The Motherof all Monads

Dan Piponi

June 2024 (originally December 2008)

```
import Control.Monad.State
```

Suppose someone stole all the monads but one, which monad would you want it to be? If you're a Haskell programmer you wouldn't be too bothered, you could just roll your own monads using nothing more than functions.

But suppose someone stole do-notation leaving you with a version that only supported one type of monad. Which one would you choose? Rolling your own Haskell syntax is hard so you really want to choose wisely. Is there a univarsal monad that encompasses the functionality of all other monads?

I often find I learn more comauter science by trying to decode random isolated sentences than from reading entire papers. About a year ago I must have skimmed this post because the line "the continuation monad is in some sense the mother of all monads" became stuck in my head. So maybe `Cont` is the monad we should choose. This post is my investigation of why exactly it's the best choice. Along the way I'll also try to give some insight into how you can make practical use the continuation monad.

So let's start with this simple piece of code

```
import Control.Monad.Cont

ex1 = do
  a <- return 1
  b <- return 10
```

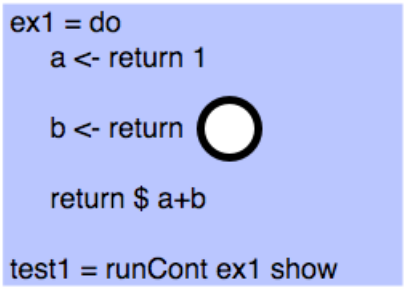
```
    return $ a+b
```

I haven't specified the monad but in almost every case we'd expect the result to have something to do with the number `11`. For the list monad we get `[11]`, for the `Maybe` monad we get `Just 11` and so on. For the `Cont` monad we get something that takes a function, and applies it to `11`. Here's an example of its use:

```
test1 = runCont ex1 show
```

`ex1` is just a function that takes as argument `show` and applies it to `11` to give the string `"11"`. `Cont` and `runCont` are just wrapping and unwrapping functions that we can mostly ignore.

We could have done that without continuations. So what exactly does the `Cont` monad give us here? Well let's make a 'hole' in the code above:



Whatever integer we place in the hole, the value of `test1` will be the result of adding one and applying `show`. So we can think of that picture as being a function whose argument we shove in the hole. Now Haskell is a functional programming language so we expect that we can somehow reify that function and get our hands on it. That's exactly what the continuation monad `Cont` does. Let's call the function we're talking about by the name `fred`. How can we get our hands on it? It's with this piece code:

```
ex1 = do
  a <- return 1
  b <- Cont (\fred -> ...)
  return $ a+b
```

The `...` is a context in which `fred` represents "the entire surrounding

computation". Such a computaton is known as a "continuation". It's a bit hard to get your head around but the `Cont` monad allows you to write subexpressions that are able to "capture" the entirety of the code around them, as far as the function provided to `runCont`. To show that this is the case let's apply `fred` to the number 10:

```
ex2 = do
  a <- return 1
  b <- Cont (\fred -> fred 10)
  return $ a+b

test2 = runCont ex2 show
```

The entire computation is applied to 10 and we get `"11"`. That's still a convoluted way of doing things. What other advantages do we get? Well the expression for `b` can do whatever it wants with `fred` as long as it returns the same type, ie. a string. So we can write this:

```
ex3 = do
  a <- return 1
  b <- Cont (\fred -> "escape")
  return $ a+b

test3 = runCont ex3 show
```

`fred` is completely ignored. The entire computation is thrown away and instead of applying `show` to a number, we simply return `"escape"`. In other words, we have a mechanism for throwing values out of a computation. So continuations provide, among other things, an exception handling mechanism. But that's curious, because that's exactly what the `Maybe` monad provides. It looks like we might be able to simulate `Maybe` this way. But rather than do that, let's do something even more radical:

```
ex4 = do
  a <- return 1
  b <- Cont (\fred -> fred 10 ++ fred 20)
  return $ a+b

test4 = runCont ex4 show
```

We've used `fred` twice. We've made the code around our "hole" run

twice, each time executing with a different starting value. Continuations allow mere subexpressions to take complete control of the expressions within which they lie. That should remind you of something. It's just like the list monad. The above code is a lot like

```
test5 = do
  a <- return 1
  b <- [10,20]
  return $ a+b
```

So can we emulate the list monad? Well instead of converting our integer to a string at the end we want to convert it to a list. So this will work:

```
ex6 = do
  a <- return 1
  b <- Cont (\fred -> fred 10 ++ fred 20)
  return $ a+b

test6 = runCont ex6 (\x -> [x])
```

We can avoid those ++ operators by using `concat`:

```
ex7 = do
  a <- return 1
  b <- Cont (\fred -> concat [fred 10,fred 20])
  return $ a+b

test7 = runCont ex7 (\x -> [x])
```

But now you may notice we can remove almost every depepndence on the list type to get:

```
ex8 = do
  a <- return 1
  b <- Cont (\fred -> [10,20] >>= fred)
  return $ a+b

test8 = runCont ex8 return
```

Note, we're using monad related functions, but when we do so we're not using do-notation. We can now do one last thing to tody this up:

```
i x = Cont (\fred -> x >>= fred)
run m = runCont m return
```

And now we have something close to do-notation for the list monad at our disposal again:

```
test9 = run $ do
  a <- i [1,2]
  b <- i [10,20]
  return $ a+b
```

I hope you can see how this works. `i x` says that the continuation should be applied to `x`, not as an ordinary function, but with `>>=`. But that's just business as usual for monads. So the above should work for any monad.

```
test10 = run $ do
  i $ print "What is your name?"
  name <- i getLine
  i $ print $ "Merry Xmas " ++ name
```

The continuation monad really is the mother of all monads.

There are some interesting consequences of this beyond Haskell. Many languages with support for continuations should be extensible to support monads. In particular, if there is an elegant notation for continuations, there should be one for monads too. This is why I didn't want to talk about the underlying mechanism of the `Cont` monad. Different languages can implement continuations in different ways. An extreme example is (non-portable) C where you can reify continuations by literally flushing out all registers to memory and grabbing the stack. In fact, I've used this to implement something like the list monad for searching in C. (Just for fun, not for real work.) Scheme as `call-with-current-continuation` which can be used similarly. And even Python's `yield` does something a little like reifying a continuation and might be usable this way. (Is that's what's going on here.? I haven't read that yet.

1 Afterword (June 2024)

At the end I mentioned an article on monads in Python. The idea that a language with something like continuations could allow the imple-

mentation of monads led me to think you couldn't implement something like a probability monad in Python. That idea eventually became `JointDistributionCoroutine` in TensorFlow probability.