Haskell
Category Theory & Monads

...

Sam Dowling

Investigating Monadic Composition & Category Theory using Haskell.

Sam Dowling

Presentation Overview

1. Haskell

1. Category Theory

1. Monads

1. The Problem

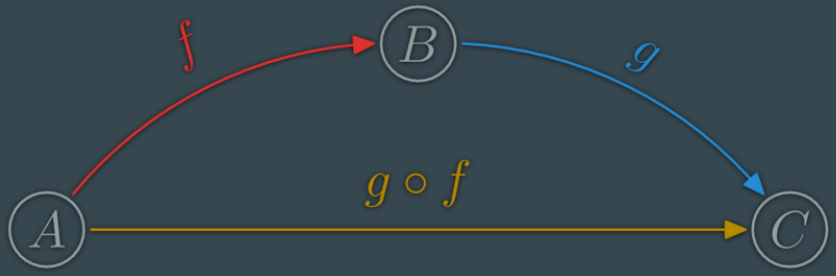
1. Difficulties Encountered

What is Haskell?

1. Haskell is a computer programming language.
2. It is a polymorphically statically typed, lazy, purely functional language.
3. Haskell programs are also shorter, clearer, and the rigorous control of side effects eliminates a lot of potential problems at compile time.

Sam Dowling

Category Theory 101



A way of representing ***things*** and ***ways to go between things***.

A *category* has three things:

1. A collection of ***objects***.
2. A collection of ***morphisms***.
3. A notion of ***composition*** of these morphisms.

Sam Dowling

What's A Monad?

The Long Answer:

A Monad in X is just a Monoid in the Category of Endofunctors of X, with product × replaced by Composition of Endofunctors and unit set by the Identity Endofunctor.

What’s A Monad?

The Short Answer:

It's a specific way of chaining operations together while observing a set of rules.

What's so good about Monads?

Reduce code duplication

Remove Side-Effects

Hide complexity

Encapsulate implementation details

Allow composability

The naïve way

This is an example of a Badly designed Function.

Dividing by Zero should not be possible.

There is no way to know whether the result is Zero or if you divided by Zero.

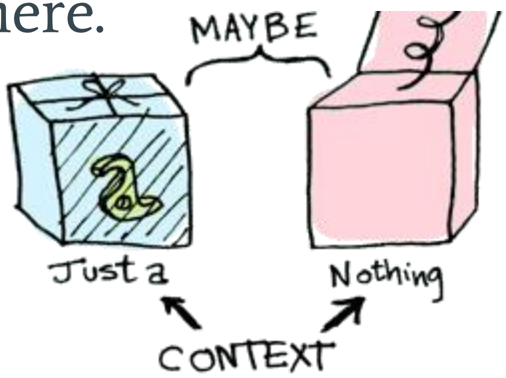
```haskell
divide :: Int -> Int -> Int
divide x 0 = 0
divide x y = quot x y

{-
- divide 10  5 = 2
- divide 10 11 = 0
- divide 10  0 = 0
-}
```

The Maybe Monad

```
Data Maybe a = Just a | Nothing

instance Monad Maybe where
    return x = Just x
    Nothing >>= f = Nothing
    Just x >>= f   = f x
    fail _ = Nothing
```

The Maybe Monad can Either have Something (Just) or Nothing.

It allows the programmer to specify something may not be there.



Monads to the Rescue!

The previous example has been upgraded with A Maybe.

Instead of returning Zero when dividing by Zero; It returns Nothing.

