
CS 421 – Spring 2007

Lecture Notes Set 35:

CPS continued

Infinite Data

Elsa L. Gunter¹
Transcribed by: Pooja Mathur

Corresponding to Slides: 26-CPS (slides 24-32) 25-infinite-data (slides 1-23)
Made available: April 23, 2007
Revision 1.0

1 Change Log

1.0 Initial Release.

2 Control Flow

2.1 Example (slides 24-26)

Last time we ended in the middle of an example where we wanted to control the flow of the program. Specifically, we wanted to write a program that did additions before it did the subtractions. In our first try, we did this:

```
# let rec eval lst k =  
  match lst with  
  | (Add x) :: xs → eval xs (fun r → add r x k)  
  | (Sub x) :: xs → eval xs (fun r → sub r x k)  
  | [] → k 0
```

But this didn't do anything to control the flow. So we tried again and came up with:

```
let ordereval lst k =  
  let rec aux lst ka ks = match lst with  
  | (Add x) :: xs → aux xs (fun r k → add r x ka k) ks  
  | (Sub x) :: xs → aux xs ka (fun r k → sub r x ks k)  
  | [] → ka 0 ks k  
  in  
  aux lst idk idk
```

Now, let's look at why this works. We have multiple continuations here. The continuation k will take the final result. The continuation ka will handle and keep the order of the additions. The continuation ks will handle and keep the order of the subtractions. Then the only question now is, which will happen first. Well, in the base case the addition continuation is the one that gets a value to work with first, so it can proceed with its work. The subtraction continuation doesn't get anything until all the additions are completed, then the value achieved from the additions get passed to the subtraction continuation and it can do its work. After all that, the final answer is returned.

¹© 2007, Share and Enjoy

2.2 Sample Run (slides 27-32)

To further understand, let's look at an example run of the above code. We want to run the code `ordereval [Add 20; Sub 5; Sub 7; Add 3; Sub 5] report`. So what happens?

```
ordereval [Add 20; Sub 5; Sub 7; Add 3; Sub 5] report ⇒
aux [Add 20; Sub 5; Sub 7; Add 3; Sub 5] idk idk report ⇒
aux [Sub 5; Sub 7; Add 3; Sub 5] (fun r1 k1 → add r1 20 idk k1) idk report ⇒
aux [Sub 7; Add 3; Sub 5] (fun r1 k1 → add r1 20 idk k1) (fun r2 k2 → sub r2 5 idk k2) report ⇒
aux [Add 3; Sub 5] (fun r1 k1 → add r1 20 idk k1) (fun r3 k3 → sub r3 7 (fun r2 k2 → sub r2 5 idk k2) r3) report
⇒
aux [Sub 5] (fun r4 k4 → add r4 3 (fun r1 k1 → add r1 20 idk k1) k4) (fun r3 k3 → sub r3 7 (fun r2 k2 → sub r2
5 idk k2) r3) report ⇒
aux [] (fun r4 k4 → add r4 3 (fun r1 k1 → add r1 20 idk k1) k4) (fun r5 k5 → sub r5 5 (fun r3 k3 → sub r3 7 (fun
r2 k2 → sub r2 5 idk k2) r3) k5) report ⇒
ka 0 ks k ⇒ (fun r4 k4 → add r4 3 (fun r1 k1 → add r1 20 idk k1) k4) 0 (fun r5 k5 → sub r5 5 (fun r3 k3 → sub
r3 7 (fun r2 k2 → sub r2 5 idk k2) r3) k5) report
```

