# CS4012 Topics in Functional Programming

Glenn Strong

# Monads

- A digression?
  - (though really, this will turn out to be fundamental structuring idea for code!)
  - So not really a digression at all
- Monads

## Monads

Last year you encountered the type `IO a` as a mechanism for dealing with certain kinds of computations. The type represents functions which are *actions* that peform side-effects.

The word "monad" was used to refer to this kind of abstraction

What's going on with these, and how do they fit in here?

## The problem of IO

Let's remind ourselves of the issues around IO. Imagine we have functions such as:

```haskell
primGetChar :: Char
primPutChar :: Char -> ()
```

For these functions to be meaningful they would be performing some side-effecting IO operations whenever they are evaluated.

A clear violation of referential transparency!

### Violating referential transparency!

It doesn't take much to see the problem. Do we know what this will do:

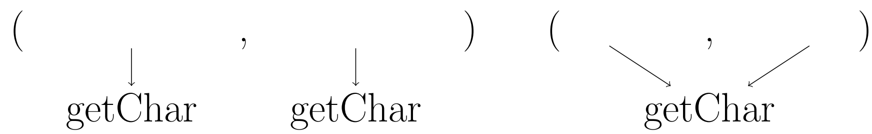
`f1 = (primGetChar,primGetChar)`

How about this:

`f2 = let x = primGetChar in (x,x)`

### Violating referential transparency!

If we draw the graphs of `f1` and `f2` we can see the problem.




**Solution**

One solution to this problem is to require some sort of “token” which will enforce the evaluation we want. We can wrap up the (unsafe) `primGetChar` function in a (safe) function that takes into account the side effects:

```
getChar :: World -> (Char,World)
```

**Solution**

The “World” here is a parameter which represents the state of the world from one moment to another

The actual details of how that is encoded, and of how `getChar` might use it are not important right now, but we assume that it is not possible to make copies of the world!

**Solution**

Now we can write our hazardous function differently:

```
f3 w = ( w2 , (ch1,ch2) )
       where
         (ch1, w1) = getChar w
         (ch2, w2) = getChar w1
```

Having to “thread” the various `w` parameters forces the evaluation to happen the way we want, as long as we are careful never to make more than one reference to any given state of the world.

**Solution**

We need to ensure that this sort of thing never happens:

```
f3 w = ( w2 , (ch1,ch2) )
       where
         (ch1, w1) = getChar w
         (ch2, w2) = getChar w
```

We would also like to make it a bit easier to write functions that use this style (manually threading the "w"'s around will get tedious quickly).

### Structure

If we make sure all our "dangerous" functions have this shape:

```
World -> (a,World)
```

we can declare a type to capture this:

```
type IO a = World -> (a,World)
```

(again, not worrying about what `World` actually is at this time)

### Structure

We get some very familiar looking types now:

```
getChar :: IO Char
putChar :: Char -> IO ()
```

### Structure

Now we can hide all the "plumbing" away in a function:

```
(>>) :: IO a -> IO b -> IO b
```

Used like this:

```
f4 = (putChar 'a') >> (putChar 'b')
```

(do the first thing, throw away the result, but keep the `World`, then do the second thing)

### Structure

A possible implementation for `>>`

```
(>>) l r = \w -> let (w1,_) = l w in r w1
```

### Structure

If the result is significant then instead of throwing it away we could keep it:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

in fact, if we have this then `>>` is easy to write:

```
(>>) l r = l >>= (\_ -> r)
```

#### Structure

I'll add one more thing - a computation which does nothing and produces a result of the correct type:

```
return :: a -> IO a
```

#### Example

```
f = getChar >>= ( \ ch1 ->
     getChar >>= ( \ ch2 ->
     return (ch1,ch2) ) )
```

### The IO Monad

What we've seen is a possible implementation for the IO monad.

### Monads

A monad is an abstraction which represents a computation. The computations have results (reflected in the type). The monad provides at least the basic operations:

- `return` produces a result
- `>>=` which binds together two computations.

Generally a monad will also provide a collection of primitive operations (like `getChar`) to make it useful.

### Sugar

Programs written in a monadic style will typically contain long chains of `>>` and `>>=` operations. Haskell provides some syntactic sugar, called the "do-notation" that allows us to write the previous program as follows:

```
f5 = do
  ch1 <- getChar
  ch2 <- getChar
  return (ch1,ch2)
```

### Sugar

There is a mechanical translation from the do-notation form to the combinator form, which we can summarize:

```
  do x
     y
=
  x >> do y

do a <- x
   y
=
  x >>= \a -> do y

do x = x
```