# An Invitation to Haskell

**Emily Pillmore** 

March 21, 2023

1

### Introduction

## Functional Programming

Foundations

Purity

Types

Consequences

### Examples

Basics

Data

Folds

### **Building Stuff**

Cabal

10

Conclusion

2

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).



5

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



#### Introduction

# Functional Programming

**Foundations** 

Purity

Types

Consequences

## Examples

Basics

Data

Folds

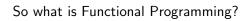
### **Building Stuff**

Caba

10

Conclusion

7



So what is Functional Programming?

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

9

## 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?

#### Introduction

# Functional Programming

**Foundations** 

Purity

Types

Consequences

## Examples

Basics

Data

Folds

### **Building Stuff**

Caba

10

Conclusion

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

Compositionality! Equational Reasoning! Sound foundational principles!

#### Introduction

# Functional Programming

Foundations

Purity

Types

Consequences

### Examples

Basics

Data

Folds

### **Building Stuff**

Caba

10

Conclusion



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*.

#### Introduction

## Functional Programming

Foundations

Purity

Types

Consequences

## Examples

Basics

Data

Folds

### **Building Stuff**

Cabal

10

Conclusion

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

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

- ▶ 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

#### Introduction

## Functional Programming

Foundations

Purity

Types

Consequences

## Examples

Basics

Data

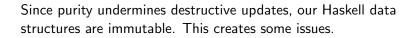
Folds

### **Building Stufl**

Caba

10

Conclusion



► 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).

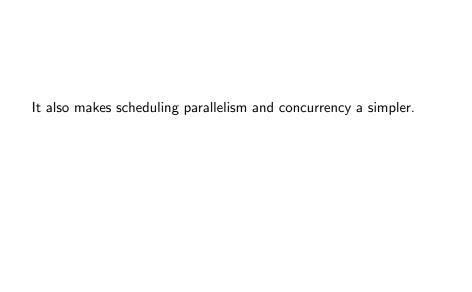
- The amortized analysis (cheap small steps paying off a more expensive larger step) needed to talk about the best/average/worst case of operations goes out the window (your amortized cost becomes your worst case for many operations).
- ► Laziness (a limited form of mutation) turns out to be enough to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

- 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.

- ► 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.



#### Introduction

## Functional Programming

Foundations

Purity

Types

Consequences

### Examples

**Basics** 

Data

Folds

### Building Stuff

Caba

10

Conclusion

#### Introduction

### Functional Programming

Foundations

Purity

Types

Consequences

## Examples

**Basics** 

Data

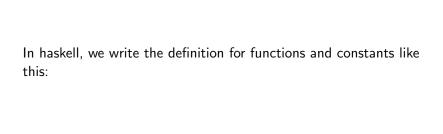
Folds

### **Building Stuff**

Caba

10

Conclusion



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

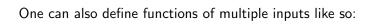
```
-- a function
square :: Int -> Int
square x = x ^ x
```

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

```
-- 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:

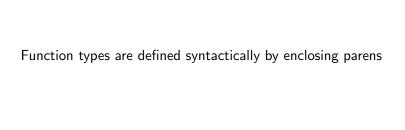
```
-- builtin lists
oneTwoThree :: [Int]
oneTwoThree = [1,2,3]
-- builtin tuples
ab :: a -> b -> (a,b)
ab a b = (a,b)
```



One can also define functions of multiple inputs like so:

```
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

44

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

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

# Table of Contents

#### Introduction

## Functional Programming

Foundations

Purity

Types

Consequences

## **Examples**

Basics

Data

Folds

#### **Building Stuff**

Caba

10

Conclusion

#### We can define data as follows:

```
-- data DataType <tyvars>
-- = Case1
-- | Case2
-- / . . .
data MyAdt = Thing1 | Thing2
data MyRec = MyRecordName
  { foo :: Int
    -- ^ foo :: MyRecordName -> Int
  , bar :: String
    -- ^ bar :: MyRecordName -> String
```

47

```
-- 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

#### Introduction

# Functional Programming

Foundations

Purity

Types

Consequences

# **Examples**

Basics

Data

Folds

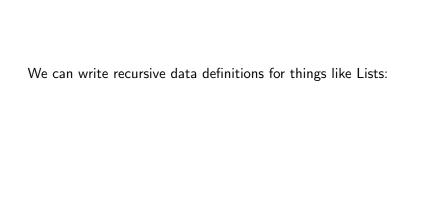
#### **Building Stuf**

Caba

10

Conclusion

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:

```
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:[]))
```

```
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:  $reduce(+)0[1,2,3]\ 1+(2+(3+0))$ 

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

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

sum :: [Int] -> Int

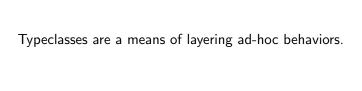
57

```
-- 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*.



```
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)
```

```
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

```
-- 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

#### Introduction

# Functional Programming

Foundations

Purity

Types

Consequences

## Examples

Basics

Data

Folds

# **Building Stuff**

Cabal

10

Conclusion

# Table of Contents

#### Introduction

# Functional Programming

Foundations

Purity

Types

Consequences

## Examples

Basics

Data

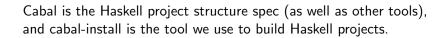
Folds

# **Building Stuff**

Cabal

10

Conclusion



# 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:

▶ 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

#### Introduction

### Functional Programming

Foundations

Purity

Types

Consequences

### Examples

Basics

Data

Folds

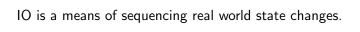
### **Building Stuff**

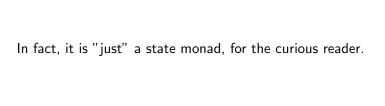
Cabal

10

Conclusion

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





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

# Table of Contents

#### Introduction

### Functional Programming

Foundations

Purity

Types

Consequences

### Examples

Basics

Data

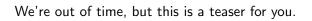
Folds

#### **Building Stuff**

Cabal

10

#### Conclusion



► Laziness + Purity = Performance + Reasoning

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

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

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

