \text{and } \mathcal{S}_{ds}[\![S_2]\!]\, s_1 = s_2. \\
\mathit{undefined}, & \text{if } \mathcal{S}_{ds}[\![S_1]\!]\, s = \mathit{undefined}, \text{ or if there exists an } s_1 \\
 & \text{such that } \mathcal{S}_{ds}[\![S_1]\!]\, s = s_1, \text{ but} \\
 & \mathcal{S}_{ds}[\![S_2]\!]\, s_1 = \mathit{undefined}.
\end{array} \right.
\end{array}$$

The conditional function *cond* used above is defined very much like the **AM** instruction **`BRANCH`** we have seen before:

$$\mathit{cond} : (\mathit{State} \to \mathbb{T}) \times (\mathit{State} \hookrightarrow \mathit{State}) \times (\mathit{State} \hookrightarrow \mathit{State}) \to (\mathit{State} \hookrightarrow \mathit{State})$$

$$\mathit{cond}\,(g_1, g_2, g_3)\, s \quad = \quad \begin{cases} g_2\, s, & \text{if } g_1\, s = \mathbf{tt} \\ g_3\, s, & \text{if } g_1\, s = \mathbf{ff} \end{cases}$$

Notice that the denotational semantics for *Boolean Expressions* is a total function, i.e. is always defined; therefore, the meaning of a conditional is '*undefined*' if *either* the predicate is true, and $\mathcal{S}_{ds}\,[\![S_1]\!]\, s$ is '*undefined*', *or* the predicate is false and $\mathcal{S}_{ds}\,[\![S_2]\!]\, s$ is '*undefined*'.

The major task in the definition of Denotational Semantics for `While` is to define the effect of '**`while`** b **`do`** S'. In view of previous results, however we define it, it should be equal to the effect of '**`if`** b **`then`** $(S$ **`;`** **`while`** b **`do`** $S)$ **`else`** **`skip`**' and therefore:

$$\begin{aligned}
\mathcal{S}_{ds}\,[\![\texttt{while}\ b\ \texttt{do}\ S]\!] &= \mathcal{S}_{ds}\,[\![\texttt{if}\ b\ \texttt{then}\ (S\,\texttt{;}\,\texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else}\ \texttt{skip}]\!] \\
&= \mathit{cond}\,(\mathcal{B}\,[\![b]\!]\,, \mathcal{S}_{ds}\,[\![S\,\texttt{;}\,\texttt{while}\ b\ \texttt{do}\ S]\!], \mathcal{S}_{ds}\,[\![\texttt{skip}]\!]) \\
&= \mathit{cond}\,(\mathcal{B}\,[\![b]\!]\,, \mathcal{S}_{ds}\,[\![\texttt{while}\ b\ \texttt{do}\ S]\!] \circ \mathcal{S}_{ds}\,[\![S]\!], \mathit{id})
\end{aligned}$$

Notice that this analysis does not brings us any closer to understanding what the denotational semantics of '**`while`** b **`do`** S' is supposed to be: $\mathcal{S}_{ds}\,[\![\texttt{while}\ b\ \texttt{do}\ S]\!]$ appears in *exactly the same form* in the expression on the right, and is not expressed, for example, as the composition of already defined functions. The main problem to tackle is to define a notion of function and to set up the right mathematical context to give a solution for this equation.

Let's have a closer look at the above expression. Assume that the denotational semantics for '**`while`** b **`do`** S' is a function f. Then the equation above expresses that this f should *at least* satisfy:

$$f \quad = \quad \mathit{cond}\,(\mathcal{B}\,[\![b]\!]\,, f \circ \mathcal{S}_{ds}\,[\![S]\!], \mathit{id})$$

Focus on the right-hand term of this equation. Replace f by x and notice that we can now define a functional F as follows:

$$F x \quad = \quad \mathit{cond}\,(\mathcal{B}\,[\![b]\!]\,, x \circ \mathcal{S}_{ds}\,[\![S]\!], \mathit{id})$$

We can see that the function f we are looking for is a special kind of input for F: when we apply F to f, we get f!

$$F f \quad = \quad \mathit{cond}\,(\mathcal{B}\,[\![b]\!]\,, f \circ \mathcal{S}_{ds}\,[\![S]\!], \mathit{id}) \quad = \quad f$$

This special property has a name: we call a point y a *fixed point* of a function h, if $h\, y = y$. So, to reformulate the above:

$\mathcal{S}_{ds}\,[\![\texttt{while}\ b\ \texttt{do}\ S]\!]$ is a fixed point of F, where $F f = \mathit{cond}\,(\mathcal{B}\,[\![b]\!]\,, f \circ \mathcal{S}_{ds}\,[\![S]\!], \mathit{id})$.

Again, this property is just natural since *any* reasonable semantics should give the same meaning to '**`while`** b **`do`** S' and '**`if`** b **`then`** $(S$ **`;`** **`while`** b **`do`** $S)$ **`else`** **`skip`**'.

Example 7.1 Fixed points of functions do exist:

- Take $f x = 1$, with $f : \mathbb{Z} \to \mathbb{Z}$, then 1 is a fixed point of f.
- Take $f x = 2 \times x$, then 0 is a fixed point of f.

Exercise 7.2 Determine the functional F, associated with:

$$\texttt{while}\ \neg(x = 1)\ \texttt{do}\ (y \texttt{ := } y \times x\,\texttt{;}\, x \texttt{ := } x{-}1)$$

Determine at least two different fixed points for F.

We will develop a context where *every* function will have a fixed point, and even define a special function that, given an input function f, constructs the fixed point of f. We will call this function *Fix*. The intention is to construct fixed points for functions like F above, so there *Fix* would be of type:

$$\mathit{Fix} : (\mathit{State} \hookrightarrow \mathit{State}) \to (\mathit{State} \hookrightarrow \mathit{State}) \to (\mathit{State} \hookrightarrow \mathit{State})$$

and, of course, the intention is that *Fix* F is a fixed point of F:

$$F(\mathit{Fix}\, F) \quad = \quad \mathit{Fix}\, F$$

In general, the fixed point construction follows the following general scheme. Let f be defined by

$$f x = \textit{‘some expression in which’ } f \textit{ ‘appears’},$$

then we can define F by:

$$F g\, x = \textit{‘some expression in which’ } g \textit{ ‘appears’},$$

and the solution for the first equation is then *Fix* F.

The intended types for the functions mentioned are:

$$\begin{array}{rcll} f & : & \alpha{\to}\beta & (f \text{ is a function}) \\ F & : & (\alpha{\to}\beta){\to}\alpha{\to}\beta & \\ \mathit{Fix} & : & ((\alpha{\to}\beta){\to}\alpha{\to}\beta){\to}\alpha{\to}\beta & \end{array}$$

Unfortunately, this does not suffice:

- There are functionals which have *more than one* fixed point. For example, the function $f x = x$ has infinitely many fixed points, and $f x = e^x - 1$ has two.

- There are functionals which have *no* fixed points at all. For example, let $g_1 \neq g_2$, and define G by:

$$G\, g \quad = \quad \begin{cases} g_1, & \text{if } g = g_2 \\ g_2, & \text{otherwise} \end{cases}$$

Our solution to these two problems is:

- to impose requirements on the fixed points such that there is *at most one* fixed point satisfying them.

- to establish a framework such that every functional does have *at least one* fixed point satisfying the requirements.

7.1 Fixed Point Construction

Before we come to a formalisation of the problem and its solution, we again have a look at the problem. Remember that we want the denotational semantics to satisfy:

$$\mathcal{S}_{ds}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] = \mathit{cond}\,(\mathcal{B}[\![b]\!], \mathcal{S}_{ds}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] \circ \mathcal{S}_{ds}[\![S]\!], \mathit{id})$$

Therefore, $\mathcal{S}_{ds}[\![\texttt{while}\ b\ \texttt{do}\ S]\!]$ should be a function f such that (using the definitions above):

$$\begin{aligned} f\,s &= \mathit{cond}\,(\mathcal{B}[\![b]\!], f \circ \mathcal{S}_{ds}[\![S]\!], \mathit{id})\ s \\ &= \begin{cases} (f \circ \mathcal{S}_{ds}[\![S]\!])\ s, & \text{if } \mathcal{B}[\![b]\!]\,s = \mathbf{tt} \\ s, & \text{if } \mathcal{B}[\![b]\!]\,s = \mathbf{ff} \end{cases} \end{aligned}$$

We observed that, therefore, $\mathcal{S}_{ds}[\![\texttt{while}\ b\ \texttt{do}\ S]\!]$ should be a fixed point of F, where F is defined by:

$$F\,f\,s = \begin{cases} (f \circ \mathcal{S}_{ds}[\![S]\!])\ s, & \text{if } \mathcal{B}[\![b]\!]\,s = \mathbf{tt} \\ s, & \text{if } \mathcal{B}[\![b]\!]\,s = \mathbf{ff} \end{cases}$$

Example 7.3 Take the `While` program '**while** $\neg(x = 0)$ **do skip**'. The intended semantics for this program, using the construction discussed above, f, viewed as a function from *State* to *State*, has the following behaviour:

$$\begin{aligned} f\,s &= \mathit{cond}\,(\mathcal{B}[\![\neg(x = 0)]\!], f \circ \mathit{id}, \mathit{id})\ s \\ &= \mathit{cond}\,(\mathcal{B}[\![\neg(x = 0)]\!], f, \mathit{id})\ s \\ &= \begin{cases} f\,s, & \text{if } s\,x \neq 0 \\ s, & \text{if } s\,x = 0 \end{cases} \end{aligned}$$

Since $f\,s$ reappears on the right-hand side, we have to use a fixed point construction to find f. As suggested above, we write

$$F\,f\,s = \begin{cases} f\,s, & \text{if } s\,x \neq 0 \\ s, & \text{if } s\,x = 0 \end{cases}$$

Now, once we have a fixed point for F, we have a solution for our problem. Well, notice that

$$h\,s = \begin{cases} \mathit{undefined}, & \text{if } s\,x \neq 0 \\ s, & \text{if } s\,x = 0 \end{cases}$$

is a fixed point of F:

$$\begin{aligned} F\,h\,s &= \begin{cases} h\,s, & \text{if } s\,x \neq 0 \\ s, & \text{if } s\,x = 0 \end{cases} \\ &= \begin{cases} \mathit{undefined}, & \text{if } s\,x \neq 0 \\ s, & \text{if } s\,x = 0 \end{cases} \\ &= h\,s \end{aligned}$$

Notice that, if f is supposed to be a function of type *State* $\hookrightarrow$ *State*, then this F is of type $(\mathit{State} \hookrightarrow \mathit{State}) \to (\mathit{State} \hookrightarrow \mathit{State})$. In particular, Ff is a function of type *State* $\hookrightarrow$ *State*, so there is some f' such that $Ff = f'$. We can now apply F to f' to obtain f'', and so forth, ad infinitum.

We will look at a few example statements, and try to establish their semantics, guided by our intuition. The idea is to approach the final, looked-for solution, but *approximating* it. We will start with the first approximation f_0, the function that carries no information, i.e. is nowhere defined. We will use that function to build the second approximation f_1, and that to build the third f_2, etc., until we have reached the solution.

