Functional Programming and Monads in Haskell

Daniel Echtler

GRK 2240 Retreat
September 2026

Heinrich-Heine-Universität Düsseldorf

Question

How are functions in mathematics different from functions in programming?

Answer

Functions in mathematics do *not* have a *hidden state* or *side effects*!

Example

This Python function always returns the argument of the *previous* function call.

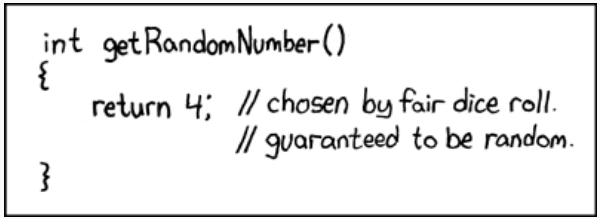
```
1  last_arg = None
2  def last(x):
3      global last_arg
4      result, last_arg = last_arg, x
5      return result
```

Question

Are side effects inherently bad?

Answer

No, side effects allow for manipulating data “in-place”, user interaction, writing files, randomness, ...



Source: xkcd.com/221

However, when formally verifying that a program works as it is supposed to, side effects make things messy. Here it can be useful to use a *purely functional* programming language, where functions are more like mathematical functions and (usually) have no side effects. One such example is *Haskell*.

Some History

1985 Miranda, one of the first commercial purely functional programming language, is released.

1987
- More than a dozen functional programming lanugages exist.
- The *International Conference on Functional Programming* takes place in Portland.
- There needs to be an open standard and a common language for future research.

1990 The first version of ~~Curry~~ Haskell is defined, named after mathematician *Haskell Curry*.



Source: wiki.haskell.org

Haskell Curry



Source: commons.wikimedia.org

Tim Curry



Source: en.wikipedia.org

Dr. Frank-N-Furter

Some code and advantages!

The Glasgow Haskell Compiler (ghc) provides an interactive environment (ghci).

Live Demo in GHCi

Advantages of functional programming

- Verifiability of programs
- Easy parallelization
- *Lazy* evaluation
- *Theorems for free!* (Philip Wadler, 1989)
- Compiler optimization

Some Mathematics!

Define the category **Hask** as follows:

Objects are all Haskell types (`Int`, `Bool`, `Float`, `[Bool]`, `[Float] -> Int`, ...).

Morphisms between two Haskell types `A`, `B` are Haskell functions `A -> B`.

Question

Why should we consider this category?

Answer

Functors, (co)limits, universal properties, ...

Example (Products and coproducts)

The product of two types `A` and `B` in **Hask** is given by `(A, B)`.

The coproduct is given by `Either A B` where `data Either a b = Left a | Right b`.

More on (co)products

The universal properties of the product an coproduct are essential for writing programs.

```
fun :: A -> (B, C)
fun x = (f x, g x)
```

```
fun :: Either A B -> C
fun (Left  x) = f x
fun (Right y) = g y
```

Thus Haskell provides tools for such constructions.

```
(&&&)  :: (a -> b) -> (a -> c) -> a -> (b, c)
either :: (a -> c) -> (b -> c) -> Either a b -> c
```

Both the product and coproduct are functorial in both variables. In Haskell this is expressed as

```
instance Functor (Either a) where
  -- fmap :: (b -> c) -> Either a b -> Either a c
  fmap f (Left  x) = Left x
  fmap f (Right y) = Right (f y)
```

Dealing with failure: Maybe

In programming, functions can fail for various reasons. In Haskell we could use **`Either String`** to keep track of errors and error messages. The simpelest solution however is to use **`Maybe`**.

```haskell
type Maybe a = Nothing | Just a    -- essentially Maybe = Either ()

safeHead :: [a] -> Maybe a
safeHead []    = Nothing
safeHead (x:_) = Just x
```

In fact, `safeHead` is a natural transformation **`[]`** $\to$ **`Maybe`**. This can be checked by hand, but follows from *Theorems for free!* as well.

The functor instance of **`Maybe`** allows us to compose possibly failing functions with usual functions from both sides.

```haskell
fmap (\x -> x^2) . safeHead :: [Int] -> Maybe Int
safeHead . fmap (\x -> x^2) :: [Int] -> Maybe Int
```

Failing twice

Question

How about composing with another function that can fail as well? Take for example

```
safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv x y = Just (x `div` y)
```

Then `fmap (safeDiv 2240) . safeHead` will be of type `[Int] -> Maybe (Maybe Int)`.

In the case of **Maybe** we can combine two **Maybe**s into one:

```
concatMaybe :: Maybe (Maybe a) -> Maybe a
concatMaybe Nothing  = Nothing
concatMaybe (Just x) = x
```

But for a functor **F** there will in general be no function of type `F (F a) -> F a`!

