Advanced Topics in Programming Languages: Operational Semantics

Idan Berkovits

15/11/2017

Contents

1 PL Semantics

When describing a PL of some kind, we usually separate between the *syntax* and the *semantics* of the language. The syntax of a programming language describes the structure of the grammar of the language. It defines, usually using a context-free grammar, which strings stand for a valid sentence or program in the language. For example, "use semi-colon to separate statements" is a C syntax rule. It describes how to interpret a string as a language statement.

The semantics of a programming language is used to describe the *meaning* of the a program in the language. This means defining the types and behavior of statements in the language, what is the expected output of the statements and what is considered a run-time error. For example, $foo(0)$; is syntactically valid, but the C semantics says it is a valid statement only if foo is a predefined function with one argument of some type that can assigned from the value 0, and the meaning of this statement (call foo with 0 as the value of the single argument) is defined in the language semantics. Formal semantics definition benefits the any user of the programming language:

1. In some languages the semantics can be described using a formal mathematical definition. In those cases it is possible to use that definition to prove qualities of programs without actually running them, like the program correctness, or the equivalence of two programs.

2. A language that is hard to define is usually a language in which programmers will have troubles implementing their programs without avoiding design mistakes, which means the language is hard to use.

3. Most languages require some kind of interpreter or compiler in order to run programs. A formal semantics definition allows us to prove the correctness of the compiler or interpreter, which is crucial for a programmer to trust the language he or she uses.

4. Formal semantics definition helps in the development process with techniques used in software engineering.

1.1 Desired Quantities of PL Semantics

- *Tractable* - PL semantics should be as simple as possible without losing the ability to express behavior accurately.
- *Abstract* - Just enough details to efficiently describe thee logic meaning of the language usage without irrelevant details.
- *Computational* - An accurate abstraction of the language run-time behavior.
- *Compositional* - The meaning of compound language construct is defined using the meaning of sub-constructs. This makes it easier to prove properties of programs (usually by induction).

1.2 Types of Formal Semantics

1. *Operational Semantics* [Plotkin] (elaborated example is given below) - Describes the meaning of a language by describing its execution and procedures. Operational semantics are classified in two categories: *structural operational semantics* which describes how each individual step of computation is preformed in some computer-based system, and *natural semantics* which describes how the overall execution is preformed. For example, languages like ML or Haskell are easier to describe using natural semantics rather then structural semantics.
2. *Denotational Semantics* [Strachey, Scott] - Formalized the meaning of a programming language by constructing mathematical objects (called *denotations* or *domains*) that represent what programs do.
3. *Axiomatic semantics* - Defines the meaning of a command be describing its effect on logical statements evaluation based on the program state (for example, whether a variable value is odd or even).

Programming languages can be generally described in any of those semantics approach, each approach complements the others, and used to discuss a different property of a program. Consider for example two different programs written in C: *"int x; x = 3; print(x);"* and *"print(3);"*. Using operational semantics they are obviously different because the performance of each program is different, but using the denotational approach they are equivalent as their output is not different.

2 The IMP Language

This section described a simple imperative language (*IMP*). The syntax of the language is simple and defined compositionally. This language will be used to demonstrate the use of different types of semantics.

2.1 The IMP Syntax

The IMP syntax contains the sets listed below:

- the set of numbers $\boldsymbol{N}$, consisting of every integer numbers (positive, negative and zero).
- truth values $\boldsymbol{T} = \{\boldsymbol{true}, \boldsymbol{false}\}$
- locations $\boldsymbol{Loc}$, contains the set of possible variables (alphabet strings of any length)
- arithmetic expressions $\boldsymbol{Aexp}$
- boolean expressions $\boldsymbol{Bexp}$
- commands $\boldsymbol{Com}$