Example 7.4 Take '**`while true do skip`**'. Following the above definition, we get:

$$\begin{array}{rcl}
F\,f\,s & = & \mathit{cond}\,(\mathcal{B}\,[\![\texttt{true}]\!]\,, f \circ \mathcal{S}_{ds}\,[\![\texttt{skip}]\!], \mathit{id})\,s \\
 & = & \begin{cases} (f \circ \mathcal{S}_{ds}\,[\![\texttt{skip}]\!])\,s, & \text{if } \mathcal{B}\,[\![\texttt{true}]\!]\,s = \mathbf{tt} \\ s, & \text{if } \mathcal{B}\,[\![\texttt{true}]\!]\,s = \mathbf{ff} \end{cases} \\
 & = & \begin{cases} (f \circ \mathit{id})\,s, & \text{if } \mathbf{tt} = \mathbf{tt} \\ s, & \text{if } \mathbf{tt} = \mathbf{ff} \end{cases} \\
 & = & f\,s
\end{array}$$

Take $f_0\,s = \mathit{undefined}$, for all s. Notice that $F f_0 = f_0$, so f_0 is a fixed point of F. Moreover, f_0 is the *intended* semantics for '**`while true do skip`**', i.e. the semantics you would want it to have. Notice that, however, *all* functions are fixed points of F; therefore, we need a process that chooses f_0 amongst all these solutions: it should put the least restrictions on the resulting function, and select the 'most undefined function'.

Example 7.5 Take '**`while`** $x = 0$ **`do`** x `:=` 5'. Following the above definition, we get:

$$\begin{array}{rcl}
F\,f\,s & = & \mathit{cond}\,(\mathcal{B}\,[\![x = 0]\!]\,, f \circ \mathcal{S}_{ds}\,[\![x \texttt{ := } 5]\!], \mathit{id})\,s \\
 & = & \begin{cases} (f \circ \mathcal{S}_{ds}\,[\![x \texttt{ := } 5]\!])\,s, & \text{if } \mathcal{B}\,[\![x = 0]\!]\,s = \mathbf{tt} \\ s, & \text{if } \mathcal{B}\,[\![x = 0]\!]\,s = \mathbf{ff} \end{cases} \\
 & = & \begin{cases} f(s[x \mapsto 5]) & \text{if } s\,x = 0 \\ s, & \text{if } s\,x \neq 0 \end{cases}
\end{array}$$

Take $f_0\,s = \mathit{undefined}$, for all s. Then

$$\begin{array}{rcl}
f_1\,s = F\,f_0\,s & = & \begin{cases} f_0\,(s[x \mapsto 5]), & \text{if } s\,x = 0 \\ s, & \text{if } s\,x \neq 0 \end{cases} \\
 & = & \begin{cases} \mathit{undefined} & \text{if } s\,x = 0 \\ s, & \text{if } s\,x \neq 0 \end{cases}
\end{array}$$

and

$$\begin{array}{rcl} f_2\, s = F f_1\, s & = & \begin{cases} f_1\,(s[x \mapsto 5]), & \text{if } s\,x = 0 \\ s, & \text{if } s\,x \neq 0 \end{cases} \\ & = & \begin{cases} \left\{ \begin{array}{ll} \textit{undefined} & \text{if } s[x \mapsto 5]\,x = 0 \\ s[x \mapsto 5], & \text{if } s[x \mapsto 5]\,x \neq 0 \end{array} \right\} & \text{if } s\,x = 0 \\ s, & \text{if } s\,x \neq 0 \end{cases} \\ & = & \begin{cases} s[x \mapsto 5], & \text{if } s\,x = 0 \\ s, & \text{if } s\,x \neq 0 \end{cases} \end{array}$$

applying the construction again, we get:

$$\begin{array}{rcl} f_3\, s = F f_2\, s & = & \begin{cases} f_2\,(s[x \mapsto 5]), & \text{if } s\,x = 0 \\ s, & \text{if } s\,x \neq 0 \end{cases} \\ & = & \begin{cases} \left\{ \begin{array}{ll} s[x \mapsto 5][x \mapsto 5] & \text{if } s[x \mapsto 5]\,x = 0 \\ s[x \mapsto 5], & \text{if } s[x \mapsto 5]\,x \neq 0 \end{array} \right\} & \text{if } s\,x = 0 \\ s, & \text{if } s\,x \neq 0 \end{cases} \\ & = & \begin{cases} s[x \mapsto 5], & \text{if } s\,x = 0 \\ s, & \text{if } s\,x \neq 0 \end{cases} \\ & = & f_2\, s \end{array}$$

So $f_2 = f_3$, and, therefore, f_2 is a fixed point of F. Moreover, since '$\texttt{while}\ x = 0\ \texttt{do}\ x \mathrel{\texttt{:=}} 5$' is essentially the same as '$\texttt{if}\ x = 0\ \texttt{then}\ x \mathrel{\texttt{:=}} 5\ \texttt{else skip}$' which has exactly f_2 as semantics, we can see that f_2 is the intended semantics for '$\texttt{while}\ x = 0\ \texttt{do}\ x \mathrel{\texttt{:=}} 5$'.

Example 7.6 Now take '$\texttt{while}\ x > 0\ \texttt{do}\ x \mathrel{\texttt{:=}} x{-}1$'. Following the above definition, we get:

$$\begin{array}{rcl} F f s & = & \textit{cond}\,(\mathcal{B}\,[\![x > 0]\!]\,, f \circ \mathcal{S}_{ds}\,[\![x \mathrel{\texttt{:=}} x{-}1]\!], \textit{id})\, s \\ & = & \begin{cases} (f \circ \mathcal{S}_{ds}\,[\![x \mathrel{\texttt{:=}} x{-}1]\!])\, s, & \text{if } \mathcal{B}\,[\![x > 0]\!]\, s = \mathbf{tt} \\ s, & \text{if } \mathcal{B}\,[\![x > 0]\!]\, s = \mathbf{ff} \end{cases} \\ & = & \begin{cases} f(s[x \mapsto (s\,x - 1)]), & \text{if } s\,x > 0 \\ s, & \text{if } s\,x \leq 0 \end{cases} \end{array}$$

Take $f_0\, s = \textit{undefined}$, for all s. Then

$$\begin{array}{rcl} f_1 = F f_0 & = & \begin{cases} f_0\,(s[x \mapsto (s\,x - 1)]), & \text{if } s\,x > 0 \\ s, & \text{if } s\,x \leq 0 \end{cases} \\ & = & \begin{cases} \textit{undefined} & \text{if } s\,x > 0 \\ s, & \text{if } s\,x \leq 0 \end{cases} \end{array}$$

and

$$
\begin{aligned}
f_2\, s = F f_1\, s &= \begin{cases} f_1\,(s[x \mapsto (s\,x-1)]), & \text{if } s\,x > 0 \\ s, & \text{if } s\,x \leq 0 \end{cases} \\
&= \begin{cases} \left.\begin{cases} \textit{undefined} & \text{if } s[x \mapsto (s\,x-1)]\,x > 0 \\ s[x \mapsto (s\,x-1)], & \text{if } s[x \mapsto (s\,x-1)]\,x \leq 0 \end{cases}\right\} & \text{if } s\,x > 0 \\ s, & \text{if } s\,x \leq 0 \end{cases} \\
&= \begin{cases} \textit{undefined} & \text{if } s\,x > 1 \\ s[x \mapsto 0], & \text{if } s\,x = 1 \\ s, & \text{if } s\,x \leq 0 \end{cases}
\end{aligned}
$$

Applying the construction again, we get:

$$
\begin{aligned}
f_3 = F f_2 &= \begin{cases} f_2\, s[x \mapsto (s\,x-1)], & \text{if } s\,x > 0 \\ s, & \text{if } s\,x \leq 0 \end{cases} \\
&= \begin{cases} \left.\begin{cases} \textit{undefined} & \text{if } s[x \mapsto (s\,x-1)]\,x > 1 \\ s[x \mapsto (s\,x-1)][x \mapsto 0], & \text{if } s[x \mapsto (s\,x-1)]\,x = 1 \\ s[x \mapsto (s\,x-1)], & \text{if } s[x \mapsto (s\,x-1)]\,x \leq 0 \end{cases}\right\} & \text{if } s\,x > 0 \\ s, & \text{if } s\,x \leq 0 \end{cases} \\
&= \begin{cases} \textit{undefined} & \text{if } s\,x > 2 \\ s[x \mapsto 0], & \text{if } s\,x = 1, 2 \\ s, & \text{if } s\,x \leq 0 \end{cases}
\end{aligned}
$$

and

$$
\begin{aligned}
f_4 = F f_3 &= \begin{cases} f_3\, s[x \mapsto (s\,x-1)], & \text{if } s\,x > 0 \\ s, & \text{if } s\,x \leq 0 \end{cases} \\
&= \begin{cases} \left.\begin{cases} \textit{undefined} & \text{if } s[x \mapsto (s\,x-1)]\,x > 2 \\ s[x \mapsto (s\,x-1)][x \mapsto 0], & \text{if } s[x \mapsto (s\,x-1)]\,x = 1, 2 \\ s[x \mapsto (s\,x-1)], & \text{if } s[x \mapsto (s\,x-1)]\,x \leq 0 \end{cases}\right\} & \text{if } s\,x > 0 \\ s, & \text{if } s\,x \leq 0 \end{cases} \\
&= \begin{cases} \textit{undefined} & \text{if } s\,x > 2 \\ s[x \mapsto 0], & \text{if } s\,x = 1, 2, 3 \\ s, & \text{if } s\,x \leq 0 \end{cases}
\end{aligned}
$$

Continuing this construction, after the n-th step we get:

$$
f_n\, s = \begin{cases} \textit{undefined} & \text{if } s\,x > n \\ s[x \mapsto 0], & \text{if } s\,x = 1, \ldots, n \\ s, & \text{if } s\,x \leq 0 \end{cases}
$$

If we continue ad infinitum, we will obtain the function

$$
f_\infty\, s = \begin{cases} s[x \mapsto 0], & \text{if } s\,x > 0 \\ s, & \text{if } s\,x \leq 0 \end{cases}
$$

which is exactly the intended semantics for '`while` $x > 0$ `do` x `:=` $x{-}1$*'.*

These last three examples share one property: for each, we constructed the functional F that underlies the definition of the semantics of the loop, and starting from $f_0\,s = \textit{undefined}$, after performing sufficiently many steps, we obtained a fixed point for F, that also gives us the intended semantics. Below, we will show that this is no coincidence: we will give the mathematical framework that shows and guarantees that this construction is always successful.

We will now formalise the requirements on *Fix*.

7.2 Partial order relations

Above, starting from $f_0\,s = \textit{undefined}$, we constructed a sequence of functions that eventually give us a fixed point for the underlying functional. Close inspection of this sequence shows that each iteration of the production process gives a result that has gained information, i.e. each of these functions shows that each new function created is a true extension of the previous, in the sense that if $f_n\,s$ is defined, then so is $f_{n+1}\,s$, and then their values coincide. So f_{n+1} is identical to f_n whenever the latter is defined, but is perhaps defined on a larger set. Such a relation will be defined formally below as a *partial order relation* on functions.

We define an ordering, $\sqsubseteq$, on the function space *State* $\rightarrow$ *State*, such that:

$$g_1 \quad \sqsubseteq \quad g_2$$

means that

$$\text{if } g_1\,s_1 = s_2, \text{ then } g_2\,s_1 = s_2$$

which expresses two properties:

- the domain of g_1 (i.e. the set where g_1 is defined) is a subset of the domain of g_2 (the converse need not hold), and

- g_1 and g_2 are identical on the domain of g_1.

