

## 1 Continuation Passing Style: CPS

It is sometimes necessary to write programs in continuation passing style (CPS) prior to staging for performance reasons. The idea behind CPS is to pass the final result of computation from a function to another function: the continuation. For the purposes of this class, it would be best to write programs in a functional manner, convert them to CPS, then finally stage them.

The example below illustrates how to write functions in continuation passing style. The first set of functions are mutually recursive and not in CPS.

```
let rec even n = if n=0 then true else odd n-1
    and odd n = if n=0 then false else even n-1
```

When you CPS a function with type  $A \rightarrow B$ , its new type should be:  $A \rightarrow (B \rightarrow \alpha) \rightarrow \alpha$ . For this example, the type of the `even` function is `int  $\rightarrow$  bool`, but after it's put into CPS, its type should be `even'`: `int  $\rightarrow$  (bool  $\rightarrow$   $\alpha$ )  $\rightarrow$   $\alpha$` . The continuation is a function, represented by the argument `k`.

```
let rec even' n k = if n=0 then k true else odd' n-1 k
    and odd' n k = if n=0 then k false else even' n-1 k
```

The second argument is now a function that takes the original output of the `even` function and uses it as input and returns something of type  $\alpha$ . This function is better known as a *continuation*. Now the `even'` function will never return a value, but instead pass its final computation to the continuation. The programmer should expect nothing to be returned from a function that has been written in CPS.

Notice the next example: the `even` function has been rewritten in terms of `even'`:

```
let even n = even' n ( $\lambda y.y$ )
```

Here, the identity function has been written as the continuation. The reason for this is because we would like to go from a type of  $A \rightarrow (B \rightarrow \alpha) \rightarrow \alpha$  to  $A \rightarrow B$ . This can be done when we use the identity function in place of  $(B \rightarrow \alpha)$  thus changing the type of the continuation to  $(B \rightarrow B)$ .

## 2 Monads

There are two monadic constructs: `return` and `bind`. `Return` is used to pass the result of a computation onto a continuation. `Bind` is used to bind the result of the last computation in a function to the `return` construct. For instance, lets look at our last example, this time written using the negation operation `not`:

```
let rec even' n k =
  if n=0 then k true
  else not(even' n-1 k)
```

`even'` has the type:  $\text{int} \rightarrow (\text{bool} \rightarrow \alpha) \rightarrow ?$ , because using `not` forces us to return something of type `bool` in the `else` branch, but the `then` branch doesn't return anything at all - its result is passed on to the continuation. Thus, we can change our definition using a new function, `not'` :

```
let rec even' n k =
  if n=0 then k true
  else not'(even' n-1 k)
```

`k` should not be passed to `even'` in the else branch: `k` should only be applied to the last computation and `even'` is not the last computation. Which part of the computation did we miss? We lost the negation of the last computation. Say we have the function `g` instead of `k` in `even'`, where `g` is defined as follows:  $g(r) = \text{not } (r)$ : this means that we can remove the `not'` and simply replace `k` with `g`!

The two monadic constructs can be defined as follows:

```
let return x =  $\lambda k.k x$ 
let bind x f =  $\lambda k.x (\lambda r.f r k)$ 
```

Intuitively we want to fit in the type of the continuation into the type of `return`:  $\text{bool} \rightarrow (\text{bool} \rightarrow \alpha) \rightarrow \alpha$ . We need to pass a continuation to the first argument of `bind` to get a hold of its output. When this is done, the output is bound to `r`, and applied to `f` with `k` passed into `f` as the last continuation. In general, the types of the simplest monad  $(\alpha \text{ m})$ , `return`, and `bind` are:

$$\alpha \text{ m} = (\alpha \rightarrow \beta) \rightarrow \beta$$

$$\text{return: } \alpha \rightarrow \alpha \text{ m}$$

$$\text{bind: } \alpha \text{ m} \rightarrow (\alpha \rightarrow \beta \text{ m}) \rightarrow \beta \text{ m}$$