An Invitation to Haskell

Emily Pillmore

March 21, 2023

Table of Contents

My name is Emily Pillmore.
I am a programmer, and a math enthusiast.

- Author/Maintainer of more than 30 packages, some bigger than others
- Served on the Haskell Core Libraries and .Org committees
- Twitter (@yandereidiot)
- Meetups in NYC: NY Homotopy Type Theory, NY Category Theory, and the NY Haskell User Group.
- All of my slides, general scribbles, research, and meetup content are hosted at cohomolo.gy.

If you ever want to talk math or programming, I'm around.

I helped start the Haskell Foundation and served on the executive leadership team as a duo (CTO) with Andrew Boardman (ED).

I now work at a company called **Kadena**, as the lead of the language, its ecosystem, and its execution layers.

Table of Contents

So what is Functional Programming?

So what is Functional Programming?

- A collection of features? (lambdas, first class HOF's, static type system...)

So what is Functional Programming?

- A collection of features? (lambdas, first class HOF's, static type system...)
- A programming style? (emphasis on recursion, "math", small static combinators, shunting as many errors to the compiler as possible)

So what is Functional Programming?

- A collection of features? (lambdas, first class HOF's, static type system...)
- A programming style? (emphasis on recursion, "math", small static combinators, shunting as many errors to the compiler as possible)
- A cult?

Table of Contents

In 1977, John Backus wrote everything we needed to know about FP.

Compositionality! Equational Reasoning! Sound foundational principles!

Table of Contents

Haskell builds a equational foundation on purity.

This means that functions may not have *side effects*. In conjunction with not allows side effects anywhere, this allows expressions to be completely deterministic, and therefore *referentially transparent*.

Table of Contents

Famously, Haskell has a type system.

Famously, Haskell has a type system.

- It has functions (read: function definitions, lambdas)

Famously, Haskell has a type system.

- It has functions (read: function definitions, lambdas)
- It has builtins (integers, IEEE floating points, machine words, characters etc.)

Famously, Haskell has a type system.

- It has functions (read: function definitions, lambdas)
- It has builtins (integers, IEEE floating points, machine words, characters etc.)
- It has generics

Haskell has a global notion of parametricity everywhere you want it which may be reasoned about equationally, and therefore free theorems you can reason about.

It has a form of ad-hoc polymorphism for generics called "Typeclasses".

For more, see:

- Wadler - Theorems for Free
- My talk - Type Arithmetic

Table of Contents

Since purity undermines destructive updates, our Haskell data structures are immutable. This creates some issues.

Since purity undermines destructive updates, our Haskell data structures are immutable. This creates some issues.

- The amortized complexity theory needed to talk about the best/average/worst case of operations goes out the window (your average case becomes your worst case for many operations).

Since purity undermines destructive updates, our Haskell data structures are immutable. This creates some issues.

- The amortized analysis (cheap small steps paying off a more expensive larger step) needed to talk about the best/average/worst case of operations goes out the window (your amortized cost becomes your worst case for many operations).
- Laziness (a limited form of mutation) turns out to be enough to recover amortized analysis

Since purity undermines destructive updates, our Haskell data structures are immutable. This creates some issues.

- The amortized analysis (cheap small steps paying off a more expensive larger step) needed to talk about the best/average/worst case of operations goes out the window (your amortized cost becomes your worst case for many operations).
- Laziness (a limited form of mutation) turns out to be enough to recover amortized analysis.
- This requires a different take on analysis (thunk counting techniques etc.) which causes you to think in a whole new paradigm.

Since purity undermines destructive updates, our Haskell data structures are immutable. This creates some issues.

- The amortized analysis (cheap small steps paying off a more expensive larger step) needed to talk about the best/average/worst case of operations goes out the window (your amortized cost becomes your worst case for many operations).
- Laziness (a limited form of mutation) turns out to be enough to recover amortized analysis.
- This requires a different take on analysis (thunk counting techniques etc.) which causes some tension.