Example 7.7

Take

$$\begin{array}{lcl}
g_1\,s & = & s \\
g_2\,s & = & \begin{cases} s, & \text{if } s\,x \geq 0 \\ \textit{undefined}, & \text{otherwise} \end{cases} \\
g_3\,s & = & \begin{cases} s, & \text{if } s\,x = 0 \\ \textit{undefined}, & \text{otherwise} \end{cases} \\
g_4\,s & = & \begin{cases} s, & \text{if } s\,x \leq 0 \\ \textit{undefined}, & \text{otherwise} \end{cases}
\end{array}$$

then

$$\begin{tikzcd}
 & g_1 & \\
g_2 \arrow[ur, phantom, sloped, "\sqsubseteq"] & & g_4 \arrow[ul, phantom, sloped, "\sqsupseteq"] \\
 & g_3 \arrow[ul, phantom, sloped, "\sqsubseteq"] \arrow[ur, phantom, sloped, "\sqsubseteq"] &
\end{tikzcd}$$

7.3 Partial Ordered Set

We will have a closer look at the relation $\sqsubseteq$ on $(\textit{State} \rightarrow \textit{State})^2$. A *partial ordered set* (poset) is a pair $\langle D, \sqsubseteq \rangle$ such that

- $\forall d \in D \,.\, d \sqsubseteq d$ (Reflexivity)

- $\forall d_1, d_2, d_3 \in D \,.\, d_1 \sqsubseteq d_2 \;\;\&\;\; d_2 \sqsubseteq d_3 \;\Rightarrow\; d_1 \sqsubseteq d_3$ (Transitivity)
- $\forall d_1, d_2 \in D \,.\, d_1 \sqsubseteq d_2 \;\;\&\;\; d_2 \sqsubseteq d_1 \;\Rightarrow\; d_1 = d_2$ (Anti-symmetry)

If $d_1 \sqsubseteq d_2$, we say that d_1 *approximates* d_2. An element $d \in D$ satisfying

$$\forall d' \in D \,.\, d \sqsubseteq d',$$

is called a *least element* of D: it is said to *contain no information.*

Exercise 7.8 If a partially ordered set $\langle D, \sqsubseteq \rangle$ has a least element, then it is unique.

We often denote the least element of D by $\bot_D$ or just $\bot$ (pronounced '*bottom*').

Exercise 7.9 Let $S \neq \emptyset$ and define

$$\mathcal{P}(S) = \{V \mid V \subseteq S\}.$$

Then $\langle \mathcal{P}(S), \subseteq \rangle$ is a poset.

Lemma 7.10 The space of partial functions from State to State, with $\sqsubseteq$ as defined above, is a poset, and the nowhere defined function is its least element $\bot_{\mathit{State} \hookrightarrow \mathit{State}}$, so, for all states s, $\bot_{\mathit{State} \hookrightarrow \mathit{State}}\, s =$ undefined.

Proof : We have to verify the three conditions:
(*Reflexivity*) : Clearly $g \sqsubseteq g$, for all $g \in \mathit{State} \hookrightarrow \mathit{State}$.
(*Transitivity*) : Assume $g_1 \sqsubseteq g_2$, and $g_2 \sqsubseteq g_3$. Then by definition of $\sqsubseteq$ on *State* $\hookrightarrow$ *State*, if $g_1\, s = s'$, then $g_2\, s = s'$, and if $g_2\, s = s'$, then $g_3\, s = s'$. So, if $g_1\, s = s'$, then $g_3\, s = s'$, so $g_1 \sqsubseteq g_3$.
(*Anti-symmetry*) : Assume $g_1 \sqsubseteq g_2$, and $g_2 \sqsubseteq g_1$. Then, if $g_1\, s = s'$, then $g_2\, s = s'$, and if $g_2\, s = s'$, then $g_1\, s = s'$. So $g_1\, s = s'$ if and only if $g_2\, s = s'$, so g_1 and g_2 coincide, and are actually the same function.
That the function '$f s =$ *undefined*' is least can be understood by the fact that, for this f, if $f\, s = s'$, then $g\, s = s'$ holds for all g, since the condition is always false.

Notice that the least element of the function space is also called $\bot$ ($\bot_{\mathit{State} \hookrightarrow \mathit{State}}$ to be precise). So, for all $g \in \mathit{State} \hookrightarrow \mathit{State}$, $\bot_{\mathit{State} \hookrightarrow \mathit{State}} \sqsubseteq g$.

We can now give a more precise statement of the requirements we want *Fix* F to satisfy:

- $F(\mathit{Fix}\, F) = \mathit{Fix}\, F$.
- *Fix* F is a *least* fixed point of F, i.e. if, for some g, $Fg = g$, then $\mathit{Fix}\, F \sqsubseteq g$.

The second requirement makes *Fix* F unique for each F.

Exercise 7.11 The effect of a function $f : X \to Y$ is expressed by its graph:

$$\mathit{graph}\,(f) \quad = \quad \{\langle x, y \rangle \in X \times Y \mid fx = y\}$$

Show that: $g_1 \sqsubseteq g_2 \iff \mathit{graph}\,(g_1) = \mathit{graph}\,(g_2)$.

7.4 Least Upper Bounds

We now develop a general theory that gives more structure to the posets and that imposes restrictions on the functionals so that they have least fixed points.

Above, we have defined a relation on functions that expresses adequately what we observed before. Starting from the function $f\,s = \mathit{undefined}$, applying the functional F repeatedly, we obtained a sequence of functions that are in the relation, $\sqsubseteq_{\mathit{State} \hookrightarrow \mathit{State}}$:

$$f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \cdots .$$

We will now focus on defining the context that guarantees that this 'path' has an end, i.e. that there exists a precisely defined function f such that, for all n, $f_n \sqsubseteq f$. This f will then be shown to be the semantic function we were looking for.

Consider $\langle D, \sqsubseteq \rangle$ and $Y \subseteq D$. An *upper bound of* Y summarises all of the information of Y:

$$\text{If } d \text{ is an upper bound of } Y, \text{ then } \forall d' \in Y \,.\, d' \sqsubseteq d.$$

An upper bound d of Y is the *least upper bound* (*lub*) of Y if and only if:

$$\text{If } d' \text{ is an upper bound of } Y, \text{ then } d \sqsubseteq d'.$$

(A *lub* of Y will add as little extra information as possible to the information already present in Y.) We denote the *lub* of Y as $\sqcup Y$.

7.5 Chains

Although we aim to build a framework that guarantees the existence of *lub*s, we will only do so for particular sets: we only need the *lub* to exist for sets $\{f_0, f_1, f_2, \ldots\}$ such that $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \cdots$. Such a set is called a chain:

We call Y a *chain* if

$$\forall d_1, d_2 \in Y \,.\, d_1 \sqsubseteq d_2 \vee d_2 \sqsubseteq d_1 .$$

So a chain is a particular subset on which the (partial) order relation $\sqsubseteq$ is total.

Example 7.12 Consider $\langle \mathcal{P}(\{a,b,c\}), \subseteq \rangle$. Then $Y' = \{\emptyset, \{a\}, \{a,c\}\}$ is a chain. The sets $\{a,b,c\}$ and $\{a,c\}$ are upper bounds of Y', and its *lub* is $\{a,c\}$. Notice that $\{a,b\}$ in *not* an upper bound, since $\{a,c\} \not\subseteq \{a,b\}$. For any chain Y in $\langle \mathcal{P}(\{a,b,c\}), \subseteq \rangle$, $\sqcup Y$ will be the largest element of Y. The set $\{\emptyset, \{a\}, \{c\}\}$ is not a chain, but its *lub* is $\{a,c\}$. Notice that also $\emptyset$ is a chain, and that $\sqcup\emptyset = \emptyset$.

Example 7.13 Let $g_n : \mathit{State} \hookrightarrow \mathit{State}$ be defined by:

$$g_n\, s \quad = \quad \begin{cases} \mathit{undefined}, & \text{if } s\,x > n \\ s[x \mapsto 1], & \text{if } 0 \leq s\,x \leq n \\ s, & \text{if } s\,x < 0 \end{cases}$$

It is easy to verify that $g_n \sqsubseteq g_m$, whenever $n \leq m$, and that, therefore, $Y = \{g_n \mid n \geq 0\}$ is a chain. Then

$$\sqcup Y\, s \quad = \quad \begin{cases} s[x \mapsto 1], & \text{if } 0 \leq s\,x \\ s, & \text{if } s\,x < 0 \end{cases}$$

7.6 CCPO

A poset $\langle D, \sqsubseteq \rangle$ is a *chain-complete* poset (ccpo) whenever $\sqcup Y$ exists for all chains Y. It is a *complete lattice* if $\sqcup Y$ exists for all subsets Y of D.

Exercise 7.14 $\langle \mathcal{P}(S), \sqsubseteq \rangle$ is a complete lattice, and (hence) a ccpo, for all non-empty S.

Exercise 7.15 If $\langle D, \sqsubseteq \rangle$ is a ccpo, it has a least element $\bot$.

Exercise 7.16 Let S be any non-empty set. Show that $\langle \wp(S), \supseteq \rangle$ is a partially ordered set and determine the least element. Draw a picture of the ordering when $S = \{a, b, c\}$.

Exercise 7.17 *State* $\hookrightarrow$ *State* is not a complete lattice.

Lemma 7.18 State $\hookrightarrow$ State is a ccpo. The least upper bound of a chain of functions Y, $\sqcup Y$, is given by graph$(\sqcup Y) = \cup \{$graph$(g) \mid g \in Y\}$, i.e.:

$$(\sqcup Y)\, s = s' \quad \iff \quad \exists g \in Y \,.\, g\, s = s'$$

7.7 Monotone functions

Let $\langle D_1, \sqsubseteq_1 \rangle$ and $\langle D_2, \sqsubseteq_2 \rangle$ be ccpos and $f : D_1 \to D_2$. We call f *monotone* if and only if, for all $d_1, d_2 \in D_1$, if $d_1 \sqsubseteq_1 d_2$, then $f d_1 \sqsubseteq_2 f d_2$.

Example 7.19 Take $f_1, f_2 : \mathcal{P}(\{a, b, c\}) \to \mathcal{P}(\{d, e\})$, defined by:

X	$\{a,b,c\}$	$\{a,b\}$	$\{a,c\}$	$\{b,c\}$	$\{a\}$	$\{b\}$	$\{a\}$	$\emptyset$
f_1X	$\{d,e\}$	$\{d\}$	$\{d,e\}$	$\{d,e\}$	$\{d\}$	$\{d\}$	$\{e\}$	$\emptyset$
f_2X	$\{d\}$	$\{d\}$	$\{d\}$	$\{e\}$	$\{d\}$	$\{e\}$	$\{e\}$	$\{e\}$

Then $f_1$is monotone, and f_2 is not.

Exercise 7.20 Let $\langle D_1, \sqsubseteq_1 \rangle$, $\langle D_2, \sqsubseteq_2 \rangle$ and $\langle D_3, \sqsubseteq_3 \rangle$ be ccpos and let $f_1 : D_1 \to D_2$ and $f_2 : D_2 \to D_3$ be monotone functions, then $f_2 \circ f_1 : D_1 \to D_3$ is a monotone function.

Lemma 7.21 Let $\langle D_1, \sqsubseteq_1 \rangle$ and $\langle D_2, \sqsubseteq_2 \rangle$ be ccpos and $f : D_1 \to D_2$ be a monotone function. If Y is a chain in D_1, then $\{f d \mid d \in Y\}$ is a chain in D_2. Furthermore,

$$\sqcup_2 \{f d \mid d \in Y\} \quad \sqsubseteq_2 \quad f(\sqcup_1 Y).$$

Exercise 7.22 Prove this lemma.

