22

As mentioned in The essence of functional programming:

> Programming with monads strongly reminiscent of continuation —passing style (CPS), and this paper explores the relationship between the two. In a sense they are equivalent: CPS arises as a special case of a monad, and any monad may be embedded in CPS by changing the answer type. But the monadic approach provides additional insight and allows a finer degree of control.

That paper is quite rigorous, and actually it doesn't quite expand on the relationship between CPS and monads. Here I attempt to give an informal, but illustrative example:

(Note: Below is an understand of Monad from a newbie (myself), though after writing it it does appear to look like one of those high-rep users' answer. Please do take it with a ton of salt)

Consider the classic `Maybe` monad

```
-- I don't use the do notation to make it
-- look similar to foo below

bar :: Maybe Int
bar =
    Just 5 >>= \x ->
    Just 4 >>= \y ->
    return $ x + y

bar' :: Maybe Int
bar' =
    Just 5 >>= \x ->
    Nothing >>= \_ ->
    return $ x

GHCi> bar
Just 9
GHCi> bar'
Nothing
```

So the computation stops as soon as `Nothing` is encountered, nothing new here. Let's try to mimic such a monadic behavior using CPS:

Here is our vanilla `add` function using CPS. We are using all `Int` here instead of algebric data type to make it easier:

```
add :: Int -> Int -> (Int -> Int) -> Int
add x y k = k (x+y)

GHCi> add 3 4 id
7
```

Notice how similar it is to a monad

```
foo :: Int
foo =
    add 1 2 $ \x -> -- 3
    add x 4 $ \y -> -- 7
    add y 5 $ \z -> -- 12
    z

GHCi> foo
12
```

OK. Suppose that we want the computation to be capped at 10. That is, whatever computation must stop when the next step would result in a value larger than 10. This is sort of like saying "a Maybe computation must stop and return `Nothing` as soon as any value in the chain is `Nothing`). Let's see how we can do it by writing a "CPS transformer"

```
cap10 :: (Int -> Int) -> (Int -> Int)
cap10 k = \x ->
    if x <= 10
    then
        let x' = k x in
        if x' <= 10 then x' else x
    else x

foo' :: Int
foo' =
    add 1 2 $ cap10 $ \x -> -- 3
    add x 4 $ cap10 $ \y -> -- 7
    add y 5 $ cap10 $ \z -> -- 12
    undefined

GHCi> foo'
7
```

Notice that the final return value can be `undefined`, but that is fine, because the evaluation stops at the 3rd step (`z`).

We can see that `cap10` "wraps" the normal continuation with some extra logic. And that's very close to what monad to -- glue computations together with some extra logic.

Let's go one step further:

```
(>>==) :: ((Int -> Int) -> Int) -> (Int -> Int) -> Int
m >>== k = m $ cap10 k

foo'' :: Int
foo'' =
    add 1 2 >>== \x -> -- 3
    add x 4 >>== \y -> -- 7
    add y 5 >>== \z -> -- 12
    undefined

GCHi> foo''
7
```

Woa! Maybe we have just invented the `Cap10` monad!

Now if we look at the [source code of Cont](), we see that `Cont` is

```
newtype Cont r a = Cont { runCont :: (a -> r) -> r }
```

The type of `runCont` is

```
Cont r a -> (a -> r) -> r
((a -> r) -> r) -> (a -> r) -> r
```

Which lines up nicely with the type of our `>>==`

Now to actually answer the question

Now after typing all this I reread the original question. The OP asked for the "difference" :P

I guess the difference is CPS gives the caller more control, where as usually the `>>=` in a monad is fully controlled by the monad's author.

answered Aug 7 '11 at 16:58
kizzx2
10.8k 13 64 75