Program Analysis/ Mooly Sagiv
Lecture 1, 31/10/2012 Operational Semantics
Notes by: Kalev Alpernas

As background to the subject of Program Analysis, we will first discuss the subject of Operational Semantics.

Syntax vs. Semantics

Before we get into the topic of Operational Semantics, let's discuss the difference between Syntax and Semantics.

The syntax is the pattern of formation of sentences or phrases in a language. It defines which strings of characters constitute a legal program. We have many tools at our disposal to check the syntactic validity of programs, e.g. Regular Expressions, Context Free Grammars, etc.

Semantics, on the other hand, is the study or science of meaning in language. Throughout this entire course we try to assign meaning to different programs. In a sense, we try to answer the abstract question of 'what does the program do?'

There are many ways to define the meaning of a program. A very natural way to understand the meaning of a program is to write an interpreter for the program, which given an input, runs the program line by line and returns the program output.

This definition is only marginally useful, as its level of granularity allows us, at best, to only discuss properties of the program as a whole. Furthermore, the reliance on an input for the definition, prevents us from proving anything more abstract.

Another way to define the meaning of a program is via the compiler – the result of the compilation is the meaning of the program. This is not, however, a very good definition of meaning, as it does not allow us to prove anything of substance regarding the meaning of the program, and is very dependent on the specific compiler.

In today's lesson we will try to define a better notion of meaning of a program; in particular one that allows us to prove certain properties of programs, e.g. that two programs are equivalent.

The Benefits of Semantics

Why do we even need semantics?

Semantics allows us to describe what a program does. This is useful in a variety of cases:

- In programming language design – Dana Scott defined a relation, which holds in most cases, between ease of definition of a programming language, ease of implementation of the programming language and ease of use of a programming language. Semantics gives us an exact definition for the meaning of programs and language constructs.

- When implementing a programming language it is very useful to have formal; exact definitions.
- When writing programs we would like to prove correctness for programs, i.e. we would like to say that a certain program computes something. Semantics gives us a way to describe what a program does, and allows us to prove certain properties of a program.
- Another useful usage of Semantics is when discussing equivalence in programs – this is especially useful for Compilers, who translate parts of a program to something different, for example more efficient, but should keep the meaning of the code – we use Semantics to define program meaning, and compare the meaning of different programs.
- Automatic generation of an interpreter – Semantic definition of a programming language allows us to automatically generate an interpreter for the language. For example the Prolog code accompanying this lecture is an interpreter for the language defined later in this lesson. The entire Prolog code consists of the Semantic definition of the programming language, and acts as an interpreter for constructs of this language.
- Compilers however are much harder to generate from Semantics, even though there is research in the field, as building a compiler requires a lot of engineering work to produce an efficient result.

Alternative Formal Semantics

There are three families of Formal Semantics:

Operational Semantics:

We describe the meaning of the program 'operationally', in a manner similar to how an interpreter would define the meaning of a program, but on an abstract level, unlike an interpreter, forgoing many of the details that an interpreter would retain, and focusing on those that ascribe meaning to the program.

This enables to prove properties of programs and programming languages in a mathematical way.

Within this family we have two classes of Semantics – Natural Operational Semantics and Structural Operational Semantics, where the NS is slightly more abstract, and the SOS is slightly more detailed. NS is an abstraction of SOS.

In **figure 1** we see an example of the operational meaning for every step of the run of a program, with a given input (x=3).

Denotational Semantics:

Denotational Semantics defines the meaning of the program as an input/output relation. It is mathematically challenging, but complicated.

In **figure 2**, we see the meaning of the same program as a lambda expression which is function that describes the operation of the program on a given input.

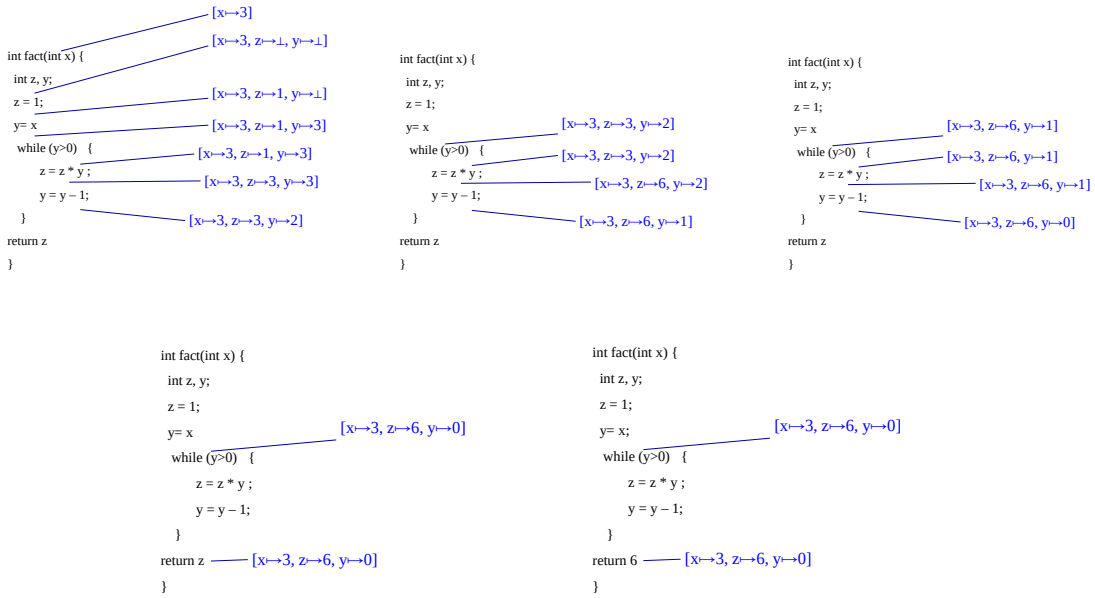

Figure 1 *– The operational meaning of a run of the factorial program on the value x=3.*

Axiomatic Semantics:

The meaning of the program is defined by a series of axioms, or assertions, throughout the program.

In **figure 3** we see the set of assertions at every point in the program, that describe, at each point, the set of state the program can be in at that point. Of special interest here is the loop invariant for the while loop, which we can see holds at the start and end of the while loop.

```
int fact(int x) {
 int z, y;
 z = 1;
 y= x ;
  while (y>0)   {
       z = z * y ;
       y = y – 1;
   }
return z;
}
```

f=λx. if x = 0 then 1 else x * f(x -1)

Figure 2*-Detonational Semantics*

```
{x=n}
 int fact(int x) {  int z, y;
 z = 1;
{x=n ∧ z=1}
 y= x
{x=n ∧ z=1 ∧ y=n}
  while
   {x=n  ∧ y ≥0 ∧ z=n! / y!}
(y>0)  {
     {x=n ∧ y >0 ∧  z=n! / y!}
      z = z * y ;
    {x=n ∧ y>0 ∧  z=n!/(y-1)!}
      y = y – 1;
    {x=n ∧ y ≥0 ∧  z=n!/y!}
  } return z} {x=n ∧ z=n!}
```

Figure 3*-Axiomatic Semantics*

Throughout this lesson we will focus on Operational Semantics.

Note:

We should note that while both Denotational and Axiomatic Semantics assign a general notion of meaning to the program, in Operational Semantics the meaning is defined for a single input for the program. However, we will deal with a general case of a single input, e.g. $x > 0$, and will deal with this general case as a set of singletons.

Operational Semantics in Static Analysis

As we've seen in last week's lesson, the process of abstract interpretation involves the application of Abstract Semantics on an Abstract Representation (**figure 4**.) This is done by applying Operational Semantics on a concretized representation, the result of which is then abstracted to get the resulting abstract representation.

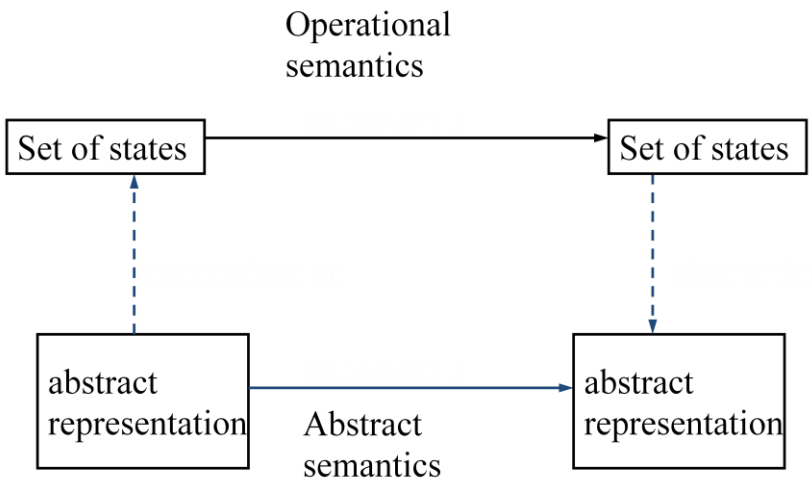

***Figure 4**-The process of Abstract (Conservative) Interpretation*

Let's look at the example of the rule of signs that we've seen in last week's lesson. **Table 1** describes the Abstract Semantic of the operator $*$. The reason that this table is sound is that it equates with the Operational Semantic of the $*$ operator – every positive number we multiply by a positive number, will result in a positive number, and so forth. **Figure 5** describes this process of concretization, application of Operational Semantics and abstraction.