Exercise 7.23 Let *graph* be as in Exercise 7.11. If $R_1 \subseteq X \times Y$ and $R_2 \subseteq Y \times Z$, the compositon of R_1 followed by R_2, denoted $R_1 \circ R_2$, is defined by:

$$R_1 \circ R_2 \quad = \quad \{\langle x, z \rangle \in X \times Z \mid \exists\, y \in Y\ [\langle x, y \rangle \in R_1\ \&\ \langle y, z \rangle \in R_2]\}$$

Show that:

$$\textit{graph}\,(f_2 \circ f_1) \quad = \quad \textit{graph}\,(f_1) \circ \textit{graph}\,(f_2)$$

Prove that F defined by:

$$F g = g_0 \circ g$$

is continuous. (Hint: you will need to use Lemma **??**.)

In general, we cannot expect that a monotone function preserves *lub*s on chains, i.e.

$$\sqcup_2\{fd \mid d \in Y\} \quad = \quad f(\sqcup_1 Y)$$

only holds in special cases, as illustrated by the next example.

Example 7.24 Consider $f : \mathcal{P}(\mathbb{N} \cup \{a\}) \to \mathcal{P}(\mathbb{N} \cup \{a\})$, defined by:

$$fX \quad = \quad \begin{cases} X, & \text{if } X \text{ is finite} \\ X \cup \{a\}, & \text{if } X \text{ is infinite} \end{cases}$$

Clearly, f is monotone. But consider the set

$$Y \quad = \quad \{\{0, 1, \ldots, n\} \mid n \geq 0\}$$

Then $\sqcup Y = \mathbb{N}$. Now

$$\sqcup\{fX \mid X \in Y\} \quad = \quad \sqcup\{X \mid X \in Y\} \quad = \quad \sqcup Y \quad = \quad \mathbb{N}$$

But

$$f(\sqcup Y) \quad = \quad f\mathbb{N} \quad = \quad \mathbb{N} \cup \{a\}.$$

7.8 Continuous Functions

We shall be interested in functions which do preserve *lub*s of chains. An $f : D_1 \to D_2$ defined on ccpos $\langle D_1, \sqsubseteq_1\rangle$ and $\langle D_2, \sqsubseteq_2\rangle$ is called *continuous* if it is monotone and, moreover:

$$\sqcup_2\{fd \mid d \in Y\} \quad = \quad f(\sqcup_1 Y)$$

holds for all *non-empty* chains Y in $\langle D_1, \sqsubseteq_1\rangle$. (When this also holds for the *empty* chain, so $\emptyset = \sqcup_2\{\emptyset\} = f(\sqcup_1\emptyset) = f(\emptyset)$, we say that f is *strict*).

Lemma 7.25 $\langle D_1, \sqsubseteq_1\rangle$, $\langle D_2, \sqsubseteq_2\rangle$ and $\langle D_3, \sqsubseteq_3\rangle$ be ccpos and let $f_1 : D_1 \to D_2$ and $f_2 : D_2 \to D_3$ be continuous functions. Then $f_2 \circ f_1 : D_1 \to D_3$ is a continuous function.

Proof : Notice that, by a previous result, $f_2 \circ f_1$ is monotone. We only need to show that $f_2 \circ f_1$ preserves *lub*s of chains, i.e.:

$$\sqcup_3\{(f_2 \circ f_1)\, d \mid d \in Y\} \quad = \quad (f_2 \circ f_1)\,(\sqcup_1 Y)$$

holds for all non-empty chains Y in $\langle D_1, \sqsubseteq_1\rangle$. We know, by continuity of f_1, that

$$\sqcup_2\{f_1\, d \mid d \in Y\} \quad = \quad f_1\,(\sqcup_1 Y),$$

for all chains Y in $\langle D_1, \sqsubseteq_1\rangle$, and by continuity of f_2, that

$$\sqcup_3\{f_2\, d \mid d \in Y\} \quad = \quad f_2\,(\sqcup_2 Y),$$

for all chains Y in $\langle D_2, \sqsubseteq_2\rangle$. Notice that, by the fact that f_1 is monotone, if Y is a chain, then also $f_1\,(Y) = \{f_1\, d_1 \mid d_1 \in Y\}$ is a chain as well. So, we obtain that

$$\begin{array}{rcll} \sqcup_3\{(f_2 \circ f_1)\,(Y)\} & = & \sqcup_3\{f_2\,(f_1\,(Y)\} & \\ & = & \sqcup_3\{f_2\, d_2 \mid d_2 \in \{f_1\,(Y)\}\} & (f_2 \text{ is continuous}) \\ & = & f_2\,(\sqcup_2\{f_1\,(Y)\}) & \\ & = & f_2\,(\sqcup_2\{f_1\,(Y)\}) & (f_1 \text{ is continuous}) \\ & = & f_2\,(f_1\,(\sqcup_1 Y)) & \\ & = & (f_2 \circ f_1)\,(\sqcup_1 Y) & \end{array}$$

7.9 The Fixed Point Theorem

Above, we have defined continuous functions as functions that are monotone, i.e. preserve the order, and continuous, i.e. preserve *lub*s of chains.

Lemma 7.26 Let $f : D \to D$ be a continuous function on the ccpo $\langle D, \sqsubseteq \rangle$ with least element $\bot$. Let

$$\begin{array}{rcl} f^0 & = & \mathit{id} \\ f^{n+1} & = & f \circ f^n, \text{ for } n \geq 0 \end{array}$$

Then

$$\mathit{Fix}\, f \quad = \quad \bigsqcup\{ f^n \bot \mid n \geq 0 \}$$

defines an element of D and this element is the least fixed point of f.

Proof : We divide the proof in two parts:

i) We first show that *Fix* f is well-defined, by showing that the set $\{f^n\bot \mid n \geq 0\}$ is a chain in D, so its *lub* is defined. So, we need to show that, for all $d_1, d_2 \in \{f^n\bot \mid n \geq 0\}$, either $d_1 \sqsubseteq d_2$ or $d_2 \sqsubseteq d_1$. We will show that $f^n\bot \sqsubseteq f^m\bot$, whenever $n \leq m$. We start showing that, for all n, $f^n\bot \sqsubseteq f^n d$; this follows by induction on numbers.

(*Base step*) : Notice that $f^0\bot = \bot$, $f^0 d = d$, and $\bot \sqsubseteq d$ for all $d \in D$.

(*Induction step*) : We can assume that $f^n\bot \sqsubseteq f^n d$, for all $d \in D$. Now, from the fact that f is monotone, we also have $f^{n+1}\bot \sqsubseteq f^{n+1}d$, for all $d \in D$.

So, for all n, $f^n\bot \sqsubseteq f^n d$.

Let $m \geq n$, and $d = f^{m-n}$, then, in particular, $f^n\bot \sqsubseteq f^n d$, so $f^n\bot \sqsubseteq f^m\bot$. Hence $\{f^n\bot \mid n \geq 0\}$ is a non-empty chain in D, and therefore, its *lub* exists because D is a ccpo, so *Fix* f exists.

ii) We will now show that *Fix* f is indeed a fixed point of f, i.e.: $f(\mathit{Fix}\, f) = \mathit{Fix}\, f$. Well:

$$\begin{array}{rcll} f(\mathit{Fix}\, f) & = & f(\bigsqcup\{f^n\bot \mid n \geq 0\}) & (f \text{ is continuous}) \\ & = & \bigsqcup\{f(f^n\bot) \mid n \geq 0\} & \\ & = & \bigsqcup\{f^n\bot \mid n \geq 1\} & (\forall Y. \bigsqcup(Y \cup \{\bot\}) = \bigsqcup Y) \\ & = & \bigsqcup(\{f^n\bot \mid n \geq 1\} \cup \{\bot\}) & \\ & = & \bigsqcup\{f^n\bot \mid n \geq 0\} & \\ & = & \mathit{Fix}\, f. & \end{array}$$

iii) We will now show that *Fix* f is the *least* fixed point. To accomplish that, we need to show: if d' is another fixed-point of f, i.e. also for d', $f(d') = d'$, then $d \sqsubseteq d'$. Well, since $\bot \sqsubseteq d'$, by continuity of f, we have $f^n\bot \sqsubseteq f^n d'$, for all $n \geq 0$. Since d' is a fixed-point, $d' = f^n d'$ for all n, so we get that $f^n\bot \sqsubseteq d'$, for all $n \geq 0$. So d' is an upper bound of $\{f^n\bot \mid n \geq 0\}$. Since d is the least upper bound of $\{f^n\bot \mid n \geq 0\}$ (so $d = \bigsqcup\{f^n\bot \mid n \geq 0\}$), $d \sqsubseteq d'$.

7.10 Denotational Semantics

Using the result above, we can now define the Denotational Semantics for a program in `While`.

The meaning of a statement S is a (partial) function from *State* to *State*:

$$\mathcal{S}_{ds} \quad : \quad \mathit{Statements} \rightarrow \mathit{State} \hookrightarrow \mathit{State}$$

$$\begin{array}{rcl}
\mathcal{S}_{ds}\,[\![x \;\texttt{:=}\; a]\!]\,s & = & s[x \mapsto \mathcal{A}\,[\![a]\!]\,s] \\
\mathcal{S}_{ds}\,[\![\texttt{skip}]\!] & = & \mathit{id} \\
\mathcal{S}_{ds}\,[\![S_1\,\texttt{;}\,S_2]\!] & = & \mathcal{S}_{ds}\,[\![S_2]\!] \circ \mathcal{S}_{ds}\,[\![S_1]\!] \\
\mathcal{S}_{ds}\,[\![\texttt{if}\;b\;\texttt{then}\;S_1\;\texttt{else}\;S_2]\!] & = & \mathit{cond}\,(\mathcal{B}\,[\![b]\!]\,, \mathcal{S}_{ds}\,[\![S_1]\!], \mathcal{S}_{ds}\,[\![S_2]\!]) \\
\mathcal{S}_{ds}\,[\![\texttt{while}\;b\;\texttt{do}\;S]\!] & = & \mathit{Fix}\,F, \\
 & & \text{where } F g = \mathit{cond}\,(\mathcal{B}\,[\![b]\!]\,, g \circ \mathcal{S}_{ds}\,[\![S]\!], \mathit{id})
\end{array}$$

Of course, given the construction above, this is only well-defined if we are in the context of *continuous functions*, something we will deal with shortly.

