

---

# CS 421 – Spring 2007

## Lecture Notes Set 33:

### CPS

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

Transcribed by: Pooja Mathur

---

**Corresponding to Slides:** 26-CPS (slides 6-17)

Made available: April 18, 2007

Revision 1.0

---

## 1 Change Log

1.0 Initial Release.

## 2 CPS

### 2.1 Terminology (slide 6)

A function is said to be in direct style when it returns its result to whatever procedure called it. However, we aren't going to use direct style. We are going to be using something else. Recall tail recursion, in CPS, instead of returning the result from a recursive call, we are going to do a tail call to its continuation. So you will always be passing on our result, instead of passing it backwards.

### 2.2 Simple example (slide 7)

But if we continue to pass results forward, we will seemingly, never have an end of the chain. So we have to put something there to say we are done. So, our end of the chain in most cases will be the report function. And what it does is simply report back the result. That is, it prints out the integer that was calculated and then prints a new line. Other times we will use the identity function. So it will take what ever you gave it and then return it right back.

But here, along with the report function, we will use the function plusk. Recall, when we were first learning OCaml and we learned how to make a simple function that added two numbers. Now we are going to take that a step farther and turn such a function into a continuation. So we are going to take three arguments and still do the addition. There are the two values to add  $a$  and  $b$  and then there is  $k$  which is what receives the result of performing some operation. So what happens is  $a$  and  $b$  get added together and then the result is handed to  $k$ .

```
# let plusk a b k = k (a + b)
val plusk : int → int → (int → a) → a = <fun>
```

So another way to think of this is that plusk doesn't give back a result, but the function  $k$  is what returns the answer. You can see this by looking at the type of plusk. The first two ints belong to  $a$  and  $b$  and then we see that plusk also takes in a function that takes an int and returns a ' $a$ '. That function is what goes into  $k$ . And that function returns the final result of that has type ' $a$ '.

Now, let's try to use plusk. We send in the values 20 and 22 and the function report. The numbers get added, and then they are sent to report. Report prints the 42 (like we told it to) and then we get a final type of type unit.

```
# plusk 20 22 report;;
42
- : unit = ()
```

---

<sup>1</sup>© 2007, Share and Enjoy

## 2.3 Recursive example (slides 8-10)

So for simple functions, you take in your argument, you do whatever computation you were supposed to do to it, but now you take one more argument which is going to be the continuation. It will be the place the result needs to go. In the recursive case, part of the work goes into calling itself. So let's use factorial as an example.

In normal OCaml code, we have the recursive factorial function take in a single argument  $n$ . We have our base case and recursive case and end up with a function that takes an int and returns an int. So when we apply factorial to 5, we get 120 for our answer. Seems simple enough.

```
# let rec factorial n =
  if n = 0 then 1 else n * factorial (n - 1);;
val factorial : int → int = <fun>
# factorial 5;;
- : int = 120
```

But for CPS, you need to write a function that not only takes in another function that tells you what to do with your result, but really what is going on here is that the recursion is doing some part of the computation and now the continuation has to tell the recursive calls how it should carry on. But usually, with recursive functions, you aren't done with your computation, so you have to build a new continuation in a sense, with these new values and move on from there.

Now let's take a look at the CPS version, piece by piece. So, like we did before, we add in the argument  $k$  in the function declaration.... ok simple enough, let's move on. In the base case, we have if  $n = 0$  then  $k$  1. What does this mean? Well, when we get to the base case, we know we don't want to be recursing anymore. We know we want to send some value back to the previous calls, just like normal recursive. But now we aren't going to only send back the value. We need to first give our base case to the continuation. So, since  $k$  is a function, will apply the function  $k$  to the value 1. That returns some value and we move on. If this doesn't make complete sense yet, let's take a look at the recursive case and then hopefully we can connect the dots.

In the recursive case we have that, like normal recursion, we are calling our recursive function `factorialk` with the reduced  $n$  as the first argument. Ok, so far so good. But what about  $k$ ... instead of sending  $k$  or report we are writing a new function... What is this function doing? Well, it's taking some argument  $m$ ...and then multiplying what it took in by  $n$  and sending that to the last  $k$ ...

```
# let rec factorialk n k =
  if n = 0 then k 1 else factorialk (n - 1) (fun m → k (n * m));;
val factorialk : int → (int → 'a) → 'a = <fun>
```

To make more sense out of this, let's look at a numerical example. We start off by making a call to `factorialk`.

```
# factorialk 5 report;;
```

Ok, looks good, what's next? Well, next we need to we check cases and find that we are going to the recursive case so we call...

```
factorialk 4 (fun m → k (5 * m)) which then calls
factorialk 3 (fun m → k (4 * m)) which then calls
factorialk 2 (fun m → k (3 * m)) which then calls
factorialk 1 (fun m → k (2 * m)) which then calls
factorialk 0 (fun m → k (1 * m)) which then calls
k 1
```