$*$ #	P	N	?
P	P	N	?
N	N	P	?
?	?	?	?

***Table 1** - The Abstract Semantics of the operator $*$.*

The While Programming Language

We will illustrate Operation Semantics by using an example – a simple programming language called while.

We define the abstract syntax of the while language as:

$$S ::= x := a \mid skip \mid S_1; S_2 \mid if\ b\ then\ S_1 else\ S_2 \mid while\ b\ do\ S$$

Where a is an arithmetic expression, b is a boolean expression and $skip$ is an expression which skips the line (does nothing).

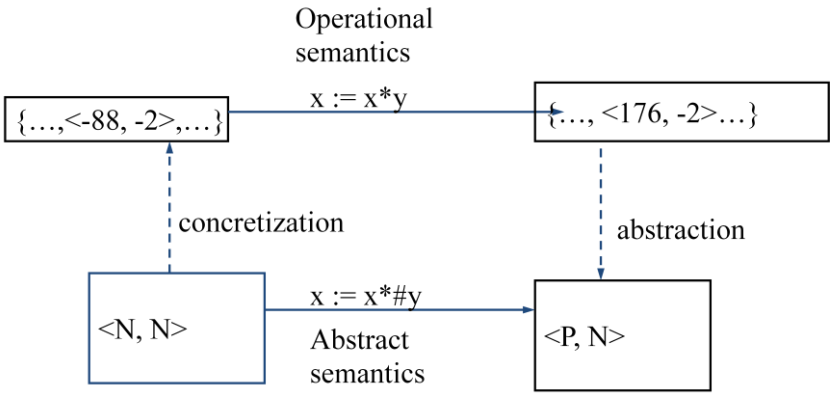

Figure 5

Note

Note that we are not interested in the syntax of the program and assume that the input is an abstract syntax tree. Thus, precedences and parentheses are not needed. Parsing is done on the 'user code' using the concrete syntax of the programming language.

Figure 6 illustrates an example program in the while language that computes the factorial of `x`.

```
y := 1;
while ¬(x=1) do (
   y := y * x;
   x := x - 1
)
```

Figure 6

But First, a Few Definitions

Syntactic Categories:

- `Var` the set of program variables.
- `Aexp` the set of arithmetic expressions.
- `Bexp` the set of boolean expressions.
- `Stm` the set of program statements.

Semantic Categories:

- Natural values
  $N = \{0,1,2, \dots\}$
- Truth values
  $T = \{ff, tt\}$
- States
  $State = Var \to N$
  where a state is the mapping of variables to Natural value.
  Sometimes State is a partial function if variables are not initialized
- Lookup the value of the variable x in a state s: $s\ x$
- Update the value of x in the state s: $s\ [x \mapsto 5]$

Example

[x↦1, y↦7, z↦16] y = 7
[x↦1, y↦7, z↦16] t = undefined
[x↦1, y↦7, z↦16][x↦5] = [x↦5, y↦7, z↦16]
[x↦1, y↦7, z↦16][x↦5] x = 5
[x↦1, y↦7, z↦16][x↦5] y = 7

Semantics of Arithmetic Expressions

Now we look at the semantics of arithmetic expressions.

We assume that in the while language arithmetic expressions are side effect free.

We define the operator $A[\![A_{exp}]\!]: State \to N$ that gets a syntactic arithmetic expression, and return semantic object. The semantic of Expressions is a total function which cannot mutate the input state.

The operator is defined by induction on the syntax tree:

- $A[\![n]\!]s = n$ – the value of a number in a state does not depend on the state.
- $A[\![x]\!]s = s\ x$
- $A[\![e_1 + e_2]\!]s = A[\![e_1]\!]s + A[\![e_2]\!]s$ – addition is compositional in our language.
- $A[\![e_1 * e_2]\!]s = A[\![e_1]\!]s * A[\![e_2]\!]s$ – multiplication, like addition, is compositional in our language.

- $A[\![(e_1)]\!]s = A[\![e_1]\!]s$ – not needed since we are working with an abstract syntax, and precedence does not interest us.
- $A[\![-e_1]\!]s = -A[\![e_1]\!]s$

We can now prove properties by structural induction on the syntax tree. For example if two states agree on all variables and values, then we can conclude equivalence.

Example

If two states s_1 and s_2 agree on the values of all variables of an arithmetic expression A_{exp}, then the states agree on the evaluation of the expression as well.

Or more formally:

$$if\ \forall x \in A_{exp}\ s_1\ x = s_2\ x\ then\ A[\![A_{exp}]\!]s_1 = A[\![A_{exp}]\!]s_2$$

Proof

The proof is by induction on the structure of A_{exp}.

Case 1: $A_{exp} = n$

$$A[\![n]\!]s_1 = n, A[\![n]\!]s_2 = n \Rightarrow A[\![n]\!]s_1 = A[\![n]\!]s_2$$

Case 2: $A_{exp} = x$

Similar to case 1.

Case 3: $A_{exp} = e_1 + e_2$

$$A[\![e_1 + e_2]\!]s_1 = A[\![e_1]\!]s_1 + A[\![e_2]\!]s_1 - follows\ from\ definition$$
$$A[\![e_1 + e_2]\!]s_2 = A[\![e_1]\!]s_2 + A[\![e_2]\!]s_2 - follows\ from\ definition$$
$$A[\![e_1]\!]s_1 = A[\![e_1]\!]s_2, A[\![e_2]\!]s_1 = A[\![e_2]\!]s_2 - Inductioin\ assumption$$
$$\Rightarrow A[\![e_1 + e_2]\!]s_1 = A[\![e_1 + e_2]\!]s_2$$

All other cases follow similarly.

■

Semantics of Boolean Expression

Similarly to the arithmetic expressions, we define an operator for boolean expressions:

$$B[\![B_{exp}]\!]: State \to T$$

Here as well, we assume that the boolean expressions are side effect free.

The operator is defined by induction on the syntax tree:

- $B[\![true]\!]s = tt$
- $B[\![false]\!]s = ff$
- $B[\![e_1 = e_2]\!]s = \begin{cases} tt \ if \ A[\![e_1]\!]s = A[\![e_2]\!]s \\ ff \ if \ A[\![e_1]\!]s \neq A[\![e_2]\!]s \end{cases}$
- $B[\![e_1 \wedge e_2]\!]s = \begin{cases} tt \ if \ B[\![e_1]\!]s = tt \ and \ B[\![e_2]\!]s = tt \\ ff \ if \ b[\![e_1]\!]s = ff \ or \ B[\![e_2]\!]s = ff \end{cases}$

Similarly we can define a notion of $B[\![e_1 \geq e_2]\!]s$ and so on…

Also, as we've seen in Arithmetic Expressions, the Boolean Expressions are also compositional.

Note

Note that we use the semantics of Arithmetic Expressions to define the semantics of Boolean Expressions.

Natural Operational Semantics

Introduced by Gilles Kahn (Natural Semantics 1987), it describes the 'overall' effect of program constructs, by the relation between the input and output of these constructs.

It ignores non terminating computations.

We denote the following:

- $< S, s >$ – The program statement S is executed on the input state s.
- s represents a terminal (final) state.

For every statement S we define meaning rules of the form: $< S, i > \to o$, which means 'if the statement S is executed on an input state i, it terminates and yields an output state o'.

Note that this Semantics can be non-deterministic, and so the meaning of a statement program S on an input state s is the set of output states o such that $< S, s > \to o$.

The meaning of compound statements is defined using the meaning immediate constituent statements. The meaning of the whole program P is just special case since P is just a statement.

Natural Semantics for While

We separate the Natural Semantics for while to two parts:

Axioms:

- $[ass_{ns}] < x := a, s >\rightarrow s[x \mapsto A[\![a]\!]s]$ – the value of a is evaluated in the state s, and is assigned to x in the resulting state.
- $[skip_{ns}] < skip, s >\rightarrow s$ – simply results in the starting state s.

Rules:

- $[comp_{ns}] \frac{< S_1, s >\rightarrow s', < S_2, s' >\rightarrow s''}{< S_1; S_2, s >\rightarrow s''}$

  If the result of $< S_1, s >$ is s' and the result of $< S_2, s' >$ is s'', then the result of $< S_1; S_2, s >$ is s''.