Example 7.27 Consider the function

$$F\,f\,s \quad = \quad \begin{cases} f\,s, & \text{if } s\,x \neq 0 \\ s, & \text{if } s\,x = 0 \end{cases}$$

The elements of $\{F^n\bot \mid n \geq 0\}$ are defined as follows:

$$\begin{array}{rcl}
F^0\,s & = & \mathit{id}\,\bot\,s \\
 & = & \bot\,s \\
 & = & \mathit{undefined}
\end{array}$$

$$\begin{array}{rcl}
F^1\bot\,s & = & (F \circ F^0)\bot\,s \\
 & = & \begin{cases} F^0 s, & \text{if } s\,x \neq 0 \\ s, & \text{if } s\,x = 0 \end{cases} \\
 & = & \begin{cases} \mathit{undefined}, & \text{if } s\,x \neq 0 \\ s, & \text{if } s\,x = 0 \end{cases}
\end{array}$$

$$\begin{array}{rcl}
F^2\bot\,s & = & (F \circ F^1)\bot\,s \\
 & = & \begin{cases} F^1 s, & \text{if } s\,x \neq 0 \\ s, & \text{if } s\,x = 0 \end{cases} \\
 & = & \begin{cases} \left\{ \begin{array}{ll} \mathit{undefined}, & \text{if } s\,x \neq 0 \\ s, & \text{if } s\,x = 0 \end{array} \right\} & \text{if } s\,x \neq 0 \\ s, & \text{if } s\,x = 0 \end{cases} \\
 & = & \begin{cases} \mathit{undefined}, & \text{if } s\,x \neq 0 \\ s, & \text{if } s\,x = 0 \end{cases}
\end{array}$$

So $F^2 = F^1$. We could continue this construction, but we would not achieve anything by it: for all $n \geq 1$, $F^n = F^{n+1}$, so

$$\bigsqcup\{F^n\bot \mid n \geq 0\} = \bigsqcup\{F^0\bot, F^1\bot\} = F^1\bot$$

Therefore,

$$\mathit{Fix}\,F \quad = \quad \begin{cases} \mathit{undefined}, & \text{if } s\,x \neq 0 \\ s, & \text{if } s\,x = 0 \end{cases}$$

Exercise 7.28 Compute the semantics of

$$z := 0\,;\,\texttt{while}\; y \le x\; \texttt{do}\; (z := z{+}1\,;\, x := x - y)$$

7.11 Existence

As already observed above, we want to achieve

$$\mathcal{S}_{ds}\,[\![\texttt{while}\; b\; \texttt{do}\; S]\!] \quad = \quad \mathit{cond}\,(\mathcal{B}\,[\![b]\!]\,, \mathcal{S}_{ds}\,[\![\texttt{while}\; b\; \texttt{do}\; S]\!] \circ \mathcal{S}_{ds}\,[\![S]\!], \mathit{id}).$$

We now want to show that the underlying functional for this definition,

$$F f \quad = \quad \mathit{cond}\,(\mathcal{B}\,[\![b]\!]\,, f \circ \mathcal{S}_{ds}\,[\![S]\!], \mathit{id})$$

is continuous. Observe that we can write $F g$ as $F_1\,(F_2\,g)$, where $F_1\,g = \mathit{cond}\,(\mathcal{B}\,[\![b]\!]\,, g, \mathit{id})$, and $F_2\,g = g \circ \mathcal{S}_{ds}\,[\![S]\!]$ are.

Then, using results shown above, we have that F is continuous if we manage to show that F_1 and F_2. We will do that in a more general setting:

Lemma 7.29 *Let* $h : \mathit{State} \hookrightarrow \mathit{State}$*,* $b : \mathit{State} \to \mathbb{T}$*, and define*

$$F g \quad = \quad \mathit{cond}\,(b, g, h)$$

Then F *is continuous.*

Proof : Notice that

$$\begin{aligned} F g\, s \quad &= \quad \mathit{cond}\,(b, g, h)\; s \\ &= \quad \begin{cases} g\, s, & \text{if } b\, s = \mathbf{tt} \\ h\, s, & \text{if } b\, s = \mathbf{ff} \end{cases} \end{aligned}$$

We prove the following two things:

(F *is monotone*) : Assume $g_1 \sqsubseteq g_2$. We have to show that $F g_1 \sqsubseteq F g_2$. Notice that, since $g_1 \sqsubseteq g_2$, $g_1\, s = s'$ implies $g_2\, s = s'$.

- ($b\, s = \mathbf{tt}$) : Then, since $F g_1\, s = g_1\, s$, the first is defined exactly when the latter is. Since $g_1\, s = s'$ implies $g_2\, s = s'$, and $g_2\, s = F g_2\, s$, we get that $F g_1\, s = s'$ implies $F g_2\, s = s'$. So $F g_1 \sqsubseteq F g_2$.
- ($b\, s = \mathbf{ff}$) : Since $F g_1\, s = h\, s = F g_2\, s$, $F g_1\, s$ is defined precisely when $h\, s$ is, which is precisely when $F g_2\, s$ is.

(F *preserves* lubs) : Let Y be a non-empty chain in *State* $\hookrightarrow$ *State*. We must show that

$$F(\sqcup Y) \quad \sqsubseteq \quad \sqcup\{F g \mid g \in Y\}$$

(Actually, we need to show this with '$=$' instead of '$\sqsubseteq$', but the part '$\sqsupseteq$' follows from the fact that F is monotone.) So, we need to show that

$$\mathit{graph}(F(\sqcup Y)) \quad \subseteq \quad \cup\,\{\mathit{graph}(F g) \mid g \in Y\}$$

Again, we have two cases ($b = \mathbf{tt}$, and $b = \mathbf{ff}$).
The proof is concluded in the exercise below.

Exercise 7.30 if $F(\sqcup Y)\, s = s'$, then there is a $g \in Y$ such that $F g\, s = s'$.

Lemma 7.31 *Let h : State $\hookrightarrow$ State, and define $Fg = g \circ h$. Then F is continuous.*

Proof : As above, we would need to show that F is monotone and preserves *lub*s, but we will skip this.

Now we are done, and the complete definition of denotational semantics for `While` becomes:

$$\mathcal{S}_{ds} : \textit{Statements} \to \textit{State} \hookrightarrow \textit{State}$$

$$\begin{array}{rcl}
\mathcal{S}_{ds}\,[\![x \texttt{ := } a]\!]\, s & = & s[x \mapsto \mathcal{A}\,[\![a]\!]\, s] \\
\mathcal{S}_{ds}\,[\![\texttt{skip}]\!] & = & \textit{id} \\
\mathcal{S}_{ds}\,[\![S_1 \texttt{;} S_2]\!] & = & \mathcal{S}_{ds}\,[\![S_2]\!] \circ \mathcal{S}_{ds}\,[\![S_1]\!] \\
\mathcal{S}_{ds}\,[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] & = & \textit{cond}\,(\mathcal{B}\,[\![b]\!]\,, \mathcal{S}_{ds}\,[\![S_1]\!], \mathcal{S}_{ds}\,[\![S_2]\!]) \\
\mathcal{S}_{ds}\,[\![\texttt{while}\ b\ \texttt{do}\ S]\!] & = & \textit{Fix}\, F, \\
 & & \text{where } Fg = \textit{cond}\,(\mathcal{B}\,[\![b]\!]\,, g \circ \mathcal{S}_{ds}\,[\![S]\!], \textit{id})
\end{array}$$

The final result to show for the denotational semantics is:

Theorem 7.32 *The semantic definition $\mathcal{S}_{ds}$ defines a total function.*

Proof : By induction on the structure of S. There is only one interesting case:
$(\texttt{while}\ b\ \texttt{do}\ S)$: By induction, $\mathcal{S}_{ds}\,[\![S]\!]$ is well-defined. Since we have shown above that the functions

$$\begin{array}{rcl}
F_1\, g & = & \textit{cond}\,(\mathcal{B}\,[\![b]\!]\,, g, \textit{id}) \\
F_2\, g & = & g \circ \mathcal{S}_{ds}\,[\![S]\!]
\end{array}$$

are continuous, so is $\mathcal{S}_{ds}\,[\![\texttt{while}\ b\ \texttt{do}\ S]\!]$.

Thus, from the fixed point theorem, *Fix* is well-defined. So $\mathcal{S}_{ds}\,[\![\texttt{while}\ b\ \texttt{do}\ S]\!]$ is well-defined.

As before, S_1 and S_2 are called *semantically equivalent* if and only if

$$\mathcal{S}_{ds}\,[\![S_1]\!] \quad = \quad \mathcal{S}_{ds}\,[\![S_2]\!]$$

Example 7.33 S; **`skip`** and S are semantically equivalent.

$$\begin{array}{rcl}
\mathcal{S}_{ds}\,[\![S; \texttt{skip}]\!] & = & \mathcal{S}_{ds}\,[\![\texttt{skip}]\!] \circ \mathcal{S}_{ds}\,[\![S]\!] \\
 & = & \textit{id} \circ \mathcal{S}_{ds}\,[\![S]\!] \\
 & = & \mathcal{S}_{ds}\,[\![S]\!]
\end{array}$$

Exercise 7.34 Show that

$$S_1 \texttt{;} (S_2 \texttt{;} S_3)$$

is semantically equivalent to

$$(S_1 \texttt{;} S_2) \texttt{;} S_3$$

7.12 Equivalence to Operational Semantics

Lemma 7.35 Notice that:

- *Let $f : D \to D$ be continuous, and let $d \in D$ satisfy $f\,d \sqsubseteq d$. Then $\mathit{Fix}\,f \sqsubseteq d$.*
- *If $\langle S_1, s_1\rangle \Rightarrow^k s_2$, then $\langle S_1\,\texttt{;}\,S_2, s_1\rangle \Rightarrow^k \langle S_2, s_2\rangle$.*
- *'$\circ$' and cond are monotone.*

We conclude the discussion of Denotational Semantics, by showing that, for the language `While`, there is no difference between this semantics and the Structural Operational Semantics.

Theorem 7.36 *For every statement S of* `While`*:*

$$\mathcal{S}_{sos}[\![S]\!] \;=\; \mathcal{S}_{ds}[\![S]\!]$$

i.e.:

$$\mathcal{S}_{sos}[\![S]\!] \;\sqsubseteq\; \mathcal{S}_{ds}[\![S]\!]$$

and

$$\mathcal{S}_{ds}[\![S]\!] \;\sqsubseteq\; \mathcal{S}_{sos}[\![S]\!]$$

Proof :

$(\mathcal{S}_{sos}[\![S]\!] \sqsubseteq \mathcal{S}_{ds}[\![S]\!])$: So, we have to show that $\langle S, s_1\rangle \Rightarrow^* s_2$ implies $\mathcal{S}_{ds}[\![S]\!]\,s_1 = s_2$. This follows by induction on the length of the derivation sequence for $\langle S, s_1\rangle \Rightarrow^k s_2$. For this, it suffices to show that:

- $\langle S, s_1\rangle \Rightarrow s_2$ implies $\mathcal{S}_{ds}[\![S]\!]\,s_1 = s_2$, and
- $\langle S, s_1\rangle \Rightarrow \langle S', s_2\rangle$ implies $\mathcal{S}_{ds}[\![S]\!]\,s_1 = \mathcal{S}_{ds}[\![S']\!]\,s_2$.

This result follows by induction over the shape of the derivation for the single step, where we focus on the last step. We just consider two cases:

$(\textsc{comp}^{\mathrm{I}}_{sos})$: Then there are S_1, S_1', S_2 such that

$$S = S_1\,\texttt{;}\,S_2,\ S' = S_1'\,\texttt{;}\,S_2,\ \text{and}\ \langle S_1, s_1\rangle \Rightarrow \langle S_1', s_2\rangle.$$

Then, by induction, $\mathcal{S}_{ds}[\![S_1]\!]\,s_1 = \mathcal{S}_{ds}[\![S_1']\!]\,s_2$, so we have

$$\begin{array}{rcll} \mathcal{S}_{ds}[\![S_1;S_2]\!]\,s_1 & = & (\mathcal{S}_{ds}[\![S_2]\!]\circ\mathcal{S}_{ds}[\![S_1]\!])\,s_1 & \\ & = & \mathcal{S}_{ds}[\![S_2]\!]\,(\mathcal{S}_{ds}[\![S_1]\!]\,s_1) & \text{(IH)} \\ & = & \mathcal{S}_{ds}[\![S_2]\!]\,(\mathcal{S}_{ds}[\![S_1']\!]\,s_2) & \\ & = & (\mathcal{S}_{ds}[\![S_2]\!]\circ\mathcal{S}_{ds}[\![S_1']\!])\,s_2 & \\ & = & \mathcal{S}_{ds}[\![S_1\,\texttt{;}\,S_2]\!]\,s_2 & \end{array}$$

$(\textsc{while}_{sos})$: Then there are S_1 and b such that $S = \texttt{while}\ b\ \texttt{do}\ S_1$, and $s_1 = s_2$, and $S' = \texttt{if}\ b\ \texttt{then}\ (S_1\,\texttt{;}\,\texttt{while}\ b\ \texttt{do}\ S_1)\ \texttt{else skip}$. By definition of $\mathcal{S}_{ds}$, $\mathcal{S}_{ds}[\![\texttt{while}\ b\ \texttt{do}\ S_1]\!] = \mathit{Fix}\,F$, where $F g = \mathit{cond}\,(\mathcal{B}[\![b]\!], g\circ\mathcal{S}_{ds}[\![S_1]\!], \mathit{id})$. Now

$$\begin{array}{rcl} \mathcal{S}_{ds}[\![\texttt{while}\ b\ \texttt{do}\ S_1]\!] & = & \mathit{Fix}\,F \\ & = & F(\mathit{Fix}\,F) \\ & = & \mathit{cond}\,(\mathcal{B}[\![b]\!], \mathcal{S}_{ds}[\![\texttt{while}\ b\ \texttt{do}\ S_1]\!]\circ\mathcal{S}_{ds}[\![S_1]\!], \mathit{id}) \\ & = & \mathit{cond}\,(\mathcal{B}[\![b]\!], \mathcal{S}_{ds}[\![S_1\,\texttt{;}\,\texttt{while}\ b\ \texttt{do}\ S_1]\!], \mathit{id}) \\ & = & \mathcal{S}_{ds}[\![\texttt{if}\ b\ \texttt{then}\ (S_1\,\texttt{;}\,\texttt{while}\ b\ \texttt{do}\ S_1)\ \texttt{else skip}]\!] \end{array}$$

So, we have shown: if $\mathcal{S}_{sos}[\![S]\!]\,s \neq \textit{undefined}$, then $\mathcal{S}_{sos}[\![S]\!]\,s = \mathcal{S}_{ds}[\![S]\!]\,s$, which, of course, is the same as $\mathcal{S}_{sos}[\![S]\!] \sqsubseteq \mathcal{S}_{ds}[\![S]\!]$.

