

# Comp511 Notes

by Kristin Y. Rozier

February 4, 2005

## 1 CPS

If we take a computation like

```
let rec fact n =
  if n = 0 then 1
  else n * fact (n-1)
```

This function has type `int → int`. This is the direct style.

We can also think of the `else`-statement as

```
else times(fact(minus n 1))
```

We can use this equivalence to do CPS on this function.

Our new function will have type `int → (int →  $\alpha$ ) →  $\alpha$` . (Remember, a CPS'ed function always has type  $A \rightarrow (B \rightarrow \alpha) \rightarrow \alpha$ , where  $A$  and  $B$  are known types and  $\alpha$  is an unknown type, represented usually by OCaml as 'a.)

```
let rec fact' n k =
  if n = 0 then k 1
  else minus' n 1 (fun a ->
    fact' a (fun b ->
      times' n b (fun c ->
        k c) ) )
```

Remember, in CPS, you never return a value, you always pass it on to a continuation instead.

We can simplify this whole expression because we are just passing the `times' n b` into `k`:

```

let rec fact' n k =
  if n = 0 then k 1
  else minus' n 1 (fun a ->
    fact' a (fun b ->
      times' n b k ))

```

## 2 Monads

We can also write the monadic version of this function. Our new function will have type  $\text{int} \rightarrow M(\text{int})$ . In OCaml this unfortunately shows up as  $\text{int} \rightarrow \text{int}(m)$ . OCaml is a little bit stupid that way.

When we use monads, we parameterize whole the computation by `return` or `bind`. We hardly ever do anything that is interesting without doing either a `return` or a `bind`.

Here are the monadic types:

monad:  $M(\alpha)$

return:  $\alpha \rightarrow M(\alpha)$  (This is like a list or a tree or another data structure of  $\alpha$ 's.)

bind:  $M(\alpha) \rightarrow (\alpha \rightarrow M(\beta)) \rightarrow M(\beta)$  (This is like a let-statement.)

```

let rec fact'' n =
  if n = 0 then return 1
  else bind (minus'' n 1) (fun a ->
    bind (fact'' a) (fun b ->
      bind (times'' n b) (fun c ->
        return c) ))

```

There are two important properties that `return` and `bind` enjoy. Firstly, it has to be always the case that `bind a` ( $\lambda x.\text{return } x$ ) has type  $\alpha \rightarrow M(\alpha)$ , where  $\alpha$  is the type of  $x$ . So,  $(\lambda x.\text{return } x)$  has type "`return`" and the statement "`bind a return`" is equivalent to just  $a$ . The intuition you want to use is that `return` takes  $a$  and makes a `bind` out of it. The `bind` takes a monadic  $a$  and looks inside that value and returns a new monadic value,  $M(b)$ . If you actually pass a `return` for the second argument than you are really doing nothing: "`bind a return`" is always equal to  $a$ .

Remember:  $\lambda x.f(x)$  is equal to the function  $f$ . (In the example above, the function  $f$  is `return`.) So, in the example above, "`fun c -> return c`" is just equal to " $c$ " and we can simplify:

```

let rec fact'' n =
  if n = 0 then return 1
  else bind (minus'' n 1) (fun a ->
    bind (fact'' a) (fun b ->
      (times'' n b) ))

```

### 3 Relationship Between CPS and Monads

We need to establish a connection between monads and CPS. We need to make functions that take CPS types and return monadic types to form a conversion between `fact''` and `fact'`.

Our mission is parameterized by 3 things:  $M(a)$ , `return`, and `bind`. Now we need to instantiate  $M(\beta)$  into something that gets us to the form of `fact'` to show that `fact'` is just a special case of `fact''`.

This is how we instantiate the monadic stuff at the level of types to what we need for CPS:

$$M(\beta) \equiv ((\beta \rightarrow \alpha) \rightarrow \alpha)$$

Now we want to find the right definitions for `return` and `bind` to complete the monadic translation.

For `return`, let's try to make the top lines of `fact'` and `fact''` look exactly the same. We will take advantage of the property that a  $\lambda$ -statement allows us to flip the first line of `fact'`:

```

let rec fact' n = fun k ->

```

We move the “`fun k ->`” to the second line of `fact'` to make the first lines identical.

Now, we compare the second lines of the functions:

```

fun k -> if n = 0 then k 1      vs      if n = 0 then return 1

```

Here, we need to use the following transformation:

```

fun x ->      if a
  if a        =>  then fun x -> b
  then b      else fun x -> c
  else c

```

An important note about this conversion: it is NOT always a semantics-preserving transformation. (Very few things are actually always correct when you push them over an if-statement. This is one of those things that should come with a disclaimer "void where prohibited.") This transformation is dangerous because the first version returns a function that waits for an  $x$  and then makes the decision of  $b$  or  $c$  based on the value of  $a$  but the second version makes a decision immediately, then returns a function which returns  $b$  or  $c$  accordingly. We can ignore this distinction in our example because for our purposes here it doesn't matter but it is important to note that this is not always the case..

Thus, the second lines of the functions become:

```
if n = 0 then (fun k -> k 1)      vs      if n = 0 then return 1
```

Now we know the type of return:

$$\text{return } a \equiv \text{fun } k \rightarrow k \ a$$

Next, we examine the third lines of `fact'` and `fact''`:

```
else (fun k -> minus'n 1 (fun a ->
                           vs      else bind (minus'' n 1) (fun a ->
```

And we know the type of bind:

$$\text{bind } x \ f \equiv \text{fun } k \rightarrow x \ (\text{fun } a \rightarrow f \ a \ k)$$

To define the bind conversion, we start with the `fun k` because we want to return something that returns a function. The next thing we want to do is to get the value out of  $x$  and the way we do that is we pass it a continuation. So, we define our own continuation to pass to  $x$ , which introduces  $\alpha$  and gives us a monadic value. The way we get the value out of that is to pass it a continuation. Then we end up with the  $\alpha$ . (The function  $f \ a$  gives us a monadic value of type  $b$ .)

In general, whenever you create a monad, you put things in that you shouldn't want to get out. In other words, you encapsulate your computation and never look inside. If you need to look inside, you can use a `show` function.

$$\begin{aligned} \text{show: } M(\alpha) &\rightarrow \alpha \\ &= \lambda a. a(\lambda x. x) \end{aligned}$$

However, you should not use this function until the end of your program.

## 4 Homework

Write the Fibonacci Function in the three styles demonstrated in class today: direct, CPS, and monadic. Run all three functions on the input value 4 and demonstrate that they all produce the same result.