- $[if_{ns}^{tt}] \frac{< S_1, s >\rightarrow s'}{< if\ b\ then\ S_1\ else\ S_2, s >\rightarrow s'}\ if\ B[\![b]\!]s = tt$

  $[if_{ns}^{ff}] \frac{< S_2, s >\rightarrow s'}{< if\ b\ then\ S_1\ else\ S_2, s >\rightarrow s'}\ if\ B[\![b]\!]s = ff$

  Here we differentiate between the case where b is true, and the case where b is false. True – the result is the same as the result of the 'then' clause. False – the result of the 'else' clause.

So far, the rules have been compositional on the sub expressions. The next rules for while are different in that they require the definition while for the rule, and so are not compositional.

- $[while_{ns}^{ff}] \frac{}{< while\ b\ do\ S, s >\rightarrow s}\ if\ B[\![b]\!]s = ff$

  $$[while_{ns}^{tt}] \frac{< S, s >\rightarrow s', < while\ b\ do\ S, s' >\rightarrow s''}{< while\ b\ do\ S, s >\rightarrow s''}\ if\ B[\![b]\!]s = tt$$

  As in the case of if we differentiate between two cases depending on the value of b. false – the state stays the same (similar to skip). True – if the result of <S,s> is s', and the result of while in the s' is s", the result of while on s is s".

A Derivation Tree

A derivation tree is the proof that $< S, i >\rightarrow o$ is true. The root of the tree is $< S, i >\rightarrow o$ the leaves are axioms, and all internal nodes are the rules.

Example

Figure 7 illustrates an example derivation tree.

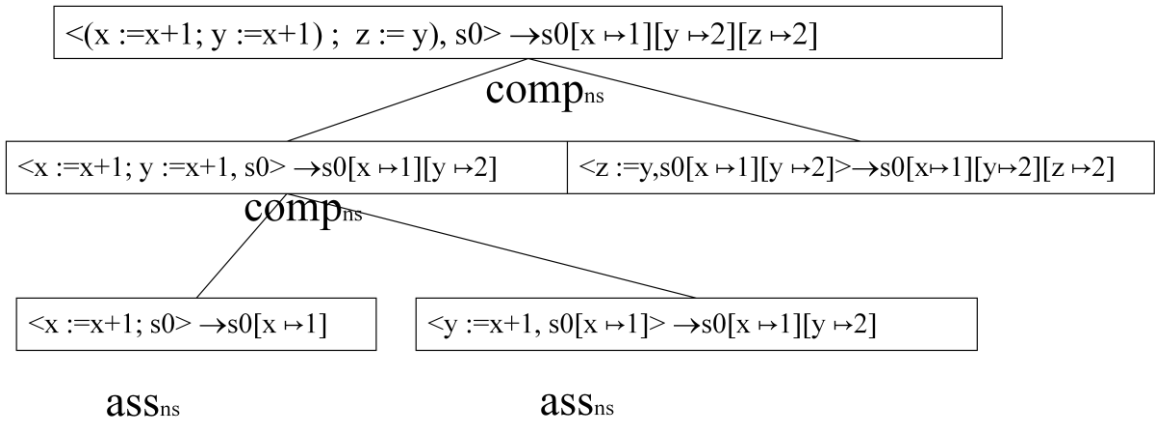

Figure 7

Also, the same derivation tree using the Logic notation:

$$\frac{\dfrac{< x := x + 1, s_0 >\rightarrow s_1 \qquad < y := x + 1, s_0[x \mapsto 1] >\rightarrow s_2}{< x := x + 1; y := x + 1, s_0 >\rightarrow s_2} \qquad < z := y, s_2 >\rightarrow s_3}{< (x := x + 1; y := x + 1); z := y, s_0 >\rightarrow s_3}$$

Where:

$s_1 = s_0[x \mapsto 1]$

$s_2 = s_0[x \mapsto 1][y \mapsto 2]$

$s_3 = s_0[x \mapsto 1][y \mapsto 2][z \mapsto 2]$

Semantic Equivalence

We can use this derivation method to prove certain properties of programs. For example, semantic equivalence.

The following excerpt from the book *Semantics with Applications* by H.Nielsen and F.Nielsen, proves the following equivalence:

> The statement *while b do S* is equivalent to the statement
> *if b then (S; while b do S) else skip*

Proof: The proof is in two stages. We shall first prove that if

$$\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \rightarrow s'' \qquad (*)$$

then

$$\langle \texttt{if}\ b\ \texttt{then}\ (S; \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else skip}, s\rangle \rightarrow s'' \qquad (**)$$

Thus, if the execution of the loop terminates then so does its one-level unfolding. Later we shall show that if the unfolded loop terminates then so will the loop itself; the conjunction of these results then prove the lemma.

Because (*) holds we know that we have a derivation tree T for it. It can have one of two forms depending on whether it has been constructed using the rule $[\text{while}^{\text{tt}}_{\text{ns}}]$ or the axiom $[\text{while}^{\text{ff}}_{\text{ns}}]$. In the first case the derivation tree T has the form:

$$\frac{T_1 \qquad T_2}{\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \rightarrow s''}$$

where T_1 is a derivation tree with root $\langle S, s\rangle \rightarrow s'$ and T_2 is a derivation tree with root $\langle \texttt{while}\ b\ \texttt{do}\ S, s'\rangle \rightarrow s''$. Furthermore, $\mathcal{B}[\![b]\!]s = \mathbf{tt}$. Using the derivation trees T_1 and T_2 as the premises for the rules $[\text{comp}_{\text{ns}}]$ we can construct the derivation tree:

$$\frac{T_1 \qquad T_2}{\langle S; \texttt{while}\ b\ \texttt{do}\ S, s\rangle \rightarrow s''}$$

Using that $\mathcal{B}[\![b]\!]s = \mathbf{tt}$ we can use the rule $[\text{if}^{\text{tt}}_{\text{ns}}]$ to construct the derivation tree

$$\frac{T_1 \qquad\qquad T_2}{\langle S;\ \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s''}$$

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

thereby showing that (**) holds.

Alternatively, the derivation tree T is an instance of $[\text{while}^{\text{ff}}_{\text{ns}}]$. Then $\mathcal{B}[\![b]\!]s = \mathbf{ff}$ and we must have that $s''{=}s$. So T simply is

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

Using the axiom $[\text{skip}_{\text{ns}}]$ we get a derivation tree

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

and we can now apply the rule $[\text{if}^{\text{ff}}_{\text{ns}}]$ to construct a derivation tree for (**):

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

This completes the first part of the proof.

For the second stage of the proof we assume that (**) holds and shall prove that (*) holds. So we have a derivation tree T for (**) and must construct one for (*). Only two rules could give rise to the derivation tree T for (**), namely $[\text{if}^{\text{tt}}_{\text{ns}}]$ or $[\text{if}^{\text{ff}}_{\text{ns}}]$. In the first case, $\mathcal{B}[\![b]\!]s = \mathbf{tt}$ and we have a derivation tree T_1 with root

$$\langle S;\ \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \rightarrow s''$$

The statement has the general form $S_1;\ S_2$ and the only rule that could give this is $[\text{comp}_{\text{ns}}]$. Therefore there are derivation trees T_2 and T_3 for

$$\langle S,\ s\rangle \rightarrow s', \text{ and}$$

$$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s'\rangle \rightarrow s''$$

for some state s'. It is now straightforward to use the rule $[\text{while}^{\text{tt}}_{\text{ns}}]$ to combine T_2 and T_3 to a derivation tree for (*).

In the second case, $\mathcal{B}[\![b]\!]s = \mathbf{ff}$ and T is constructed using the rule $[\text{if}^{\text{ff}}_{\text{ns}}]$. This means that we have a derivation tree for

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

and according to axiom $[\text{skip}_{\text{ns}}]$ it must be the case that $s{=}s''$. But then we can use the axiom $[\text{while}^{\text{ff}}_{\text{ns}}]$ to construct a derivation tree for (*). This completes the proof. □

Deterministic Semantics of While

The following excerpt from the book shows that the Semantics for while are deterministic, or more formally:

$$if < S, s > \rightarrow s_1 \ and < S, s > \rightarrow s_2 \ then \ s_1 = s_2$$

Proof: We assume that $\langle S,\ s\rangle \rightarrow s'$ and shall prove that

$$\text{if } \langle S,\ s\rangle \rightarrow s'' \text{ then } s' = s''.$$

We shall proceed by induction on the shape of the derivation tree for $\langle S,\ s\rangle \rightarrow s'$.

The case $[\text{ass}_{\text{ns}}]$: Then S is $x{:=}a$ and s' is $s[x \mapsto \mathcal{A}[\![a]\!]s]$. The only axiom or rule that could be used to give $\langle x{:=}a,\ s\rangle \rightarrow s''$ is $[\text{ass}_{\text{ns}}]$ so it follows that s'' must be $s[x \mapsto \mathcal{A}[\![a]\!]s]$ and thereby $s' = s''$.

The case $[\text{skip}_{\text{ns}}]$: Analogous.

