
CS 421 – Spring 2007

Lecture Notes Set 31:

Natural Semantics Continued

Transition Semantics

Elsa L. Gunter¹

Transcribed by: Pooja Mathur

Corresponding to Slides: 20-nat-sem (slides 31-end) 20a-trans-sem (slides 1-10)
Made available: April 13, 2007
Revision 1.0

1 Change Log

1.0 Initial Release.

2 Last notes on Natural Semantics

2.1 Commands - Continued (slide 31)

If you had the Let-in command. Under the line, you might have something that looks like (let $Id = E$ in C , m). Then, the first thing you want to do is evaluate your expression. So, above the line you take the E and evaluate it, (E, m) and that returns some value v . Next, you want run the command C , but first you need to change your memory and make Id point to this new value v . So you run C with the changed memory, $(C, m[Id \rightarrow v])$. Running that command is going to change your original memory to a m' . And that is what you return to below the line. You don't add the $Id \rightarrow v$ to the final memory since that assignment is only in scope during the time the command, C , runs.

But there is an issue. We need to know what to do when inside C , we make an assignment to the identifier we just used in the let statement. So if we had:

$$\text{let } x = 5 \text{ in } (x := 7; y := x)$$

Then we would evaluate the $x = 5$ and all we would have to do is evaluate:

$$(x := 7; y := x) [x \rightarrow 5]$$

...but what do we do, do we give y the 5 or the 7? Well, what would happen is we would see that we need to reassign x , so that is what we would do. Then we would go to the next command and that is to give x to y . So we assign 7 to y . Then when we are done. Since the x 's were local and only in the scope of the let-in, the x 's disappear from memory. If x was in memory to begin with, the after the commands are done and we are about to send $y := 7$ back to down the tree, we will give x 's original assignment back to it, what ever that may be.

2.2 Let-in Example (slides 32-33)

Let's say we were trying to evaluate (let $x = 5$ in $(x := x + 3)$, $\{x \rightarrow 17\}$) (that's level 0 of our tree). The on the next level, we would first evaluate the expression in the let part. All it is, is the number 5 and if you recall from last time, numbers just evaluate to themselves. So that is what we get. And then what we do is change memory and add $x \rightarrow 5$ in it. Here, it replaces the current assignment of x . Then all that is left is to evaluate the $x := x + 3$ part. This is an assignment, and the assignment rule says to evaluate the expression first.

¹© 2007, Share and Enjoy

So onto level 2 of the tree. Here we need to evaluate $x + 3$. That is an arithmetic expression, and that requires us to evaluate each side of the plus, so we do. So on the 3rd level of the tree, we look at the x , and that maps to 5. Then we look at the 3 and that maps to 3. Now here we can finally perform the operation of addition. $3 + 5 = 8$, so we return 8 to level 2 of the tree. Level 2 returns the completed assignment of $x \rightarrow 8$ to level 1. And then level 1 is going to return the assignment to level 0. However, that gets thrown away, because all that evaluation was local in scope, so we end up returning $x \rightarrow 17$ as our final memory.

2.3 One last case

Now, we have looked at the case when you create a local assignment in memory, when there was no assignment before. Then we looked at when there was an assignment and we overwrote it with a local one. Now, what happens when there was no local assignment and there wasn't any previous assignment. Let's say you have $(x := x + 3, \{\})$ in level 0 of your tree. Level 1 will try to evaluate your expression, the $x + 3$ part. Then you will try to evaluate the parts of the addition. First you will look at the x ... well to evaluate the x , you need to look in memory... but there isn't anything in memory about the x ...so your stuck. You can evaluate the 3 of course. 3 just maps to 3. But you're still stuck on the x ... so this is one of those cases where this command means nothing with the semantics we are currently using.

2.4 Comments on scope (slide 34)

Part of the point of all this is to make sure you are being careful when using the same variables and such continuously in your programs, but for different reasons. You need to make sure these paradigms work well together, and when they do, it is often referred to as being orthogonal. But there are times when they don't work well together, and this usual stems from incorrect use of local and global variables. So this starts to add traps for programmers. But one thing that semantics do is try to help you point out where it is that your language may lead to complications, so you know what to look out for.

2.5 Interpretation vs. Compiler (slides 35-37)

Another use for semantics is, is to help you write something that will actually do what the language is supposed to. What we mean is, writing compilers or interpreters. The difference between to two is gray since it is often the case that parts of each are involved in doing the other's job.

A compiler is a program that will take in something in one language and it will produce an output that is in another language. So, you can basically think of it as a translator. It is not necessary that you have a particular machine language as your target language during compilation. Compilers, for example, only need to look at a loop once.

An interpreter is a program, written in one language, that executes commands that happen in another language. It interprets them. It says what to do with them. So to write an interpreter, you start with doing the lexing and the parsing. So we extract the literals and the abstract syntax trees. The we build up the interpreter by saying how to build up small pieces first, like the literals, and then we move up to variables. After that we can start working on expressions. And after that is complete, we can finish with commands/declarations. So your interpreter tells you how to evaluate each of these things in terms of the parts that are simpler. That is, you evaluate variables by going back and evaluating the literals. So you were evaluating something 'complex' by evaluating something simple first. Going back to the loop example, with an interpreter, we look at each iteration of the loop.