Let's take a minute right here to look at what just happened. The first step was just calling the function which then called the auxiliary function. The next step, we saw an Add, and did the match statement, and then passed the value in the Add to the addition continuation. In the next step, we saw a Sub, so we sent the value to the subtraction continuation. But wait, what about doing the adds first? Well...let's just see how things pan out. In the next step we see another Sub, so we send the 7 to continuation, but that continuation's last call, was another subtraction continuation, so we now have nested continuations. Then we see another Add and nest the additions. And then we see another Sub and further nest the subtractions. Then we get to the base case where we send the addition continuation the value 0 to finish off. So now what?

```
add 0 3 (fun r1 k1 → add r1 20 idk k1) (fun r5 k5 → sub r5 5 (fun r3 k3 → sub r3 7 (fun r2 k2 → sub r2 5 idk k2)
r3) k5) report ⇒
(fun r1 k1 → add r1 20 idk k1) 3 (fun r5 k5 → sub r5 5 (fun r3 k3 → sub r3 7 (fun r2 k2 → sub r2 5 idk k2) r3)
k5) report ⇒
add 3 20 idk (fun r5 k5 → sub r5 5 (fun r3 k3 → sub r3 7 (fun r2 k2 → sub r2 5 idk k2) r3) k5) report ⇒
idk 23 ⇒
(fun r5 k5 → sub r5 5 (fun r3 k3 → sub r3 7 (fun r2 k2 → sub r2 5 idk k2) r3) k5) 23 report ⇒
sub 23 5 (fun r3 k3 → sub r3 7 (fun r2 k2 → sub r2 5 idk k2) r3) k5) report ⇒
(fun r3 k3 → sub r3 7 (fun r2 k2 → sub r2 5 idk k2) r3) 18 report ⇒
sub 18 7 (fun r2 k2 → sub r2 5 idk k2) report ⇒
(fun r2 k2 → sub r2 5 idk k2) 11 report ⇒
sub 11 5 idk report ⇒
idk 6 report ⇒ report 6 ⇒ 6
```

So, let's look at what happened here. We place the 0 inside the add, and do $3 + 0$. That result gets passed to the other add. And then we do $3 + 20$. That gets sent to the next continuation, which is the subtraction continuation. Then we take 23 and start that sequence. So we first do $23 - 5$ and then $18 - 7$ and then $11 - 5$ and we end up with 6. Send that to idk and then to report and we're done. The whole point of this is to see that the add continuation sends its value to the subtract continuation. And in fact, the subtract continuation is an argument of the add continuation. Take a look, add takes two continuations, idk and k1? What is in k1's place, the whole subtraction continuation. And because of that, this works, because add passes its values to subtract we can do the adds before the subtracts.

3 Infinite Data

To be able to deal with infinite data, we are going to look at three methods of evaluation.

3.1 Call by Value (eager) (new slide set slides 2-14)

Recall eager evaluation, where you want to evaluate your argument first. And then you can use it. One way to remember this is to think that we are just soooooo eager to do all this work that we will evaluate everything we run into. You can associate eager with call by value by thinking, since we evaluated everything, everything must be a value and not an expression.

Let's look at a quick example. If we had: let $f\ x = (x+1)*(x+1)$ and then tried to run $f\ ((3*4)/6)$. The first thing we would do is evaluate the $((3*4)/6)$. So we would get:

```
f ((3*4)/6)
f ((12)/6)
f (2)
(2+1)*(2+1)
3*(x+1)
3*3
9
```

We only had to evaluate the expression and then pass that value along. Because of that, it can be very efficient, and it's easy to implement. However, sometimes we might evaluate something that we didn't need to, so it could also be wasteful. And because it evaluates everything, we might end up trying to evaluate something that may result in an infinite loop, but if we didn't evaluate everything, we would have been fine.

For example of this nontermination, if we had the function let $rec\ foo\ x = foo\ (x + 1)$ and the function let $ftue\ a\ b = a$. So, foo is going to call itself with an incremented argument...basically forever. $ftue$ is just going to return it's first argument. It doesn't need to do anything with the second argument. However with eager evaluation, if you tried to run $ftue\ 5\ (foo\ 10)$, you would evaluate the 5...ok done, and then evaluate the $foo\ 10$...but that would need to evaluate $foo\ 11$ and then $foo\ 12$, $foo\ 13$, $foo\ 14$... and you would never be done. But if you only evaluated the expression, you would have only evaluated the 5 and then would have been done.

3.2 Call by Name (lazy) (slides 15-23)

Now, we will look at call by name. Here we only want to evaluate what we have to, we're too lazy to do anything more. And you can associate call by name to lazy by thinking that we need to name whole expressions and call those, because of our laziness to do anything else with the expressions.

Let's look at the same example but with lazy evaluation:

```
f ((3*4)/6)
(((3*4)/6)+1)*(((3*4)/6)+1)
(((12)/6)+1)*(((3*4)/6)+1)
(2+1)*(((3*4)/6)+1)
(3)*(((3*4)/6)+1)
(3)*(((12)/6)+1)
(3)*(2+1)
(3)*(3)
9
```

Here we have to do more steps. We have to, in fact, evaluate the same function twice. So the evaluation of functions can be inefficient, but at least if a function can terminate, it will.

To use this in OCaml, you can lambda-lift expressions. Then, you can only evaluate something when you want to. So with the example, you can make a function that takes a unit and then to evaluate the expression, you need to give it a unit.