The case $[\text{comp}_{\text{ns}}]$: Assume that

$$\langle S_1;S_2,\ s\rangle \rightarrow s'$$

holds because

$$\langle S_1,\ s\rangle \rightarrow s_0 \text{ and } \langle S_2,\ s_0\rangle \rightarrow s'$$

for some s_0. The only rule that could be applied to give $\langle S_1;S_2,\ s\rangle \to s''$ is $[\text{comp}_{\text{ns}}]$ so there is a state s_1 such that

$$\langle S_1,\ s\rangle \to s_1 \text{ and } \langle S_2,\ s_1\rangle \to s''$$

The induction hypothesis can be applied to the premise $\langle S_1,\ s\rangle \to s_0$ and from $\langle S_1,\ s\rangle \to s_1$ we get $s_0 = s_1$. Similarly, the induction hypothesis can be applied to the premise $\langle S_2,\ s_0\rangle \to s'$ and from $\langle S_2,\ s_0\rangle \to s''$ we get $s' = s''$ as required.

The case $[\text{if}^{\text{tt}}_{\text{ns}}]$: Assume that

$$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \to s'$$

holds because

$$\mathcal{B}[\![b]\!]s = \mathbf{tt} \text{ and } \langle S_1,\ s\rangle \to s'$$

From $\mathcal{B}[\![b]\!]s = \mathbf{tt}$ we get that the only rule that could be applied to give the alternative $\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \to s''$ is $[\text{if}^{\text{tt}}_{\text{ns}}]$. So it must be the case that

$$\langle S_1,\ s\rangle \to s''$$

But then the induction hypothesis can be applied to the premise $\langle S_1,\ s\rangle \to s'$ and from $\langle S_1,\ s\rangle \to s''$ we get $s' = s''$.

The case $[\text{if}^{\text{ff}}_{\text{ns}}]$: Analogous.

The case $[\text{while}^{\text{tt}}_{\text{ns}}]$: Assume that

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

because

$$\mathcal{B}[\![b]\!]s = \mathbf{tt},\ \langle S,\ s\rangle \to s_0 \text{ and } \langle \texttt{while}\ b\ \texttt{do}\ S,\ s_0\rangle \to s'$$

The only rule that could be applied to give $\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s''$ is $[\text{while}^{\text{tt}}_{\text{ns}}]$ because $\mathcal{B}[\![b]\!]s = \mathbf{tt}$ and this means that

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

must hold for some s_1. Again the induction hypothesis can be applied to the premise $\langle S,\ s\rangle \to s_0$ and from $\langle S,\ s\rangle \to s_1$ we get $s_0 = s_1$. Thus we have

$$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s_0\rangle \to s' \text{ and } \langle \texttt{while}\ b\ \texttt{do}\ S,\ s_0\rangle \to s''$$

Since $\langle \texttt{while}\ b\ \texttt{do}\ S,\ s_0\rangle \to s'$ is a premise of (the instance of) $[\text{while}^{\text{tt}}_{\text{ns}}]$ we can apply the induction hypothesis to it. From $\langle \texttt{while}\ b\ \texttt{do}\ S,\ s_0\rangle \to s''$ we therefore get $s' = s''$ as required.

The case $[\text{while}^{\text{ff}}_{\text{ns}}]$: Straightforward. □

The Semantic Function S_{ns}

Now that we have shown that the semantics is deterministic we define the semantic function

$$S_{ns}: Stm \to (State \hookrightarrow State)$$

Where S_{ns} is a function, that given the statement Stm, gives us the meaning of the statement, which is a partial function from State to State:

$$S_{ns}[\![S]\!]s = s' if < S, s > \to s' \; and \; S_{ns}[\![S]\!]s \; is \; undefined \; otherwise$$

Examples:

$S_{ns}[\![skip]\!]s = s$

$S_{ns}[\![x := 1]\!]s = s[x \mapsto 1]$

$S_{ns}[\![while \; true \; do \; skip]\!]s = undefined$ – since it does not terminate.

Structural Operational Semantics

This Semantics is similar to the Normal semantics, where for every statement S, we write meaning rules $< S, i > \Rightarrow \gamma$ which means 'If the first step of executing the statement S on an input state i leads to γ'

Where we have two possibilities for γ:

- $\gamma = < S', s' >$ – The execution of S is not completed, S' is the remaining computation which need to be performed on s'.
- $\gamma = o$ – The execution of S has terminated with a final state o
- γ is a stuck configuration when there are no transitions

The meaning of a program P on an input state s is the set of final states that can be executed in arbitrary finite steps.

Since our program meaning is now a series of derivation, this semantics is commonly referred to as Small Step Semantics – every derivation step is one step of the program execution.

This Semantics emphasizes the individual step, and is usually more suitable for analysis.

Structural Semantics for While

Axioms:

- $[ass_{sos}] < x := a, s > \Rightarrow s[x \mapsto A[\![a]\!]s]$
- $[skip_{sos}] < skip, s > \Rightarrow s$
- $[if^{tt}_{sos}] < if \; b \; then \; S_1 \; else \; S_2, s > \Rightarrow < S_1, s > \; if \; B[\![b]\!]s = tt$
- $[if^{ff}_{sos}] < if \; b \; then \; S_1 \; else \; S_2, s > \Rightarrow < S_2, s > \; if \; B[\![b]\!]s = ff$
- $[while_{sos}] < while \; b \; do \; S, s > \Rightarrow < if \; b \; then \; (S; while \; b \; do \; S) \; else \; skip, s >$

Rules:

- $[comp^1_{sos}]\dfrac{< S_1, s >\Rightarrow< S_1', s' >}{< S_1; S_2, s >\Rightarrow< S_1'; S2, s' >}$
- $[comp^2_{sos}]\dfrac{< S_1, s >\Rightarrow s'}{< S_1; S_2, s >\Rightarrow< S_2, s' >}$

As we can see, unlike in NS, now we have Statements as the result as well.

Derivation Sequences

In this Semantics, since the each derivation step defines a small step of the execution, a single statement is not as useful as in the NS case. However, we are interested now in the notion of Derivation Sequence.

Unlike the NS we now include non-terminating programs, and so we distinguish between two cases:

- A finite derivation sequence starting at $< S, s >, \gamma_0, \gamma_1, \gamma_2, \dots, \gamma_k$ such that:
  - $\gamma_0 =< S, s >$
  - $\gamma_i \Rightarrow \gamma_{i+1}$
  - γ_k is either a 'stuck configuration', i.e. a state with no further derivation rules, or a final state.
- An infinite derivation sequence starting at $< S, s >, \gamma_0, \gamma_1, \gamma_2, \dots$ such that:
  - $\gamma_0 =< S, s >$
  - $\gamma_i \Rightarrow \gamma_{i+1}$

We also denote:

- $\gamma_0 \Rightarrow^i \gamma_i$ – i derivation steps.
- $\gamma_0 \Rightarrow^* \gamma_i$ – a finite number of derivation steps.

Program Termination

We can now define Program Termination:

Given a statement S and input s

- *S terminates* on s if there exists a finite derivation sequence starting at $< S, s >$
  - Examples:
    `<x=2;x=x+y,` $[x \mapsto 0]$`>`
- *S terminates successfully* on s if there exists a finite derivation sequence starting at $< S, s >$ leading to a final state
  - Example:
    `<x=2;while(x>0) do (x=x-1),` $[x \mapsto 0]$`>`
- *S loops* on s if there exists an infinite derivation sequence starting at $< S, s >$
  - Example:
    `<x=2;while(x>0) do (x=x+1),` $[x \mapsto 0]$`>`

Properties of the Semantics

Semantic Equivalence

S_1 and S_2 are semantically equivalent if:

- for all s and γ which is either final or stuck,
$$< S_1, s > \Rightarrow^* \gamma \; if \; and \; only \; if < S_2, s > \Rightarrow^* \gamma$$
- there is an infinite derivation sequence starting at $< S_1, s >$ if and only if there is an infinite derivation sequence starting at $< S_2, s >$.

Determinism of the Semantics

$$if < S, s > \Rightarrow^* s_1 \; and < S, s > \Rightarrow^* s_2 \; then \; s_1 = s_2$$

Sequential Composition

The execution of $S_1; S_2$ on an input can be split into two parts:

1. Execute S_1 on s yielding a state s'.
2. Execute S_2 on s'.

If $< S_1; S_2, s > \Rightarrow^k s''$ then there exists a state s' and numbers k_1 and k_2 such that:

- $< S_1, s > \Rightarrow^{k_1} s'$
- $< S_2, s' > \Rightarrow^{k_2} s''$
- And $k = k_1 + k_2$

Unlike previously where we the proof was by induction on the length of the derivation tree, here the proof is by induction on the derivation sequence.

First we prove that the property holds for all derivation sequences of length 0.

Then we prove that the property holds for all other derivation sequences: we show that the property holds for sequences of length $k + 1$ using the fact it holds on all sequences of length k.