Immutability + Laziness, though, is a super power. Friedman-Wise posed an important question back in 1976.

Inherently easy to spread about on multiple cores. With commutative, associative, and unital (see: commutative monoidal) functions, map-reduce is possible.

It also makes scheduling parallelism and concurrency a simpler.

Table of Contents

Table of Contents

In haskell, we write the definition for functions and constants like this:

In haskell, we write the definition for functions and constants like this:

```haskell
-- a function
square :: Int -> Int
square x = x ^ x
```

In haskell, we write the definition for functions and constants like this:

```haskell
-- a function
square :: Int -> Int -- a type signature
square x = x ^ x

-- Constants
pi_trunc :: Double
pi_trunc = 3.14159265359

charizard :: Char
charizard = 'c'

stringy :: String
stringy = "Hi, SEMIBUG!"
```

In haskell, we write the definition for functions and constants like this:

```haskell
-- builtin lists
oneTwoThree :: [Int]
oneTwoThree = [1,2,3]

-- builtin tuples
ab :: a -> b -> (a,b)
ab a b = (a,b)
```

One can also define functions of multiple inputs like so:

One can also define functions of multiple inputs like so:

```haskell
plus :: Int -> Int -> Int
plus x y = x + y

-- Int -> Int -> Int is the
-- same as Int -> (Int -> Int)
plus' :: Int -> Int -> Int
plus' x = \y -> x + y
```

Function types are defined syntactically by enclosing parens

Function types are defined syntactically by enclosing parens

```haskell
apply :: (a -> b) -> a -> b
-- apply = id
apply f a = f a

compose
  :: (b -> c)
  -> (a -> b)
  -> (a -> c)
compose f g = \a -> f (g a)
```

Defining infix notation is easy as well (not pictured: fixity):

```
(^+) :: Int -> Int -> Int
(^+) x y = x + y
-- usage: x ^+ y
```

Table of Contents

We can define data as follows:

```haskell
-- data DataType <tyvars>
--   = Case1
--   | Case2
--   | ...
data MyAdt = Thing1 | Thing2

data MyRec = MyRecordName
  { foo :: Int
    -- ^ foo :: MyRecordName -> Int
  , bar :: String
    -- ^ bar :: MyRecordName -> String
  }
```

```haskell
-- data DataType <tyvars>
--   = Case1
--   | Case2
--   | ...
data MyAdt = Thing1 | Thing2

data MyRec = MyRecordName
  { foo :: Int
    -- ^ foo :: MyRecordName -> Int
  , bar :: String
    -- ^ bar :: MyRecordName -> String
  }
```

Pattern matching is the means by which one destructs sum types.

```
let
  x :: MyDataType
  x = Case1

in case x of
  Case1 -> "hi!"
  Case2 -> "Death!"
```

Table of Contents

As mentioned in the beginning of the talk, immutable data structures let us achieve some remarkable properties.

We can write recursive data definitions for things like Lists:

We can write recursive data definitions for things like Lists:

```
data List a = Nil | Cons a (List a)
-- builtin: data [a] = [] | (:) a [a]
-- usage: Cons 1 (Cons 2 (Cons 3 Nil))
-- usage: [1,2,3] := 1:(2:(3:[]))
```

```haskell
reduce :: (a -> b -> b) -> b -> [a] -> b
reduce step accum lst = case lst of
  [] -> accum
  first:rest ->
    let stepped = step first accum
    in reduce step stepped rest
```

reduce is commonly referred to as a "fold". In fact, it's a *"right fold"* in the sense that the values are accumulated thusly: reduce (+) 0 [1,2,3] $1 + (2 + (3 + 0))$

reduce is commonly referred to as a "fold". In fact, it's a *"right fold"* in the sense that the values are accumulated thusly:

```
reduce (+) 0 [1,2,3]
-- ~ reduce (+) 0 (1:(2:(3:[])))
-- ~ 1 + reduce (+) 0 (2:(3:[]))
-- ~ 1 + (2 + (reduce (+) 0 (3:[])))
-- ~ 1 + (2 + (3 + reduce (+) 0 []))
-- ~ 1 + (2 + (3 + 0))
-- ~ 1 + (2 + 3)
-- ~ 1 + 5
-- ~ 6
```