...ok....what's  $k$  at this point? Well, if we look at the call we just made, the function we sent in for  $k$  was  $(\text{fun } m \rightarrow k (1 * m))$ . So we apply 1 to that function:

```
(fun m → k (1 * m)) 1 ⇒ (k (1 * 1)) ⇒ k 1
```

Now what's  $k$ ? Well, the call before last had  $k$  be  $(\text{fun } m \rightarrow k (2 * m))$ . So we apply the 1 to that and get:

```
(fun m → k (2 * m)) 1 ⇒ (k (2 * 1)) ⇒ k 2
```

The value of  $k$  before that was  $(\text{fun } m \rightarrow k (3 * m))$ . So apply 2 to it:

```
(fun m → k (3 * m)) 2 ⇒ (k (3 * 2)) ⇒ k 6
```

Next:

```
(fun m → k (4 * m)) 6 ⇒ (k (4 * 6)) ⇒ k 24
```

Then:

```
(fun m → k (5 * m)) 24 ⇒ (k (5 * 24)) ⇒ k 120
```

Wait, there's another  $k$ ? But we just got the answer we wanted. However, don't forget in our original call to `factorialk` whose  $k$  function was the report function. So we call,

```
report 120
```

and get:

```
120
- : unit = ()
```

So the whole idea here is that we keep adjusting  $k$  so that we can do our computation. We have to build these intermediate  $k$  functions that tell us what more work there is to be done.

Important note: the way I wrote the steps above, I passed into `factorialk` the  $(n-1)$  without stopping to actually leave a step to do the subtraction. So it may look like I am recursing right away... But that isn't true, this isn't a forward recursive function. The very very first step is not recursion, it is to evaluate the  $(n-1)$ . But we can't just do the recursive call there either. In the steps above, I also filled in the value of  $n$  in the intermediate  $k$ . That also counting as computation is creating the closure for the function  $k$ . That essentially is making it a value. So, now you have your two arguments and you can move on. Now you make the recursive call. Then it repeats the steps, do the computation and then recursion, computation and then recursion, etc, at each step. This means that our function, `factorialk`, is tail recursive. It does the recursive call last.

After all this, the final work is done in the continuation function. We get our recursive value  $m$ , finish our computation at this level, and then send it to the continuation that was passed to us.

## 2.4 Nesting CPS (slide 11)

Another thing you can do is write nested functions with CPS. That is, you use one CPS function in another CPS function. So let's say you want find out the length of a list of elements. There are two ways you can right this. Now, this can easily be written without nesting. It can be done like so.

If we see the empty list. We say there, we pass back 0 to  $k$ . Otherwise, our function  $k$  takes in an argument  $r$  and adds one to it and then returns that incremented value back.

```
# let rec lengthk list k = match list with [] → k 0
| x :: xs → lengthk xs (fun r → k (r + 1));;
val lengthk : 'a list → (int → 'b) → 'b = <fun>
```

But earlier, we wrote `plusk`, and since we are soooooo proud of our very first CPS function ever, we want to use it. So instead of saying  $k (r + 1)$ ... we say `plusk r 1 k`... Wait, what? We aren't passing a value back to  $k$ ...we are including  $k$  in our continuation... what is going on?! Hold on, let's think about this. To make it easier, let's do this step by step with an example. We have the list `[2;4;6;8]` and our original call is `lengthk [2;4;6;8] report`.

So step 1 is, since we don't see the empty list, we go to the other case and break off the head element and then for our  $k$  we have `plusk r 1 k`...

```
lengthk [4;6;8] (fun r → plusk r 1 k)
```

...hmmm still not exactly clear, let's keep going. The next step is to break off our new head element.

```
lengthk [6;8] (fun r → plusk r 1 k) and then..
```

```
lengthk [8] (fun r → plusk r 1 k) and then...
```

```
lengthk [] (fun r → plusk r 1 k)
```

```
k 0
```

Ok, that looks somewhat familiar. We got to a base case and now we use the continuation that was sent to us That was `(fun r → plusk r 1 k)`. So we do that.

`(fun r → plusk r 1 k) 0 ⇒ plusk 0 1 k`...what's  $k$  here?...well it was the continuation passed in at this level... which happens to be `(fun r → plusk r 1 k)`. That means we have:

`plusk 0 1 (fun r → plusk r 1 k)`. So we evaluate the addition, get 1 and remember, in the function `plusk`, we send our sum to our  $k$  value after the addition is performed. That is we had  $k (a + b)$ . Here,  $k$  is `(fun r → plusk r 1 k)` and  $a + b$  is 1 so we have

```
plusk 1 1 (fun r → plusk r 1 k) ⇒ k (1 + 1) ⇒ (fun r → plusk r 1 k) 2
```

And then

```
plusk 2 1 (fun r → plusk r 1 k) ⇒ k (2 + 1) ⇒ (fun r → plusk r 1 k) 3
```

And finally

```
plusk 3 1 report ⇒ k (3 + 1) ⇒ report 4 ⇒ 4 (and prints out unit() )
```

So this might be a little confusing because for the most part,  $k$  was the same function ( $\text{fun } r \rightarrow \text{plusk } r \ 1 \ k$ ). But if you look at the steps, we called  $\text{lengthk}$  4 times with that function as  $k$  and then during the evaluation steps of the functions, it appeared 4 times. It appeared once as we applied it to 0, 1, 2 and 3. So with that, you can match up which call goes with which  $k$ .