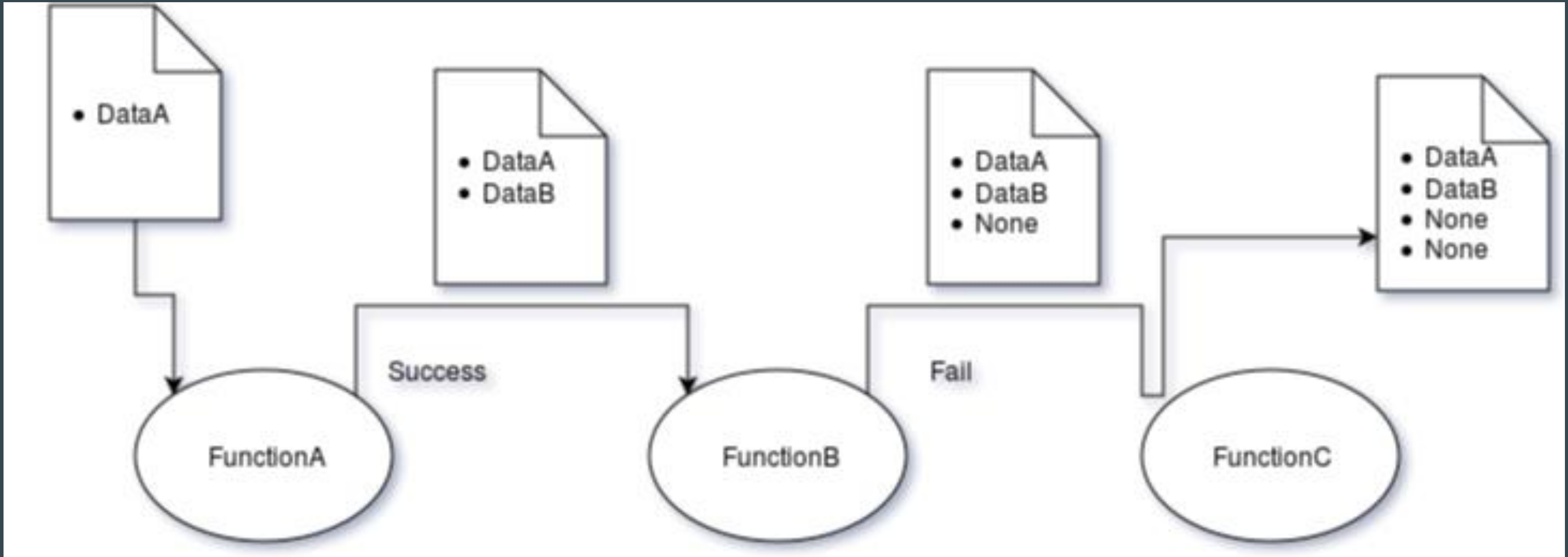
```haskell
divideM :: Int -> Int -> Maybe Int
divideM x 0 = Nothing
divideM x y = Just (quot x y)
{-
- divideM 10  5 = Just 2
- divideM 10 11 = Just 0
- divideM 10  0 = Nothing
-}
```

Monadic Composition

The purpose of this program is to subtract 1 from a given Positive number, while making sure the number stays Positive (> 0)

```haskell
type Positive = Int

subOne :: Positive -> Positive
subOne = (subtract 1)

check :: (Positive -> Positive) -> Positive -> Maybe Positive
check f n
    | f n > 0 = Just (f n)
    | otherwise = Nothing

safeSubOne :: Positive -> Maybe Positive
safeSubOne = check subOne

{-
- Just 2 >>= safeSubOne = Just 1
- Just 2 >>= safeSubOne >>= safeSubOne = Nothing
-}
```

The Problem



• DataA
• DataA
• DataB
• DataA
• DataB
• None
• DataA
• DataB
• None
• None
Success
Fail
FunctionA
FunctionB
FunctionC

My Solution: Monads!

1. First we create our data types

2. Then we create our Monad, which contains a list of MyData

3. We then create our functions A, B and C all of which take a list of MyData and returns a DataM Monad

4. next we define our Monadic Operator >=>

```
data MyData = DataA | DataB | DataC | None deriving Show

data DataM = Success [MyData] | Failure [MyData] deriving Show

functionA :: [MyData] -> DataM
functionA x = Success $ DataB : x -- Success!

functionB :: [MyData] -> DataM
functionB x = Failure $ None : x -- This Function Fails!

functionC :: [MyData] -> DataM
functionC x = Success $ DataC : x -- Success!

(>=>) :: DataM -> ([MyData] -> DataM) -> DataM
x >=> f = case x of
             Success (x) -> f x
             Failure (x) -> Failure $ None : x

run = Success [DataA] >=> functionA >=> functionB >=> functionC
-- Failure [None,None,DataB,DataA]
```

Encountered Problems

- Due to Haskell’s purity there are strict regulations on I/O which makes writing I/O heavy applications in Haskell very time-consuming.

- Gentle entry-level tutorials are few and hard to find & explanations are largely math based, so the initial learning curve is pretty steep.

More Examples?

All research available online at https://github.com/Sam-Dowling/HaskellResearch

- Catamorphisms
- Concurrency
- Conduits
- Error Handling
- I/O
- Monoids
- Orderings
- Persistence
- Database Interactivity
- Mutable State
- Natural Transformations
- Parallelism
- Kleisli Categories

The End

Thank you for your time.

All research availiable online at
https://github.com/Sam-Dowling/HaskellResearch