
CS 421 – Spring 2007

Lecture Notes Set 30: Natural Semantics

Elsa L. Gunter¹

Transcribed by: Pooja Mathur

Corresponding to Slides: 20-nat-sem (slides 2-30)

Made available: April 11, 2007

Revision 1.0

1 Change Log

1.0 Initial Release.

2 Semantics overview

2.1 Idea

We have been talking about λ -calculus as a way of expressing meaning of functions. Now, we are going to look at another way of doing this and that is through natural semantics. And these ways will turn out to be more useful when it comes to writing compilers and interpreters.

So when we first started talking about λ -calculus, we said that it gave semantics to the syntax, that is, it gave meaning to it. But this so-called meaning can only come from what the expression looks like. Which means it needs to be understood without being able to execute the expression. That, in turn, means we have to do type checking and type inference to make sure we are on the right track.

2.2 Static (slide 2)

For example, you think you know what "+" means. But without more information, (about the language and the expression) you don't know if it's talking about the addition of integers, or floats, or in some language it can refer to the concatenation of strings. $2 + 3$ would equal 5, but "two" + "three" can result in "twothree". So there needs to be some way of making sure everything has the meaning you think it has. So, we have checked to see if something has meaning through lexing, parsing, and type checking. But that is all part of the static analysis of semantics. That is, this is all done before the program runs. This is analysis that you can do based on the syntax of a language and the expressions used in a program.

2.3 Dynamic (slide 3)

But we want to take things a step farther, we want to take into account the meaning of an executing program, the dynamic semantics. This kind of analysis is done by looking at the different actions that take place in a program.

3 Types of Dynamic Semantics (slide 4)

There are operational semantics that describe what a naive machine would do, what result should it give, if given a certain expression. There are mathematical semantics, where you find some function or expression is the coded

¹© 2007, Share and Enjoy

version of some mathematical term or function. There are axiomatic semantics which involve making assumptions before running some program, and because you made some assumptions, you can make following assumptions about the world after running the program. The idea is that different types of semantics are better suited towards different languages. That is because the types of semantics focus on different aspects of programming.

3.1 Operational (slide 5)

In more detail, with operational semantics, we start out with some notion of a virtual machine. Then, from there, you describe how the program should work by describing the how each statement should work. Then we learn what the meaning of the program is based on how the machine state changes during the program's execution.

3.2 Axiomatic (slides 6-7)

In axiomatic semantics, you have a collection of assertions. You have a logical statement that involves variables, where the variables are your program variables. But you end up treating your variables like they are the variables of your logic. You give a statement about them concerning what is true about them before you run your program. It is possible that it could be nothing, that is, you don't know anything about your variables. Then you have your program, and this assertion is saying that, if your program runs and it terminates, then you will know that the conditions about the end of your program will be true afterwards.

So you may have some precondition that x is equal to 5. And maybe your program figures out the factorial. So then your postcondition may be that y should be equal to $5!$. But this is just an assertion. You have to be able to find out if your assertion is true. So there needs to be a way of proving it. And that is what axiomatic proofs are about, giving you the proof steps you need to prove the assertions.

There is another concept, weakest precondition, that comes from the idea that you may not know what you want to be true before the program starts, but you know how you want it to end. So, using these inference rules, you figure out, what was the least you needed to know for you to end up with the conclusion you wanted.

3.3 Denotational (slides 8-9)

Denotational semantics is when you give a mathematical function which maps from syntax/strings to mathematical objects. These mathematical objects can be λ terms, for example. Then we say the meaning of our program is made from pieces. So, the meaning of the `if_then_else`, will be built up out of doing something with the initial boolean expression, then the meaning of the then-clause, the meaning of the else-clause and then will be put together from there.

This is used mainly for proving properties about program, like that one program is equivalent to another.

4 Natural Semantics

4.1 Idea (slide 10)

What we are going to do with natural semantics is give a semantics where you find out how to evaluate an expression all the way to the final value. Whereas other semantics only tell you how to take one step. The rules are going to take one of two forms. There are commands and expressions. (In the mp, we used declarations instead of commands.) So commands will alter the memory (the environment) we are working in. And command rules take the form: $(C, m) \Downarrow m'$. So this is saying that we get a command and a memory and after performing this command, we have the new memory, m' . The expressions will evaluate to values and take the form: $(E, m) \Downarrow v$. This is saying that you have an expression and a memory, and after working out the expression, the result, the output from that is the value v .

4.2 Example language (slide 11)

We will use the following language for the purpose of examples and explanations.

$\text{Id} \in \text{Identifiers}$

By identifiers, we mean variable names.

$N \in \text{Numerals}$

By numerals, we mean, numbers.

$B ::= \text{true} \mid \text{false} \mid B \ \& \ B \mid B \ \text{or} \ B \mid \text{not } B \mid E < E \mid E = E$

We have booleans and boolean expressions.

$E ::= N \mid \text{Id} \mid E + E \mid E * E \mid E - E \mid - E$

Here, we have expressions.