The definitions of the first three sets is given above. As for the other three, they are defined recursively and intuitively:

$$\boldsymbol{Aexp} \ni a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$$

$$\boldsymbol{Bexp} \ni b ::= \boldsymbol{true} \mid \boldsymbol{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b_0 \mid b_0 \wedge b_1 \mid b_0 \vee b_1$$

$$\boldsymbol{Com} \ni c ::= \boldsymbol{skip} \mid X := a_0 \mid c_0 c_1 \mid \boldsymbol{if}\, b_0\, \boldsymbol{then}\, c_0\, \boldsymbol{else}\, c_1 \mid \boldsymbol{while}\, b_0\, \boldsymbol{do}\, c_0$$

where $n \in \boldsymbol{N}$, $X \in \boldsymbol{Loc}$, $a_0, a_1 \in \boldsymbol{Aexp}$, $b_0, b_1 \in \boldsymbol{Bexp}$ and $c_0, c_1 \in \boldsymbol{Com}$. This syntax description tells us how to build strings that will be syntactically valid, but not how to determine which way the string was built. For example, $2 + 3 \times 4 - 5$ is a valid arithmetic expression, but can be generated in various ways: $(2 + (3 \times 4)) - 5$, $((2 + 3) \times 4) - 5$, $(2 + 3) \times (4 - 5)$ etc. We will use brackets so cases like those will not stay ambivalent.

Another delicate issue is that these rules do not describe how an arithmetic expression is evaluated, which means $5+3$ is **not** the same arithmetic expression as $3+5$ or 8.

2.2 Program Examples

The language semantics was not defined yet, but using the intuitive meaning of the statements we can look at some simple examples of IMP programs. Obviously, IMP is Turing complete.

2.2.1 Factorial

```
Y:=1;
while !(X=1) do
        Y:=y*X;
        x:=X-1;
```

2.2.2 Greatest common divisor

```
while !(X=Y) do
        if Y<=X then
        X:=X–Y
    else
        Y:Y–X
```

2.3 The IMP Semantics

The semantics pf the IMP language is based on the definition of a *state* $\sigma \in \Sigma$ (the set of all states). A state σ is a function from the set of locations to the set of numbers - $\sigma : \boldsymbol{Loc} \to \boldsymbol{N}$. The state describes a machine snapshot in the sense that for each location (or variable) X and state σ, $\sigma(X)$ is the value, or contents, of location X. σ_0 is marked to be the initial state, a special state in which for every $X \in \boldsymbol{Loc}$, $\sigma(X) = 0$

2.3.1 Evaluation of arithmetic expressions

The evaluation of an arithmetic expression $a \in \boldsymbol{Aexp}$ requires a state $\sigma \in \Sigma$ and is represented by the pair $\langle a, \sigma \rangle$. We define an evaluation relation between such pairs and numbers $\langle a, \sigma \rangle \to n$, which means an expression a in state σevaluates to n. The rules that

define how an expression is evaluated in a state $\sigma$are give from the syntax rules that define how an arithmetic expression is built:

- $\langle n, \sigma \rangle \to n$
- $\langle X, \sigma \rangle \to \sigma(X)$

The recursive rules are quite trickier. Say we are to evaluate the expression $a = a_0 + a_1$ in some state σ. It is automatic to suggest evaluating the expressions a_0, then evaluating the expression a_1, and define the evaluation of a in σ to be the sum of those two evaluations. The problem is that it would imply the *way* an expressions should be evaluated. To fix that we define a relation $\frac{A}{B}$ that means *assuming* A then B. When A is empty, we can call B an axiom. So a more precise way of defining the evaluation of a in σ is the following:

- $\overline{\langle n, \sigma \rangle \to n}$
- $\overline{\langle X, \sigma \rangle \to \sigma(X)}$
- $\frac{\langle a_0, \sigma \rangle \to n_0 \; \langle a_1, \sigma \rangle \to n_1}{\langle a_0 + a_1, \sigma \rangle \to n_0 + n_1}$
- $\frac{\langle a_0, \sigma \rangle \to n_0 \; \langle a_1, \sigma \rangle \to n_1}{\langle a_0 - a_1, \sigma \rangle \to n_0 - n_1}$
- $\frac{\langle a_0, \sigma \rangle \to n_0 \; \langle a_1, \sigma \rangle \to n_1}{\langle a_0 \times a_1, \sigma \rangle \to n_0 \times n_1}$

Notice that any expression can be replaced in a_0 and a_1, and any number can be replaces in n_0 and n_1. That means this is a valid rule:

$$\frac{\langle 2, \sigma \rangle \to 2 \; \langle 3, \sigma \rangle \to 3}{\langle 2 \times 3, \sigma \rangle \to 6}$$

But this rule is also valid:

$$\frac{\langle 2, \sigma \rangle \to 3 \; \langle 3, \sigma \rangle \to 4}{\langle 2 \times 3, \sigma \rangle \to 12}$$

though the premises can never be derived.

