Operational Semantics

Lecturer: Dr. Steffen van Bakel
Room: Huxley 425

Spring 2001

Recommended book:

Semantics with Applications A Formal Introduction

by Hanne Riis Nielson and Flemming Nielson

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

Lecture time

Monday: 10:00-11:00 am

Thursday: 10:00-12:00 am

Tutorial

Will not be organised separately; exercises are a part of the course.

Question time: Anytime during those two days.

Course Design

- Preliminaries:
  Introduction, Induction, **`While`**.
- Natural Semantics
- Structural Operational Semantics
- Correct implementation
- Denotational Semantics
- Extensions to **`While`**:
  Parallelism, Blocks and Procedures

Why formal semantics?

Syntax - 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 - Meaning of sentences

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

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

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

1.) the user with an unambiguous description of the effect of a program.
2.) a yardstick for implementation.
3.) a basis for program analysis and synthesis.
   - Transformation.
   - Optimisation.
   - Verification.

States

Semantics deals with the meaning of programs that are executing. We need the status of the memory during execution.

The *state* of the memory s is a function that maps variable names to values;

State is 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.

There are three approaches

Operational Semantics - How the effect of a computation is produced- abstraction of machine execution.

Denotational Semantics - What is the effect of a computation.

Axiomatic 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.

Operational Semantics

- To execute a sequence of statements separated by '`;`', execute the individual statements one after the other from left to right.
- To execute 'x `:=` y', determine the value of y and assign it to x.

Notation '$\langle p, s \rangle$', to be read as 'the semantics of program p in state s'.

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

Denotational Semantics

Effect of a program is a function in *State* $\to$ *State*.

- The effect of '`;`' is the functional composition:

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

  (Notice the inversion of s_1 and s_2.)

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

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

In other words,

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

Take $s_1 = [x \mapsto 5, y \mapsto 7, z \mapsto 0]$, $s_2 = s_1[z \mapsto 5]$, $s_3 = s_2[x \mapsto 7]$, and $s_4 = s_3[y \mapsto 5]$, then

$$\begin{aligned} \mathcal{S}_{ds}\,[\![z := x\ \texttt{;}\ x := y\ \texttt{;}\ y := z]\!]\,(s_1) &= \\ \mathcal{S}_{ds}\,[\![y := z]\!] \circ \mathcal{S}_{ds}\,[\![x := y]\!] \circ \mathcal{S}_{ds}\,[\![z := x]\!]\,(s_1) &= \\ \mathcal{S}_{ds}\,[\![y := z]\!] \circ \mathcal{S}_{ds}\,[\![x := y]\!]\,(s_2) &= \\ \mathcal{S}_{ds}\,[\![y := z]\!]\,(s_3) &= s_4 \end{aligned}$$

Axiomatic Semantics

Partial correctness (with respect to *Pre-* and *Post-condition*)

Take

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

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

But how to work on

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

is not easy to see.

Mathematical Induction

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

Base case - Prove $P(0)$.

Inductive Case - For every k, using the assumption that $P(k)$ holds, prove $P(k+1)$.

Then $P(n)$ holds for all n.

In Logic :

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

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

Proof : By induction.

Base case - Trivial.

Inductive Case -

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

Why does this work?

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.

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

Theorem: *Suppose X satisfies the properties*

- $0 \in X$
- *if $x \in X$, then $x + 1 \in X$.*

then IN $\subseteq X$.

Proof : By mathematical induction over IN. Take $k \in$ IN. Then either $k = 0$, or $k = k' + 1$, with $k' \in$ IN. To show: $k \in X$.

$k = 0$ - By definition of X, we have $0 \in X$.

$k = k' + 1$ - Since $k' \in$ IN, by induction also $k' \in X$. Then, by definition of X, also $k + 1 \in X$.

So, for all $k \in$ IN, $k \in X$.

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

Theorem: *There are infinitely many prime numbers.*

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 $P(k+1)$.

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

In Logic:

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

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

Other induction

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

Example: Ev is the least set such that:

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

Induction for Ev: 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* $P(k+2)$.

Theorem: *If* $n \in$ *Ev and* $m \in$ *Ev, then* $n+m \in$ *Ev.*

Proof : $P(x) = \forall m \in$ *Ev*$.[x+m \in$ *Ev*$]$. 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 also $\forall m \in$ *Ev*$.[(x+m)+2 \in$ *Ev*$]$.

So $\forall n \in$ *Ev*$.\forall m \in$ *Ev*$.[n+m \in$ *Ev*$]$.

Structural induction

Take: *List*(IN).

- $[\,] \in$ *List*(IN).
- If $n \in$ IN, and $l \in$ *List*(IN), then $n : l \in$ *List*(IN).

An other representation technique is through *rules*:

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

that has the intended meaning:

if *premises*, then *conclusions*.

An inductive proof for $P(l)$ with $l \in$ *List*(IN), would follow:

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

Inductive Case - Assuming $P(l)$, prove $P(n : l)$.