The Semantic Function S_{sos}

As was the case for NS, now that we have seen that the semantics is deterministic, we define the semantic function

$$S_{sos}: Stm \rightarrow (State \hookrightarrow State)$$

which for every statement Stm returns the meaning of that statement, which is a partial function from $State$ to $State$.

Where:

$$S_{sos}[\![S]\!]s = s' if < S, s > \Rightarrow^* s' and \; otherwise \; S_{sos}[\![S]\!]s \; is \; undefined.$$

Equivalence of Semantic Functions

The following excerpt from the book proves the following theorem:

Theorem

For every statement S of the while language:

$$S_{nat}[\![S]\!] = S_{sos}[\![S]\!]$$

The theorem is proved in two stages as expressed by Lemma 2.27 and Lemma 2.28 below. We shall first prove:

Lemma 2.27 For every statement S of **While** and states s and s' we have

$$\langle S, s\rangle \to s' \text{ implies } \langle S, s\rangle \Rightarrow^* s'.$$

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

Proof: The proof proceeds by induction on the shape of the derivation tree for $\langle S, s\rangle \to s'$.

The case [ass_{ns}]: We assume that

$$\langle x := a, s\rangle \to s[x \mapsto \mathcal{A}[\![a]\!]s]$$

From [ass_{sos}] we get the required

$$\langle x := a, s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$$

The case [skip_{ns}]: Analogous.

The case [comp_{ns}]: Assume that

$$\langle S_1;S_2, s\rangle \to s''$$

because

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

The induction hypothesis can be applied to both of the premises $\langle S_1, s\rangle \to s'$ and $\langle S_2, s'\rangle \to s''$ and gives

$$\langle S_1, s\rangle \Rightarrow^* s' \text{ and } \langle S_2, s'\rangle \Rightarrow^* s''$$

From Exercise 2.21 we get

$$\langle S_1;S_2, s\rangle \Rightarrow^* \langle S_2, s'\rangle$$

and thereby $\langle S_1;S_2, s\rangle \Rightarrow^* s''$.

The case $[\text{if}^{\text{tt}}_{\text{ns}}]$: Assume that

$$\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, s\rangle \rightarrow s'$$

because

$$\mathcal{B}[\![b]\!]s = \mathbf{tt} \text{ and } \langle S_1, s\rangle \rightarrow s'$$

Since $\mathcal{B}[\![b]\!]s = \mathbf{tt}$ we get

$$\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, s\rangle \Rightarrow \langle S_1, s\rangle \Rightarrow^* s'$$

where the first relationship comes from $[\text{if}^{\text{tt}}_{\text{sos}}]$ and the second from the induction hypothesis applied to the premise $\langle S_1, s\rangle \rightarrow s'$.

The case $[\text{if}^{\text{ff}}_{\text{ns}}]$: Analogous.

The case $[\text{while}^{\text{tt}}_{\text{ns}}]$: Assume that

$$\langle \texttt{while } b \texttt{ do } S, s\rangle \rightarrow s''$$

because

$$\mathcal{B}[\![b]\!]s = \mathbf{tt}, \langle S, s\rangle \rightarrow s' \text{ and } \langle \texttt{while } b \texttt{ do } S, s'\rangle \rightarrow s''$$

The induction hypothesis can be applied to both of the premises $\langle S, s\rangle \rightarrow s'$ and $\langle \texttt{while } b \texttt{ do } S, s'\rangle \rightarrow s''$ and gives

$$\langle S, s\rangle \Rightarrow^* s' \text{ and } \langle \texttt{while } b \texttt{ do } S, s'\rangle \Rightarrow^* s''$$

Using Exercise 2.21 we get

$$\langle S; \texttt{while } b \texttt{ do } S, s\rangle \Rightarrow^* s''$$

Using $[\text{while}_{\text{sos}}]$ and $[\text{if}^{\text{tt}}_{\text{sos}}]$ (with $\mathcal{B}[\![b]\!]s = \mathbf{tt}$) we get the first two steps of

$$\begin{aligned}&\langle \texttt{while } b \texttt{ do } S, s\rangle\\ &\quad\Rightarrow \langle \texttt{if } b \texttt{ then } (S; \texttt{while } b \texttt{ do } S) \texttt{ else skip}, s\rangle\\ &\quad\Rightarrow \langle S; \texttt{while } b \texttt{ do } S, s\rangle\\ &\quad\Rightarrow^* s''\end{aligned}$$

and we have already argued for the last part.

The case $[\text{while}^{\text{ff}}_{\text{ns}}]$: Straightforward. □

This completes the proof of Lemma 2.27. The second part of the theorem follows from:

Lemma 2.28 For every statement S of **While**, states s and s' and natural number k we have that

$$\langle S, s\rangle \Rightarrow^{k} s' \text{ implies } \langle S, s\rangle \rightarrow s'.$$

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

Proof: The proof proceeds by induction on the length of the derivation sequence $\langle S, s\rangle \Rightarrow^{k} s'$, that is by induction on k.

If k=0 then the result holds vacuously.

To prove the induction step we assume that the lemma holds for $k \leq k_0$ and we shall then prove that it holds for k_0+1. We proceed by cases on how the first step of $\langle S, s\rangle \Rightarrow^{k_0+1} s'$ is obtained, that is by inspecting the derivation tree for the first step of computation in the structural operational semantics.

The case $[ass_{sos}]$: Straightforward (and $k_0 = 0$).

The case $[skip_{sos}]$: Straightforward (and $k_0 = 0$).

The cases $[comp^1_{sos}]$ and $[comp^2_{sos}]$: In both cases we assume that

$$\langle S_1;S_2, s\rangle \Rightarrow^{k_0+1} s''$$

We can now apply Lemma 2.19 and get that there exists a state s' and natural numbers k_1 and k_2 such that

$$\langle S_1, s\rangle \Rightarrow^{k_1} s' \text{ and } \langle S_2, s'\rangle \Rightarrow^{k_2} s''$$

where $k_1+k_2=k_0+1$. The induction hypothesis can now be applied to each of these derivation sequences because $k_1 \leq k_0$ and $k_2 \leq k_0$. So we get

$$\langle S_1, s\rangle \rightarrow s' \text{ and } \langle S_2, s'\rangle \rightarrow s''$$

Using $[\text{comp}_{\text{ns}}]$ we now get the required $\langle S_1;S_2,\ s\rangle \to s''$.

The case $[\text{if}^{\text{tt}}_{\text{sos}}]$: Assume that $\mathcal{B}[\![b]\!]s = \mathbf{tt}$ and that

$$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle S_1,\ s\rangle \Rightarrow^{k_0} s'$$

The induction hypothesis can be applied to the derivation sequence $\langle S_1,\ s\rangle \Rightarrow^{k_0} s'$ and gives

$$\langle S_1,\ s\rangle \to s'$$

The result now follows using $[\text{if}^{\text{tt}}_{\text{ns}}]$.

The case $[\text{if}^{\text{ff}}_{\text{sos}}]$: Analogous.

The case $[\text{while}_{\text{sos}}]$: We have

$$\begin{aligned}&\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle\\ &\quad\Rightarrow \langle \texttt{if}\ b\ \texttt{then}\ (S;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else}\ \texttt{skip},\ s\rangle\\ &\quad\Rightarrow^{k_0} s''\end{aligned}$$

The induction hypothesis can be applied to the k_0 last steps of the derivation sequence and gives

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

and from Lemma 2.5 we get the required

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

Proof of Theorem 2.26: For an arbitrary statement S and state s it follows from Lemmas 2.27 and 2.28 that if $\mathcal{S}_{\text{ns}}[\![S]\!]s = s'$ then $\mathcal{S}_{\text{sos}}[\![S]\!]s = s'$ and vice versa. This suffices for showing that the functions $\mathcal{S}_{\text{ns}}[\![S]\!]$ and $\mathcal{S}_{\text{sos}}[\![S]\!]$ must be equal: if one is defined on a state s then so is the other, and therefore, if one is not defined on a state s then neither is the other. □

Extensions to the While Programming Language

An addition of an 'abort' statement

Abstract Syntax:

$$S ::= x := a \mid skip \mid S_1; S_2 \mid if\ b\ then\ S_1\ else\ S_2 \mid while\ b\ do\ S \mid abort$$

'abort' terminates the execution.

No new rules are needed in neither Natural nor Structural Operational Semantics.

Examples:

- `x:=12;y:=3;if (x=0) then abort else z:=x*y`
- `x:=10;while(x>0) if (x=3) then abort else skip`

Theorem

`while true do skip` is semantically equivalent to `abort` in NS.

Proof

In natural semantics we are only concerned with executions that terminate properly. In either case there is no derivation tree for $< S, s > \rightarrow o$, in the case of abort this is because the derivation enters a stuck configuration because there is no derivation rule. In the case of while true do skip it is because the derivation has entered a looping configuration.

Since in NS we cannot differentiate between the two cases, they are semantically equivalent.