Consider the next example of evaluating an arithmetic expression: $a \equiv (Init{+}5){+}(7{+}9)$ and the initial state σ_0:

$$\frac{\frac{\langle Init, \sigma_0 \rangle \to 0 \; \langle 5, \sigma_0 \rangle \to 5}{\langle Init{+}5, \sigma_0 \rangle \to 5} \; \frac{\langle 7, \sigma_0 \rangle \to 7 \; \langle 9, \sigma_0 \rangle \to 9}{\langle 7{+}9, \sigma_0 \rangle \to 16}}{\langle (Init + 5) + (7 + 9), \sigma \rangle \to 21}$$

This structure is called a *derivation tree* and is built using the recursive instances of sub expressions of the arithmetic expression at the bottom. In general, we would say $\langle a, \sigma \rangle \to n$ iff there exists a derivation tree with $\langle a, \sigma \rangle \to n$ at the bottom and all nodes at the top are axioms.

Notice that the evaluation of an arithmetic expression cannot change the actual state σ, unlike evaluating expressions in C, for example. That is why it does not matter which branch of the derivation tree you would take in order to compute the evaluation of an arithmetic expression a.

Now we can actually comparing arithmetic expressions. We say that two expressions a_0 and a_1 are equivalent ($a_0 \sim a_1$) iff

$$\forall n \in \boldsymbol{N}\, \forall \sigma \in \Sigma\ \langle a_0, \sigma \rangle \to n \Leftrightarrow \langle a_1, \sigma \rangle \to n$$

Which means two arithmetic expressions are equivalent if the evaluate to the same value in all states. Using that relation we can now way that $5 + 3 \sim 3 + 5 \sim 8$

2.3.2 Evaluation of boolean expressions

Similar to the previous section, we now define how to evaluate boolean expressions to truth values (**true**, **false**) with the following rules:

- $\dfrac{}{\langle \boldsymbol{true}, \sigma \rangle \to \boldsymbol{true}}$
- $\dfrac{}{\langle \boldsymbol{false}, \sigma \rangle \to \boldsymbol{false}}$
- $\dfrac{\langle a_0, \sigma \rangle \to n \ \langle a_1, \sigma \rangle \to m}{\langle a_0 = a_1, \sigma \rangle \to \boldsymbol{true}}$ if $n = m$
- $\dfrac{\langle a_0, \sigma \rangle \to n \ \langle a_1, \sigma \rangle \to m}{\langle a_0 = a_1, \sigma \rangle \to \boldsymbol{false}}$ if $n \neq m$
- $\dfrac{\langle a_0, \sigma \rangle \to n \ \langle a_1, \sigma \rangle \to m}{\langle a_0 \leq a_1, \sigma \rangle \to \boldsymbol{true}}$ if $n \leq m$
- $\dfrac{\langle a_0, \sigma \rangle \to n \ \langle a_1, \sigma \rangle \to m}{\langle a_0 \leq a_1, \sigma \rangle \to \boldsymbol{false}}$ if $n > m$
- $\dfrac{\langle b, \sigma \rangle \to \boldsymbol{true}}{\langle \neg b, \sigma \rangle \to \boldsymbol{false}}$, $\dfrac{\langle b, \sigma \rangle \to \boldsymbol{false}}{\langle \neg b, \sigma \rangle \to \boldsymbol{true}}$
- $\dfrac{\langle b_0, \sigma \rangle \to t_0 \ \langle b_1, \sigma \rangle \to t_1}{\langle b_0 \wedge b_1, \sigma \rangle \to t}$ where $t = \boldsymbol{true}$ when $t_0 = t_1 = \boldsymbol{true}$ and $t = \boldsymbol{false}$ otherwise
- $\dfrac{\langle b_0, \sigma \rangle \to t_0 \ \langle b_1, \sigma \rangle \to t_1}{\langle b_0 \vee b_1, \sigma \rangle \to t}$ where $t = \boldsymbol{false}$ when $t_0 = t_1 = \boldsymbol{false}$ and $t = \boldsymbol{true}$ otherwise