So an interpreter, after getting the abstract syntax trees as the input, will, for each non-terminal (expression, command, etc) will create a procedure, or rule for it. So you will find out what you are supposed to do if you see something of that type. Now you have to write the code that implements the rules. So if you were doing natural semantics, you would be finding out how to get the final solution from the original code that you are interpreting. On the other hand, with transition semantics, you are finding out how to get the next "state" of your code. So if you wanted to, you can use it to find the final solutions, but you would have to put your code in a loop, to keep it iterating from state to state.

2.6 Sample code for Natural Semantics (slides 38-39)

Here, we will look at sample code for implementing natural semantics. So, to evaluate an expression, you have to know how to evaluate all the pieces of an expression. That means you have a case for variables, for integers, etc. So for variables, if we want to evaluate it, we simply look it up in memory, like we said to earlier. If we wanted to evaluate an integer, we look it up in the sense that we know numbers map to themselves. To evaluate addition, you evaluate the left hand side, then the right hand side and then add them. Basically, you will end up coding the rules we talked about this lecture and the last one.

For commands, let's look at the `if_then_else` first. This command will get a boolean expression, then two commands and a memory. It will start by evaluating the boolean expression, and after getting the expression back, we check to see if we should go to the then case or if we should go to the else case. In the then case, we evaluate the first command with the memory we got, otherwise we evaluate the second expression with the given memory. If we are doing a while loop, then we, again, have cases. If the boolean expression evaluates to false, return the given memory, otherwise, run the command once and then run the while with and updated memory of what the command returned.

Note, that with the while loop, we may fail to terminate. Either there is an infinite loop, or there loop just runs soooooo many times that it fills up the entire stack. Both of these cases will return garbage data if anything at all.

3 Transition Semantics

3.1 Idea (20a-trans-sem slides 2-3)

Transition semantics can be used as the basis of an interpreter or a compiler. It is an operational semantics. It only, however, tells you how to make one step, instead of how to evaluate the whole expression. So our rules look like $(C, m) \rightarrow (C', m')$. That is, we have a command C and a memory m and then after running one step of the code, the command now has the form C' and the memory recognizes that something has changed and so you get m' . Or in the case that you have reached the end of the command, then you simply return the changed memory.

One thing about transition semantics is that you can tell the difference between a code that doesn't finish because of an infinite loop and the code the doesn't finish because it is trying to access something in memory that doesn't exist.

3.2 Transitions for Expressions (slides 5-8)

Note that we are going the continue using the same language we were using before.

So for identifiers, in one step, it will evaluate to a value, provided there is a value for it to evaluate to. So, we go from (Id, m) to $m(Id)$. If there's no value, then there is no arrow.

Numerals immediately evaluate to themselves.

Now, before we move on, let's talk about the memory update. If you had in memory that $Id \rightarrow V$ and your update is Y (the new value), then if Y is equal to Id , then keep V as what Id is mapped to, otherwise, update Id with Y .

When evaluating boolean expression, most rules that use the values true or false, only need one step to finish the evaluation. For example, $(false \& B, m)$ will return false. $(true \text{ or } B, m)$ will return true. If you had $(true \& B, m)$ or $(false \text{ or } B, m)$ then, both of those expressions would take you to the next step of evaluating the expression (B, m) . Then if you only had expressions, and no values. Then, the first step with and, or, not, would be to evaluate (B, m) and that would result in some B'' and m , and then your next step would be to evaluate $(B'' \& B, m)$, $(B'' \text{ or } B, m)$, or $(\text{not } B'', m)$.

With relations, your first step is to evaluate the first expression E with memory m , and that gives an (E'', m) , and then you take that E'' and use it in the relation with the original E' and memory m for your next step. That is, if you had $(E > E', m)$, your first step would be to evaluate (E, m) . That gives an (E'', m) and then you would replace that E with E'' , so you would get $(E'' > E', m)$. And that expression is the next step of the tree.

Arithmetic operations are the same. You have an expression, E , operation, op , and then second expression, E' and then a memory. You go and evaluate the E , and the returns and E'' . Then replace the E with E'' for the next step. That is, we start with $(E \text{ op } E', m)$ and then evaluate (E, m) to get (E'', m) and then take that to get the next expression we need to calculate, $(E'' \text{ op } E', m)$.

3.3 Transitions for Commands (slide 9-10)

If you see a skip, you're done. So if you see (skip, m) you simply return m .

If you need to evaluate an assignment, your first step is to evaluate the expression. So when you see $(\text{Id} ::= E, m)$, you then take the E , and evaluate it, (E, m) . That returns an (E', m) and then you replace the E with E' and get, $(\text{Id} ::= E', m)$. And that is your next step in the calculation if you decide to continue evaluating. If you already have a value, then your next step is to update memory. That looks like $(\text{Id} ::= V, m)$, and the updated memory looks like $m[\text{Id} \rightarrow] V$.

As for sequences, your job is to work on the first command in the sequence of commands. That is, if you have $(C; C', m)$, a command C , a command C' , and memory m . Then you would take that first command C , and take one step with it to get (C'', m) . Then you replace the C with C'' . So you have $(C''; C', m')$ to work on next. When you get to the point that you have finished working on the first command, that is when you had $(C; C', m)$ and you went to evaluate (C, m) and after that one last evaluation you found out you were done and that memory that was returned, the m' , well you take that m' and you work on C' with it. So you have (C', m') and that is your next step.