Program Analysis/ Noam Rinetzky and Mooly Sagiv
Lecture 2, 03.03.2005 Operational Semantics
Notes by: Binun Alexander

Syntax and Semantics – Why do we need them?

Programming language consists of syntax, and formal semantics. When we write a program in some program language, we merely type chains of symbols (tokens - after a scanner processes them , syntax trees – after a parser processes). Say,

```
y:=1;
While not (x=1)
 {
    y:=y*x; x:=x-1;
 }
```

The syntax of a programming language is concerned with the grammatical structure of programs such as regular expressions, and context-free grammars. The semantics of a program is the meaning of grammatically correct programs (incorrect ones – are filtered out by compilers). The level of understanding the semantics can be divided into three categories (deeper and deeper…)

1) Understand which actions to be performed - to write high-quality programs in this language
2) Divide these actions into smaller ones, select a "kernel subset" and think how to implement them. Which syntactic constructions can describe them? This knowledge is required to write compilers (interpreters) for our language.
3) Understand which motivations led to our language. Which observations can be made? Formulate "strategic goals" laid in the foundation of our language. If they change, how our language will change? This knowledge allows to design program languages.

The While programming language

Syntactic Categories

n ranges over natural, **Num**,

x ranges over variables, **Var**,

a ranges over arithmetic expressions, **Aexp**,

b ranges over Boolean expressions, **Bexp**,

S range over program statements, **Stm**.

Semantic Categories

Natural values: **N={0,1,2,….}**

Truth values: **T={ff,tt}**

States: **State = Var→ N** (i.e., function from Var to Natural).

Lookup in a state **s: s x** (fetching a value from a state).

Update of a state **s: s [x:=5]** (updating a value in a state).

Grammar rules

a::= n | x | a1 + a2 | a1 * a2 | a1 – a2 – arithmetic expressions

b::= tt | ff | a1 = a2 | a1 <= a2 | ~b | b1 ^ b2 – Boolean expressions

S::= x := a | **skip** | S1 ; S2 | **if** b **then** S1 **else** S2 | **while** b do S - statements

The kinds of semantics

The While language presents two kinds of actions used to express the meaning of the statements (expressions): atomic and combined. Atomic actions are returning of a constant value (integer, boolean), returning the variable value from a state and updating a state. Combined actions are composed , in particular, using the subexpressions of the original expression

Operational semantics

The meaning of the actions is specified by the computation it induces when it is executed on the machine. Formally, our semantics is a function that, given an expression as a symbolic value (and maybe a set of current variables – the state) returns the value of this expression - interprets it (or the next state – for a statement). This can be subdivided into the following categories:

1) Simple Operational Semantics**:** arithmetic (A) and boolean (B)

A[n]s = n - Number semantic meaning is it's numerical value

A[x]s = sx - Variable semantic meaning - lookup in States.

A[e_1+e_2]s= A[e_1]s+A[e_2]s - Addition

A[e_1*e_2]s = A[e_1]s*A[e_2]s - Multiplication

B[true]s = tt - true semantic meaning is truth value.

B[false] = ff - false semantic meaning is the false value.

B[x]s = sx - Variable semantic meaning - lookup in States.

B[$e_1 \wedge e_2$]s = {**tt** - if **B[e_1]s = tt** and **B[e_2]s = tt**, **ff** – otherwise}, similar OR

B[$\neg$e]s = not B[e]s - Negation

Note that the overall meaning of the expression is obtained ONLY from the meaning of the subexpressions!

These rules are also the part of denotational semantics. The meaning of a complex expression is defined using the meaning of subexpressions.

2) **Operational Natural semantics** – (Divide/Conquer technique) the overall result of the execution of the statement **S** from the start state **s** to the final state **s'** (written: **<S,s> → s'**) is divided recursively into the overall effects of the computations of the S's subexpressions S_i from their start substates **(<S_i,s_i> → s'$_i$**) and combining these "start substates" into **s'** (conquer). For example: to compute **<S_1, S_2, s> → s'** its components are computed separately: **<S_1, s>→ s_{11}**, **<S_2, s_{11}> → s'** , the intermediate states are "piped".

Therefore, the computation process is presented as a tree and it will be suitable to prove properties by induction on its shape. We will return to this later. The problem is to express looping – the natural semantics can express ONLY terminating constructs whose computations are OK, since we start from the overall terminating construct and divide it.