and we define equivalence the same way:

$$b_0 \sim b_1 \, \boldsymbol{iff} \, \forall t \in \boldsymbol{T} \, \forall \sigma \in \Sigma \; \langle b_0, \sigma \rangle \to t \Leftrightarrow \langle b_1, \sigma \rangle \to t$$

Notice that according to those rules, in order to evaluate $b_0 \wedge b_1$ we must evaluate b_0 and b_1 first. To make the semantics simpler we can add the following rules:

- $\frac{\langle b_0,\sigma\rangle \to \boldsymbol{false}}{\langle b_0 \wedge b_1,\sigma\rangle \to \boldsymbol{false}}$, $\frac{\langle b_1,\sigma\rangle \to \boldsymbol{false}}{\langle b_0 \wedge b_1,\sigma\rangle \to \boldsymbol{false}}$
- $\frac{\langle b_0,\sigma\rangle \to \boldsymbol{true}}{\langle b_0 \vee b_1,\sigma\rangle \to \boldsymbol{true}}$, $\frac{\langle b_1,\sigma\rangle \to \boldsymbol{true}}{\langle b_0 \vee b_1,\sigma\rangle \to \boldsymbol{true}}$

Added to the list above, those rules keep the semantics complete, but let's us build smaller derivation trees.

2.3.3 Execution of commands

Unlike expressions, the role of executing a command is to *change the state*. Therefore we shall write for a command $c \in \boldsymbol{Com}$ and states $\sigma, \sigma' \in \Sigma$, $\langle c, \sigma \rangle \to \sigma'$ if the command c executes from a state σ to a state σ'. This means the *full* execution of command c in the state σ terminates on a state σ'.

Notation: Let $\sigma \in \Sigma$, $m \in \boldsymbol{N}$, $X \in \boldsymbol{Loc}$. The state $\sigma[m/X] \in \Sigma$ is defined to be:

$$\sigma[m/X](Y) = \begin{cases} m & if \, Y = X \\ \sigma(Y) & otherwise \end{cases}$$

Now we can define the derivation rules for executing commands in IMP:

- $\overline{\langle \boldsymbol{skip}, \sigma \rangle \to \sigma}$
- $\frac{\langle a,\sigma\rangle \to m}{\langle X := a,\sigma\rangle \to \sigma[m/X]}$
- $\frac{\langle c_0,\sigma\rangle \to \sigma'' \; \langle c_1,\sigma''\rangle \to \sigma'}{\langle c_o ; c_1,\sigma\rangle \to \sigma'}$
- $\frac{\langle b,\sigma\rangle \to \boldsymbol{true} \, \langle c_0,\sigma\rangle \to \sigma'}{\langle \boldsymbol{if} \, b_0 \, \boldsymbol{then} \, c_0 \, \boldsymbol{else} \, c_1,\sigma\rangle \to \sigma'}$, $\frac{\langle b,\sigma\rangle \to \boldsymbol{false} \, \langle c_1,\sigma\rangle \to \sigma'}{\langle \boldsymbol{if} \, b_0 \, \boldsymbol{then} \, c_0 \, \boldsymbol{else} \, c_1,\sigma\rangle \to \sigma'}$
- $\frac{\langle b,\sigma\rangle \to \boldsymbol{false}}{\langle \boldsymbol{while} \, b_0 \, \boldsymbol{do} \, c_0,\sigma\rangle \to \sigma}$, $\frac{\langle b,\sigma\rangle \to \boldsymbol{true} \, \langle c,\sigma\rangle \to \sigma'' \, \langle \boldsymbol{while} \, b_0 \, \boldsymbol{do} \, c_0,\sigma''\rangle \to \sigma'}{\langle \boldsymbol{while} \, b_0 \, \boldsymbol{do} \, c_0,\sigma\rangle \to \sigma'}$

and the equivalence relation is defined similarly:

$$c_0 \sim c_1 \, \boldsymbol{iff} \, \forall \sigma_0, \sigma_1 \in \Sigma \; \langle c_0, \sigma \rangle \to \sigma' \Leftrightarrow \langle c_1, \sigma \rangle \to \sigma'$$

An important point to notice is that the conditional rules do not require executing both of the commands in the statement, but only only the one that would effect the state after the condition executing.