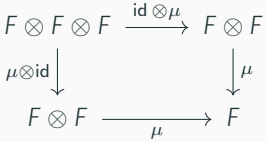
Back to Mathematics: Monoids and Monads

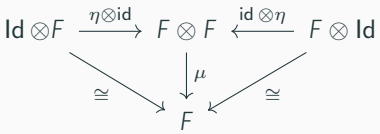
The category $\mathsf{End}(\mathsf{Hask})$, of endofunctors $\mathsf{Hask} \to \mathsf{Hask}$, is a monoidal category with

Tensor product given by composition of functors.

Unit object given by the identity functor $\mathsf{Id} : \mathsf{Hask} \to \mathsf{Hask}$.

So a function `F (F a) -> F a` is essentially a morphism $\mu : F \otimes F \to F$ in $\mathsf{End}(\mathsf{Hask})$. Together with another morphism $\eta : \mathsf{Id} \to F$, making





commute, the functor F is a *monoid* in $\big(\mathsf{End}(\mathsf{Hask}), \otimes, \mathsf{Id}\big)$.

A monoid in the category of endofunctors of $\mathcal{C}$ is also known as a *monad* on $\mathcal{C}$.

Say we are given a functor `F` together with maps `eta :: a -> F a`, which in Haskell is usually called `return`, and `mu :: F (F a) -> F a` satisfying the monad laws. We can now apply a function of type `a -> F b` to a value of type `F a` using the *bind* operator

```
(>>=) :: F a -> (a -> F b) -> F b
v >>= fun = (mu . (fmap fun)) v
```

We can recover `mu` as `mu = (>>= id)`. Moreover we define the *Kleisli composition* as

```
(<=<) :: (b -> F c) -> (a -> F b) -> a -> F c   -- left fish
g <=< f = (\x -> f x >>= g)

(>=>) :: (a -> F b) -> (b -> F c) -> a -> F c   -- right fish
(>=>) = flip (<=<)
```

Examples of Monads in Haskell

```haskell
instance Monad Maybe where
  return x = Just x

  Nothing  >>= _ = Nothing
  (Just x) >>= f = f x
```

```haskell
instance Monad [] where
  return x = [x]

  []     >>= _ = []
  (x:xs) >>= f = f x ++ (xs >>= f)
```

To make composition of monadic functions easier, Haskell provides **do**-notation:

```haskell
safeHeadDiv :: [Int] -> Maybe Int
safeHeadDiv xs = do
  x <- safeHead xs
  safeDiv 2240 x
```

```haskell
twice :: [a] -> [a]
twice xs = do
  x <- xs
  [x, x]
```

These are essentially equivalent to

```haskell
safeHead xs >>= safeDiv 2240
```

```haskell
xs >>= (\x -> [x,x])
```

For now this might not seem like much of a simplification, but for more complicated expressions with multiple >>= operations, **do**-notation becomes way easier to read.

Monads and adjoint functors

One way of constructing monads is using adjoint functors: Let $F\colon \mathcal{C} \to \mathcal{D}$ be left-adjoint to $G\colon \mathcal{D} \to \mathcal{C}$. Moreover, denote by $\varepsilon\colon F \circ G \to \mathsf{Id}_{\mathcal{D}}$ and $\eta\colon \mathsf{Id}_{\mathcal{D}} \to G \circ F$ the counit and unit of the adjunction. Then $T = G \circ F$ together with η and

$$\mu\colon T \circ T = G \circ F \circ G \circ F \xrightarrow{\mathrm{id}_G \circ \varepsilon \circ \mathrm{id}_F} G \circ \mathsf{Id}_{\mathcal{D}} \circ F = G \circ F = T$$

is a monad on $\mathcal{C}$.

In fact every monad is of this form! Let $M = (T, \eta, \mu)$ be a monad on $\mathcal{C}$. The *Kleisli category* $\mathcal{C}_M$ consist of the same objects as $\mathcal{C}$ with morphisms are given by $\mathsf{Hom}_{\mathcal{C}_M}(X, Y) = \mathsf{Hom}_{\mathcal{C}}\big(X, M(Y)\big)$. The identities are given by $\eta_X\colon X \to M(X)$ and composition is the *Kleisli composition*. Then

$$\begin{aligned} F\colon \mathcal{C} &\longrightarrow \mathcal{C}_M \\ X &\longmapsto X \\ \big(f\colon X \to Y\big) &\longmapsto \eta_Y \circ f \end{aligned} \qquad \begin{aligned} G\colon \mathcal{C}_M &\longrightarrow \mathcal{C} \\ X &\longmapsto T(X) \\ \big(f\colon X \to T(Y)\big) &\longmapsto \mu_Y \circ T(f) \end{aligned}$$

are adjoint and the construction above recovers M.

Back to Haskell



Source: xkcd.com/1312

The IO monad

For input-output operations Haskell provides the **IO** monad. This can be thought of as

```
IO a = Universe -> (Universe, a)
```

So for example `putStr "Hello, World!" :: IO ()` is a function which takes the current universe and produces a parallel universe where `"Hello, World!"` was printed in our terminal.

This allows us to finally interact with the outside world!

```
greet :: IO ()
greet = do
  putStrLn "What is your name?"
  name <- getLine
  let greeting = "Hello " ++ name ++ "!"
  putStrLn greeting
```