4) **Operational structural semantics**: Take the first "atomic" action (update state, etc) from the overall expression computation **<S,s>,** execute it (passing to the **state s' - <S,s>=>s'**). The remainder of **S - S'** may to be computed already from **s'** (**<S', s'>=>*s''**). Note that a computation is modeled by the derivation relation □ and therefore cycling also can be formally expressed. Abnormal states also can be modeled (by marking one state as "aborting". The computations thus are modeled by derivation sequences and properties can be proven by induction on the derivation sequence length.

To compute **<S_1, S_2, s>=> s'** two options can be considered:

a) the execution of the first part has not completed: if **<S_1, s>=><S'_1, s'>** then

<S_1; S_2, s>=><S'_1;S_2, s'>

the execution of the first part has completed – pass to the second part: if **<S_1, s>□s'** then **<S_1;S_2, s>=><S_2, s'>**

Denotational semantics

The essence of Denotational Semantics is that semantic functions are defined compositionally. IN other words, for each basic element of syntactic category there is a semantic clause interpreting it. For each method to construct a composite syntactic element, we define a semantic clause defined in the terms of semantic functions applied to the subexpressions.

For example, for each natural number its meaning **M[n]s = n**. For each variable, **M[x]s = s(x)**. For arithmetical (boolean) operations, **M[a op b]s = M[a]s op M[b]s**. So, the simple structural semantics is applicable here.

The meaning of each statement is a mathematical function from **States** to **States**). The effect of a complex expression is a combination of the functions of its components. For a sequence **S1; S2** the overall effect is the composition of the meanings of **S1**and **S2**.

The essence of this approach is that the meaning of an expression is expressed only by reasoning about mathematical objects; we are not concerned with how a given expression is executed. However, given a firm mathematical basis for a functions; it will be easy to rigorously prove many results using this basis.

Axiomatic semantics

It is used to formally prove the program correctness – i.e. to prove whether a program satisfies some property. To do this, we write the pre- and postcondition for our program – i.e. express the wanted properties in some logic-based language. Then we try to verify whether these properties can be obtained from the program components using some predefined derivation rules, repeat this process for the components and so on – until the basic pre- and postcondition (for atomic expressions) are reached. We prove ONLY the properties that interest us – therefore this process is called "proving partial correctness" and the meaning (semantics) of a program expressed by our properties is also partial. For example, for variable exchanging program (the precondition is written before the statement, the postcondition – after)

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

Using a tree-like derivation:

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

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

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

Operational Semantics in Details

Natural Semantics of While Language

Axioms:

$\langle x:=a, s\rangle \rightarrow s[x:=A[a]]$

$\langle skip,s\rangle \rightarrow s$ - Skip statement

$\underline{\langle S_1,s\rangle \rightarrow s', \langle S_2,s'\rangle \rightarrow s''}$ - Composition of two statements.

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

Rules:

If

$\underline{<S_1,s> \rightarrow s'}$ - if B[b]s=tt (If-Execute then-branch if true)

<if b the S_1 else S_2,s>→S'

$\underline{<S_2,s> \rightarrow s'}$ - if B[b]s=ff (If-Execute else-branch if false)

<if b the S_1 else S_2,s>→S'

That is, if the test evaluates to TRUE, execute then-branch from the current state, otherwise execute else-branch from the current state.

While:

$\underline{<S,s> \rightarrow s', <\text{while b do S}, s'> \rightarrow S''}$ - if B[b]s=tt (Do S while true)

<while b do S, s> → s''

$\underline{\qquad\qquad\qquad}$ - if B[b]s=ff

<while b do S, s> → s''

Thus, if the test is false, the execution is stopped. Otherwise , (**S; while b do S**) is executed. This is an example of non-structural semantics – i.e. the evaluation of the entire structure **(while b do S**) depends NOT on the values of its substructures **(b, S**) but also on itself (**while b do S** is submitted to execution again after S is successfully executed).

Therefore, the natural semantics analysis can be viewed as a **partial** function S from States to States. The function is partial since is is not defined for inputs the computation diverges on.

Denote:

S_{nat}: Statement→(State→State)

S_{nat}[S]s = s' if <S,s>→s' otherwise it is undefined.

Examples:

S_{nat}[x:=24]s = s[x=24]

S_{nat}[while true do skip]s = undefined

Several examples:

Let s_0 be the initial state, which assigns zero to all program variables.

<x:=x+1, s_0> → s_0[x:=1] - Assignment statement that increase **x** by **1**, since **x** is 0 initially, in the result state x will be 1.