### 3 Exceptions

Another thing CPS can do for you is handle control flow. You can have a certain amount of extra precision, and with that - flexibility, when it comes to handling what order your codes runs.

#### 3.1 Example (slides 12-14)

So with the idea of exceptions, we are going to add the idea of exceptions to make things more efficient. Let's say we want to multiply together all the elements in a list. So the old way we did things was to just iterate (or recurse) through the list and multiply the elements one by one. However, if any of the elements was a 0, and then a bunch of other elements appeared after the zero, we just wasted a whole bunch of time multiplying numbers together just to get a 0 back, when we knew we would get a zero back when we first saw the 0.

Instead, we could write things this way. First we state there is some exception, and we call the exception zero and we get back that it has type exception Zero.

```
# exception Zero;;  
exception Zero
```

Next we write our function. We say, if we see the empty list, return 1, it is our base case. If we see a zero, raise the exception. Otherwise do the recursive multiplication.

```
# let rec list_mult_aux list =  
  match list with [] → 1  
  | x :: xs →  
    if x = 0 then raise Zero  
    else x * list_mult_aux xs;;  
val list_mult_aux : int list → int = <fun>
```

Next we say what we do with this exception when we raise it. So we write another function that will try and do `list_mult_aux`, but if the exception is raised at any time, 0 will be returned right there and then.

```
# let rec list_mult list =  
  try list_mult_aux list with Zero → 0;;  
val list_mult : int list → int = <fun>
```

So, if we called `list_mult` on the list `[3;4;2]`, we get 24 returned. On `[7;4;0]`, 0 is returned. If we called `list_mult_aux` on `[7;4;0]` though, we would get "Exception: Zero" returned to us.

When this exception was raised, what happened was, everything that was supposed to happen after will never be called. The program counter basically runs back through the instruction stack until the matching handler is found for that exception. Then anything calls/work that was supposed to be done is now canceled.

#### 3.2 Implementation (slides 15-17)

So, how can we write our own exceptions. Well, as an example we will write the multiplication of a list with our own exceptions. The first thing we need to do is write a CPS function for multiplication. So we have:

```
# let multkp m n k =  
  let r = m * n in  
  (print_string "product result: ";  
  print_int r; print_string "\n";  
  k r);;
```

```
val multkp : int → int → (int → 'a) → 'a = <fun>
```

Now we have a function that takes a list and a  $k$  and *kexcep* which will be the continuation for the exception case. So, like the other version we wrote, if we see the empty list, we return 1, except now we pass it to our regular continuation to get returned. If we see a 0 in the list we will give the 0 to the exception continuation. Otherwise, we make our recursive call with an intermediate continuation.

```
# let rec list_multk_aux list k kexcp =  
  match list with [] → k 1  
  | x :: xs → if x = 0 then kexcp 0  
  else list_multk_aux xs (fun r → multkp x r k) kexcp;;  
val list_multk_aux : int list → (int → 'a) → (int → 'a) → 'a = <fun>
```

Lastly, we need to write the function to call our auxiliary function. So call the function with  $k$  as both the initial regular continuation and the exception case continuation. Note that you can write a function that takes in two completely different continuations, it just happens to work out here, that we can send in the same continuation and have the function do exactly what we want.

```
# let rec list_multk list k = list_multk_aux list k k;;  
val list_multk : int list → (int → 'a) → 'a = <fun>
```

So if you sent in [3;4;2] and report to list\_multk we would some work and recurse to the end and then we start being able to evaluate our continuations. So we first see a 2 printed and then an 8 and then 24 and that is our final result.

But if we sent in [7;4;0]. The first thing that would happen is we would get our intermediate continuations and recurse to the [4;0] case. Then repeat and get [0]. And then when trying to evaluate that, we would see that  $x = 0$  and we would send 0 to the kexcp continuation. That continuation happened to be report. (Notice that our recursive call was always to list\_multk\_aux, so we had to always send in a continuation for  $k$  and kexcp and if you look, we never gave kexcp an intermediate continuation, it was always the same, which was report.)

So we call report with 0 and that prints a 0 and then unit and we are done. We don't waste any time with unnecessary multiplications, which is what we wanted to do with exceptions.

Basically, idea here, to write exceptions with continuations is to write one continuation per handler. And then anywhere you need to raise an exception, you make sure there is some case that leads to it.