Example: *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
```

Let $P(l) = \texttt{sum}\, l \leq \texttt{maximum}\, l \times \texttt{length}\, l$, *then for all* $l \in$ *List*(ℕ), $P(l)$.

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

Inductive Case -

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

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 - *constants* $a_1, \ldots, a_m$ are assumed 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:

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

Closure - X is defined as the smallest set satisfying the above two rules.

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 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:

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

or, alternatively

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

Constructors are '$\cdot \times \cdot$' and '$(\cdot)$', '$\texttt{zero}$' is a constant.

Property: $P(E) = \#'(' \text{ in } E = \#')' \text{ in } E$.

Proof : Induction on the structure of *Exp*.

Base case - $\#'(' \text{ in } \texttt{zero} = 0 = \#')' \text{ in } \texttt{zero}$.

Inductive Case 1 - $E = E_1 \times E_2$, and by induction $P(E_1)$ and $P(E_2)$. Let $\#'(' \text{ in } E_1 = n_1 = \#')' \text{ in } E_1$, and $\#'(' \text{ in } E_2 = n_2 = \#')' \text{ in } E_2$. Then $\#'(' \text{ in } E = n_1 + n_2 = \#')' \text{ in } E$.

Inductive Case 2 - $E = (E_1)$, and by induction $P(E_1)$. Let $\#'(' \text{ in } E_1 = n_1 = \#')' \text{ in } E_1$, then $\#'(' \text{ in } E = n + 1 = \#')' \text{ in } E$.

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

$$S ::= *E*$$
$$E ::= +S \mid **$$

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

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

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

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.

How to use rules

(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'.

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

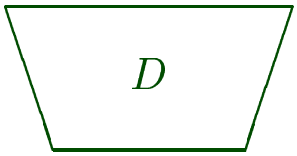
These rules describe how to *build derivations*.

For any set Γ of formulae, if A occurs in Γ, then

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

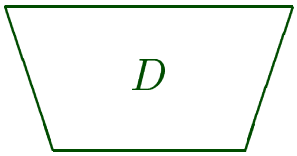
is a correct derivation.

Suppose we have *derived* $\Gamma \cup \{A\} \vdash B$.



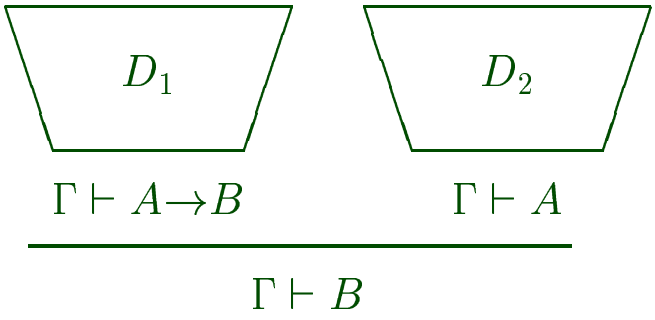
$$\Gamma \cup \{A\} \vdash B$$

then by the second rule $(\rightarrow I)$, the following is a correct derivation:



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

And, similarly, if we have D_1 and D_2, one for $\Gamma \vdash A \to B$, the other for $\Gamma \vdash A$, then by $(\to E)$, the following is a correct derivation.



These three steps are the only permitted to construct derivations; 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.

Correct denotation for objects defined by these rules would be $D :: \Gamma \vdash A$. We use $\Gamma \vdash A$ 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. An inductive proof over the structure of derivations would have 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*.

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

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

($\rightarrow I$) - We have a derivation D' of the structure

$$\dfrac{\begin{array}{c}\boxed{D}\\ \Gamma \cup \{A\} \vdash B\end{array}}{\Gamma \vdash A \rightarrow B}$$

The derivation D with conclusion $\Gamma \cup \{A\} \vdash B$ is a subderivation of D' for $\Gamma \vdash A \rightarrow B$. We can assume that the property to prove holds for D, and use that to prove that it holds for D'.

($\rightarrow E$) - We have a derivation D' of the structure

$$\dfrac{\begin{array}{cc}\boxed{D_1} & \boxed{D_2}\\ \Gamma \vdash A \rightarrow B & \Gamma \vdash A\end{array}}{\Gamma \vdash B}$$

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'.

(Apparent) Static and Reverse Induction

Take the 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: *Show: If $M \sim N$, then either $M \equiv N$, or $M \equiv A$ and $N \equiv B$, or $M \equiv B$ and $N \equiv A$.*

Exercise: *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 $\mathbb{N} \subseteq X$.

Concrete syntax

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

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

Ambiguous: precedence $4 \times 2 - 1$, *associativity* .

Instead, take

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

The bracketing now forces the computation.

Abstract syntax

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

Syntactic categories

$$e \in \textit{exp}$$
$$\textbf{op} \in \textit{Op}$$
$$n \in \textit{Numeral}$$

Definitions

$$\textbf{op} ::= + \mid - \mid \times \mid /$$
$$e ::= n \mid (e_1 \ \textbf{op} \ e_2)$$

Natural Semantics

Syntactic Categories:

$$a \in \textit{Arithmetic Expressions}$$
$$n \in \textit{Numeral}$$
$$x \in \textit{Variables}$$

where *Variables* is an infinite set of variable names.

Definitions:

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

Natural Semantics of Arithmetic Expressions

Semantics as a transition system.

Configurations:

$\langle a, s \rangle$ – *Arithmetic Expressions* and *state*

v – *an integer* (in $\mathbb{Z}$, the final state)

We consider *two* sets of 'numbers'.

- *Numeral*, syntactic representation of numbers.
- $\mathbb{Z}$, the actual numbers.

For example, the forty-second positive number can be represented as $42_{10} = 101010_2 = 222_4 = 52_8 = 2A_{16} = 10_{42}$. We will use underline to distinguish. So, $\underline{4} \in$ *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 $\mathbb{N}$.

Natural Semantics deals 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).

The rules :

$$(\text{NUM}_{ns}) \qquad \frac{}{\langle n, s\rangle \to \underline{n}}$$

$$(\text{VAR}_{ns}) \qquad \frac{}{\langle x, s\rangle \to s\, x}$$

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

where '**op**' is any of '$+, \times, -$'. 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: *Semantics for $(3 \times x) + 8$, in a state s such that $s\, x = 4$. The last rule applied: $(+_{ns})$.*

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

The last rule for the left-hand result was: $(\times_{ns})$.

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

By rules (NUM_{ns}) and (VAR_{ns}), the three lacking derivations are easy to construct:

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

Determinism

The Natural Semantics for natural numbers is deterministic:

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

Proof : By induction on the structure of derivations, where we focus on the last rule applied.

