An Introduction to Haskell, Type Systems, and Functional Programming

Allele Dev (@queertypes)

```haskell
{-# LANGUAGE OverloadedStrings #-}
import Data.Text (Text)

meta :: [(Text, Text)]
meta = [
  ("Author", "Allele Dev")
  , ("Email", "allele.dev@gmail.com")
  , ("Objectives", "Introduce: Haskell, Types, FP")
  ]

main :: IO ()
main = print meta
```

Contact Me!

- Github: cabrera
- Twitter: @cppcabrera
- Blog: Read, Review, Refactor

Goal

Let's use the wisdom of more than **four decades** worth of programming language theory to write better software.

Goal: What Will This Entail?

- Haskell as a medium

Goal: What Will This Entail?

- Haskell as a medium
    - Just enough Haskell

Goal: What Will This Entail?

- Haskell as a medium
  - Just enough Haskell
  - Just enough myth-smashing

Goal: What Will This Entail?

- Haskell as a medium
  - Just enough Haskell
  - Just enough myth-smashing
  - Just enough evidence

Goal: What Will This Entail?

- Haskell as a medium
  - Just enough Haskell
  - Just enough myth-smashing
  - Just enough evidence
- Just enough functional programming

Goal: What Will This Entail?

- Haskell as a medium
    - Just enough Haskell
    - Just enough myth-smashing
    - Just enough evidence
- Just enough functional programming
- Just enough type theory

Goal: What Will This Entail?

- Haskell as a medium
  - Just enough Haskell
  - Just enough myth-smashing
  - Just enough evidence
- Just enough functional programming
- Just enough type theory
- A sprinkle of category theory

An Aside on Typed FP Languages

- Haskell is only a medium in this presentation

An Aside on Typed FP Languages

- Haskell is only a medium in this presentation
- Other languages in a similar vein (with similar capacities) include:

An Aside on Typed FP Languages

- Haskell is only a medium in this presentation
- Other languages in a similar vein (with similar capacities) include:
  - Scala: Typed-FP/OO hybrid on JVM

An Aside on Typed FP Languages

- Haskell is only a medium in this presentation
- Other languages in a similar vein (with similar capacities) include:
  - Scala: Typed-FP/OO hybrid on JVM
  - F#: Typed-FP on .NET

An Aside on Typed FP Languages

- Haskell is only a medium in this presentation
- Other languages in a similar vein (with similar capacities) include:
  - Scala: Typed-FP/OO hybrid on JVM
  - F#: Typed-FP on .NET
  - Ocaml: Typed-FP, ML-derived

An Aside on Typed FP Languages

- Haskell is only a medium in this presentation
- Other languages in a similar vein (with similar capacities) include:
  - Scala: Typed-FP/OO hybrid on JVM
  - F#: Typed-FP on .NET
  - Ocaml: Typed-FP, ML-derived
  - Standard ML: Typed-FP, ML-derived

An Aside on Typed FP Languages

- Haskell is only a medium in this presentation
- Other languages in a similar vein (with similar capacities) include:
  - Scala: Typed-FP/OO hybrid on JVM
  - F#: Typed-FP on .NET
  - Ocaml: Typed-FP, ML-derived
  - Standard ML: Typed-FP, ML-derived
- And others, still: Elm, Idris, Agda

Why Haskell as a Medium?

- Personal bias: I am fond of Haskell

Why Haskell as a Medium?

- Personal bias: I am fond of Haskell
- Purely functional: forces one to solve problems functionally

Why Haskell as a Medium?

- Personal bias: I am fond of Haskell
- Purely functional: forces one to solve problems functionally
- Very clean syntax

Why Haskell as a Medium?

- Personal bias: I am fond of Haskell
- Purely functional: forces one to solve problems functionally
- Very clean syntax
- Great resources available for free, everywhere

Why Haskell as a Medium?

- Personal bias: I am fond of Haskell
- Purely functional: forces one to solve problems functionally
- Very clean syntax
- Great resources available for free, everywhere
- Runs on: Windows, Linux, Mac OS X, iOS, Android

Overview

- A tour of Haskell

Overview

- A tour of Haskell
  - Syntax

Overview

- A tour of Haskell
  - Syntax
  - Abstraction facilities

Overview

- A tour of Haskell
  - Syntax
  - Abstraction facilities
  - Modules

Overview

- A tour of Haskell
  - Syntax
  - Abstraction facilities
  - Modules
  - Myths, ecosystem, and related alternatives

Overview

- A tour of Haskell
  - Syntax
  - Abstraction facilities
  - Modules
  - Myths, ecosystem, and related alternatives
- Why functional programming mattters

Overview

- A tour of Haskell
  - Syntax
  - Abstraction facilities
  - Modules
  - Myths, ecosystem, and related alternatives
- Why functional programming mattters
- Why types matter

What's Haskell?