$C ::= \text{skip} \mid C; C \mid \text{Id} ::= E \mid \text{if } B \text{ then } C \text{ else } C \text{ fi} \mid \text{while } B \text{ do } C \text{ od}$

To bracket the if_then_else there is a "fi" at the end. To bracket the while loop, there is an "od" at the end. That way we don't have to worry about when they end.

4.3 Evaluating base cases (slide 12)

To evaluate an identifier, you write $(\text{Id}, m) \Downarrow m(\text{Id})$. So some identifier is in memory m .

To evaluate a numeral, you write $(N, m) \Downarrow N$. This means that there is in memory m , the numeral N evaluates to N .

So you can think of the same thing when looking at booleans. $(\text{true}, m) \Downarrow \text{true}$. That is, in memory m , true evaluates to true. And if you had $(\text{false}, m) \Downarrow \text{false}$. Then, false evaluates to false in memory m as well.

4.4 Evaluating boolean expressions (slide 13)

So, from that, what do you do if you want to evaluate two booleans?

$$\begin{array}{ccc}
 \frac{(\text{B}, m) \Downarrow \text{false}}{(\text{B} \ \& \ \text{B}', m) \Downarrow \text{false}} & \frac{(\text{B}, m) \Downarrow \text{true} \quad (\text{B}', m) \Downarrow b}{(\text{B} \ \& \ \text{B}', m) \Downarrow b} & \frac{(\text{B}, m) \Downarrow \text{true}}{(\text{B} \ \text{or} \ \text{B}', m) \Downarrow \text{true}} \\
 \frac{(\text{B}, m) \Downarrow \text{false} \quad (\text{B}', m) \Downarrow b}{(\text{B} \ \text{or} \ \text{B}', m) \Downarrow b} & \frac{(\text{B}, m) \Downarrow \text{true}}{(\text{not } \text{B}, m) \Downarrow \text{false}} & \frac{(\text{B}, m) \Downarrow \text{false}}{(\text{not } \text{B}, m) \Downarrow \text{true}}
 \end{array}$$

Well, for the first tree above, we are evaluating $B \ \& \ B'$. So, the first thing we do is evaluate B . So that is the part above the line. We evaluate B , and we find out that its false. Since, overall, we are trying to evaluate an $\&$, we can stop here and just send the conclusion of false downwards.

In the next tree, moving right, we, again, are evaluating $B \ \& \ B'$. But here, when we evaluated B above the line, we found that it was true. So we had to move on to B' and evaluate that. Here, we simply say that B' evaluates to some boolean value b , and whatever that value is is sent down the tree, since B evaluated to true.

In the next statement, to the right, we check $B \ \text{or} \ B'$. If we get the B is true we're done. Otherwise, like in the first tree on the second row, if we get a false, we have to evaluate B' and see what that is. Then, whatever that evaluates to, we send down the tree.

Then the last two trees depict how to evaluate not statements. If B is true, then we send false down and vice versa.

Note, that we are using eager evaluation to solve these.

Let's take a minute and look at how you would make sure you evaluated B and B' when trying to evaluate $B \ \& \ B'$. Well, above the line you would have $(\text{B}, m) \Downarrow b' \quad (\text{B}', m) \Downarrow b'' \quad (b' \ \& \ b'') = b$. So you would get true or false for b' . Then you would get true or false for b'' and then you would $\&$ those to get b . Then send that value down the tree.

4.5 Evaluating relational expressions (slide 14)

Here, we will look at how to evaluate relational expressions, like less than and equals to, we, will denote any relational operator, $<$, $=$, as \sim . So, below the line, we want to evaluate the whole expression $(E \ \sim \ E', m) \Downarrow b$. Then above the line in the tree we would have to evaluate each expression on its own. So first we would have $(E, m) \Downarrow U$. That is, evaluating the expression E with memory m , gives us the value U . And then $(E', m) \Downarrow V$ means evaluating the expression E' with memory m , gives us the value V . And then after that, we have $U \ \sim \ V = b$. So we just have two values and we figure out how they compare to each other. Doing that results in a boolean and that is what we send down the tree.

4.6 Evaluating Arithmetic expressions (slide 15)

Here we do something extremely similar. Let's denote the arithmetic operation, $+$, $-$, $*$, $/$, being performed as "op". So below the line in the tree, we have $(E \text{ op } E', m) \Downarrow N$. Above the line we have $(E, m) \Downarrow U$ and then $(E', m) \Downarrow V$. And then we take the two values we got from evaluating those expressions and we get an actual numeral from them, $U \text{ op } V = N$. That N is what is sent down the tree.

4.7 Evaluating Commands (slides 16-18)

There is this skip command that just means, if you see this command with memory m , return memory m . It means we are done with what ever we were doing. It looks like $(\text{skip}, m) \Downarrow m$.

Then there is assignment, giving an identifier an expression to go with it. Below the line, if we wanted to say $(\text{Id} := E, m) \Downarrow$ "to be determined"... Then we take the expression E , we would evaluate it above the line, and that would return a value V . Then when we sent that V down and create a new memory with I assigned to that V . So $m[\text{Id} \rightarrow V]$ would replace the "to be determined".