Exercise: This semantics is terminating, i.e., for every $a \in$ *Arithmetic Expressions, for all states* s*, there is a* v *such that* $\langle a, s\rangle \to v$*.*

Denotational Semantics for Arithmetic Expressions

Memory is modeled by functions of type

$$\mathit{State} = \mathit{Variables} \to \mathbb{Z}.$$

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

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

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

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

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

Free variables

$$
\begin{aligned}
\mathit{FV}\,(n) &= \emptyset \\
\mathit{FV}\,(x) &= \{x\} \\
\mathit{FV}\,(a_1 \textbf{ op } a_2) &= \mathit{FV}\,(a_1) \cup \mathit{FV}\,(a_2)
\end{aligned}
$$

Theorem: *Let s and s' be such that $s\,x = s'x$, for all $x \in \mathit{FV}\,(a)$. Then $\mathcal{A}\,[\![a]\!]\,s = \mathcal{A}\,[\![a]\!]\,s'$.*

Proof : By induction on the structure of terms in *Arithmetic Expressions*.

The language **While**

Abstract syntax:

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

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

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

Example:

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

Semantics of Boolean expressions

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

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

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

Natural Semantics of `While`

Configurations:

$\langle S, s\rangle$ – S is to be executed from state s,

s – a terminal state, or value.

Transitions: $\langle S, s\rangle \to s'$ Rules: (extended by (NUM_{ns}), (VAR_{ns}), and (OP_{ns}))

(SKIP_{ns}) $\dfrac{}{\langle \textbf{skip}, s\rangle \to s}$ (ASS_{ns}) $\dfrac{\langle a, s\rangle \to v}{\langle x := a, s\rangle \to s[x \mapsto v]}$

(COMP_{ns}) $\dfrac{\langle S_1, s_1\rangle \to s_2 \;\; \langle S_2, s_2\rangle \to s_3}{\langle S_1 \,\textbf{;}\, S_2, s_1\rangle \to s_3}$

($\text{WHILE}^{\text{F}}_{ns}$) $\dfrac{}{\langle \textbf{while}\ b\ \textbf{do}\ S, s\rangle \to s}\ \mathcal{B}[\![b]\!]\, s_1 = \textbf{ff}$

($\text{WHILE}^{\text{T}}_{ns}$) $\dfrac{\langle S, s_1\rangle \to s_2 \langle \textbf{while}\ b\ \textbf{do}\ S, s_2\rangle \to s_3}{\langle \textbf{while}\ b\ \textbf{do}\ S, s_1\rangle \to s_3}\ \mathcal{B}[\![b]\!]\, s_1 = \textbf{tt}$

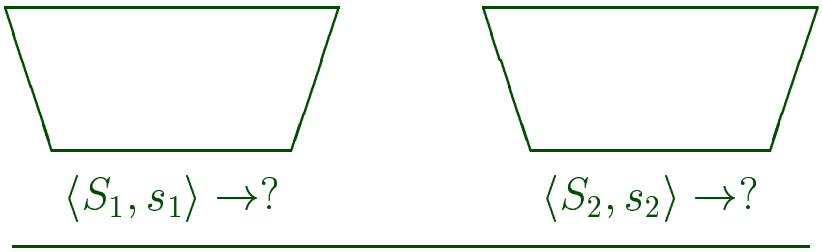
($\text{COND}^{\text{T}}_{ns}$) $\dfrac{\langle S_1, s_1\rangle \to s_2}{\langle \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2, s_1\rangle \to s_2}\ \mathcal{B}[\![b]\!]\, s_1 = \textbf{tt}$

($\text{COND}^{\text{F}}_{ns}$) $\dfrac{\langle S_2, s_1\rangle \to s_2}{\langle \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2, s_1\rangle \to s_2}\ \mathcal{B}[\![b]\!]\, s_1 = \textbf{ff}$

When looking for the semantics of a certain statement S in a specific state s, we would first need to construct the derivation

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

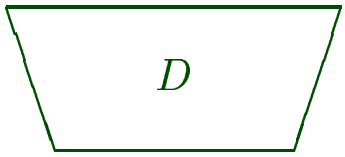
The syntax of S guides the construction of the derivation



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

We will, normally, end up looking for derivations for $\langle n, s'\rangle \to ?$ or $\langle \texttt{skip}, s'\rangle \to ?$, which is easy.

The only problem is rule ($\text{WHILE}^{\text{T}}_{ns}$).



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

Suppose $\mathcal{B}[\![b]\!]\, s = \mathbf{tt}$, then the derivation is as follows:

$$\frac{\begin{array}{cc} \boxed{D_1} & \boxed{D_2} \\ \langle S, s\rangle \to ?_1 & \langle \mathbf{while}\ b\ \mathbf{do}\ S, ?_1\rangle \to ?_2 \end{array}}{\langle \mathbf{while}\ b\ \mathbf{do}\ S, s\rangle \to ?_2}$$

Does this always terminate?

$$\frac{\boxed{D}}{\langle \mathbf{while\ true\ do\ skip}, s\rangle \to ?}$$

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

$$\frac{\begin{array}{cc} \quad & \boxed{D_2} \\ \langle \mathbf{skip}, s\rangle \to s & \langle \mathbf{while\ true\ do\ skip}, s\rangle \to ? \end{array}}{\langle \mathbf{while\ true\ do\ skip}, s\rangle \to ?}$$

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 $\langle S, s\rangle \to s'$.

Semantic Equivalence

Two statements S_1 and S_2 are *semantically equivalent* if for all states s and s':

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

Lemma: *The statements*

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

and

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

are semantically equivalent.

Proof : On the Black/White Board.

Theorem: *If* $\langle S, s\rangle \to s^1$ *and* $\langle S, s\rangle \to s^2$*, then* $s^1 = s^2$*.*

Proof : On the Black/White Board.

Meaning of statements:

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

$$\begin{aligned} \mathcal{S}_{ns}\,[\![S]\!]\,s &= s', && \text{if } \langle S, s\rangle \to s' \\ &= \mathit{undef}, && \text{otherwise} \end{aligned}$$

Well-defined, since $\langle \cdot, \cdot\rangle \to \cdot$ is deterministic.

Structural Operational Semantics

Focuses on individual steps of the execution. Transitions: $\langle S, s\rangle \Rightarrow \gamma$, γ of the form $\langle S', s'\rangle$ or s'.

A configuration is stuck if there is no γ such that $\langle S, s\rangle \Rightarrow \gamma$.

Rules:

$$(\text{ASS}_{sos}) \quad \langle x := a, s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$$

$$(\text{SKIP}_{sos}) \quad \langle \mathbf{skip}, s\rangle \Rightarrow s$$

$$(\text{COMP}^T_{sos}) \quad \frac{\langle S_1, s_1\rangle \Rightarrow s_2}{\langle S_1 \,;\, S_2, s_1\rangle \Rightarrow \langle S_2, s_2\rangle}$$

$$(\text{COMP}^I_{sos}) \quad \frac{\langle S_1, s_1\rangle \Rightarrow \langle S_1', s_2\rangle}{\langle S_1 \,;\, S_2, s_1\rangle \Rightarrow \langle S_1' \,;\, S_2, s_2\rangle}$$

$$(\text{WHILE}_{sos}) \quad \langle \mathbf{while}\ b\ \mathbf{do}\ S, s\rangle \Rightarrow \langle \mathbf{if}\ b\ \mathbf{then}\ (S\ ;\mathbf{while}\ b\ \mathbf{do}\ S)\ \mathbf{else}\ \mathbf{skip}, s\rangle$$

$$(\text{COND}^T_{sos}) \quad \langle \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2, s\rangle \Rightarrow \langle S_1, s\rangle, \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

$$(\text{COND}^F_{sos}) \quad \langle \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2, s\rangle \Rightarrow \langle S_2, s\rangle, \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

Sequences

Sequence of S in s: *configurations* $\gamma_0, \gamma_1, \gamma_2, \ldots$, such that $\gamma_0 = \langle S, s \rangle$, $\gamma_i \Rightarrow \gamma_{i+1}$ for $0 \leq i$.

A sequence is a *derivation sequence* if:

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

Notation: $\gamma_0 \Rightarrow^i \gamma_i$ $\gamma_0 \Rightarrow^* \gamma_i$

Also, for each step there is a derivation tree.

Remark:

- no stuck configurations for the Structural Operational Semantics.
- semantics is deterministic (later), so only 1 derivation sequence for each configuration.

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: *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 of derivation sequences.

S_1 and S_2 are *semantically equivalent* if for all states s:

- $\langle S_1, s\rangle \Rightarrow^* \gamma$ if and only if $\langle S_2, s\rangle \Rightarrow^* \gamma$ (γ 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.

Meaning of statements:

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

$$\begin{aligned} \mathcal{S}_{sos}[\![S]\!]\, s &= s', && \text{if } \langle S, s\rangle \Rightarrow^* s' \\ &= \mathit{undef}, && \text{otherwise} \end{aligned}$$

Equivalence of the two semantics.

Theorem: *For every* S, $\mathcal{S}_{ns}[\![S]\!] = \mathcal{S}_{sos}[\![S]\!]$.

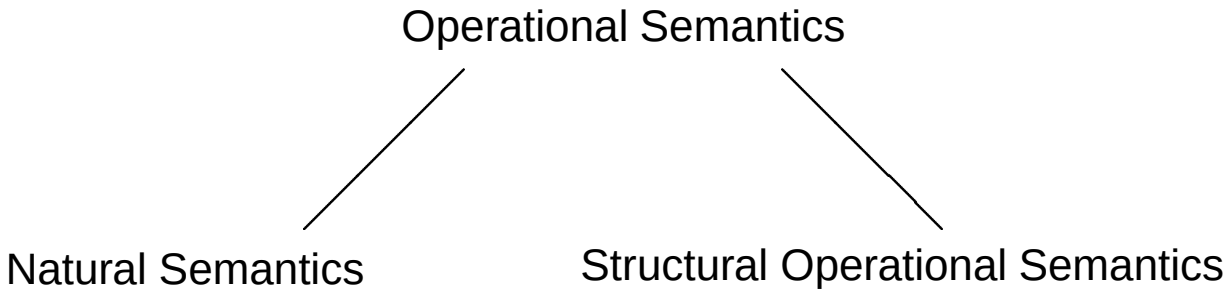
Follows from the following lemmae.

Lemma: *If* $\langle S_1, s_1\rangle \Rightarrow^k s_2$, *then* $\langle S_1 \,;\, S_2, s_1\rangle \Rightarrow^k \langle S_2, s_2\rangle$.

Lemma: $\langle S, s_1\rangle \to s_2$ *implies* $\langle S, s_1\rangle \Rightarrow^* s_2$.

Lemma: $\langle S, s_1\rangle \Rightarrow^k s_2$ *implies* $\langle S, s_1\rangle \to s_2$.

Comparison



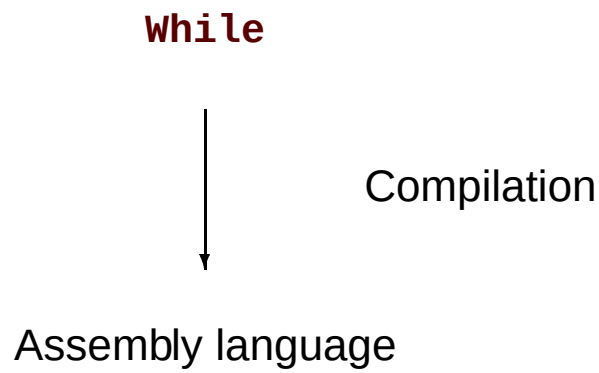
Natural Semantics - Gives, for each statement, a mapping from an intial state to the final state; it gives no detail on how the latter is obtained from the first.

It can be used to show program equivalence, etc . . . .

Structural Operational Semantics - Gives a derivation sequence $\gamma_0 \Rightarrow \gamma_1 \Rightarrow$, where each ' $\Rightarrow$ ' represents a small step in the computation.

In SOS, we can express some properties that are impossible to express in Natural Semantics, like parallelism (later).

Provably correct implementation



Steps

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

Correctness : if we translate a program into code and execute that code on the abstract machine, we get the same result as was specified by $\mathcal{S}_{ns}$ or $\mathcal{S}_{sos}$.

The abstract machine $\mathcal{AM}$ has configurations $\langle c, e, s \rangle$ where

- c is the *code* to be executed,
- e is the *evaluation stack*, and
- s is the *storage*.

The evaluation stack is used to evaluate arithmetic and boolean expressions:

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

We also have $s \in \mathit{State}$, as before.

The abstract machine language

Syntactic categories:

$$\textit{inst} \in \textit{Machine Instructions}$$
$$c \in \textit{Code} \qquad \text{sequence of instructions.}$$

Abstract syntax:

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

Configurations:

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

$\langle c, e, s \rangle$ is terminal if $c \equiv \epsilon$.

The transition relation $\triangleright$:

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

specifies how to execute instructions.

$$
\begin{aligned}
\langle \textbf{PUSH}{-}n : c, e, s\rangle &\ \triangleright\ \langle c, \mathcal{N}\,[\![n]\!] : e, s\rangle \\
\langle \textbf{ADD} : c, z_1 : z_2 : e, s\rangle &\ \triangleright\ \langle c, z_1 + z_2 : e, s\rangle \\
&\ldots \\
\langle \textbf{TRUE} : c, e, s\rangle &\ \triangleright\ \langle c, \textbf{tt} : e, s\rangle \\
\langle \textbf{FALSE} : c, e, s\rangle &\ \triangleright\ \langle c, \textbf{ff} : e, s\rangle \\
\langle \textbf{EQ} : c, z_1 : z_2 : e, s\rangle &\ \triangleright \\
&\quad \langle c, (z_1 = z_2) : e, s\rangle \text{ if } z_1, z_2 \in \mathbb{Z} \\
&\ldots \\
\langle \textbf{AND} : c, b_1 : b_2 : e, s\rangle &\ \triangleright \\
&\quad \langle c, (b_1 \ \&\ b_2) : e, s\rangle \text{ if } b_1, b_2 \in \mathbb{T} \\
&\ldots \\
\langle \textbf{FETCH}{-}x : c, e, s\rangle &\ \triangleright\ \langle c, (s\ x) : e, s\rangle \\
\langle \textbf{STORE}{-}x : c, z : e, s\rangle &\ \triangleright\ \langle c, e, s[x \mapsto z]\rangle \\
\langle \textbf{NOOP} : c, e, s\rangle &\ \triangleright\ \langle c, e, s\rangle \\
\langle \textbf{BRANCH}\,(c_1, c_2) : c, b : e, s\rangle &\ \triangleright\ \langle c_1 : c, e, s\rangle \text{ if } b = \textbf{tt} \\
\langle \textbf{BRANCH}\,(c_1, c_2) : c, b : e, s\rangle &\ \triangleright\ \langle c_2 : c, e, s\rangle \text{ if } b = \textbf{ff} \\
\langle \textbf{LOOP}\,(c_1, c_2) : c, e, s\rangle &\ \triangleright \\
&\quad \langle c_1 : \textbf{BRANCH}\,(c_2, \textbf{LOOP}\,(c_1, c_2), \textbf{NOOP}) : c, e, s\rangle
\end{aligned}
$$

Define a *computation sequence* by analogy to a derivation sequence.

Note: initial configurations always have an empty evaluation stack.

Example: *Take the code*

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

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

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

$\texttt{LOOP}\ (\texttt{TRUE}, \texttt{NOOP})$ is non-terminating.

$\langle \texttt{ADD}, \epsilon, s\rangle$ is stuck.

By analogy with the SOS-style semantics of **`While`** we can prove properties of $\mathcal{AM}$.

Lemma:

$$\langle c_1, e_1, s\rangle \rhd^k \langle c', e', s'\rangle \Rightarrow$$
$$\langle c_1 : c_2, e_1 : e_2, s\rangle \rhd^k \langle c' : c_2, e' : e_2, s'\rangle$$

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

Theorem: *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.

The meaning of a sequence of instructions can be expressed as a partial function from *State* to *State*.

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

$$\mathcal{M}[\![c]\!]\, s = s', \quad \text{if } \langle c, \epsilon, s\rangle \rhd \langle \epsilon, \epsilon, s'\rangle$$
$$= \mathit{undef}, \text{ otherwise}$$

Translation of expressions

$\mathcal{CA}$: *Arithmetic Expressions* $\rightarrow$ *Code*

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

$\mathcal{CB}$: *Boolean Expressions* $\rightarrow$ *Code*

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

Translation of statements

$$\mathcal{CS} \;:\; \mathit{Statements} \to \mathit{Code}$$

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

The meaning of a statement S can now be obtained by first translating it into code for $\mathcal{AM}$ and next executing the code on the abstract machine:

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

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

Correctness of translation

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

or