What happens when a command never terminates? For example, consider $\omega = \textbf{\textit{while}}\, \textbf{\textit{true}}\, \textbf{\textit{do}}\, \textbf{\textit{skip}}$. The answer is that formally there are no states σ, σ' such that $\langle \omega, \sigma \rangle \to \sigma'$. This is because if there were two states like that, the last derivation rule in the tree would have to be the second rule (because the boolean expression is $\textbf{\textit{true}}$), which means the tree would have to be infinite.

Using execution of commands and evaluation of expressions we can now execute IMP programs from any initial state in Σ, assuming the execution of the program on σterminates.

2.4 Using the IMP semantics

The definition of the IMP syntax and semantics is hardly compositional, which suggests a proof technique for providing properties of the operational semantics of IMP. For example, consider the following proposition:

Proposition 2.8 (p. 21) Let $\omega = \textbf{\textit{while}}\, b\, \textbf{\textit{do}}\, c$, then $\omega \sim \textbf{\textit{if}}\, b\, \textbf{\textit{then}}\, c; \omega\, \textbf{\textit{else}}\, \textbf{\textit{skip}}$

The proof of the proposition is fairly straight, and does not require any induction steps. One needs to show that the existence of a derivation tree of ω concludes the existence of such tree of $\textbf{\textit{if}}\, b\, \textbf{\textit{then}}\, c; \omega\, \textbf{\textit{else}}\, \textbf{\textit{skip}}$ on some arbitrary states $\sigma_0, \sigma_1 \in \Sigma$. This is done by considering the various possible forms of derivations, thus avoiding actual executing of the program.

2.5 Extending the IMP semantics

2.5.1 Program abortion and non-determinism

The semantics can be easily extended to include more properties. For example, an $\textbf{\textit{abort}}$ axiomatic command can be added to the definition of commands. Because the execution rules do not specify how to handle $\textbf{\textit{abort}}$ that will mean hitting an $\textbf{\textit{abort}}$ command will result in no legal state which means the program is not valid. The thing is that command can be conditioned, or in some loop, which will mean a program validity is dependent on the initial state.

Another possible extension is adding non-determinism feature - say w add the command building rule $c_0|c_1$ and the derivation rules:

$$\frac{\langle c_0, \sigma\rangle \to \sigma'}{\langle c_o | c_1, \sigma\rangle \to \sigma'}$$

$$\frac{\langle c_1, \sigma\rangle \to \sigma'}{\langle c_o | c_1, \sigma\rangle \to \sigma'}$$

That way in order to execute $c_0|c_1$ one needs to execute *either* c_0 or c_1. This means one of the commands might not terminate (infinite loop or an ***abort*** command) but the command $c_o|c_1$ will still terminate.

2.5.2 Parallel execution

An extension of parallelism is not possible using the natural semantics defined here, because we cannot properly define the atomic operation. In order to do that we need to use *small-step semantics*: in small-step semantics we replace the relation $\langle a, \sigma\rangle \to n$ with the relation $\langle a, \sigma\rangle \to_1 \langle a', \sigma'\rangle$, which means one step in the evaluation of a in state σ yields a' in state σ'. Using this notation, we can set derivation rules that will allow us to evaluate arithmetic expression from left to right:

- $\frac{\langle a_0,\sigma\rangle \to \langle a_0',\sigma\rangle}{\langle a_0+a_1,\sigma\rangle \to \langle a_0'+a_1,\sigma\rangle}$
- $\frac{\langle a_1,\sigma\rangle \to \langle a_1',\sigma\rangle}{\langle n+a_1,\sigma\rangle \to \langle n+a_1',\sigma\rangle}$
- $\overline{\langle n+m,\sigma\rangle \to \langle p,\sigma\rangle}$ where $p = n + m$

using those rule, the derivation tree of arithmetic expression show how the expression is evaluated step by step from left to right. Command execution relation will change as well to be $\langle c, \sigma\rangle \to_1 \langle c', \sigma'\rangle$, which describes one step in executing the command c. Notice that the way we will define the rules is very important when we use them to extend the IMP semantics to include parallel execution: which rule do we take into consideration:

$$\frac{\langle b, \sigma\rangle \to_1 \langle \boldsymbol{true}, \sigma'\rangle}{\langle \boldsymbol{if}\, b\, \boldsymbol{then}\, c_0\, \boldsymbol{else}\, c_1, \sigma\rangle \to_1 \langle c_0, \sigma'\rangle}$$

or

$$\frac{\langle b, \sigma\rangle \to_1 \langle \boldsymbol{true}, \sigma'\rangle}{\langle \boldsymbol{if}\, b\, \boldsymbol{then}\, c_0\, \boldsymbol{else}\, c_1, \sigma\rangle \to_1 \langle \boldsymbol{if}\, \boldsymbol{true}\, \boldsymbol{then}\, c_0\, \boldsymbol{else}\, c_1, \sigma'\rangle}$$

As we recall, the single step semantics define which execution steps are atomic in context of parallel execution.

2.5.3 Arithmetic evaluation side effects

As mentioned above, evaluating an arithmetic expression does not effect the state in which the expression is evaluated, unlike C programs. The semantics could be changed to include those kind of effect, with not so minor changes. First, the relation $\langle a, \sigma\rangle \to n$ will now be changed to $\langle a, \sigma\rangle \to \langle n, \sigma\rangle$. Also we would add a new form of arithmetic expression, in the form of "c ***resultis*** a" where $c \in \boldsymbol{Com}$ and a is an arithmetic expression. Its deviation rule would be:

$$\frac{\langle c, \sigma\rangle \to \sigma'' \ \langle a, \sigma''\rangle \to \langle n, \sigma'\rangle}{\langle c\, \boldsymbol{resultis}\, a, \sigma\rangle \to \langle n, \sigma'\rangle}$$

Adding the changed state to the arithmetic evaluation changes the property that was discussed before - the order of evaluation obviously matters now, and is determined by the order of states changed, like in the rule of command concatenation.

3 References

1. Winskel, Glynn. "2." The Formal Semantics of Programming Languages: an Introduction, MIT Press, 2001, pp. 11–26.

2. Plotkin, GD, "A structural approach to operational semantics ." Lecture Notes, University of Aarhus, Denemark, 1981 (printed at 1991)

