1 Monads

1.1 Writing Apps with Monads

Why is it so hard to write a program that prints "`Hello world!`" in Haskell?

Note that Haskell is *pure*. A program is an expression that evaluates to a value and does *nothing else*. So, a function of type `Int -> Int` computes a *single integer output* from a single integer input and does *nothing else*. Moreover, it always returns the same output given the same integer. Specifically, evaluation must not have any side effects.

Haskell has a special type called `IO`, which we can think of as a `Recipe`.

```
type Recipe a = IO a
```

So, when executing a program, Haskell looks for a special value

```
main :: Recipe ()
```

This is a Recipe for everything a program should do, and does not return a special value.

Note that

- A function of type `Int -> Int` still computes a single integer output from a single integer input and does nothing else.
- A function of type `Int -> Recipe Int` computes an Int-recipe from a single integer input and does nothing else.
- Handing this recipe to `main` will possibly result in these "side effects."

So, writing "`Hello World`" is as simple as

```
main :: Recipe ()
main = putStrLn "Hello, world!"
```

Note that `putStrLn` has the following definition

```
putStrLn :: String -> Recipe ()
```

Compiling and running like so gives us

```
$ ghc hello.hs
$ ./hello
Hello, world!
```

How would we have multiple print statements, though?

(Quiz.) Suppose we have a function `combine` that lets us combine recipes like so:

```
main :: Recipe ()
main = combine (putStrLn "Hello,") (putStrLn "World!")
```

(a) `() -> () -> ()`

(b) `Recipe () -> Recipe () -> Recipe ()`

(c) `Recipe a -> Recipe a -> Recipe a`

(d) `Recipe a -> Recipe b -> Recipe b`

(e) `Recipe a -> Recipe b -> Recipe a`

The answer depends. For this particular example, **B** is the answer. However, this could be generalized, which we will discuss later.

1.1.1 Using Intermediate Results

Suppose we wanted to write a program that

- asks for the user's `name` using

```
getLine :: Recipe String
```

- prints out a greeting with that name using

```
putStrLn :: String -> Recipe ()
```

How do we pass the output of the first recipe into the second recipe?

(Quiz.) Suppose you have two recipes.

```
crack       :: Recipe Yolk
eggBatter   :: Yolk -> Recipe Batter
```

and we want to get

```
mkBatter :: Recipe Batter
mkBatter :: crack `combineWithResult` eggBatter
```

What should the type of `combineWithResult` be?

(a) `Yolk -> Batter -> Batter`

(b) `Recipe Yolk -> (Yolk -> Recipe Batter) -> Recipe Batter`

(c) `Recipe a -> (a -> Recipe a) -> Recipe a`

(d) `Recipe a -> (a -> Recipe b) -> Recipe b`

(e) `Recipe Yolk -> (Yolk -> Recipe Batter) -> Recipe ()`

The answer is **D**. Note that B is a more specific case of D. As noted, this is similar in nature to the type of `>>=` (bind).

In fact, since `Recipe`s are `Monad`s, we can do

```
main :: Recipe ()
main = getLine >>= \name -> putStrLn ("Hello, " ++ name ++ "!")
```

or (in a more procedural style),

```
main :: Recipe ()
main = do
          name <- getLine
          putStrLn ("Hello, " ++ name ++ "!")
```

Now, expanding on this program to ask the user for the name multiple times gives us

```
doQuit :: Recipe a
doQuit =  exitWith ExitSuccess

putStrFlush :: String -> Recipe ()
putStrFlush str = do
    putStr str
    hFlush stdout

main :: Recipe ()
main = do
            putStrFlush "Your name: "
            name <- getLine
            if name == ":quit"
                then doQuit
                else do
                    putStrLn ("Hello, " ++ name ++ "!")
                    main
```