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

These two results need proof, and express that, if we first translate a statement into code for $\mathcal{AM}$ 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

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

and

$$\langle \mathcal{CB}\,[\![b]\!], \epsilon, s\rangle \quad \rhd^* \quad \langle \epsilon, \mathcal{B}\,[\![b]\!]\,s, s\rangle$$

and use these results to show

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

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: $\langle \mathcal{CA}[\![a]\!], \epsilon, s\rangle \ \triangleright^* \ \langle \epsilon, \mathcal{A}[\![a]\!]\, s, s\rangle$

Proof : By induction on the structure of arithmetic expressions. Booleans are dealt with in a similar way.

Correctness for Statements

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

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

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.

Lemma: *For every* S, s_1, s_2,

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

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

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.

Denotational Semantics

Operational approach: *how* a program is executed. Denotational approach: *the effect* 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.

Examples: $\mathcal{A}$, $\mathcal{B}$, and non-examples: $\mathcal{S}_{ns}$, $\mathcal{S}_{sos}$.

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

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

id is the identity function on states.

$$\begin{aligned}
&\mathcal{S}_{ds}[\![S_1 \texttt{ ; } S_2]\!]\, s \\
&\quad = (\mathcal{S}_{ds}[\![S_2]\!] \circ \mathcal{S}_{ds}[\![S_1]\!])\, s \\
&\quad = \mathcal{S}_{ds}[\![S_2]\!]\, (\mathcal{S}_{ds}[\![S_1]\!]\, s) \\
&\quad = \begin{cases} s_2, & \text{if } \mathcal{S}_{ds}[\![S_1]\!]\, s = s_1, \text{ and} \\ & \mathcal{S}_{ds}[\![S_2]\!]\, s_1 = s_2. \\ \mathit{undef}, & \text{if } \mathcal{S}_{ds}[\![S_1]\!]\, s = \mathit{undef}, \text{ or} \\ & \mathcal{S}_{ds}[\![S_1]\!]\, s = s_1, \text{ but} \\ & \mathcal{S}_{ds}[\![S_2]\!]\, s_1 = \mathit{undef}. \end{cases}
\end{aligned}$$

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

What is the effect of '**while** b **do** S'. We want:

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

Not compositional. Assume $\mathcal{S}_{ds}\,[\![\textbf{while}\ b\ \textbf{do}\ S]\!]$ is a function f. Then the equation above expresses that this f should *at least* satisfy:

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

We can now define a functional F as follows:

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

Then, in particular:

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

Then f is a *fixed point* of F.

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

Example: *Fixed points of functions do exist:*

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

We will define a special function, *Fix*, that, given an input function f, constructs the fixed point of f. The intention is that *Fix* F is a fixed point of F:

$$F(\mathit{Fix}\ F) \;=\; \mathit{Fix}\ F$$

Let f be defined by

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

then we can define F by:

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

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

The intended types for the functions mentioned are:

$$\begin{aligned} f &: * \to * && (f \text{ is a function}) \\ F &: (* \to *) \to * \to * \\ \mathit{Fix} &: ((* \to *) \to * \to *) \to * \to * \end{aligned}$$

Unfortunately, this does not suffice:

- There are functionals which have *more than one* fixed point. For example, the function $fx = x$ has infinitely many fixed points, and $fx = e^x - 1$ has two.
- There are functionals which have *no* fixed points at all. For example, let $g_1 \neq g_2$, and define G by:

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

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.

Fixed Point Construction

Remember that we want the denotational semantics to satisfy:

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

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

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

So $\mathcal{S}_{ds}\,[\![\textbf{while}\ b\ \textbf{do}\ S]\!]$ should be a fixed point of F, where F is defined by:

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

Example: '**while** $\neg(x = 0)$ **do skip**'. *The intended semantics for this program, using the construction discussed above,* f *is:*

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

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*

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

is a fixed point of F*:*

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

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

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

*Take $f_0\,s = \mathit{undef}$, 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.*

Example: *Take '*$\texttt{while}\ x = 0\ \texttt{do}\ x\ \texttt{:=}\ 5$*'. Following the above definition, we get:*

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

Take $f_0\,s =$ *undef, for all* s*. Then*

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

and

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

applying the construction again, we get:

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

So $f_2 = f_3$, and, therefore, f_2 is a fixed point of F. Also, f_2 is the intended semantics for '`while` $x = 0$ `do` x `:=` 5'.

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

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

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

$$f_1 = F\,f_0 = \begin{cases} \mathit{undef} & \text{if } s\,x > 0 \\ s, & \text{if } s\,x \leq 0 \end{cases}$$

and (where $s' = s[x \mapsto (s\,x - 1)]$*)*

$$f_2\,s = F\,f_1\,s = \begin{cases} f_1\,(s'), & \text{if } s\,x > 0 \\ s, & \text{if } s\,x \leq 0 \end{cases}$$
$$= \begin{cases} \left\{ \begin{array}{ll} \mathit{undef} & \text{if } s'\,x > 0 \\ s', & \text{if } s'\,x \leq 0 \end{array} \right\} & \text{if } s\,x > 0 \\ s, & \text{if } s\,x \leq 0 \end{cases}$$
$$= \begin{cases} \mathit{undef} & \text{if } s\,x > 1 \\ s[x \mapsto 0], & \text{if } s\,x = 1 \\ s, & \text{if } s\,x \leq 0 \end{cases}$$

Applying the construction again, we get:

$$
\begin{aligned}
f_3 = F\, f_2 \;&= \begin{cases} f_2\, s', & \text{if } s\, x > 0 \\ s, & \text{if } s\, x \leq 0 \end{cases} \\
&= \begin{cases} \left\{ \begin{array}{ll} \textit{undef} & \text{if } s'\, x > 1 \\ s'[x \mapsto 0], & \text{if } s'\, x = 1 \\ s', & \text{if } s'\, x \leq 0 \end{array} \right\} & \text{if } s\, x > 0 \\ s, & \text{if } s\, x \leq 0 \end{cases} \\