$(\mathcal{S}_{ds}[\![S]\!] \sqsubseteq \mathcal{S}_{sos}[\![S]\!])$: This part is shown by induction on the structure of statements. We only consider two cases:

$(S_1\,\texttt{;}\,S_2)$: Notice that:

- '$\circ$' is monotone in both arguments, and
- if $\langle S_1, s_1\rangle \Rightarrow s_2$, then $\langle S_1\,\texttt{;}\,S_2, s_1\rangle \Rightarrow \langle S_2, s_2\rangle$.

By induction, we obtain both $\mathcal{S}_{ds}[\![S_1]\!] \sqsubseteq \mathcal{S}_{sos}[\![S_1]\!]$ and $\mathcal{S}_{ds}[\![S_2]\!] \sqsubseteq \mathcal{S}_{sos}[\![S_2]\!]$. So,

$$\begin{array}{rcll}
\mathcal{S}_{ds}[\![S_1\,\texttt{;}\,S_2]\!] & = & \mathcal{S}_{ds}[\![S_2]\!]\circ\mathcal{S}_{ds}[\![S_1]\!] & \text{(IH)} \\
& \sqsubseteq & \mathcal{S}_{sos}[\![S_2]\!]\circ\mathcal{S}_{sos}[\![S_1]\!] & (\circ \text{ is monotone}) \\
& \sqsubseteq & \mathcal{S}_{sos}[\![S_1\,\texttt{;}\,S_2]\!] &
\end{array}$$

$(\textbf{while}\ b\ \textbf{do}\ S)$: Since $\mathcal{S}_{ds}[\![\textbf{while}\ b\ \textbf{do}\ S]\!]$ is the fixed point of a functional F, where $Fg = \textit{cond}(\mathcal{B}[\![b]\!], g\circ\mathcal{S}_{ds}[\![S_1]\!], \textit{id})$, it is sufficient to prove:

$$F(\mathcal{S}_{sos}[\![\textbf{while}\ b\ \textbf{do}\ S]\!]) \quad \sqsubseteq \quad \mathcal{S}_{sos}[\![\textbf{while}\ b\ \textbf{do}\ S]\!]$$

since,

$$\textit{Fix}\,F \sqsubseteq \mathcal{S}_{sos}[\![\textbf{while}\ b\ \textbf{do}\ S]\!] \quad \Rightarrow$$
$$\mathcal{S}_{ds}[\![\textbf{while}\ b\ \textbf{do}\ S]\!] \sqsubseteq \mathcal{S}_{sos}[\![\textbf{while}\ b\ \textbf{do}\ S]\!]$$

Notice that we have, by induction, $\mathcal{S}_{ds}[\![S]\!] \sqsubseteq \mathcal{S}_{sos}[\![S]\!]$, so

$$\begin{array}{rcl}
F(\mathcal{S}_{sos}[\![\textbf{while}\ b\ \textbf{do}\ S]\!]) & = & \textit{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{sos}[\![\textbf{while}\ b\ \textbf{do}\ S]\!]\circ\mathcal{S}_{ds}[\![S]\!], \textit{id}) \\
\textit{(IH)} & \sqsubseteq & \textit{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{sos}[\![\textbf{while}\ b\ \textbf{do}\ S]\!]\circ\mathcal{S}_{sos}[\![S]\!], \textit{id}) \\
& \sqsubseteq & \textit{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{sos}[\![S\,\texttt{;}\,\textbf{while}\ b\ \textbf{do}\ S]\!], \textit{id}) \\
& = & \mathcal{S}_{sos}[\![\textbf{while}\ b\ \textbf{do}\ S]\!]
\end{array}$$

8 Extensions to `While`

We have seen that Operational Semantics are good for formally describing implementation aspects of programming languages. Furthermore:

- Structural Operational Semantics are good for describing low-level details (abstract machine).
- Natural Semantics are good for reasoning (more abstract - more intuitive)

We will now see some other differences. We will do that by defining extensions to the language `While`, adding new language constructs.

8.1 Aborting

We start by adding the new statement

`abort`.

Intuitively, attempting to execute **`abort`** stops the execution of the whole program. There are various ways of interpreting this instruction, depending on what the *intention* of the instruction is: do you want a program just to halt without executing further, or do you want an error message to appear. Using this difference, we will consider two approaches to giving semantics to the new statement.

(*Approach 1*) : Consider $\langle \texttt{abort}, s\rangle$ to be a *stuck configuration*. In this case, there is no need to add an extra rule to either the Natural or the Structural Operational Semantics, and we can use the original definition of both $\langle\cdot,\cdot\rangle \to \cdot$ and $\langle\cdot,\cdot\rangle \Rightarrow \cdot$. Since no new rule is added, no transformation for **`abort`** is defined, so in both systems $\langle \texttt{abort}, s\rangle$ will not evaluate, and therefore be stuck.

However, now there is a difference between the two systems. Note that

$$\langle \texttt{while true do skip}, s_1\rangle \to s_2 \quad \text{implies} \quad \langle \texttt{abort}, s_1\rangle \to s_2$$

and

$$\langle \texttt{abort}, s_1\rangle \to s_2 \quad \text{implies} \quad \langle \texttt{while true do skip}, s_1\rangle \to s_2$$

are both vacuously true, so, in the Natural Semantics, the statements '**`while true do skip`**' and '**`abort`**' are equivalent. This is not true in Structural Operational Semantics, the statement '**`while true do skip`**' generates an infinite number of steps,

$$\begin{array}{ll}\langle \texttt{while true do skip}, s\rangle & \\ \qquad\Rightarrow & \langle \texttt{if true then } (\texttt{skip ; while true do skip}) \texttt{ else skip}, s\rangle \\ \qquad\Rightarrow & \langle \texttt{skip ; while true do skip}, s\rangle \\ \qquad\Rightarrow & \langle \texttt{while true do skip}, s\rangle \\ \qquad\vdots & \end{array}$$

and '**`abort`**' none

$$\langle \texttt{abort}, s\rangle \text{ is stuck}$$

So, in Natural Semantics, looping and abnormal termination are equivalent, whereas in Structural Operational Semantics, abnormal termination is equivalent to reaching a stuck configuration.

(*Approach 2*) : Add a new terminal configuration to the system, **`error`**, and extend the definition of Natural Semantics by

$$(\text{ABORT}_{ns}) \quad \frac{}{\langle \texttt{abort}, s_1\rangle \to \texttt{error}}$$

and that of Structural Operational Semantics by

$$(\text{ABORT}_{sos}) \quad \frac{}{\langle \texttt{abort}, s_1\rangle \Rightarrow \texttt{error}}$$

Because we have added a terminal state that cannot be reached by any other statement, we can now distinguish between '**`while true do skip`**' and '**`abort`**' in both semantics.

8.2 Non-determinism

We add the statement construct ‘ **or** ’, so extend the definition of *Statements* by ‘S_1 **or** S_2’, that has the intuitive semantics that either S_1 or S_2 will be executed, which one being chosen non-deterministically. For example:

$$(x := 1) \quad \texttt{or} \quad (x := 2\,;\, x := x+2)$$

could result in a statewhere x has value 1, or in a state where x has value 4. The rules for Natural Semantics are exended by adding:

$$(\textsc{or}^{\mathrm{L}}_{ns}) \quad \left(\frac{\langle S_1, s_1\rangle \to s_2}{\langle S_1 \;\texttt{or}\; S_2, s_1\rangle \to s_2}\right) \qquad (\textsc{or}^{\mathrm{R}}_{ns}) \quad \left(\frac{\langle S_2, s_1\rangle \to s_2}{\langle S_1 \;\texttt{or}\; S_2, s_1\rangle \to s_2}\right)$$

Then we have the following derivations:

$$\dfrac{\dfrac{}{\langle x := 1, s\rangle \to s[x \mapsto 1]}}{\langle x := 1 \;\texttt{or}\; (x := 2\,;\, x := x+2), s\rangle \to s[x \mapsto 1]}$$

and

$$\dfrac{\dfrac{\dfrac{}{\langle x := 2, s\rangle \to s[x \mapsto 2]} \qquad \dfrac{}{\langle x := x+2, s[x \mapsto 2]\rangle \to s[x \mapsto 4]}}{\langle x := 2\,;\, x := x+2, s\rangle \to s[x \mapsto 4]}}{\langle x := 1 \;\texttt{or}\; (x := 2\,;\, x := x+2), s\rangle \to s[x \mapsto 4]}$$

Notice that non-determinism suppresses looping (if possible), since we can derive

$$\langle(\texttt{while true do skip}) \;\texttt{or}\; (x := 2\,;\, x := x+2), s\rangle \to s[x \mapsto 4]$$

Likewise, the definition of Structural Operational Semantics is extended by:

$$\langle S_1 \;\texttt{or}\; S_2, s\rangle \Rightarrow \langle S_1, s\rangle \quad \text{and} \quad \langle S_1 \;\texttt{or}\; S_2, s\rangle \Rightarrow \langle S_2, s\rangle$$

Then we have:

$$\begin{aligned} \langle x := 1 \;\texttt{or}\; (x := 2\,;\, x := x+2), s\rangle &\Rightarrow \langle x := 1, s\rangle \\ &\Rightarrow s[x \mapsto 1] \end{aligned}$$

and

$$\begin{aligned} &\langle x := 1 \;\texttt{or}\; (x := 2\,;\, x := x+2), s\rangle \\ &\qquad \Rightarrow \langle x := 2\,;\, x := x+2, s\rangle \\ &\qquad \Rightarrow \langle x := x+2, s[x \mapsto 2]\rangle \\ &\qquad \Rightarrow s[x \mapsto 4] \end{aligned}$$

But replacing 'x **:=** 1' by '**while true do skip**' will still give two derivation sequences; one will be infinite:

$$\begin{aligned}
&\langle(\texttt{while true do skip})\ \texttt{or}\ (x \texttt{ := } 2\,\texttt{;}\,x \texttt{ := } x+2), s\rangle\\
&\qquad\Rightarrow\ \langle\texttt{while true do skip}, s\rangle\\
&\qquad\Rightarrow\ \langle\texttt{if true then}\ (\texttt{skip}\,\texttt{;}\,\texttt{while true do skip})\ \texttt{else skip}, s\rangle\\
&\qquad\Rightarrow\ \langle\texttt{skip}\,\texttt{;}\,\texttt{while true do skip}, s\rangle\\
&\qquad\Rightarrow\ \langle\texttt{while true do skip}, s\rangle\\
&\qquad\ \vdots
\end{aligned}$$

and the other is finite:

$$\begin{aligned}
&\langle(\texttt{while true do skip})\ \texttt{or}\ (x \texttt{ := } 2\,\texttt{;}\,x \texttt{ := } x+2), s\rangle\\
&\qquad\Rightarrow\ \langle x \texttt{ := } 2\,\texttt{;}\,x \texttt{ := } x+2, s\rangle\\
&\qquad\Rightarrow\ \langle x \texttt{ := } x+2, s[x \mapsto 2]\rangle\\
&\qquad\Rightarrow\ s[x \mapsto 4]
\end{aligned}$$

So, in a certain sense, Structural Operational Semantics can 'choose the wrong branch'.