- A **statically-typed**, **pure**, **lazy**, **functional** programming language
- At least 24 years old (Report 1.0 released on April 1, 1990)

Definitions

- **Statically-typed**: Type checks occur at compile-time

Definitions

- **Statically-typed**: Type checks occur at compile-time
- **pure**: side-effects are carefully isolated

Definitions

- **Statically-typed**: Type checks occur at compile-time
- **pure**: side-effects are carefully isolated
- **lazy**: function arguments are evaluated only when needed

Definitions

- **Statically-typed**: Type checks occur at compile-time
- **pure**: side-effects are carefully isolated
- **lazy**: function arguments are evaluated only when needed
- **functional**: programs as composition of functions

What Does it Look Like?

```haskell
-- Hello.hs
main = print "Hello, World"
```

What Does it Look Like?

```haskell
-- Hello.hs
hello :: String
hello = "Hello, world"

main = print hello
```

How Do I Make it Run?

```
$ ghc Hello
[1 of 1] Compiling Main             ( Hello.hs, Hello.o )
Linking Hello ...
$ ./Hello
"Hello, world!"
```

How Do I Make it Run? (interactive version!)

```
$ ghci Hello
ghci Hello
GHCi, version 7.8.2: http://www.haskell.org/ghc/
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
[1 of 1] Compiling Main             ( Hello.hs, ... )
Ok, modules loaded: Main.
*Main> main
"Hello, world!"
```

Literals

- Nums/Ints/Integers: 1

Literals

- Nums/Ints/Integers: `1`
- Fractionals/Floats: `1.0`

Literals

- Nums/Ints/Integers: `1`
- Fractionals/Floats: `1.0`
- Chars: `'a'`

Literals

- Nums/Ints/Integers: `1`
- Fractionals/Floats: `1.0`
- Chars: `'a'`
- Booleans: False, True

Literals

- Nums/Ints/Integers: `1`
- Fractionals/Floats: `1.0`
- Chars: `'a'`
- Booleans: False, True
- Lists: `[1, 2, 3]`, `"Char List"` – homogenous

Literals

- Nums/Ints/Integers: `1`
- Fractionals/Floats: `1.0`
- Chars: `'a'`
- Booleans: False, True
- Lists: `[1, 2, 3]`, `"Char List"` – homogenous
    - “A list” :: [Char] == String

Literals

- Nums/Ints/Integers: `1`
- Fractionals/Floats: `1.0`
- Chars: `'a'`
- Booleans: False, True
- Lists: `[1, 2, 3]`, `"Char List"` – homogenous
  - "A list" :: [Char] == String
- Tuples: (1, 'a', [False, True]) – not homogenous

Functions

```haskell
factorial :: Num a => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Functions

```haskell
-- function_name :: (type contraints) =>
-- arg_type1 -> arg_type2 -> return type
factorial :: Num a => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Functions

```haskell
factorial :: Num a => a -> a
-- function_name arg1 arg2 = implementation
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

A Brief Aside

- Learn to read type signatures

A Brief Aside

- Learn to read type signatures
  - Extremely helpful early investment w/ Haskell

A Brief Aside

- Learn to read type signatures
  - Extremely helpful early investment w/ Haskell
  - If in doubt, inspect the types!

A Brief Aside

- Learn to read type signatures
  - Extremely helpful early investment w/ Haskell
  - If in doubt, inspect the types!
  - Open GHCi, and ask away:

Type Inspection with GHCi

```
> :t 1
```

Type Inspection with GHCi

```
> :t 1
1 :: Num a => a
```

Type Inspection with GHCi

```
> :t 1.0
```

Type Inspection with GHCi

```
> :t 1.0
1.0 :: Fractional a => a
```

Type Inspection with GHCi

```
> :t [1, 2, 3]
```

Type Inspection with GHCi

```
> :t [1, 2, 3]
[1, 2, 3] :: Num t => [t]
```

Type Inspection with GHCi

```
> :t (1, 'a', False)
```

Type Inspection with GHCi

```
> :t (1, 'a', False)
(1, 'a', False) :: Num t => (t, Char, Bool)
```

Type Inspection with GHCi

```
> :t map
```

Type Inspection with GHCi

```
> :t map
map :: (a -> b) -> [a] -> [b]
```

Type Inspection with GHCi

```
> :t (+)
```

Type Inspection with GHCi

```
> :t (+)
(+) :: Num a => a -> a -> a
```

A Note on Operators

- All operators are just built-in functions

A Note on Operators

- All operators are just built-in functions
- It's common to define custom infix ops in Haskell

A Note on Operators

- All operators are just built-in functions
- It's common to define custom infix ops in Haskell
  - <*> appears with Applicatives

A Note on Operators

- All operators are just built-in functions
- It's common to define custom infix ops in Haskell
  - <*> appears with Applicatives
  - . function composition

A Note on Operators

- All operators are just built-in functions
- It's common to define custom infix ops in Haskell
  - <*> appears with Applicatives
  - . function composition
  - >>= sequencing

Golden Rule About Haskell Functions

- Every function takes just one argument

Golden Rule About Haskell Functions

- Every function takes just one argument
  - All arguments are automatically curried

Golden Rule About Haskell Functions

- Every function takes just one argument
  - All arguments are automatically curried
  - (or Schonfinkled – a story for another time)

Golden Rule About Haskell Functions

- Every function takes just one argument
  - All arguments are automatically curried
  - (or Schonfinkled – a story for another time)
- Use this to your advantage

Type-inspection Search Engine: hoogle

- Haskell supports a type-signature search engine

Type-inspection Search Engine: hoogle

- Haskell supports a type-signature search engine
- Looking for a particular function, use hoogle

Type-inspection Search Engine: hoogle

- Haskell supports a type-signature search engine
- Looking for a particular function, use hoogle
    - Can also be installed in ghci

Curry in Action