3. Sagiv, Mooly, "Operational Semantics", http://www.cs.tau.ac.il/~msagiv/courses/apl17/os.pdf [Online]

4 Appendix

4.1 Haskell implementation of the IMP semantics

```
data Boolean = TT | FF

or(TT,TT) = TT;or(TT,FF) = TT;or(FF,TT) = TT;or(FF,FF) = FF;
not(TT) = FF;not(FF) = TT;
and(b1, b2) = not(or(not(b1), not(b2)))
cond(TT,tval,fval) = tval;cond(FF,tval,fval) = fval
from_bool b = if b then TT else FF

:{
data Aexp = Number(Int) |
            Loc(String) |
            Plus(Aexp, Aexp) |
            Minus(Aexp, Aexp) |
            Mul(Aexp, Aexp)
:}

:{
data Bexp = Val(Boolean) |
            Eq(Aexp, Aexp) |
            Leq(Aexp, Aexp) |
            Not(Bexp) |
            Land(Bexp, Bexp) |
            Lor(Bexp, Bexp)
:}

:{
data Comm = Skip |
            Ass(String, Aexp) |
            Concat(Comm, Comm) |
            If(Bexp, Comm, Comm) |
```

```
            Loop(Bexp, Comm)
:}

:{
factorial_program =
    Concat(
        Ass("Y", Number(1)),
        Loop(Not(Leq(Loc("X"), Number(1))),
            Concat(
                Ass("Y", Mul(Loc("Y"), Loc("X"))),
                Ass("X", Minus(Loc("X"), Number(1)))
            )
        )
    )
:}

:{
gcd_program =
    Loop(Not(Eq(Loc("X"), Loc("Y"))),
        If(Not(Leq(Loc("X"), Loc("Y"))),
            Ass("X", Minus(Loc("X"), Loc("Y"))),
            Ass("Y", Minus(Loc("Y"), Loc("X")))
        )
    )

:}

:{
let evalAexp :: Aexp -> (String -> Int) -> Int
    evalAexp (Number n)(s) = n
    evalAexp (Loc var)(s) = s(var)
    evalAexp (Plus(a1, a2))(s) = evalAexp(a1)(s) + evalAexp(a2)(s)
    evalAexp (Minus(a1, a2))(s) = evalAexp(a1)(s) - evalAexp(a2)(s)
    evalAexp (Mul(a1, a2))(s) = evalAexp(a1)(s) * evalAexp(a2)(s)
```

```
:}

:{
let evalBexp :: Bexp -> (String -> Int) -> Boolean
    evalBexp (Val(b))(s) = b
    evalBexp (Eq(a1, a2))(s) = from_bool (evalAexp(a1)(s) == evalAexp(a2)(s))
    evalBexp (Leq(a1, a2))(s) = from_bool (evalAexp(a1)(s) <= evalAexp(a2)(s))
    evalBexp (Not(b1))(s) = not (evalBexp(b1)(s))
    evalBexp (Land(b1, b2))(s) = and (evalBexp(b1)(s),evalBexp(b2)(s))
    evalBexp (Lor(b1, b2))(s) = or (evalBexp(b1)(s),evalBexp(b2)(s))
:}

:{
let execComm :: Comm -> (String -> Int) -> String -> Int
    execComm (Skip)(s)(y) = s(y)
    execComm (Ass(loc, a))(s)(y) = if loc == y then evalAexp(a)(s) else s(y)
    execComm (Concat(c1,c2))(s)(y) = execComm(c2)((execComm(c1)(s)))(y)
    execComm (If(b1, c1, c2))(s)(y) = cond(evalBexp(b1)(s),
                                           execComm(c1), execComm(c2))(s)(y)
    execComm (Loop(b1, c1))(s)(y) = cond(evalBexp(b1)(s),
                              execComm(Loop(b1, c1))(execComm(c1)(s)), s)(y)
:}


state_X_Y (x,y) loc = if loc == "X" then x else if loc == "Y" then y else 0

factorial m = execComm(factorial_program)(state_X_Y(m,0))("Y")
gcd x y = execComm(gcd_program)(state_X_Y(x,y))("X")

:{
test_factorial m =
        putStrLn ("factorial(" ++ show(m) ++ ")=" ++ show(factorial(m)))
:}
```

```
:{
test_gcd x y =
        putStrLn ("gcd(" ++ show(x) ++ ", " ++ show(y) ++ ")=" ++ show(gcd x y))
:}

test_factorial 0
-- factorial(0)=1
test_factorial 1
-- factorial(1)=1
test_factorial 2
-- factorial(2)=2
test_factorial 3
-- factorial(3)=6
test_factorial 4
-- factorial(4)=24
test_factorial 5
-- factorial(5)=120
test_factorial 6
-- factorial(6)=720
test_factorial 7
-- factorial(7)=5040

test_gcd 50 6
-- gcd(50, 6)=2
test_gcd 31 4
-- gcd(31, 4)=1
test_gcd 15 5
-- gcd(15, 5)=5
test_gcd 1 1
-- gcd(1, 1)=1
```

Notice that the lazy property of the Haskell language it ensures the *cond* function work like you would want it to - it only computes the condition branch that is needed. A C function would not do that. Specifically, the way *execComm* is computed on loop values

would be infinite otherwise.

Also, the very nature of the semantics promise cannot be implemented. For example, It is not possible to use both the "or" rule that checks only the first boolean expression *and* the rule that checks only the second term. Another good example to that problem is the inability of implementing the non-determinism extension of the semantics.