8.3 Parallelism

We can now add the statement construct ' **par** ', so extend the definition of *Statements* by the alternative 'S_1 **par** S_2', and expect the execution of S_1 and S_2 to be 'interleaved'. For example, the program

$$(x \texttt{ := } 1)\ \texttt{par}\ (x \texttt{ := } 2\,\texttt{;}\,x \texttt{ := } x+2)$$

has the following different ways to execute:

Step 1	Step 2	Step 3	Result
x := 1	x := 2	x := $x+2$	$s\,x = 4$
x := 2	x := 1	x := $x+2$	$s\,x = 3$
x := 2	x := $x+2$	x := 1	$s\,x = 1$

For the Structural Operational Semantics, to deal with the new construct, the derivation rules need to be extended with:

$$(\text{PAR}^{\text{LT}}_{sos})\ \frac{\langle S_1, s_1\rangle \Rightarrow s_2}{\langle S_1\ \texttt{par}\ S_2, s_1\rangle \Rightarrow \langle S_2, s_2\rangle} \qquad (\text{PAR}^{\text{LI}}_{sos})\ \frac{\langle S_1, s_1\rangle \Rightarrow \langle S_1', s_2\rangle}{\langle S_1\ \texttt{par}\ S_2, s_1\rangle \Rightarrow \langle S_1'\ \texttt{par}\ S_2, s_2\rangle}$$

$$(\text{PAR}^{\text{RT}}_{sos})\ \frac{\langle S_2, s_1\rangle \Rightarrow s_2}{\langle S_1\ \texttt{par}\ S_2, s_1\rangle \Rightarrow \langle S_1, s_2\rangle} \qquad (\text{PAR}^{\text{RI}}_{sos})\ \frac{\langle S_2, s_1\rangle \Rightarrow \langle S_1', s_2\rangle}{\langle S_1\ \texttt{par}\ S_2, s_1\rangle \Rightarrow \langle S_1\ \texttt{par}\ S_1', s_2\rangle}$$

Exercise 8.1 Verify that the three results above are allowed by this semantics.

To accomplish the same expressive power in Natural Semantics, we run into problems. Assume the rules that need to be added are:

$$\frac{\langle S_1, s_1\rangle \to s_2 \quad \langle S_2, s_2\rangle \to s_3}{\langle S_1 \ \texttt{par} \ S_2, s_1\rangle \to s_3} \qquad \frac{\langle S_2, s_1\rangle \to s_2 \quad \langle S_1, s_2\rangle \to s_3}{\langle S_1 \ \texttt{par} \ S_2, s_1\rangle \to s_3}$$

Although these rules look perfectly reasonable, they only manage to express that *the order in which* S_1 and S_2 are executed is arbitrary (which could be desirable for a language that allows the specification of threads). Although we are free to devide a statement S into S_1 and S_2 as we see fit, still the rules express that either first S_1 runs, and then S_2, or vice versa; the required interleaving is *not* expressed. Using Natural Semantics, we cannot describe the intuitive semantics, because Natural Semantics is defined using the *immediate constituents* of a program, not the individual computation step. In a sense, Natural Semantics is too abstract.

Exercise 8.2 Specify a SOS-style semantics for arithmetic expressions where the individual parts of an expression can be computed in parallel. Try to prove that you still obtain the result that was specified by $\mathcal{A}$.

Exercise 8.3 Consider an extension of `While` that in addition to the **`par`**-construct also contains the construct

$$\texttt{protect}\ S\ \texttt{end}$$

The idea is that the statement S has to be executed as an atomic entity; so that for example:

$$x \texttt{ := } 1 \ \texttt{par protect}\ (x \texttt{ := } 2\,\texttt{;}\, x \texttt{ := } x{+}2)\ \texttt{end}$$

only has two possible outcomes, namely 1 and 4. Extend the SOS semantics to express this. Can you specify a natural semantics for the extended language?

8.4 Blocks

We now focus on how to deal with *declarations*, the *scope* for declared identifiers and *blocks* in semantics. We start by considering a new programming language **`Block`**, that is an extension of `While`. Its abstract syntax is given by:

$$\begin{array}{rcl}
n & \in & \textit{Numeral} \\
x & \in & \textit{Variables} \\
a & \in & \textit{Arithmetic Expressions} \\
b & \in & \textit{Boolean Expressions} \\
D_v & \in & \textit{Declared Variables} \\
S & \in & \textit{Statements}
\end{array}$$

$$\begin{array}{rcl}
a & ::= & n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \\
b & ::= & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \ \& \ b_2 \\
D_v & ::= & \texttt{var}\ x \texttt{ := } a\,\texttt{;}\, D_v \mid \epsilon \\
S & ::= & x \texttt{ := } a \mid \texttt{skip} \mid S_1\,\texttt{;}\,S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \\
 & & \mid \texttt{while}\ b\ \texttt{do}\ S \mid \texttt{begin}\ D_v\ S\ \texttt{end}
\end{array}$$

The idea is that variables are local to the block in which they are declared, and it is possible to use an identifier more than once in a declaration in a program.

Example 8.4 The following is an example of a program in **Block**:

$$\texttt{\textbf{begin}}\ \texttt{\textbf{var}}\ y \ \texttt{:=}\ 1\,\texttt{;}\ x\ \texttt{:=}\ 1\,\texttt{;}\ \texttt{\textbf{begin}}\ \texttt{\textbf{var}}\ x\ \texttt{:=}\ 2\,\texttt{;}\ y\ \texttt{:=}\ x+1\ \texttt{\textbf{end}}\,\texttt{;}\ x\ \texttt{:=}\ y+x\ \texttt{\textbf{end}}$$

In more readable form, this becomes

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

The semantics of statements is defined in the same way as was done for `While`, so using the same rules as before, but with the addition of the following rule for the block construct:

$$(\textsc{block}_{ns})\quad \frac{\langle D_v, s_1\rangle \to s_2 \quad \langle S, s_2\rangle \to s_3}{\langle \texttt{\textbf{begin}}\ D_v\ S\ \texttt{\textbf{end}}, s_1\rangle \to s_3[DV(D_v) \mapsto s_1]}$$

Notice that this rules uses some features that need specification, like '$\langle D_v, s_1\rangle \to s_2$'. First of all, the function *DV*, applied to a list of variable declarations, determines the set of variables declared in D_v:

$$\begin{aligned} DV(\texttt{\textbf{var}}\ x\ \texttt{:=}\ a\,\texttt{;}\ D_v) &= \{x\} \cup DV(D_v) \\ DV(\epsilon) &= \emptyset \end{aligned}$$

and $s_1[V \mapsto s_2]$ is a state as s_1, except for at variables in the set V where it is specified by s_2:

$$s_1[V \mapsto s_2]\,x \quad = \quad \begin{cases} s_2\,x & \text{if } x \in V \\ s_1\,x & \text{if } x \notin V \end{cases}$$

For the semantics of variable declarations, we have:

$$(\textsc{decl}_{ns})\quad \frac{\langle D_v, s_1[x \mapsto \mathcal{A}[\![a]\!]s_1]\rangle \to s_2}{\langle \texttt{\textbf{var}}x\ \texttt{:=}\ a\,\texttt{;}\ D_v, s_1\rangle \to s_2}$$

$$(\textsc{no-decl}_{ns})\quad \frac{}{\langle \epsilon, s\rangle \to s}$$

Which completes the definition of Natural Semantics for **Block**. It is much more difficult to define the Structural Operational Semantics (although possible, just not done here).

8.5 Procedures

We extend the language **`Block`** to **`Proc`**, a language that allows for the definition of (parameterless) *procedure declarations* , by

$$\begin{array}{rcl} n & \in & \textit{Numeral} \\ x & \in & \textit{Variables} \\ a & \in & \textit{Arithmetic Expressions} \\ b & \in & \textit{Boolean Expressions} \\ p & \in & \textit{Procedure Names} \\ D_v & \in & \textit{Declared Variables} \\ D_p & \in & \textit{Declared Procedures} \\ S & \in & \textit{Statements} \end{array}$$

$$\begin{array}{rcl} a & ::= & n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \\ b & ::= & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \,\&\, b_2 \\ D_v & ::= & \texttt{var}\, x \texttt{ := } a \texttt{;} D_v \mid \epsilon \\ D_p & ::= & \texttt{proc}\, p \texttt{ is } S \texttt{;} D_p \mid \epsilon \\ S & ::= & x \texttt{ := } a \mid \texttt{skip} \mid S_1 \texttt{;} S_2 \mid \texttt{if}\, b \texttt{ then } S_1 \texttt{ else } S_2 \\ & & \mid \texttt{while}\, b \texttt{ do } S \mid \texttt{begin}\, D_v\, D_p\, S \texttt{ end} \mid \texttt{call}\, p \end{array}$$

For this new language, in particular for the added features, we shall give three different semantics, that differ in how a procedure declaration is dealt with:

- *Dynamic* scope for variables as well as procedures.
- Dynamic scope for variables, and *static* scope for procedures.
- Static scope for variables as well as procedures.

The difference in binding is important in programming languages where it is possible to re-use variable and procedure names inside new blocks. Imagine a program in which you call a procedure p that is defined outside the current block, that calls another procedure q that is defined *both in-* and *outside* the block.

(*Dynamic*) : With this kind of binding, only the last definition counts; the call to q will use the most recent declaration.

(*Static*) : With static binding, calling q would give you the definition that was declared in the block where p is defined.

Example 8.5 Consider the following program:

```
begin  var $x$ := $0$;
       proc $p$ is $x$ := $x \times 2$;
       proc $q$ is call $p$;
       begin  var $x$ := $5$;
              proc $p$ is $x$ := $x + 1$;
              call $q$;
              $y$ := $x$
       end
end
```

The different scope rules have the following effect for the end result of the program:

- With dynamic scope for variables as well as procedures, the result of this program will be that $y = 6$. The value of y depends on the value of x, which depends on the '**call** q' on the line above. q will do a '**call** p', which will used the 'last' declaration for p (in time), i.e. the one stating '**proc** p **is** x **:=** $x + 1$'. Since it will use the 'last' declaration for x (in time), '**var** x **:=** 5', the result of '**call** p' will be that x has value 6, which will be assigned to y.

- With dynamic scope for variables, and static scope for procedures, the result of this program will be that $y = 10$. Now q will use '**proc** p **is** x **:=** $x \times 2$', since that is the one defined in the block where q is declared. However, since it will use the 'last' declaration for x (in time), '**var** x **:=** 5', the result of **call** p will be that x has value 10, which will be assigned to y.