$\underline{<\text{skip, s0}> \rightarrow \text{s0}}$

<if x = 0 then skip else x:=x+1, s_0> → s_0 – since **x=0** then nothing was changed

$\underline{<\text{skip, s0}> \rightarrow \text{s0}}$

<if x = 1 then skip else x:=x+1, s_0> → s_0 [x:=1]– since **x=0** then x changes

Assume **s = (x=1, y=0)**. The more complicated derivation tree is:

<while (x > 0) do {x:=x-1; y:=y+2}, s> → s22

<{x:=x-1; y:=y+2}, s> →s11 <while (x>0) do {x:=x-1; y:=y+2}, s11> → s22

<x:=x-1, s>→ **s112** **<y:=y+2, s112>→ s11**

The intermediate states are **s11 = (x=0, y=2), s112 = (x=0).**

Now let's define the semantics for **repeat S until b**_(i.e. enter the loop without first checking the condition; b is the exit condition):

<S,s>→ s' if B[b]s = tt – exit if b is true

<S,s>→ s', <repeat S until b, s'>→ s'' - if B[b]s=ff

<repeat S until b, s>→ s''

Two statements S_1 and S_2 are semantically equivalent if for all states s, s' <S_1, s>→ s' iff <S_2, s>>→ s' (i.e. their derivation trees are identical).

The proof idea is to reconstruct the derivation tree for S_2 from the parts of the derivation tree for S_1 and vice versa (maybe using induction on the shape of the derivation tree).

while b do S is semantically equivalent to **if b then (S; while b do S) else skip**

To prove it we first show that:

<while b do S, s> → s'' is divided into <S, s>→**s' followed by < while b do S, s'>→s''** when B[**b]s** is tt – in other words, this is a derivation tree for **S; while b do S. When B[b]s = ff,** we obtain **skip.**

On the other side, according to the **while** semantics, **(S; while b do S)** is the definition for **while** semantics provided **B[b]=tt**; skip is the definition for the **while** semantics provided **B[b]=ff –** so, **while b do S** is reconstructed from **if b then (S; while b do S) else skip**

Theorem: **repeat S until b** is semantically equivalent to **S; while ¬b do S.**

The proof idea is to construct a derivation tree for <**S; while ¬b do S, s>→s'>** using the components of the derivation tree of <**repeat S until b, s**> → s' (and vice versa). May use induction on the shape of the derivation tree.

First we prove the "if" direction:

In the first case (**B[b]s=tt**) we have the tree:

<S, s>→ s'

<repeat S until b, s>→ s'>

Denote T = <S, s>→ s'. Using T as a premise, we can construct a tree:

<S,s>→ s'

<(S; skip), s>→ s'.

Since **skip** is the derivation tree for **while a do S** where **B[a]=ff** (or, equivalently, B[**¬b]s=ff (since B[b]s=tt),** we obtain:

<S,s>→ s' when **B[b]s=tt**

<(S; while ¬b do S), s>→ s'.

If **B[b]ss=ff**, we have **<S,s>→ s', <repeat S until b, s'>→ s'' .**

<repeat S until b, s>→ s''

Denote T_1 = <S,s>→ s', T_2 = **<repeat S until b, s'>→ s''**. By the induction hypothesis (on the shape of the derivation tree) T_2 is equivalent to <**S; while ¬b do S, s'>→ s''.** Then **T_1; T_2 is <S,s>→ s', <S, s'> → s'_1, < while ¬b do S, s'_1>→ s''.** Since **S,s>→ s', <S, s'> → s'_1** can be reduced to **<S, s> → s'_1**, together with **< while ¬b do S, s'_1>→ s''** we reconstruct from **T_1; T_2** the derivation tree for **< while ¬b do S, s>→ s''.**

The same technique is used to reconstruct the derivation tree **<repeat S until b, s>→ s'** from <**S; while ¬b do S, s>→s'>.**

We say language is *deterministic* if the following holds:

"if <S,s>→s1 and <S,s>→s2 then s1=s2".

In other words, in decomposing the execution statement (expression) each time we choose the decomposition rule uniquely.

The **while** language is *a deterministic one.*. It can be proved by induction. First, we show it holds on the axioms, and then we show it holds for each rule.

Structural Operational Semantics

In structural operation semantics the emphasis is on the individual steps of the executions, that is the execution of assignments and tests. The translation relation has the form:

<S,s>=>γ

γ is either of the form

<S',s'> - we choose the leftmost basic execution step, execute it. The remainder is the expression that yet should be computed and the current state.

s' – the execution has completed, s' is the final state.

<S,s> is called *stuck* if there is no γ such that <S,s> $=>^*$ γ.

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.

