Introductory Haskell, polyconf'15, Alexander Ulrich

[Partially based on material by Torsten Grust]

Monads

Prerequisites: type classes (including class and instance definition), `Maybe`.

Plan

Monad is propably the word most often associated with Haskell.

So we ask: *What is a monad?*

"A monad is just a monoid in the category of endofunctors, what's the problem?"

This is technically (propably) correct, but not exactly helpful. Involving category theory at this point is not what we need. And by the way, the quote is actually not from Phil Wadler.

What, then is a monad? There is a vast number of tutorials that try to answer the question.

Show https://www.haskell.org/haskellwiki/Monad_tutorials_timeline

A lot of them use questionable metaphors and try to tell you that a monad is somewhat like a *burrito*. Or a *space suit*. Or a *sock*. Or whatever.

Here's a better question: *How can we model effectful computations (e.g. computations that have side effects, do IO) in a pure language*?

Here's the plan: We'll look at two simple examples and try to generalize what we find there. By abstracting, we will obtain the notion of a monad.

Maybe

We saw the `Maybe` type earlier. `Maybe` models the case that we either have a value or not have a value. Put another way, a function that returns a `Maybe` models a computation that might go wrong.

Let's look at a very simple example of computations that might go wrong.

Task: map english words that represent digits to chinese numerals.

We already have three functions that implement the meat of the task.

1. `numeralToDigit` maps a digit word to a the corresponding character. It might fail (word does not represent a digit), therefore returns a `Maybe`.

2. `digitToVal`: Maps a digit character to its numeric value. Might fail (character is not a digit).
3. `chineseNumeral`: Map a number to the corresponding chinese numeral. Might fail (number is not a single digit).

We can implement the task by combining those three functions (of course, that's not the best implementation).

We write a function that represents the composition of the three functions:

```
chinese :: String -> Maybe Char
chinese x = case numeralToDigit x of
                Just n -> case digitToVal n of
                              Just o -> chineseNumeral o
                              Nothing -> Nothing
                Nothing -> Nothing
```

For each function, we have to account for the possibility of failure. That makes the code rather awkward to read and write. What we need, is a clean way to *sequence/chain computations that might fail.* In the case of failure, the chain should be short-circuited.

Define an operator that chains two computations and does the boring work of checking for failure.

```
seqMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b
seqMaybe (Just a) f = f a
seqMaybe Nothing  _ = Nothing
```

Chinese conversion now becomes easy, because we have hidden all the boilerplate.

```
chinese' :: String -> Maybe Char
chinese' w = numeralToDigit w `seqMaybe` digitToVal `seqMaybe` chineseNumeral
```

State

How can we generate pseudo random numbers in Haskell?

```
> import System.Random
> :t random
```

Let's get some random numbers:

```
> g <- getStdGen
> fst (random g) :: Int
> fst (random g) :: Int
```

Crap! But it's rather clear that this would happen. As every Haskell function, `random` is pure (or *referentially transparent*). It can only take its parameter (the random generator) into account. If we pass in the same generator twice, we get the same result. `random` gives us back a *new* generator, from which we can get a new number.

```
> let (i, g') = random g :: (Int, StdGen)
> i
> fst (random g') :: Int
```

Functions that work with random numbers have to look like this:

```
addRand :: Int -> StdGen -> (Int, StdGen)
addRand i g = let (i', g') = random g in (i + i', g')

subRand :: Int -> StdGen -> (Int, StdGen)
subRand i g = let (i', g') = random g in (i - i', g')
```

Let's write a function that combines several functions with random numbers (take a number, add two random numbers and subtract one.

```
doRandomStuff :: Int -> StdGen -> (Int, StdGen)
doRandomStuff i g = let (i', g') = addRand i g
                        (i'', g'') = addRand i' g'
                    in subRand i'' g''
```

Again, sequencing of random functions involves lots of boilerplate and is awkward.

Let's at least make the types nicer: A random computation takes a generator and returns a value and a new generator.

```
type Rand a = StdGen -> (a, StdGen)
```

change types of functions to Rand

We might also say: `Rand` represents computations that have a state - namely the generator. When a random number is obtained, the state is updated.

Could we hide the boilerplate involved in sequencing just as in the `Maybe` example? Let's define a sequencing operator. This one is a bit more tricky.

```
seqRand :: Rand a -> (a -> Rand b) -> Rand b
seqRand ra f = \g -> let (a, g') = ra g
                     in f a g'
```

Again, we hide the boilerplate by using the sequencing operator:

```
doRandomStuff' :: Int -> StdGen -> (Int, StdGen)
doRandomStuff' i = addRand i `seqRand` addRand `seqRand` subRand
```

Also, we can define the most basic random computation: the one that actually gets a new random value:

```
getRandom :: Random a => Rand a
getRandom = random
```

In addition, we define another function that injects a regular value into a randomness context. You give me a value and I give you a random computation that returns this value.

```
simply :: a -> Rand a
simply a = \g -> (a, g)
```

With those, we can also reformulate the functions that actually obtained random numbers.

```
addRand' :: Int -> Rand Int
addRand' i = getRandom `seqRand` \i' -> simply (i + i')
```

Abstraction

Let's look at the types of the sequencing functions again.

```
seqMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b
seqRand  :: Rand  a -> (a -> Rand  b) -> Rand  b
```

They are identical, except for the type constructor `Rand` or `Maybe`. We have obtained the same structure from two very different scenarios - a sure sign that something principled is going on. This calls for abstraction. We abstract over the type constructor:

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

But this does not make sense for all type constructors `m`, just for a subset of all possible type constructors. So we define a class:

```
class EffectfulComputation m where
    (>>=) :: m a -> (a -> m b) -> m b
```

Now we get a name conflict for this strangely named operator, which is apparently already defined in the Haskell standard library.

```
> :info (>>=)
```

Actually, we are about to re-define the monad type class! Let's take a look at that:

```
> :info Monad
```

There is the bind operator (`>>=`) for sequencing two computations and there is a `return` function, which is the same as the `simply` function we defined for `Rand`: it gives a computation that simply returns a value.

Attention: Do not confuse the `return` combinator with the `return` keyword in imperative languages.

Interestingly, `Maybe` is already an instance of `Monad`. So we can just write:

```
chinese' :: String -> Maybe Char
chinese' w = numeralToDigit w >>= digitToVal >>= chineseNumeral
```

Question: How is `return` defined in the `Monad Maybe` instance?

monad instance for Rand

For technical reasons, we have to make the type alias `Rand` stronger by defining a proper data type.

```
data Rand a = R (StdGen -> (a, StdGen)

instance Monad Rand where
    return a = R $ \g -> (a, g)
    (R ra) >>= f = R $ \g -> let (a, g') = ra g
                                 R f'    = f a
                             in f' g'
```

Remark 1: If we abstract over the state that we use (it does not have to be a random number generator), we get the state monad.

Remark 2: `IO` is also an instance of `Monad`! Monads are how we sequence multiple computations that can do IO.

Syntactic Sugar

Monads are omnivalent in Haskell. Therefore, the language offers special syntactic sugar for sequencing monadic computations.

```
addRand :: Int -> Rand Int
addRand i = do
    j <- getRandom
    return (j + i)

doRandomStuff :: Int -> Rand Int
doRandomStuff i = do
    j <- addRand i
    k <- addRand j
    subRand k

echo :: IO String
echo = do
    l <- getLine
    putStrLn l
    return l
```