&= \begin{cases} \textit{undef} & \text{if } s\, x > 2 \\ s[x \mapsto 0], & \text{if } s\, x = 1, 2 \\ s, & \text{if } s\, x \leq 0 \end{cases}
\end{aligned}
$$

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

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

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 **`:=`** $x - 1$’.*

Partial order relations

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

$$g_1 \sqsubseteq g_2$$

means that 'if $g_1\, s_1 = s_2,$ then $g_2\, s_1 = s_2$' which expresses two properties:

- the domain of g_1 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:

Take

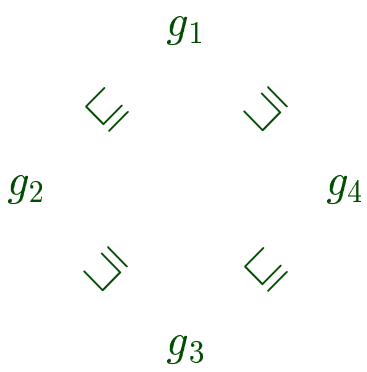
$$g_1\, s = s$$

$$g_2\, s = \begin{cases} s, & \text{if } s\, x \geq 0 \\ \textit{undef}, & \text{otherwise} \end{cases}$$

$$g_3\, s = \begin{cases} s, & \text{if } s\, x = 0 \\ \textit{undef}, & \text{otherwise} \end{cases}$$

$$g_4\, s = \begin{cases} s, & \text{if } s\, x \leq 0 \\ \textit{undef}, & \text{otherwise} \end{cases}$$

then



Partial Ordered Set

A *partial ordered set* is a pair $\langle D, \sqsubseteq \rangle$ such that

- $\forall d \in D \;.\; d \sqsubseteq d$
- $\forall d_1, d_2, d_3 \in D \;.\; d_1 \sqsubseteq d_2 \;\&\; d_2 \sqsubseteq d_3 \;\Rightarrow\; d_1 \sqsubseteq d_3$
- $\forall d_1, d_2 \in D \;.\; d_1 \sqsubseteq d_2 \;\&\; d_2 \sqsubseteq d_1 \;\Rightarrow\; d_1 = d_2$

If $\forall d' \in D \;.\; d \sqsubseteq d'$, d is called a *least element* of D:

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

The least element of D is $\bot_D$ or just $\bot$.

Exercise: 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: *$\langle State \to State, \sqsubseteq \rangle$ is a poset, and $\bot s = undef$ is its least element.*

Fix F needs 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 *Fix* $F \sqsubseteq g$.

Least Upper Bounds

Consider $\langle D, \sqsubseteq \rangle$ and $Y \subseteq D$. If $\forall d' \in Y \,.\, d' \sqsubseteq d$, then d is an *upper bound* of Y. d is the *least upper bound* (*lub*) of Y if and only if:

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

We denote the *lub* of Y as $\sqcup Y$.

Chains

We call Y a *chain* if $\forall d_1, d_2 \in Y \,.\, d_1 \sqsubseteq d_2 \vee d_2 \sqsubseteq d_1$.

Example: $\langle \mathcal{P}(\{a, b, c\}), \subseteq \rangle$.

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

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

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*

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

CCPO

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

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

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

Exercise: $\mathit{State} \hookrightarrow \mathit{State}$ *is not a complete lattice.*

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

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

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:

X	$\{a,b,c\}$	$\{a,b\}$	$\{a,c\}$	$\{b,c\}$	$\{a\}$	$\{b\}$	$\{a\}$	$\emptyset$
$f_1 X$	$\{d,e\}$	$\{d\}$	$\{d,e\}$	$\{d,e\}$	$\{d\}$	$\{d\}$	$\{e\}$	$\emptyset$
$f_2 X$	$\{d\}$	$\{d\}$	$\{d\}$	$\{e\}$	$\{d\}$	$\{e\}$	$\{e\}$	$\{e\}$

Exercise: 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: *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,*

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

Exercise: Prove this lemma.

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

$$\sqcup_2\{f\,d \mid d \in Y\} \;=\; f(\sqcup_1 Y)$$

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

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

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

Clearly, f *is monotone. But consider the set*

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

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

$$\sqcup\{f\,X \mid X \in Y\} \;=\; \sqcup\{X \mid X \in Y\} \;=\; \sqcup Y \;=\; \mathbb{N}$$

But

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

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:

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

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: *$\langle D_1, \sqsubseteq_1 \rangle$, $\langle D_2, \sqsubseteq_2 \rangle$ 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.*

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

$$f^0 = \mathit{id}$$
$$f^{n+1} = f \circ f^n, \text{ for } n \geq 0$$

Then

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

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

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*:

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

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

Only well-defined in the context of *continuous functions*.

Example: *Consider the function*

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

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

$$F^0\,s \;=\; \mathit{id}\,\bot\,s \;=\; \bot\,s \;=\; \mathit{undef}$$

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

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

$$F^2 \bot\, s \;=\; (F \circ F^1)\bot\, s = \begin{cases} F^1 s, & \text{if } s\,x \neq 0 \\ s, & \text{if } s\,x = 0 \end{cases}$$

$$= \begin{cases} \left\{ \begin{array}{ll} \mathit{undef}, & \text{if } s\,x \neq 0 \\ s, & \text{if } s\,x = 0 \end{array} \right\} & \text{if } s\,x \neq 0 \\ s, & \text{if } s\,x = 0 \end{cases}$$

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

So:

$$\sqcup\{F^n \bot \mid n \geq 0\} \;=\; \sqcup\{F^0 \bot, F^1 \bot\} \;=\; F^1 \bot \;=\; \mathit{Fix}\,F$$

Existence

We need to show that:

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

is continuous. So:

Lemma: *Let $h : State \hookrightarrow State$, $b : State \to \mathbb{T}$, and define $F\,g = \mathit{cond}\,(b, g, h)$. Then F is continuous.*

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

Theorem: *$\mathcal{S}_{ds}$ is a total function.*

Thus, from the fixed point theorem, *Fix* F 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 $\mathcal{S}_{ds}\,[\![S_1]\!] = \mathcal{S}_{ds}\,[\![S_2]\!]$

Example: $S; \texttt{skip}$ *and* S *are semantically equivalent.*

Equivalence to Operational Semantics

Lemma: *Notice that:*

- *Let $f : D \to D$ be continuous, and let $d \in D$ satisfy $f\,d \sqsubseteq d$. Then 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: *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

$$\mathcal{S}_{ds}[\![S]\!] \sqsubseteq \mathcal{S}_{sos}[\![S]\!]$$

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.

Aborting