■

Theorem

`while true do skip` is not semantically equivalent to `abort` in SOS.

Proof

$< abort, s >$ is the only derivation sequence for `abort` (a stuck configuration).

Whereas the derivation sequence for `while true do skip` is:

$< while\ true\ do\ skip, s >$

$\Rightarrow < if\ true\ then\ (skip; while\ true\ do\ skip)\ else\ skip, s >$

$\Rightarrow < skip; while\ true\ do\ skip, s >$

$\Rightarrow < while\ true\ do\ skip, s >$

$\Rightarrow \cdots$

We see that the two cannot be equivalent – one is a stuck configuration with no derivations, whereas the other is an infinite derivation sequence.

■

An addition of Non-Determinism

Abstract Syntax:

$$S ::= x := a \mid skip \mid S_1; S_2 \mid if\ b\ then\ S_1\ else\ S_2 \mid while\ b\ do\ S \mid S_1\ or\ S_2$$

In the non-deterministic statement, either S_1 or S_2 are executed.

Additions to NS:

New Rules:

- $[or_{ns}^1] \dfrac{< S_1, s >\rightarrow s'}{< S_1\ or\ S_2, s >\rightarrow s'}$
- $[or_{ns}^2] \dfrac{< S_2, s >\rightarrow s'}{< S_1\ or\ S_2, s >\rightarrow s'}$

Additions to SOS:

New Axioms:

- $[or_{sos}^1] < S_1\ or\ S_2, i > \Rightarrow < S_1, i >$
- $[or_{sos}^2] < S_1\ or\ S_2, i > \Rightarrow < S_2, i >$

Examples:

- `x:=10; while x>0 do x:=x+1 or x:=x-1`
- `x:=4;y:=10; while (y>x) do x:=x+1 or x:=x-1 or y:=y+1 or y:=y-1`

Theorem

`while true do skip or x:=1` is semantically equivalent to `x:=1` in NS

proof

We look at the derivation trees of the two statements:

$< while\ true\ do\ skip\ or\ x := 1, s >\rightarrow s[x \mapsto 1]$

$< x := 1, s >\rightarrow s[x \mapsto 1]$

As we can see, each one of the statements has exactly one derivation tree, with the same resulting end state. Hence, they are equivalent.

■

Theorem

`while true do skip or x:=1` is not semantically equivalent to `x:=1` in SOS

Proof

We look at the derivation trees of the two statements:

- As before, `x:=1` has only one, finite derivation tree:
  $< x \coloneqq 1, s > \Rightarrow s[x \mapsto 1]$
- The second statement, while true do skip or x:=1, however, has two derivation trees:
  - One is a finite derivation tree:
    $< while\ true\ do\ skip\ of\ x \coloneqq 1, s > \Rightarrow^{*} s[x \mapsto 1]$
  - And the other an infinite derivation sequence:
    $< while\ true\ do\ skip\ or\ x \coloneqq 1, s >$
    $\Rightarrow < while\ true\ do\ skip, s >$
    $\Rightarrow^{3} < while\ true\ do\ skip, s >$
    $\Rightarrow \cdots$

Hence, the two cannot be equivalent.

■

An addition of Parallelism

Abstract Syntax:

$$S ::= x \coloneqq a \mid skip \mid S_1; S_2 \mid if\ b\ then\ S_1\ else\ S_2 \mid while\ b\ do\ S \mid S_1\ par\ S_2$$

Additions to SOS:

New Rules:

- $$[par_{sos}^{1}] \frac{< S_1, s > \Rightarrow < S_1', s' >}{< S_1\ par\ S_2, s > \Rightarrow < S_1'\ par\ S_2, s >}$$
- $$[par_{sos}^{2}] \frac{< S_1, s > \Rightarrow s'}{< S_1\ par\ S_2, s > \Rightarrow < S_2, s' >}$$
- $$[par_{sos}^{3}] \frac{< S_2, s > \Rightarrow < S_2', s' >}{< S_1\ par\ S_2, s > \Rightarrow < S_1\ par\ S_2', s' >}$$
- $$[par_{sos}^{4}] \frac{< S_2, s > \Rightarrow s'}{< S_1\ par\ S_2, s > \Rightarrow < S_1, s' >}$$

New Rules in NS

Cannot be defined in NS, since NS defines the behavior in terms of input and output, the Semantic cannot be used to differentiate the cases in this level of detail.

Example:

```
x:=5; while x>0 do x:=x-1 par while x<10 do x:=x+1
```

An addition of local variables and procedures

Abstract Syntax:

$S ::= x := a | skip | S_1; S_2 | if\ b\ then\ S_1\ else\ S_2 | while\ b\ do\ S | begin\ D_v\ D_p\ S\ end$|call p

$D_v ::= var\ x := a; D_v | \varepsilon$

$D_p ::= proc\ p\ is\ S; D_p | \varepsilon$

Example

```
begin var f:=x; var r:=1;proc fact is while f>1 do
(r:=r*f; f:=f-1) end; x:=5; call fact; x:=8; call fact
```

What do we need to add to NS and SOS?

In SOS we need to define a new mechanism, a new data structure e.g. a stack, that will retain the state in the outer scope.

In NS we do not need any such mechanism, since the rules of NS are defined inductively, the premise of the rule can hold a different state than the result, and the different state may hold the different states in different scopes.

Transition Systems

We define another form notion of Semantics – a Transition System.

This form is a much lower level of Semantics than we have previously seen.

The meaning of a program is a relation $\tau \subseteq \Sigma \times \Sigma$.

An execution of a program is a finite sequence of states.

Example

```
1:y:=1
While 2:¬(x=1) do (
   3: y:=y*x;
   4: x:=x-1
)
5:
```

The set of states for the above program:

$pc = 1 \wedge pc' = 2 \wedge y' = 1 \wedge x' = x$

$pc = 2 \wedge x = 1 \wedge pc' = 5 \wedge x' = 5 \wedge y' = y$

$pc = 2 \wedge x \neq 1 \wedge pc' = 3 \wedge x' = x \wedge y' = y$

$pc = 3 \wedge pc' = 4 \wedge x' = x \wedge y' = y * x$

$pc = 4 \wedge pc' = 2 \wedge x' = x - 1 \wedge y' = y$

Appendix – Operational Semantics Implemented in Prolog

Following is a Prolog implementation of both NS and SOS, as well as the results of a few example runs.

The following three parts are shared between NS and SOS:

Environment Operations:

These are operations on the state:

```
%%% Update Value
upd([], Var, Val, [Var/Val]).
upd([Var/_ | T], Var, Val, [Var/Val | T]).
upd([Varp/V | T], Var, Val, [Varp/V | Envp])  :- Varp \= Var, upd(T, Var, Val, Envp).

%% Retrieve Values
eval_env([Var/Val |_], Var, Val).
eval_env([Varp/_ | T], Var, Val)  :- Varp \= Var, eval_env(T, Var, Val).
```

Arithmetic Expressions:

The semantics of arithmetic expressions, defined using the native Prolog arithmetic operations:

```
eval_exp(Num, _, Num) :- number(Num).
eval_exp(Var, Env, Val) :- atom(Var), eval_env(Env, Var, Val).
eval_exp(plus(E1, E2), Env, Val) :- eval_exp(E1, Env, V1), eval_exp(E2, Env, V2),
    Val is V1 + V2.
eval_exp(minus(E1, E2), Env, Val) :- eval_exp(E1, Env, V1), eval_exp(E2, Env, V2),
    Val is V1 - V2.
eval_exp(mul(E1, E2), Env, Val) :- eval_exp(E1, Env, V1), eval_exp(E2, Env, V2),
    Val is V1 * V2.
```

Boolean Expressions:

The semantics of Boolean operations, defined using the native Prolog Boolean operations, except for the definition of ‘not’ which is defined as ‘anything which is not true’:

```
eval_boolean_exp(true,_).
eval_boolean_exp(eq(E1, E2), Env) :- eval_exp(E1, Env, V1), eval_exp(E2, Env, V2),
    V1 = V2.
eval_boolean_exp(le(E1, E2), Env) :- eval_exp(E1, Env, V1), eval_exp(E2, Env, V2),
    V1 =< V2.
eval_boolean_exp(lt(E1, E2), Env) :- eval_exp(E1, Env, V1), eval_exp(E2, Env, V2),
    V1 < V2.
eval_boolean_exp(land(E1, E2), Env) :- eval_boolean_exp(E1, Env), eval_boolean_exp(E2,
Env).
eval_boolean_exp(lor(E1, _), Env) :- eval_boolean_exp(E1, Env).
eval_boolean_exp(lor(_, E2), Env) :- eval_boolean_exp(E2, Env).
eval_boolean_exp(lneg(E), Env) :- \+ eval_boolean_exp(E, Env).
```

Following are the definitions of both Operational Semantics in Prolog. We can see that the definitions in Prolog match one-to-one the derivation rules and axioms we've previously seen.