The definition of => (derivation relation) for While language is given in the following semantics rules:

Axioms:

[ass] <x:=a, s> => s[x:=A[a]s] - Assignment

[skip] <skip,s> => s - Skip statement

Rules:

[comp1] $\frac{\langle S_1,s\rangle \Rightarrow \langle S_1',s'\rangle}{\langle S_1;S_2,s\rangle \Rightarrow \langle S_1';S_2,s\rangle}$ - Composition **(S_1 yet does not terminate)**

[comp2] $\frac{\langle S1,s\rangle \Rightarrow s'}{\langle S_1;S_2,s\rangle \Rightarrow \langle S_2,s\rangle}$ - Composition **(S_1 has ended)**

[if-tt] **<if b then S_1 else S_2,s> => <S_1,s> - if B[b]s=tt**

[if-ff] **<if b then S_1 else S_2,s> => <S_2,s> - if B[b]s=ff**

[while] **<while b do S1,s> => <if b then (S_1;while b do S_1) else skip,s>**

Derivation Sequences for Structural Semantics

A *derivation sequence* of a statement S starting at state s is either:

1. A *finite derivation sequence* $\gamma_0, \gamma_1, \ldots \gamma_k$ such that:
   $\gamma_0 = \langle S,s\rangle$
   $\gamma_i \Rightarrow \gamma_{i+1}$
   γ_k is the stuck configuration or a final state.
2. A *infinite derivation sequence* $\gamma_0, \gamma_1, \gamma_2 \ldots$ such that:
   $\gamma_0 = \langle S,s\rangle$
   $\gamma_i \Rightarrow \gamma_{i+1}$

The Simplest Example

Let s_0 be the initial state {x = 5, y = 7}

Let S = (z:=x; x:=y); y:=z

<(z:=x; x:=y);y:=z, s_0>

=> <x:=y; y:=z, s_0[z=5]>

=> <y:=z, s_0[z=5][x=7]

=> <s0[z=5][x=5][y=5]>

Let's look at another example (y^x)

Assume s = {x = 3, y=4, p=1}

<while (x>0) do (p:=p*y; x:=x-1), s>

=> <p:=p*y; x:=x-1; while (x>1) do (p:=p*y; x:=x-1), s> **- execute while-tt**

=> <while (x>1) do (p:=p*y; x:=x-1), s[p=4, x=2]]> **- execute sequence p:=p*y; x:=x-1**

=> <p:=p*y; x:=x-1; while (x>1) do (p:=p*y; x:=x-1), s[p=4,x=2]> **execute while-tt**

=> <while (x>1) do (p:=p*y; x:=x-1), s[p=16,x=1]> **- execute sequence p:=p*y; x:=x-1**

=> <p:=p*y; x:=x-1; while (x>1) do (p:=p*y; x:=x-1), s[p=16,x=1]> **- execute while-tt**

=> <while (x>1) do (p:=p*y; x:=x-1), s[p=64,x=0]> **- execute while-ff**

=> s[p=64,x=0]

The semantic equivalence notion for the structural semantics is something stronger than in the natural. On the one side, S_1 and S_2 are semantically equivalent if $<S_1, s> \Rightarrow$ ***γ iff** $<S_2, s> \Rightarrow$ ***γ if γ is a terminal or stuck configuration.** This is similar to the natural semantics. **Also, S_1 and S_2** are semantically equivalent **if S_1** may loop from any state s ($<S_1, s>$ is infinite) iff **S_2** may loop from s.

The natural and the structural semantics are equivalent – i.e. either the execution of any statement S terminates in both semantics in the same state or it loops in both semantics.

The equivalence is proved as follows. Assuming we have <S, s> □ s' we construct a derivation sequence $<S_1, s> \Rightarrow$ ***s' and vice versa.** We do this for each atomic case (skip and assignment). For each rule, we do the following:

1) Use the appropriate semantic rule to decompose the original statement tree into subtrees
2) By the induction hypothesis on the shape of the derivation tree (sequence), assume that all is proven for subtrees
3) Combine the subtrees

For example, assume $<S_1; S_2, s>$□ s''. Decomposing the rule in the natural semantics, obtain the subtrees $<S_1, s>$□ s' and $<S_2, s'>$□ s''. By the induction hypothesis, assume $<S_1, s> \Rightarrow$ ***s' and** $<S_2, s'> \Rightarrow$ ***s''.** This is combined into **<S1; S2, s>** $\Rightarrow$ ***s'** .

<Bibliography>

1. Hanne Riis Nielson, Flemming Nielson. Semantics with Applications – A Formal Introduction. The Revised Edition (John Wiley&Sons) July, 1999