

---

# CS 421 – Spring 2007

## Lecture Notes Set 32:

### Transition Semantics Continued and CPS

Elsa L. Gunter<sup>1</sup>

Transcribed by: Pooja Mathur

---

**Corresponding to Slides:** 20a-trans-sem (slides 12-end) 26-CPS (slides 1-3)

Made available: April 16, 2007

Revision 1.0

---

## 1 Change Log

1.0 Initial Release.

## 2 Last notes on Transition Semantics

### 2.1 Commands (slides 12-13)

So last time, we finished talking about some of the commands. We ended with skip, assignment, and sequence. Now, we will talk about the `if_then_else` case. Here there are two overall cases. First if the expression is for the if statement is already a boolean value, then the next step would simply be to evaluate the case the if statement points to. If the expression needs to be evaluated, that's what you do first.

Then there is the while command. To do this, you expand the while into an `if_then_else` expression. What you do to translate it is take the boolean expression that starts the while loop and then that becomes the if statement. Then a single iteration of the command that you perform during the while loop becomes the then case along with another call to the while loop. The else case, the case that is called when the if statement is no longer true just has a skip in it. To clarify, you do one of three things, you either, have to evaluate the boolean expression (if it hasn't been evaluated yet), run one iteration of the while loop and call it again, or complete the while loop with the else case and move on.

### 2.2 Example (slides 14-21)

So, we want to evaluate  $(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \rightarrow 7\}) \rightarrow ??$  (Recall this is the same example that we showed how to solve with natural semantics.

So what do we do first? What is the first step? Well, it is to evaluate the boolean expression. So that is what we do. We need to evaluate  $(x > 5, \{x \rightarrow 7\}) \rightarrow ?$  Ok, from here, what would be our next step? We need to evaluate the  $x$ , so that is what we do:  $(x, \{x \rightarrow 7\}) \rightarrow 7$ . So at this point your tree should look like:

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

Now that you have the 7, you can iterate that down the tree and now you have your next step of evaluating the boolean expression. So you have  $(7 > 5, \{x \rightarrow 7\})$  on the right side of the arrow on the second level. Next you can iterate that expression down, so now your new problem to solve is:  $(\text{if } 7 > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \rightarrow 7\})$ . Now... that was just the first step of evaluating the original command and you should have a tree that looks like:

---

<sup>1</sup>© 2007, Share and Enjoy

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

So what should the second step be? Well, we have to continue evaluating the boolean expression. So the bottom of the tree has:  $(if\ 7 > 5\ then\ y := 2 + 3\ else\ y := 3 + 4\ fi,\ \{x \rightarrow 7\}) \rightarrow ??$  And then what do we do, we evaluate  $7 > 5$  and that returns true, so we send that down and get:  $(if\ true\ then\ y := 2 + 3\ else\ y := 3 + 4\ fi,\ \{x \rightarrow 7\})$ . After the second step, our tree looks like:

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

For the third step, we figure out which case our boolean is telling us to take:

$$\frac{(if\ true\ then\ y := 2 + 3\ else\ y := 3 + 4\ fi,\ \{x \rightarrow 7\}) \rightarrow (y := 2 + 3, \{x \rightarrow 7\})}$$

In the fourth step, we need to start evaluating the expression of the assignment, so we go and evaluate the  $2 + 3$ . We get 5 back and then we know that the next thing we want to evaluate is  $(y := 5, \{x \rightarrow 7\})$ . Our tree for this step is:

$$\frac{(2+3, \{x \rightarrow 7\}) \rightarrow 5}{(y := 2+3, \{x \rightarrow 7\}) \rightarrow (y := 5, \{x \rightarrow 7\})}$$

Our final step is to make the finalized assignment. So we put  $y \rightarrow 5$  in memory. This gives us the tree:

$$\frac{(y := 5, \{x \rightarrow 7\}) \rightarrow \{y \rightarrow 5, x \rightarrow 7\}}$$

If you look at all the bottom levels, you can watch the evaluation happen step by step. That is, you have:

$$\begin{aligned} &(if\ x > 5\ then\ y := 2 + 3\ else\ y := 3 + 4\ fi,\ \{x \rightarrow 7\}) \rightarrow \\ &(if\ 7 > 5\ then\ y := 2 + 3\ else\ y := 3 + 4\ fi,\ \{x \rightarrow 7\}) \rightarrow \\ &(if\ true\ then\ y := 2 + 3\ else\ y := 3 + 4\ fi,\ \{x \rightarrow 7\}) \rightarrow \\ &(y := 2 + 3, \{x \rightarrow 7\}) \rightarrow \\ &(y := 5, \{x \rightarrow 7\}) \rightarrow \\ &\{y \rightarrow 5, x \rightarrow 7\} \end{aligned}$$

## 2.3 Overall idea of evaluation (slides 22,29-31)

So the idea behind transition semantics is that you have a string of proof trees. That is every transition from one step to the next has a proof tree associated with it. The tree can be really small or really big, but it's there. In comparison, in natural semantics, you create one huge proof tree above your initial expression.

Basically, if you look at the tree we built for natural semantics the other day and then the trees for the evaluate this way, you have a perfect example. The natural semantics tree ends up being on big one with many levels. Here the "proof tree" has steps. So, the tree in step one points to the tree for step two, which points to the tree for step four...all the way to six, where we cannot evaluate any farther.

Where transition semantics shows the relations between specific steps in evaluations, natural semantics show the relationship between steps and your final value.

So why do we have both? Well, with natural semantics, it is much more intuitive to expression recursive programs while transition semantics make it easy to show iterative programs. While natural semantics are more concise, we have to still worry about our program never terminating.

## 2.4 Adding Local Declarations (slides 23-25)

Let's now add in LetIn, functions, and function applications.