If we had a sequence of commands, for example $(C; C', m)$ was below the line. To evaluate it, we would first run the command C , and that would return a memory, m' . Then we would take that memory, since it is affect the next command that runs and run C' with m' and that will return an m'' . That is the final memory we send down the tree, that is what goes to the right of the \Downarrow . As clarification, if you don't understand why m' should be run with C' , think of this. If you had the expression $x = 5$. And then had a command that changed x to 7 in memory, then for the next command you would want to run it with $x = 7$ in memory. Otherwise you would never make any progress when you have a sequence of commands, because the memory is never being updated properly.

In the `if_then_else` command. Below the line, you may have something that may look like, $(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m)$. The first thing you have to do is evaluate the boolean expression. So you evaluate (B, m) . If you get true returned, then you know you want to run the first command C and so that's what you do, you run (C, m) and that returns and altered memory m' . And that is what is sent down the tree. On the other hand, if you get a false return from evaluating the boolean expression, you run the second command, and that looks like $(C', m) \Downarrow m'$. And this m' (which could be, and most likely is, different from the m' we had if we evaluated to true) is the m' we send down the tree. The m' is what goes to the right of the \Downarrow .

For while loops, below the line we will have $(\text{while } B \text{ do } C \text{ od}, m)$. Again, first off, we evaluate the boolean expression. So if we get false from (B, m) , then the while loop doesn't run and we send the original memory m back. If we get true back though, this will get a little messy. We run the command C once, (C, m) and that returns a new memory m' . After that we write the while loop expression again, but here we send in a the new memory, m' . That looks like $(\text{while } B \text{ do } C \text{ od}, m')$ after running this, we will eventually get back an m'' , which is what we send down the tree. So that is where the messy part comes in though. The rule has a call to itself in it and because of that, we end up with the possibility of non-termination. So if you had $(\text{while true do } x = x \dots)$ that loop would run forever. That command, any program that had that command in it would mean nothing we these semantics. And even if it does terminate...if the while loop was supposed to run for 100 iterations...them you suddenly have a tree with branch of length 100. And that branch at the very end, sends it's m'' down, one branch, and then the 99^{th} branch send m'' to the 98^{th} branch, and then the 98^{th} branch... and finally the original while command gets the m'' .

4.8 Example Evaluation (slides 19-30)

So you want to evaluate $(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi}, \{x \rightarrow 7\}) \Downarrow ?$.

The first thing you would want to do is evaluate the boolean expression to see if you should run C (the $y := 2 + 3$ part) or C' (the $y := 3 + 4$ part). So the boolean expression is $(x > 5)$, your memory is $\{x \rightarrow 7\}$. Next you have to evaluate x , so on the next level of the tree you do $(x, \{x \rightarrow 7\})$. Well, that evaluates to 7. So your value to the right of the \Downarrow is 7. The you evaluate the 5, and remember numerals map to themselves, so you have 5. And the you evaluate the relation and get $7 > 5$, that equals true, so you send that down the tree and now you know to evaluate the first command. At this point, your tree should look like:

$$\frac{\frac{(x, \{x \rightarrow 7\}) \Downarrow 7 \quad (5, \{x \rightarrow 7\}) \Downarrow 5 \quad (7 < 5 = \text{true})}{(x > 5, \{x \rightarrow 7\}) \Downarrow \text{true}}}{(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \rightarrow 7\}) \Downarrow ?}$$

So now we go and do the evaluation for C. We have $(y := 2 + 3, \{x \rightarrow 7\})$. To move farther with assignment, we have to evaluate the expression $2 + 3$. So we go up a level in the tree and do that. To evaluate the mathematical expression itself, we need to evaluate each side of the expression. So we go up another level. Now we evaluate the 2, and that returns a 2. Then we evaluate the 3, and that returns a 3. Now we can add the 2 + 3, and get 5. So we send 5 down the tree until we get to the assignment level. Here we add $y \rightarrow 5$ to the memory, and we send that memory down. And that is our final answer. So our final tree looks like:

$$\frac{\frac{(x, \{x \rightarrow 7\}) \Downarrow 7 \quad (5, \{x \rightarrow 7\}) \Downarrow 5 \quad (7 < 5 = \text{true})}{(x > 5, \{x \rightarrow 7\}) \Downarrow \text{true}} \quad \frac{\frac{(2, \{x \rightarrow 7\}) \Downarrow 2 \quad (3, \{x \rightarrow 7\}) \Downarrow 3 \quad (2 + 3 = 5)}{(2 + 3, \{x \rightarrow 7\}) \Downarrow 5}}{(y := 2 + 3, \{x \rightarrow 7\}) \Downarrow \{y \rightarrow 5, x \rightarrow 7\}}}{(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \rightarrow 7\}) \Downarrow \{y \rightarrow 5, x \rightarrow 7\}}$$