Natural Semantics:

Axioms:

```
%% Skip Statement
nat(skip, Env, Env).
%% Simple Assignment Statement - assignment axiom
nat(ass(LHS, RHS), Env, Envp) :- eval_exp(RHS, Env, Val), upd(Env, LHS, Val, Envp).
```

Derivation Rules:

```
%% Sequential Composition
nat(seq(S1, S2), Env, Envp) :- nat(S1, Env, Envpp), nat(S2, Envpp, Envp).

%% Conditional Statements
nat(if(B, S1, _), Env, Envp) :- eval_boolean_exp(B, Env), nat(S1, Env, Envp).
nat(if(B, _, S2), Env, Envp) :- \+ eval_boolean_exp(B, Env), nat(S2, Env, Envp).

%% Loops
nat(while(B,S), Env, Envp) :- eval_boolean_exp(B, Env), nat(S, Env, Envpp),
nat(while(B,S), Envpp, Envp).
nat(while(B,_), Env, Env) :- \+ eval_boolean_exp(B, Env).
```

Sequential Operational Semantic:

Axioms:

```
%% Skip Statement
sos(skip, Env, Env).
%% Simple Assignment Statement - assignment axiom
sos(ass(LHS, RHS), Env, Envp) :- eval_exp(RHS, Env, Val), upd(Env, LHS, Val, Envp).

%% Conditional Statements
sos(if(B, S1, _), Env, S1, Env) :- eval_boolean_exp(B, Env).
sos(if(B, _, S2), Env, S2, Env) :- \+ eval_boolean_exp(B, Env).

%% Loops
sos(while(B,S), Env, if(B, seq(S, while(B,S)), skip), Env).
```

Derivation Rules:

```
%% Sequential Composition
sos(seq(S1, S2), Env, S2, Envp) :- sos(S1, Env, Envp).
sos(seq(S1, S2), Env, seq(S1p, S2), Envp) :- sos(S1, Env, S1p, Envp).

%% Transitive Closure
rtc_sos(S, Env, S, Env).
rtc_sos(S, Env, Sp, Envp) :- sos(S, Env, Spp, Envpp), rtc_sos(Spp, Envpp, Sp, Envp).

%% Atomic termination
rtc_sos(S, Env, Envp) :- rtc_sos(S, Env, Sp, Envpp), sos(Sp, Envpp, Envp).
```

Run Example:

We'll see an example run of the following 'while' program - factorial:

```
y:=1

while (x!=1) do (

    y:=y*x;

    x:=x-1;

)
```

As can be expected, the results are exactly the same when running the program in either Semantics. e.g. for the input y=2:

NS:

```
?- nat(seq(ass(y, 1), while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1))))), [x/2], E).
E = [x/1, y/2] .
```

SOS:

```
?- rtc_sos(seq(ass(y, 1), while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1))))), [x/2], E).
E = [x/1, y/2] .
```

Lets now have a look at the derivation sequence in Prolog for the two different Semantics:

NS:

```
?- nat(seq(ass(y, 1), while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1))))), [x/1], E).
   Call: (6) nat(seq(ass(y, 1), while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1))))), [x/1], _G2086) ? creep
   Call: (7) nat(ass(y, 1), [x/1], _G2196) ? creep
   Call: (8) eval_exp(1, [x/1], _G2196) ? creep
   Call: (9) number(1) ? creep
   Exit: (9) number(1) ? creep
   Exit: (8) eval_exp(1, [x/1], 1) ? creep
   Call: (8) upd([x/1], y, 1, _G2197) ? creep
   Call: (9) x\=y ? creep
   Exit: (9) x\=y ? creep
   Call: (9) upd([], y, 1, _G2188) ? creep
   Exit: (9) upd([], y, 1, [y/1]) ? creep
   Exit: (8) upd([x/1], y, 1, [x/1, y/1]) ? creep
   Exit: (7) nat(ass(y, 1), [x/1], [x/1, y/1]) ? creep
   Call: (7) nat(while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1)))), [x/1, y/1], _G2086) ? creep
   Call: (8) eval_boolean_exp(lneg(eq(x, 1)), [x/1, y/1]) ? creep
   Call: (9) eval_boolean_exp(eq(x, 1), [x/1, y/1]) ? creep
   Call: (10) eval_exp(x, [x/1, y/1], _G2208) ? creep
   Call: (11) number(x) ? creep
   Fail: (11) number(x) ? creep
   Redo: (10) eval_exp(x, [x/1, y/1], _G2208) ? creep
   Call: (11) atom(x) ? creep
   Exit: (11) atom(x) ? creep
   Call: (11) eval_env([x/1, y/1], x, _G2208) ? creep
   Exit: (11) eval_env([x/1, y/1], x, 1) ? creep
   Exit: (10) eval_exp(x, [x/1, y/1], 1) ? creep
   Call: (10) eval_exp(1, [x/1, y/1], _G2208) ? creep
   Call: (11) number(1) ? creep
   Exit: (11) number(1) ? creep
   Exit: (10) eval_exp(1, [x/1, y/1], 1) ? creep
   Call: (10) 1=1 ? creep
   Exit: (10) 1=1 ? creep
   Exit: (9) eval_boolean_exp(eq(x, 1), [x/1, y/1]) ? creep
```

```
   Fail: (8) eval_boolean_exp(lneg(eq(x, 1)), [x/1, y/1]) ? creep
   Redo: (7) nat(while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1)))), [x/1, y/1], _G2086) ? creep
   Call: (8) eval_boolean_exp(lneg(eq(x, 1)), [x/1, y/1]) ? creep
   Call: (9) eval_boolean_exp(eq(x, 1), [x/1, y/1]) ? creep
   Call: (10) eval_exp(x, [x/1, y/1], _G2208) ? creep
   Call: (11) number(x) ? creep
   Fail: (11) number(x) ? creep
   Redo: (10) eval_exp(x, [x/1, y/1], _G2208) ? creep
   Call: (11) atom(x) ? creep
   Exit: (11) atom(x) ? creep
   Call: (11) eval_env([x/1, y/1], x, _G2208) ? creep
   Exit: (11) eval_env([x/1, y/1], x, 1) ? creep
   Exit: (10) eval_exp(x, [x/1, y/1], 1) ? creep
   Call: (10) eval_exp(1, [x/1, y/1], _G2208) ? creep
   Call: (11) number(1) ? creep
   Exit: (11) number(1) ? creep
   Exit: (10) eval_exp(1, [x/1, y/1], 1) ? creep
   Call: (10) 1=1 ? creep
   Exit: (10) 1=1 ? creep
   Exit: (9) eval_boolean_exp(eq(x, 1), [x/1, y/1]) ? creep
   Fail: (8) eval_boolean_exp(lneg(eq(x, 1)), [x/1, y/1]) ? creep
   Redo: (7) nat(while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1)))), [x/1, y/1], [x/1, y/1]) ? creep
   Exit: (7) nat(while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1)))), [x/1, y/1], [x/1, y/1]) ? creep
   Exit: (6) nat(seq(ass(y, 1), while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1))))), [x/1], [x/1, y/1]) ?
creep
E = [x/1, y/1] .
```

