Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Week 2
Monads

CS4012
Topics in Functional Programming

Glenn Strong <glenn.Strong@scss.tcd.ie>

Monads

- **Certain patterns of computation come up over and over**
- **I want to look at a way to structure programs that can solve lots of problems for us.**
- **You've already seen this, but not from the ground up…**

Monads

- Last year you encountered the IO a type as a mechanism for dealing with certain kinds of computations.
- Specifically…
- Functions that represent *actions which have side-effects*.

- The word “*monad”* was used to refer to this abstraction
- What’s going on with these functions, and how do they fit into my claim that we have a useful way to structure programs?

Monads

The problem of IO

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

```
primGetChar :: Char
primPutChar :: Char → ()
```

- For these functions to be meaningful they would have to be performing some side-effecting IO operations whenever they are evaluated.
- That's clearly going to violate the principle of referential transparency, and make us sad!

Monads

Referential transparency?

- It won't take much to illustrate the problem.
- Do we know what this will do?

```
f1 = (primGetChar, primGetChar)
```

- How about this?

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

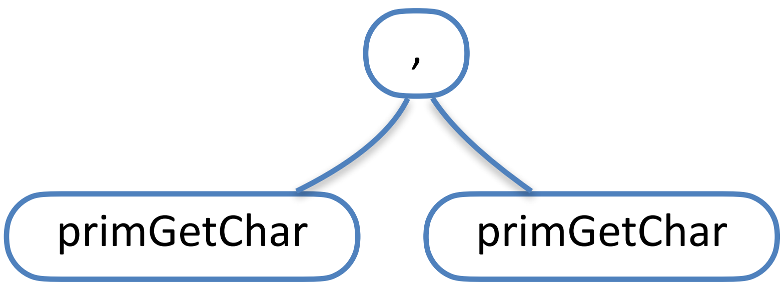
Monads

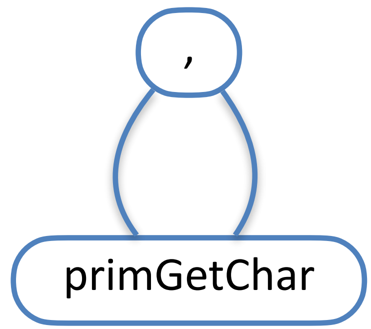
Referential transparency?

- **If we draw the graphs for the two expressions we can see the problem**

```
f1 = (primGetChar, primGetChar)
```

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





- **In the right-hand case we will get only one IO action performed, while on the left there will be two. But the expressions were supposed to be the same!**
- **Side-effects really mess things up…**

Monads

Token passing

- I’m going to propose a solution.
- We want the two uses of “primGetChar” to be different somehow, so I will add an argument to the function, and use that to distinguish them.

```
getChar :: World → (Char,World)
```

- The (unsafe) primGetChar will be wrapped up in this (safe) function that takes the side-effects into account.

Monads

World?

- 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.
- We will just assume that it is not possible to make copies of the World, nor to magic-up a World out of nowhere.

Monads

State passing

- **With this idea in place we must write our hazardous function differently:**

```
f3 :: World → ((Char, Char), World)
f3 w0 = ((ch1, ch2), w2)
       where
         (ch1, w1) = getChar w0
         (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 moment in the state of the world!**

Monads

State passing

- **We need to ensure that this sort of nonsense never happens:**

```
f3 :: World → ((Char, Char), World)
f3 w = ( (ch1, ch2), w2 )
       where
         (ch1, w1) = getChar w
         (ch2, w2) = getChar w
```

- **While we are at it, we would like to make it easier to write this style of function**
- **Manually “threading” those various “w”s around will get tedious very quickly!**

Monads

Structured state passing

- **Let's assume that all our world-mangling functions have this "shape":**

```
f :: World → (a, World)
```

- **We can declare a type that captures this, which will clean up our code a bit.**

```
type IO a = World → (a, World)
```

- **Again, not really worrying about what World actually is at this time**

Monads

Structured state passing

- **Then we get some types that will look rather familiar to you**

```
getChar :: IO Char
putChar :: Char → IO ()
```

Monads

Structured state passing

- The second thing we can do is hide all the "plumbing" involved in threading the state.

- For example, if I declare an (infix) function with this type:

```
(>>) :: IO a → IO b → IO b
```

- Used like this:

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

- Read it as "do the first thing, throw away the result but keep the World, then do the second thing".

- The results of printing are all just "()" values so there's nothing of interest dropped.

Monads

Structured state passing

- **A possible implementation for (>>)**

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

Monads

Structured state passing?

- If the result is significant then instead of throwing it away we can keep it.
- Our first thought might be to have some function like this:

```
pair :: IO a → IO b → IO (a,b)
```

- But that's not really the right structure.
- For one thing, that type doesn't really say that the left thing happens *before* the right thing. If we wrote "pair" to do it in the opposite order that would still be the type!
- We will often want to do something in the second action that makes use of the result of the first action

Monads

Structured state passing

- So we'll really do something like this:

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

- In fact, if we have this then it's easy to write >>

```
(>>) l r = l >>= (\_ → r)
```

Monads

Structured state passing

- To write useful programs I'll add one utility function
- A computation that does nothing (has no side effect) but which produces a value

```
return :: a → IO a
```

- This is useful if we want to do combine IO and non-IO computations.

Monads

Using the monad functions

- **Let's see this in use**
- **Here's a function that reads two characters in order, then returns them as a string**

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

Monads

The IO Monad

- **What’s we’ve seen is a possible implementation for the IO monad in Haskell.**

Monads

Monads in general

- A monad is an abstraction which represents a computation.
- The computations have results (reflected in the type).
- The monad provides at least two basic operations:
  - return, which produces a result without doing anything
  - >>=, which binds together two computations in the monad
- Of course a monad will also provide a collection of primitive operations (like getChar), in order to be useful.

Monads

Syntactic sugar

- One thing with this style of programming is that you can end up with long chains of >>= and >> operations.
- Haskell provides some syntactic sugar, called the “do-notation”, that allows us to write the previous program like this:

```
f = do
      ch1 ← getChar
      ch2 ← getChar
      return [ch1,ch2]
```

Monads

Syntactic sugar

- **The notation is automatically de-sugared into the combinatory form.**
- **A summary of the rules:**

```
do x
   y
```

→

```
x >> y
```

```
do a ← x
   y
```

→

```
x >>= \a → do y
```

```
do x
```

→

```
x
```

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Thank you

glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/