- With static scope for variables as well as procedures, the result of this program will be that $y = 5$. Now q will use '**proc** p **is** x **:=** $x \times 2$', since that is the one defined at the time q is declared. It will also use the local declaration for x, '**var** x **:=** 0', and multiply that variable. The assignment 'y **:=** x', however, uses *its locally declared* x, '**var** x **:=** 5', that still has value 5 (it is not touched), so the result of '**call** p' will be that y has value 5.

8.6 Dynamic / Dynamic

We introduce a procedure environment

$$\mathit{env}_p \in \mathit{Env}_p \quad = \quad \mathit{Procedure\ Names} \hookrightarrow \mathit{Statements}$$

which records the body of each procedure, by linking procedure names to statements. Since only the last declaration counts, a simple connection between procedure name and body suffices; when leaving a block, the environment of the context of the block will be used, and the connections within the block are lost.

The transition rules for the system will now be of the form:

$$\mathit{env}_p \quad \vdash \quad \langle S, s_1 \rangle \to s_2$$

(so only a minor change of the previous notation; this should look familiar).

Most of the rules are similar to the original with the addition of the extra environment information, like, for example,

$$\frac{\mathit{env}_p \vdash \langle S_1, s_1\rangle \to s_2 \quad \mathit{env}_p \vdash \langle S_2, s_2\rangle \to s_3}{\mathit{env}_p \vdash \langle S_1 \,\texttt{;}\, S_2, s_1\rangle \to s_3}$$

For the semantics of blocks, we need to be able to update the environment. *Update*$_p$ is a procedure that, given a list of procedure declarations, produces an environment that links procedure names to procedure bodies.

$$\begin{array}{rcl} \mathit{Update}_p(\texttt{proc}\; p \;\texttt{is}\; S \,\texttt{;}\, D_p, \mathit{env}_p) & = & \mathit{Update}_p(D_p, \mathit{env}_p[p \mapsto S]) \\ \mathit{Update}_p(\epsilon, \mathit{env}_p) & = & \mathit{env}_p \end{array}$$

We add two rules, one that deals with the new language construct '**call** p', the other for the blocks.

$$(\text{CALL}_{ns}^{\text{rec}}) \quad \frac{\mathit{env}_p \vdash \langle S, s_1\rangle \to s_2}{\mathit{env}_p \vdash \langle \texttt{call}\; p, s_1\rangle \to s_2} \; (\mathit{env}_p\, p = S)$$

$$(\text{BLOCK}_{ns}) \quad \frac{\langle D_v, s_1\rangle \to s_2 \quad \mathit{Update}_p\,(D_p, \mathit{env}_p) \vdash \langle S, s_2\rangle \to s_3}{\mathit{env}_p \vdash \langle \texttt{begin}\; D_v\; D_p\; S\; \texttt{end}, s_1\rangle \to s_3[DV(D_v) \mapsto s_1]}$$

Note that procedures can always be recursive, since we use the environment in the premisse of the rule $(\text{CALL}_{ns}^{\text{rec}})$. Moreover, the way in which the semantics for the structure '**begin** D_v D_p S **end**' is established shows that the statement inside is evaluated in a changed environment, where all the local declarations have been considered. For the block as a whole, the environment used is not influenced by the declarations in D_v or D_p.

8.7 Dynamic / Static

In this setting, procedures will now use those declarations for the procedures they call that are defined at the moment the procedure itself was declared, not just the last one. So we now also need to state the environment in which procedures are defined, linking the names of procedures called to their bodies and the environment in which they were defined:

$$\mathit{Env}_p \quad = \quad \mathit{Procedure\ Names} \hookrightarrow \mathit{Statements} \times \mathit{Env}_p$$

(Note that the concrete procedure environments are always built from smaller ones.) *Update*$_p$ is now defined as a procedure that, given a list of procedure declarations, produces an environment that links procedure names to a body and a procedure context.

$$\begin{array}{rcl} \mathit{Update}_p(\texttt{proc}\; p \;\texttt{is}\; S \,\texttt{;}\, D_p, \mathit{env}_p) & = & \mathit{Update}_p(D_p, \mathit{env}_p[p \mapsto (S, \mathit{env}_p)]) \\ \mathit{Update}_p(\epsilon, \mathit{env}_p) & = & \mathit{env}_p \end{array}$$

We then just need to update the rule (CALL_{ns}). If procedures in **Proc** are non-recursive, we use:

$$(\text{CALL}_{ns}) \quad \frac{\mathit{env}'_p \vdash \langle S, s_1\rangle \to s_2}{\mathit{env}_p \vdash \langle \texttt{call}\; p, s_1\rangle \to s_2} \; (\mathit{env}_p\, p = (S, \mathit{env}'_p))$$

But if procedures are recursive, we need:

$$(\text{CALL}_{ns}^{\text{rec}}) \quad \frac{env'_p[p \mapsto (S, env'_p)] \vdash \langle S, s_1 \rangle \to s_2}{env_p \vdash \langle \texttt{call}\, p, s_1 \rangle \to s_2} \ (env_p\, p = (S, env'_p))$$

Exercise 8.6 Try to construct a statement which illustrates the difference between these two rules.

8.8 Static / Static

To specify static scope not only for procedures, but also for variables, we not only use the approach defined above, but also need to model the fact that variables can appear in more than one declaration. This implies that we can no longer simply use states, that mapped variables names to values, but need to replace the state with two mappings:

$$\begin{aligned} env_v \in Env_v &= Variables \to Locations \\ store \in Store &= Locations \cup \{next\} \to \mathbb{Z} \end{aligned}$$

where, essentially, $Locations = \mathbb{Z}$, and *next* is a special token which holds the next free location. The idea is that the machine is now represented not as a mapping from variable names to values, but as a infinite number of *locations*, and a variable x now will point to a location through a mapping that is called a *variable environment*, env_v. The location $env_v\, x$ points at will hold its value through the function *store* that maps locations to values. Entering a new block can make that x will be redeclared, without losing the old declaration. This is modelled by making x point to a new location (the old location is kept, but not pointed at), and to store the new value for x there. Since we have the need for a '*new location*' on a regular basis, we also use a function '*new*' that produces the number of the next free location.

$$new : Locations \to Locations$$

$$new\, l \ = \ l + 1$$

(Note that $s = store \circ env_v$.) The initial variable environment could map all variables to location 0, and the initial store might map *next* to 1.

Since the semantics of programs now also depend on what the value (location) is that a variable is currently (in this block) pointing at, we need to consider transitions of the form

$$\langle D_v, env_v, store_1 \rangle \to (env'_v, store_2)$$

For evaluating the variable declarations, we have rules

$$\frac{\langle D_v, env_v[x \mapsto l], store[l \mapsto v][next \mapsto new\, l] \rangle \to (env'_v, store_2)}{\langle \texttt{var}\ x \ \texttt{:=}\ a\, \texttt{;}\ D_v, env_v, store \rangle \to (env'_v, store_2)}$$

$$\frac{}{\langle \epsilon, env_v, store \rangle \to (env_v, store)}$$

where

$$\begin{aligned} v &= \mathcal{A}[\![a]\!]\,(store \circ env_v) \\ l &= store\, next \end{aligned}$$

In view of the fact that also variable environments are used, we must further update procedure environments:

$$\mathit{Env}_p \quad = \quad \mathit{Procedure\ Names} \hookrightarrow \mathit{Statements} \times \mathit{Env}_v \times \mathit{Env}_p$$

$$\begin{aligned}\mathit{Update}_p\,(\textbf{proc}\ p\ \textbf{is}\ S\ \textbf{;}\ D_p, \mathit{env}_v, \mathit{env}_p) &= \mathit{Update}_p\,(D_p, \mathit{env}_v, \mathit{env}_p[p \mapsto (S, \mathit{env}_v, \mathit{env}_p)]) \\ \mathit{Update}_p\,(\epsilon, \mathit{env}_v, \mathit{env}_p) &= \mathit{env}_p\end{aligned}$$

The transition system for statements now has rules of the form

$$\mathit{env}_v, \mathit{env}_p \quad \vdash \quad \langle S, \mathit{store}_1 \rangle \to \mathit{store}_2$$

Most rules are similar to their original, like:

$$\frac{\mathit{env}_v, \mathit{env}_p \vdash \langle S_1, \mathit{store}_1 \rangle \to \mathit{store}_2 \quad \mathit{env}_v, \mathit{env}_p \vdash \langle S_2, \mathit{store}_2 \rangle \to \mathit{store}_3}{\mathit{env}_v, \mathit{env}_p \vdash \langle S_1\ \textbf{;}\ S_2, \mathit{store}_1 \rangle \to \mathit{store}_3}$$

but the rule for blocks is modified:

$$\frac{\langle D_v, \mathit{env}_v, \mathit{store}_1 \rangle \to (\mathit{env}'_v, \mathit{store}_2) \quad \mathit{env}'_v, \mathit{env}'_p \vdash \langle S, \mathit{store}_2 \rangle \to \mathit{store}_3}{\mathit{env}_v, \mathit{env}_p \vdash \langle \textbf{begin}\ D_v\ D_p\ S\ \textbf{end}, \mathit{store}_1 \rangle \to \mathit{store}_3}$$

where

$$\mathit{env}'_p \quad = \quad \mathit{Update}_p\,(D_p, \mathit{env}'_v, \mathit{env}_p)$$

(Why is there no analogue of $s_2[\mathit{DV}\,(D_v) \mapsto s]$?)

And the new rules for $\textbf{call}\ p$ are:

$$(\text{CALL}_{ns}) \quad \frac{\mathit{env}'_v, \mathit{env}'_p \vdash \langle S, \mathit{store}_1 \rangle \to \mathit{store}_2}{\mathit{env}'_v, \mathit{env}_p \vdash \langle \textbf{call}\ p, \mathit{store}_1 \rangle \to \mathit{store}_2}$$

$$(\text{CALL}_{ns}^{\text{rec}}) \quad \frac{\mathit{env}'_v, \mathit{env}'_p[p \mapsto (S, \mathit{env}'_v, \mathit{env}'_p)] \vdash \langle S, \mathit{store}_1 \rangle \to \mathit{store}_2}{\mathit{env}'_v, \mathit{env}_p \vdash \langle \textbf{call}\ p, \mathit{store}_1 \rangle \to \mathit{store}_2}$$

where

$$\mathit{env}_p \quad = \quad (S, \mathit{env}'_v, \mathit{env}'_p).$$

Exercise 8.7 Modify the synax of procedure declarations so that procedures take two *call-by-value* parameters:

$$\begin{aligned} D_p &::= \textbf{proc}\ p(x_1, x_2)\ \textbf{is}\ S\ \textbf{;}\ D_p \mid \epsilon \\ S &::= x\ \textbf{:=}\ a \mid \textbf{skip} \mid S_1\ \textbf{;}\ S_2 \mid \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2 \\ &\quad\ \mid \textbf{while}\ b\ \textbf{do}\ S \mid \textbf{begin}\ D_v\ D_p\ S\ \textbf{end} \mid \textbf{call}\ p(a_1, a_2) \end{aligned}$$

Procedure environments will now be elements of:

$$\mathit{Env}_p \quad = \quad \mathit{Procedure\ Names} \hookrightarrow \mathit{Variables} \times \mathit{Variables} \times \mathit{Statements} \times \mathit{Env}_v \times \mathit{Env}_p$$

Modify the semantics to handle this language. In particular, provide new rules for procedure calls: one for non-recursive and another for recursive procedures.