

---

# CS 421 – Spring 2007

## Lecture Notes Set 34:

### CPS continued

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

Transcribed by: Pooja Mathur

---

**Corresponding to Slides:** 26-CPS (slides 18-31)

Made available: April 20, 2007

Revision 1.0

---

## 1 Change Log

1.0 Initial Release.

## 2 Terminology review (slides 18-26)

Let's take a moment to have a little review of terminology. We discussed before that an expression in the tail position is the position such that it is the last thing to be evaluated. So in let  $x = 5$  in  $x + 4$ , the  $x+4$  is in the tail position.

Following that idea, we have a tail call, which, still happens at the end, but instead of it being a simple evaluation of an expression, but a function call.

Next there is the idea of availability. Sometimes a function call might not be available because it is hidden by some command or other function. For example in an `if_then_else` statement, if there was a function call in the then case and the else case, one call would never run because you can only run one of the two choices. Basically, you can tell if something is hidden or not based on whether or not you can reduce/simplify an expression. If you can't get to it, or can't simplify it yet, then it must be unavailable.

## 3 CPS Transformation

### 3.1 Steps (slides 20-21)

The first step is very simple. You alter your function declaration to have the continuation function as part of its arguments. So instead of `let func arg1 arg2 ... = etc...` you have `let func arg1 arg2 ... k = etc`. Basically, you are just adding the  $k$  in at this point.

The next step is to instead of return your answer, you need to pass it on to  $k$ . By doing that generally means that you apply  $k$  to your answer so you have a function and its arguments. That is, if you have `return a`, you would instead write `k a`. Which means, you have your function  $k$  and you are applying the argument  $a$  to it.

On the other side of things, if you don't have your final answer to return, but you need to pass along your current continuation to your function calls you just add  $k$  to your function calls. So instead of `return f arg` you will have `f arg k`. So you will be calling your function  $f$  again with the argument  $arg$  and the continuation  $k$ .

The last step is to add intermediate continuations, in cases where you need to do more work. So let's say you wanted to do some operation on what gets returned from calling  $f arg$ . And after you do that operation, you want to return that answer. Well, with continuations instead of returning something, you end up sending it to  $k$ . So in the first case you might have `return op (f arg)` as your code. But with continuations, you can write `f arg (fun r → k(op r))`. That is you are doing the  $f arg$  call before. You will be getting some result from it and then it turns out that that value will be given to  $r$  and have  $op$  applied to it. Then that result will be passed to  $k$ . So both codes do similar things.

---

<sup>1</sup>© 2007, Share and Enjoy

And that's it, that's how you convert from regular code to CPS.

### 3.2 Example (slide 22)

So let's say we had the following code to add all the elements in a list.

```
let rec add_list lst =  
  match lst with  
  | [] → 0  
  | 0 :: xs → add_list xs  
  | x :: xs → (+) x (add_list xs);;
```

The first thing we do is to add  $k$  to the declaration:

```
let rec add_listk lst k =
```

Next, you take your simple returns and pass them to  $k$  instead. So here, in the original code, if we saw the empty list, we returned 0, but now, we will be returning  $k$  0.

```
  match lst with  
  | [] → k 0
```

The third step says to pass the continuation function along when you make function calls.

```
  | 0 :: xs → add_listk xs k
```

For the last step, we still have work to do, so here is where we make an intermediate continuation. So you start out making your function call like normal. But instead of the  $k$  we have been using, we are going to say that our continuation is going to add the head to the result so far and then send that to the old  $k$ .

```
  | x :: xs → add_listk xs (fun r → k ((+) x r));;
```

## 4 Continuations Example (slides 23-26)

Now let's look at an example that uses continuations to do its work. Let's say we started out with the following code. The first function is a continuation that adds two numbers together. The next one subtracts one from the other. The third one reports the value calculated back to you. The next one just gives the value back, it's the identity function. Then at the end we have a type declaration.

```
let add a b k = print_string "Add "; k (a + b);;  
let sub a b k = print_string "Sub "; k (a - b);;  
let report n = print_string "Answer is: ";  
  print_int n;  
  print_newline ();;  
let idk n k = k n;;  
type calc = Add of int | Sub of int
```

For the example's sake, let's just say that this set of code has a problem with negative numbers. The code just can't handle it. So when we do a sequence of adds and subtracts, we want to do all the adds first. How can we make sure that happens?

```
# 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
```

Well, like usual, we have our base case, where we send 0 to the continuation. That result gets returned to the call before it and you can now finish the call to another continuation. With this code what we are doing is, we are saying ok, if you see something of type add, do the recursive call, but send your result of whatever the recursive call returns to the new intermediate continuation and perform the add. Otherwise, if you see a subtract send what is returned to the subtract function. But...this code...doesn't make the adds add on their own. It can't subtract before all the adds are finished. So what do we do? We still need to separate the functions. Start thinking of ideas on how to do this with maybe more than one continuation.

(Note that the function does the adds and subtracts from right to left. That is, if you had a list of adds and subtracts. The function would have to recurse to the end of the list, pass  $k$  the 0 and then it can continue to perform the adds and subtracts from there.

Well, to start off thinking about multiple continuations in one function, what if we made adding and subtracting happen in the same part of the code. Instead of two cases, just have one. So, we can start with:

```
# add 3 5 (fun r → sub r 2 report);;
```

Here, what happens is we add two numbers, and send that result to subtract and subtract 2 from that and then report back the value. There are two continuations, report and the fun r function. But what if we wanted to pass more the fun r function, what if we had:

```
# add 3 5 ((fun k r → sub r 2 k) report);;
```

Here, what is happening is the same, but instead of directly giving the function report, we are passing it in as an argument. Notice that it says fun k r and not fun r k. The reason for that is because the addition of 3 and 5 will appear on the outside of the report, so report will be applied for k and 8 will eventually be applied for r.

Ok, so let's go back to our old problem. We want the additions to happen first. In this code, we are going to be using three continuations total. The original, and then there is ka for the addition continuation and then ks for the subtraction continuation. Looking at the base case, with the empty list, we give ka the value 0, but the other continuations get nothing. What that is say is, the additions are going to get to work first since they are the only one with values. And we will leave things here for now. See if you can understand things on your own for now. The rest of the explanation will come with the next lecture.

```
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
```