So, for LetIn, the rule is going to look like:  $(\text{let } I = V \text{ in } E, m)$ . That is to say, let some identifier  $I$  be equal to some value  $V$  in the some expression  $E$  (and that goes with the memory  $m$ ). So, here, since we already have the value, we can just stick it in the expression. That is to say, anywhere we see the identifier  $I$  in the expression  $E$ , we are going to replace with the value  $V$ . However, if the value has not been evaluated yet, we have to start evaluating it. So that is, if the whole thing has the structure, let some identifier  $I$  be equal to some expression  $E$  in some other expression  $E'$ , the first thing we have to do is evaluate the expression  $E$ . So that is what we do, and we get an  $E''$  back and then we can say that the identifier  $I$  is equal to  $E''$  in the other expression  $E'$ .

For functions, we can have  $(\text{fun } I \rightarrow E) V$  with some memory  $m$ . What this is saying is that there is a function that takes an argument  $I$  and puts that argument into the expression  $E$ . In that expression,  $I$  gets the value  $V$  applied to it. So that's what we get, we say that in the expression  $E$ , for any place that has the argument  $I$ , we will replace it with  $V$ . That, however, is the case that the value we are passing in is already a value. But what if we are passing in an expression. In this case we have to evaluate the expression first, and then move on with the replacement and everything in the next step.

Note, that this has all been eager evaluation. The reason is because we have 'eagerly' been evaluating every expression we come across. (Eager evaluation is also known as call-by-value evaluation.) The opposite would be with lazy evaluation, where we would have not done any work (evaluating) until we had to. (This is also known as call-by-name evaluation.)

The next question is what is the difference between doing  $(\text{let } I = E \text{ in } E', m)$  and  $((\text{fun } I \rightarrow E') E, m)$ ... Well one is a function and then other is local declaration. But look at what they both go to with lazy evaluation. They both end up being  $(E' [E/I], m)$ , that is, in both cases, you replace every instance of the identifier  $I$  with the expression  $E$  in the other expression  $E'$ . Well, sometimes, there might not be a difference. It all depends on the language you are programming in. For example, if you had a print statement in a let statement, that is, you have  $(\text{let } x = \text{print\_int } 4; 7 + 2 \text{ in } y)$ , then in eager evaluation, you will have to evaluate the let expression first. So that means printing 4 and then adding 7 and 2, and then assigning 9 to  $x$ . After all that do you see that you didn't actually need to figure out  $x$ , because the expression doesn't use it, it just uses  $y$ . In lazy evaluation, however, you just see that there is some expression that is equal to  $x$ , and then you go off and replace all the instances of  $x$  in  $E'$ . Well,  $E'$  is just  $y$ , so you don't do any replacements, and since you never actually use  $x$ , you never end up evaluating that expression and in turn, you never print 4.

## 2.5 Church-Rosser Property (slides 26-27)

Trying to figure out if it matters or not how we get a value, is the same as trying to figure out, with our transition system, is our language confluent. Does it have the diamond property? Basically what this all is asking is, if you do evaluations in a different order or way, will you end up in the same place. For example, if you wanted to evaluate  $2 + 3 + 4$ , you can do  $(2 + 3)$  first or you can do  $(3 + 4)$  first. In the former case, you have  $5 + 4$ , in the latter case you  $2 + 7$ . Then when you do the final evaluation, both end up at 9. So it didn't matter which path I took to evaluate that expression. I was able to end up in the same place. However, if you had  $2 * 3 + 4$ ...  $2 * 3$  is 6 so you then have  $6 + 4$  which is 10. The you have  $3 + 4$  is 7 and  $2 * 7$  is 14. Here, we ended up in different places, so then we can say that addition and multiplication do not have the Church-Rosser Property, while addition alone does.

Then taking that idea, you can apply the idea to languages. A language is Church-Rosser if you always end up at the same value, otherwise it isn't. So generally, languages with side-effects do not have confluence when there is a combination of call-by-name and call-by value instructions.  $\lambda$ -calculus however is confluence.

## 2.6 Transitions for $\lambda$ -calculus (slide 28)

So in  $\lambda$ -calculus transition semantics look every similar to what we have seen. The only real difference is that we see  $\lambda$  appear every now and then. So application can look like one of two things. In lazy evaluation, we can say  $(\lambda x. E) E' \rightarrow E[E'/x]$ . That is, if we have an expression with the argument  $x$ , we want to apply the expression  $E'$  to  $E$  through the argument  $x$ . So for every instance of  $x$  in  $E$  we will replace it with  $E'$ .

With eager evaluation, we are going to say, well, if we have a value already, it looks like:  $(\lambda x. E) V \rightarrow E[V/x]$ . That is we are applying a value  $V$  to the expression  $E$  through the argument  $x$ . Then that ends up meaning, for every  $x$  we see in  $E$ , we replace it with  $V$ . On the other hand, if we have  $(\lambda x. E) E' \rightarrow ?$  We have to evaluate  $E'$  first. So  $E$  goes to some  $E''$  after evaluation and then our next step is to evaluate  $(\lambda x. E) E''$ . That replace the question mark.

### 3 CPS (new slide set: slides 2-3)

So far with transition semantics, we have looked at what does one step do for us. What we want to do is use these tiny steps to control the flow of our programs. The idea is going to be, that when we write a function, instead of returning a result, we are going to give the function an additional argument, called continuation, that it will take the value we just calculated and move on with that result. So instead of running a function and returning a result we are running a function and then passing on the result.

The point of this is to make functions much more like what you do in imperative programming, that is something that has a large sequence of commands. So you do something...then you have the next job and then the next and then next, etc.

Next time we are going to talk about using continuation passing style to use these continuations. So we will pass along results instead of return them, which is basically tail recursion, taken to the next step.