We start by adding the new statement **`abort`**.

Approach 1 - Consider $\langle \texttt{abort}, s \rangle$ to be a *stuck configuration*.

No extra rule to either the Natural or the Structural Operational Semantics added.

Note that

$$\langle \texttt{while true do skip}, s_1 \rangle \to s_2 \text{ implies } \langle \texttt{abort}, s_1 \rangle \to s_2$$

$$\langle \texttt{abort}, s_1 \rangle \to s_2 \text{ implies } \langle \texttt{while true do skip}, s_1 \rangle \to s_2$$

so equivalent in the Natural Semantics. In Structural Operational Semantics '**`while true do skip`**' generates infinite number of steps, and '**`abort`**' none

Approach 2 - Add a new terminal configuration to the system, **`error`**, and add

$$\langle \texttt{abort}, s_1 \rangle \to \texttt{error}$$

$$\langle \texttt{abort}, s_1 \rangle \Rightarrow^* \texttt{error}$$

We can now distinguish between the statements in both semantics.

Non-determinism

We add 'S_1 **or** S_2'. For example:

$$(x \texttt{ := } 1) \texttt{ or } (x \texttt{ := } 2 \texttt{;} x \texttt{ := } x + 2)$$

result state: x has value 1, or 4. Natural Semantics:

$$(\mathsf{OR}^{\mathsf{L}}_{ns}) \quad \frac{\langle S_1, s_1\rangle \to s_2}{\langle S_1 \texttt{ or } S_2, s_1\rangle \to s_2}$$

$$(\mathsf{OR}^{\mathsf{R}}_{ns}) \quad \frac{\langle S_2, s_1\rangle \to s_2}{\langle S_1 \texttt{ or } S_2, s_1\rangle \to s_2}$$

Then we have the following derivations:

$$\frac{\dfrac{}{\langle x \texttt{ := } 1, s\rangle \to s[x \mapsto 1]}}{\langle x \texttt{ := } 1 \texttt{ or } (x \texttt{ := } 2 \texttt{;} x \texttt{ := } x + 2), s\rangle \to s[x \mapsto 1]}$$

and

$$\frac{\dfrac{\dfrac{}{\langle x \texttt{ := } 2, s\rangle \to s[x \mapsto 2]} \qquad \dfrac{}{\langle x \texttt{ := } x + 2, s[x \mapsto 2]\rangle \to s[x \mapsto 4]}}{\langle x \texttt{ := } 2 \texttt{;} x \texttt{ := } x + 2, s\rangle \to s[x \mapsto 4]}}{\langle x \texttt{ := } 1 \texttt{ or } (x \texttt{ := } 2 \texttt{;} x \texttt{ := } x + 2), s\rangle \to s[x \mapsto 4]}$$

Non-determinism suppresses looping

$$\langle (\texttt{while true do skip}) \texttt{ or } (x \texttt{ := } 2 \texttt{;} x \texttt{ := } x + 2), s\rangle \to s[x \mapsto 4]$$

Likewise, :

$$\langle S_1 \texttt{ or } S_2, s\rangle \Rightarrow \langle S_1, s\rangle \text{ and } \langle S_1 \texttt{ or } S_2, s\rangle \Rightarrow \langle S_2, s\rangle$$

Then we have:

$$\begin{aligned}\langle x \texttt{ := } 1 \texttt{ or } (x \texttt{ := } 2 \texttt{; } x \texttt{ := } x+2), s\rangle &\Rightarrow \langle x \texttt{ := } 1, s\rangle \\ &\Rightarrow s[x \mapsto 1]\end{aligned}$$

and

$$\begin{aligned}&\langle x \texttt{ := } 1 \texttt{ or } (x \texttt{ := } 2 \texttt{; } x \texttt{ := } x+2), s\rangle \\ &\quad\Rightarrow \langle x \texttt{ := } 2 \texttt{; } x \texttt{ := } x+2, s\rangle \\ &\quad\Rightarrow \langle x \texttt{ := } x+2, s[x \mapsto 2]\rangle \\ &\quad\Rightarrow s[x \mapsto 4]\end{aligned}$$

But replacing 'x `:=` 1' by '`while true do skip`' will still give two derivation sequences; one will be infinite:

$$\begin{aligned}&\langle(\texttt{while true do skip}) \texttt{ or } (x \texttt{ := } 2 \texttt{; } x \texttt{ := } x+2), s\rangle \\ &\quad\Rightarrow \langle \texttt{while true do skip}, s\rangle \\ &\quad\vdots\end{aligned}$$

and the other is finite:

$$\begin{aligned}&\langle(\texttt{while true do skip}) \texttt{ or } (x \texttt{ := } 2 \texttt{; } x \texttt{ := } x+2), s\rangle \\ &\quad\Rightarrow \langle x \texttt{ := } 2 \texttt{; } x \texttt{ := } x+2, s\rangle \\ &\quad\Rightarrow \langle x \texttt{ := } x+2, s[x \mapsto 2]\rangle \\ &\quad\Rightarrow s[x \mapsto 4]\end{aligned}$$

Parallelism

Add the construct '**par**', and expect the execution of S_1 and S_2 to be 'interleaved'. For example, the program

$$(x := 1) \textbf{ par } (x := 2 ; x := x + 2)$$

has the following different ways to execute:

$x := 1$	$x := 2$	$x := x + 2$	$s\, x = 4$
$x := 2$	$x := 1$	$x := x + 2$	$s\, x = 3$
$x := 2$	$x := x + 2$	$x := 1$	$s\, x = 1$

Rules for Structural Operational Semantics

$$(\text{PAR}^{\text{LT}}_{sos}) \quad \frac{\langle S_1, s_1\rangle \Rightarrow s_2}{\langle S_1 \textbf{ par } S_2, s_1\rangle \Rightarrow \langle S_2, s_2\rangle}$$

$$(\text{PAR}^{\text{LI}}_{sos}) \quad \frac{\langle S_1, s_1\rangle \Rightarrow \langle S_1', s_2\rangle}{\langle S_1 \textbf{ par } S_2, s_1\rangle \Rightarrow \langle S_1' \textbf{ par } S_2, s_2\rangle}$$

$$(\text{PAR}^{\text{RT}}_{sos}) \quad \frac{\langle S_2, s_1\rangle \Rightarrow s_2}{\langle S_1 \textbf{ par } S_2, s_1\rangle \Rightarrow \langle S_1, s_2\rangle}$$

$$(\text{PAR}^{\text{RI}}_{sos}) \quad \frac{\langle S_2, s_1\rangle \Rightarrow \langle S_1', s_2\rangle}{\langle S_1 \textbf{ par } S_2, s_1\rangle \Rightarrow \langle S_1 \textbf{ par } S_1', s_2\rangle}$$

Exercise: Verify the results above.

To accomplish the same expressive power in Natural Semantics, we run into problems. Assume the rules that need to be added are:

$$\frac{\langle S_1, s_1\rangle \rightarrow s_2 \quad \langle S_2, s_2\rangle \rightarrow s_3}{\langle S_1 \ \textbf{par} \ S_2, s_1\rangle \rightarrow s_3}$$

$$\frac{\langle S_2, s_1\rangle \rightarrow s_2 \quad \langle S_1, s_2\rangle \rightarrow s_3}{\langle S_1 \ \textbf{par} \ S_2, s_1\rangle \rightarrow s_3}$$

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.

Blocks

New programming language **`Block`**

Abstract syntax:

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

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

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:

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

For the Natural Semantics, take

$$s_1[V \mapsto s_2]\, x = \begin{cases} s_2\, x & \text{if } x \in V \\ s_1\, x & \text{if } x \notin V \end{cases}$$

$$DV(\textbf{var}\ x\ \textbf{:=}\ a\ \textbf{;}\ D_v) = \{x\} \cup DV(D_v)$$

$$DV(\epsilon) = \emptyset$$

We add the following rules:

$$(\text{DECL}_{ns}) \quad \frac{\langle D_v, s_1[x \mapsto \mathcal{A}[\![a]\!]\, s_1]\rangle \to s_2}{\langle \textbf{var}\, x\ \textbf{:=}\ a, s_1\rangle \to s_2}$$

$$(\text{NO-DECL}_{ns}) \quad \frac{}{\langle \epsilon, s\rangle \to s}$$

$$(\text{BLOCK}_{ns}) \quad \frac{\langle D_v, s_1\rangle \to s_2 \qquad \langle S, s_2\rangle \to s_3}{\langle \textbf{begin}\ D_v\ S\ \textbf{end}, s_1\rangle \to s_3[DV(D_v) \mapsto s_2]}$$

Procedures

Proc, a language that allows for the definition of (parameterless) *procedure declarations*

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

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

Three different semantics

- *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.

Example:

```
begin var x := 0;
      proc p is x := x × 2;
      proc q is call p;
      begin var x := 5;
            proc p is x := x + 1;
            call q;
            y := x
      end
end
```

- $y = 6$
- $y = 10$
- $y = 5$

Dynamic / Dynamic

Procedure environment

$$env_p \in Env_p = \textit{Procedure Names} \hookrightarrow \textit{Statements}$$

The transition rules will now be of the form:

$$env_p \vdash \langle S, s_1 \rangle \to s_2$$

We need to be able to update the environment.

$$Update_p(\texttt{proc}\ p\ \texttt{is}\ S\ \texttt{;}\ D_p, env_p) = Update_p(D_p, env_p[p \mapsto S])$$

$$Update_p(\epsilon, env_p) = env_p$$

We add two rules

$$(\text{CALL}^{\text{rec}}_{ns}) \quad \frac{env_p \vdash \langle S, s_1 \rangle \to s_2}{env_p \vdash \langle \texttt{call}\ p, s_1 \rangle \to s_2} \quad (env_p\, p = S)$$

$$(\text{BLOCK}_{ns}) \quad \frac{\langle D_v, s_1 \rangle \to s_2 \quad Update_p\,(D_p, env_p) \vdash \langle S, s_2 \rangle \to s_3}{env_p \vdash \langle \texttt{begin}\ D_v\ D_p\ S\ \texttt{end}, s_1 \rangle \to s_3[DV\,(D_v) \mapsto s_1]}$$

Procedures can always be recursive.

Dynamic / Static

Procedures will use those declarations that are defined at the moment the procedure itself was declared. We also need to state the environment in which procedures are defined:

$$\mathit{Env}_p = \mathit{Procedure\ Names} \hookrightarrow \mathit{Statements} \times \mathit{Env}_p$$

$$\mathit{Update}_p(\texttt{proc}\ p\ \texttt{is}\ S\ \texttt{;}\ D_p, \mathit{env}_p) = \mathit{Update}_p(D_p, \mathit{env}_p[p \mapsto (S, \mathit{env}_p)])$$

$$\mathit{Update}_p(\epsilon, \mathit{env}_p) = \mathit{env}_p$$

We then just need to update the rule (CALL_{ns}). If procedures in **Proc** are non-recursive, we use:

$$(\text{CALL}_{ns})\ \frac{\mathit{env}'_p \vdash \langle S, s_1\rangle \to s_2}{\mathit{env}_p \vdash \langle \texttt{call}\ p, s_1\rangle \to s_2}\ (\mathit{env}_p\, p = (S, \mathit{env}'_p))$$

But if procedures are recursive, we need:

$$(\text{CALL}^{\text{rec}}_{ns})\ \frac{\mathit{env}'_p[p \mapsto (S, \mathit{env}'_p)] \vdash \langle S, s_1\rangle \to s_2}{\mathit{env}_p \vdash \langle \texttt{call}\ p, s_1\rangle \to s_2}\ (\mathit{env}_p\, p = (S, \mathit{env}'_p))$$

Exercise: Try to construct a statement which illustrates the difference between these two rules.

Static / Static

Approach defined above, and variables can appear in more than one declaration. We need to replace the state with two mappings:

$$\mathit{env}_v \in \mathit{Env}_v = \mathit{Variables} \to \mathit{Locations}$$

$$\mathit{store} \in \mathit{Store} = \mathit{Locations} \cup \{\mathit{next}\} \to \mathbb{Z}$$

$\mathit{Locations} = \mathbb{Z}$, and *next* is a special token which holds the next free location. A machine is now represented 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 $\mathit{env}_v\, x$ 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, making x point to a new location. We also use a function '*new*' that produces the number of the next free location.

$$\mathit{new} : \mathit{Locations} \to \mathit{Locations}$$

$$\mathit{new}\, l = l + 1$$

(Note that $s = \mathit{store} \circ \mathit{env}_v$.)

We need to consider transitions of the form

$$\langle D_v, \mathit{env}_v, \mathit{store}_1 \rangle \to (\mathit{env}'_v, \mathit{store}_2)$$

For evaluating the variable declarations, we have rules

$$\frac{\langle D_v, \mathit{env}_v[x \mapsto l], \mathit{store}[l \mapsto v][\mathit{next} \mapsto \mathit{new}\, l] \rangle \to (\mathit{env}'_v, \mathit{store}_2)}{\langle \texttt{var}\ x\ \texttt{:=}\ a\ \texttt{;}\ D_v, \mathit{env}_v, \mathit{store} \rangle \to (\mathit{env}'_v, \mathit{store}_2)}$$

$$\frac{}{\langle \epsilon, \mathit{env}_v, \mathit{store} \rangle \to (\mathit{env}_v, \mathit{store})}$$

where

$$\begin{aligned} v &= \mathcal{A}[\![a]\!]\,(\mathit{store} \circ \mathit{env}_v) \\ l &= \mathit{store}\,\mathit{next} \end{aligned}$$

We must further update procedure environments:

$$\mathit{Env}_p = \mathit{Procedure\ Names} \hookrightarrow \mathit{Statements} \times \mathit{Env}_v \times \mathit{Env}_p$$

$$\begin{aligned} &\mathit{Update}_p\,(\texttt{proc}\ p\ \texttt{is}\ S\ \texttt{;}\ D_p, \mathit{env}_v, \mathit{env}_p) = \\ &\quad \mathit{Update}_p\,(D_p, \mathit{env}_v, \mathit{env}_p[p \mapsto (S, \mathit{env}_v, \mathit{env}_p)]) \\ &\qquad \mathit{Update}_p\,(\epsilon, \mathit{env}_v, \mathit{env}_p) = \mathit{env}_p \end{aligned}$$

The transition system for statements now has rules of the form

$$env_v, env_p \ \vdash \ \langle S, store_1 \rangle \to store_2$$

Most rules are similar to their original, but the rule for blocks is modified:

$$\frac{\langle D_v, env_v, store_1 \rangle \to (env'_v, store_2) \qquad env'_v, env'_p \vdash \langle S, store_2 \rangle \to store_3}{env_v, env_p \vdash \langle \textbf{begin}\ D_v\ D_p\ S\ \textbf{end}, store_1 \rangle \to store_3}$$

where

$$env'_p \ = \ Update_p\,(D_p, env'_v, env_p)$$

And the new rules for **`call`** are:

$$(\text{CALL}_{ns}) \ \frac{env'_v, env'_p \vdash \langle S, store_1 \rangle \to store_2}{env'_v, env_p \vdash \langle \textbf{call}\ p, store_1 \rangle \to store_2}$$

$$(\text{CALL}_{ns}^{\text{rec}}) \ \frac{env'_v, env'_p[p \mapsto (S, env'_v, env'_p)] \vdash \langle S, store_1 \rangle \to store_2}{env'_v, env_p \vdash \langle \textbf{call}\ p, store_1 \rangle \to store_2}$$

where

$$env_p \ = \ (S, env'_v, env'_p).$$