SOS:

```
?- rtc_sos(seq(ass(y, 1), while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1))))), [x/1], E).
   Call: (6) rtc_sos(seq(ass(y, 1), while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1))))), [x/1], _G1005) ?
creep
   Call: (7) rtc_sos(seq(ass(y, 1), while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1))))), [x/1], _G1115,
_G1116) ? creep
   Exit: (7) rtc_sos(seq(ass(y, 1), while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1))))), [x/1],
seq(ass(y, 1), while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1))))), [x/1]) ? creep
   Call: (7) sos(seq(ass(y, 1), while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1))))), [x/1], _G1005) ?
creep
   Fail: (7) sos(seq(ass(y, 1), while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1))))), [x/1], _G1005) ?
creep
   Redo: (7) rtc_sos(seq(ass(y, 1), while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1))))), [x/1], _G1115,
_G1116) ? creep
   Call: (8) sos(seq(ass(y, 1), while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1))))), [x/1], _G1115,
_G1116) ? creep
   Call: (9) sos(ass(y, 1), [x/1], _G1115) ? creep
   Call: (10) eval_exp(1, [x/1], _G1115) ? creep
   Call: (11) number(1) ? creep
   Exit: (11) number(1) ? creep
   Exit: (10) eval_exp(1, [x/1], 1) ? creep
   Call: (10) upd([x/1], y, 1, _G1116) ? creep
   Call: (11) x\=y ? creep
   Exit: (11) x\=y ? creep
   Call: (11) upd([], y, 1, _G1107) ? creep
   Exit: (11) upd([], y, 1, [y/1]) ? creep
   Exit: (10) upd([x/1], y, 1, [x/1, y/1]) ? creep
   Exit: (9) sos(ass(y, 1), [x/1], [x/1, y/1]) ? creep
   Exit: (8) sos(seq(ass(y, 1), while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1))))), [x/1],
while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1)))), [x/1, y/1]) ? creep
   Call: (8) rtc_sos(while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1)))), [x/1, y/1], _G1127, _G1128) ?
creep
   Exit: (8) rtc_sos(while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1)))), [x/1, y/1], while(lneg(eq(x,
1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1)))), [x/1, y/1]) ? creep
   Exit: (7) rtc_sos(seq(ass(y, 1), while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1))))), [x/1],
while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1)))), [x/1, y/1]) ? creep
   Call: (7) sos(while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1)))), [x/1, y/1], _G1005) ? creep
   Fail: (7) sos(while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1)))), [x/1, y/1], _G1005) ? creep
   Redo: (8) rtc_sos(while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1)))), [x/1, y/1], _G1127, _G1128) ?
creep
   Call: (9) sos(while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1)))), [x/1, y/1], _G1127, _G1128) ? creep
```

```
   Exit: (9) sos(while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1)))), [x/1, y/1], if(lneg(eq(x, 1)),
seq(seq(ass(y, mul(y, x)), ass(x, minus(x, 1))), while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1))))),
skip), [x/1, y/1]) ? creep

   Call: (9) rtc_sos(if(lneg(eq(x, 1)), seq(seq(ass(y, mul(y, x)), ass(x, minus(x, 1))), while(lneg(eq(x, 1)), seq(ass(y,
mul(y, x)), ass(x, minus(x, 1))))), skip), [x/1, y/1], _G1137, _G1138) ? creep

   Exit: (9) rtc_sos(if(lneg(eq(x, 1)), seq(seq(ass(y, mul(y, x)), ass(x, minus(x, 1))), while(lneg(eq(x, 1)), seq(ass(y,
mul(y, x)), ass(x, minus(x, 1))))), skip), [x/1, y/1], if(lneg(eq(x, 1)), seq(seq(ass(y, mul(y, x)), ass(x, minus(x,
1))), while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1))))), skip), [x/1, y/1]) ? creep

   Exit: (8) rtc_sos(while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1)))), [x/1, y/1], if(lneg(eq(x, 1)),
seq(seq(ass(y, mul(y, x)), ass(x, minus(x, 1))), while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1))))),
skip), [x/1, y/1]) ? creep

   Exit: (7) rtc_sos(seq(ass(y, 1), while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1))))), [x/1],
if(lneg(eq(x, 1)), seq(seq(ass(y, mul(y, x)), ass(x, minus(x, 1))), while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x,
minus(x, 1))))), skip), [x/1, y/1]) ? creep

   Call: (7) sos(if(lneg(eq(x, 1)), seq(seq(ass(y, mul(y, x)), ass(x, minus(x, 1))), while(lneg(eq(x, 1)), seq(ass(y,
mul(y, x)), ass(x, minus(x, 1))))), skip), [x/1, y/1], _G1005) ? creep

   Fail: (7) sos(if(lneg(eq(x, 1)), seq(seq(ass(y, mul(y, x)), ass(x, minus(x, 1))), while(lneg(eq(x, 1)), seq(ass(y,
mul(y, x)), ass(x, minus(x, 1))))), skip), [x/1, y/1], _G1005) ? creep

   Redo: (9) rtc_sos(if(lneg(eq(x, 1)), seq(seq(ass(y, mul(y, x)), ass(x, minus(x, 1))), while(lneg(eq(x, 1)), seq(ass(y,
mul(y, x)), ass(x, minus(x, 1))))), skip), [x/1, y/1], _G1137, _G1138) ? creep

   Call: (10) sos(if(lneg(eq(x, 1)), seq(seq(ass(y, mul(y, x)), ass(x, minus(x, 1))), while(lneg(eq(x, 1)), seq(ass(y,
mul(y, x)), ass(x, minus(x, 1))))), skip), [x/1, y/1], _G1137, _G1138) ? creep

   Call: (11) eval_boolean_exp(lneg(eq(x, 1)), [x/1, y/1]) ? creep

   Call: (12) eval_boolean_exp(eq(x, 1), [x/1, y/1]) ? creep

   Call: (13) eval_exp(x, [x/1, y/1], _G1137) ? creep

   Call: (14) number(x) ? creep

   Fail: (14) number(x) ? creep

   Redo: (13) eval_exp(x, [x/1, y/1], _G1137) ? creep

   Call: (14) atom(x) ? creep

   Exit: (14) atom(x) ? creep

   Call: (14) eval_env([x/1, y/1], x, _G1137) ? creep

   Exit: (14) eval_env([x/1, y/1], x, 1) ? creep

   Exit: (13) eval_exp(x, [x/1, y/1], 1) ? creep

   Call: (13) eval_exp(1, [x/1, y/1], _G1137) ? creep

   Call: (14) number(1) ? creep

   Exit: (14) number(1) ? creep

   Exit: (13) eval_exp(1, [x/1, y/1], 1) ? creep

   Call: (13) 1=1 ? creep

Exit: (13) 1=1 ? creep

   Exit: (12) eval_boolean_exp(eq(x, 1), [x/1, y/1]) ? creep

   Fail: (11) eval_boolean_exp(lneg(eq(x, 1)), [x/1, y/1]) ? creep

   Redo: (10) sos(if(lneg(eq(x, 1)), seq(seq(ass(y, mul(y, x)), ass(x, minus(x, 1))), while(lneg(eq(x, 1)), seq(ass(y,
mul(y, x)), ass(x, minus(x, 1))))), skip), [x/1, y/1], _G1137, _G1138) ? creep

   Call: (11) eval_boolean_exp(lneg(eq(x, 1)), [x/1, y/1]) ? creep

   Call: (12) eval_boolean_exp(eq(x, 1), [x/1, y/1]) ? creep

   Call: (13) eval_exp(x, [x/1, y/1], _G1137) ? creep

   Call: (14) number(x) ? creep

   Fail: (14) number(x) ? creep

   Redo: (13) eval_exp(x, [x/1, y/1], _G1137) ? creep

   Call: (14) atom(x) ? creep

   Exit: (14) atom(x) ? creep

   Call: (14) eval_env([x/1, y/1], x, _G1137) ? creep

   Exit: (14) eval_env([x/1, y/1], x, 1) ? creep

   Exit: (13) eval_exp(x, [x/1, y/1], 1) ? creep

   Call: (13) eval_exp(1, [x/1, y/1], _G1137) ? creep

   Call: (14) number(1) ? creep

   Exit: (14) number(1) ? creep

   Exit: (13) eval_exp(1, [x/1, y/1], 1) ? creep

   Call: (13) 1=1 ? creep

   Exit: (13) 1=1 ? creep

   Exit: (12) eval_boolean_exp(eq(x, 1), [x/1, y/1]) ? creep

   Fail: (11) eval_boolean_exp(lneg(eq(x, 1)), [x/1, y/1]) ? creep

   Redo: (10) sos(if(lneg(eq(x, 1)), seq(seq(ass(y, mul(y, x)), ass(x, minus(x, 1))), while(lneg(eq(x, 1)), seq(ass(y,
mul(y, x)), ass(x, minus(x, 1))))), skip), [x/1, y/1], skip, [x/1, y/1]) ? creep

   Exit: (10) sos(if(lneg(eq(x, 1)), seq(seq(ass(y, mul(y, x)), ass(x, minus(x, 1))), while(lneg(eq(x, 1)), seq(ass(y,
mul(y, x)), ass(x, minus(x, 1))))), skip), [x/1, y/1], skip, [x/1, y/1]) ? creep

   Call: (10) rtc_sos(skip, [x/1, y/1], _G1137, _G1138) ? creep

   Exit: (10) rtc_sos(skip, [x/1, y/1], skip, [x/1, y/1]) ? creep

   Exit: (9) rtc_sos(if(lneg(eq(x, 1)), seq(seq(ass(y, mul(y, x)), ass(x, minus(x, 1))), while(lneg(eq(x, 1)), seq(ass(y,
mul(y, x)), ass(x, minus(x, 1))))), skip), [x/1, y/1], skip, [x/1, y/1]) ? creep
```

```
   Exit: (8) rtc_sos(while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1)))), [x/1, y/1], skip, [x/1, y/1]) ?
creep
   Exit: (7) rtc_sos(seq(ass(y, 1), while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1))))), [x/1], skip,
[x/1, y/1]) ? creep
   Call: (7) sos(skip, [x/1, y/1], _G1005) ? creep
   Exit: (7) sos(skip, [x/1, y/1], [x/1, y/1]) ? creep
   Exit: (6) rtc_sos(seq(ass(y, 1), while(lneg(eq(x, 1)), seq(ass(y, mul(y, x)), ass(x, minus(x, 1))))), [x/1], [x/1,
y/1]) ? creep
E = [x/1, y/1] .
```

In the derivation above we can see that while the SOS derives a single step at each derivation step (hence, small-step semantics), the NS attempts to derive the whole statement.