Claim: this function corresponds with a kind of "canonical way to reduce a list recursively". As a result, one may define many interesting properties of a list in terms of this formulation.

```haskell
sum :: [Int] -> Int
sum lst = reduce (+) 0 lst

product :: [Int] -> Int
product lst = reduce (*) 1 lst

filter :: (a -> Bool) -> [a] -> [a]
filter p lst = reduce
  (\a acc -> if p a then a:acc else acc)
  [] lst

map :: (a -> b) -> [a] -> [b]
map f lst = reduce
 (\a acc -> (f a):acc) [] lst
```

```haskell
-- data Bool = True | False
all :: [Bool] -> Bool
all bs = reduce (&&) True bs

any :: [Bool] -> Bool
any bs = reduce (||) False bs
```

Graham Hutton's paper A tutorial on the universality and expressiveness of fold is a great resource on the subject.

These are definable in any language for any list. But in haskell, with the right foundation, they're understandable one-liners.

Further, we have laziness. This implies a thing called *shortcircuiting*.

Typeclasses are a means of layering ad-hoc behaviors.

```haskell
class Functor (f :: Type -> Type) where
  fmap :: (a -> b) -> f a -> f b

instance Functor List where
  -- fmap :: (a -> b) -> List a -> List b
  fmap f Nil = Nil
  fmap f (Cons h t) = Cons (f h) (fmap f h)
```

```haskell
functorFloor :: Functor f => f Double -> f Int
functorFloor dubs = fmap floor dubs

-- class Show a where show :: a -> String
stringify :: [Int] -> [String]
stringify = fmap show
```

```haskell
-- instances:
-- Int, <> = +, <> = *, etc.
class Semigroup a where
  (<>) :: a -> a -> a

-- instances:
-- Int, unit = 0, unit = 1
class Semigroup a => Monoid a where
  unit :: a
```

Table of Contents

Table of Contents

You can get everything you need from a tool called "ghcup".

GHCup is the canonical toolchain installer. It will install the LSP, the compiler, and the build tools.

Get it here: get-ghcup

Table of Contents

Cabal is the Haskell project structure spec (as well as other tools), and cabal-install is the tool we use to build Haskell projects.

Useful commands:

- init
- build
- repl
- test
- run
- publish

Dependencies are located by default in a community service called "Hackage". Cabal-install knows how to talk to this service.

Pragmatically, Haskell works like any other language:

Pragmatically, Haskell works like any other language:

- Define a main

Pragmatically, Haskell works like any other language:

- Define a main
- Do stuff in sequence

Pragmatically, Haskell works like any other language:

- Define a main
- Do stuff in sequence
- Exit

Table of Contents

Without spending too much on the dreaded "m-word" (it's monad), we have a monad called "IO".

IO is a means of sequencing real world state changes.

In fact, it is "just" a state monad, for the curious reader.

```haskell
main :: IO ()
main = putStrLn "Hello, World!"
```

Table of Contents

We’re out of time, but this is a teaser for you.

Takeaways:

Takeaways:

- Laziness + Purity = Performance + Reasoning

Takeaways:

- Laziness + Purity = Performance + Reasoning
- Reasoning + Consistency = Laws

Takeaways:

- Laziness $+$ Purity $=$ Performance $+$ Reasoning
- Reasoning $+$ Consistency $=$ Laws
- Laws $+$ Types $=$ Fewer Bugs

Takeaways:

- Laziness + Purity = Performance + Reasoning
- Reasoning + Consistency = Laws
- Laws + Types = Fewer Bugs
- Fewer Bugs = Less Stressful Programs

Haskell: lose your hair *before* you run the program.

...and then probably after too (it's still programming).