```
> :t (+)
```

Curry in Action

```
> :t (+)
(+) :: Num a => a -> a -> a
```

Curry in Action

```
> :t (+2)
```

Curry in Action

```
> :t (+2)
(+2) :: Num a => a -> a
```

Curry in Action

```
> :t (+2)
(+2) :: Num a => a -> a
> -- (+2) is a valid term; a "section",
```

Curry in Action

```
> :t (+2)
(+2) :: Num a => a -> a
> -- (+2) is a valid term; a "section",
> -- a partially applied function
```

Syntax for Defining Functions

- Type signature

Syntax for Defining Functions

- Type signature
- Equations

Syntax for Defining Functions

- Type signature
- Equations
- Guards

Syntax for Defining Functions

- Type signature
- Equations
- Guards
- Case expression and pattern matching

Type Signatures

- Rarely required, due to powerful type inference engine

Type Signatures

- Rarely required, due to powerful type inference engine
- Serves more as compiler-checked documentation of intent

Type Signatures

- Rarely required, due to powerful type inference engine
- Serves more as compiler-checked documentation of intent
    - Can also aid Type Driven Development

Type Signature Examples

```
id :: a -> a
```

Type Signature Examples

```
map :: (a -> b) -> [a] -> [b]
```

Type Signature Examples

```
filter :: (a -> Bool) -> [a] -> [a]
```

Type Signature Examples

```
(+) :: Num a => a -> a -> a
```

Type Signature Examples

```haskell
(<) :: Ord a => a -> a -> Bool
```

Type Signature Examples

```
(==) :: Eq a => a -> a -> Bool
```

Type Signature Examples

```
(/=) :: Eq a => a -> a -> Bool
```

```
map _ [] = []
map f (x:xs) = f x : map f xs
```

Guarded Functions

```
factorial n
  | n <= 0 = 1
  | otherwise = n * factorial (n - 1)
```

Case Expressions

```haskell
describeList :: [a] -> String
describeList xs = case xs of
  [] -> "empty"
  [_] -> "singleton"
  _ -> "longer list"
```

Where: Inline Definitions After the Fact

```haskell
validArea :: (Ord a, Num a) => a -> a -> Bool
validArea x y
  | area x y >= 0 = True
  | otherwise = False
  where area x' y' = x' * y'
```

Let-In: Inline Definitions as a Prologue

```haskell
analyzeNumber :: (Ord a, Num a) => a -> Bool
analyzeNumber n =
  let analyze n' = (n' * n')
      reasonable n' = analyze n' > 2
  in
    reasonable n
```

As Patterns: Named Capture of the Whole in a Pattern Match

```
starter :: String -> String
starter "" = "empty"
starter all_xs@(x:_) = all_xs ++ " starts with " ++ [x]
```

A Little More Syntax: Abstraction

- We can now define functions

A Little More Syntax: Abstraction

- We can now define functions
- Let's define our own types!

Abstract Data Types: Simple

```haskell
-- week.hs
data Weekday =
  Monday
  | Tuesday
  | Wednesday
  | Thursday
  | Friday
  | Saturday
  | Sunday deriving (Show, Eq)
```

Simple: Sum Types, Deriving

- Most typed-FP languages allow for **sum types**

Simple: Sum Types, Deriving

- Most typed-FP languages allow for **sum types**
  - discriminated unions checked at compile-time

Simple: Sum Types, Deriving

- Most typed-FP languages allow for **sum types**
  - discriminated unions checked at compile-time
- `deriving`: compiler automatically implements certain interfaces

Simple Weekday

```haskell
-- week.hs
next :: Weekday -> Weekday
next day = case day of
  Tuesday -> Wednesday
  Wednesday -> Thursday
  Thursday -> Friday
  Friday -> Saturday
  Saturday -> Sunday
  Sunday -> Monday
```

Simple Types in Action

```
$ ghci -Wall week
GHCi, version 7.8.2: http://www.haskell.org/ghc/
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
[1 of 1] Compiling Main     ( week.hs, interpreted )

week.hs:12:12: Warning:
Pattern match(es) are non-exhaustive
In a case alternative: Patterns not matched: Monday
Ok, modules loaded: Main.
```

Simple Types in Action

- Compiler knows how to check for all cases in a sum type

Simple Types in Action

- Compiler knows how to check for all cases in a sum type
- It just told us we forgot about Monday

Simple Types in Action

- Compiler knows how to check for all cases in a sum type
- It just told us we forgot about Monday
  - We're human! Sometimes we forget what day of the week we're on

Simple Types in Action

- Compiler knows how to check for all cases in a sum type
- It just told us we forgot about Monday
  - We're human! Sometimes we forget what day of the week we're on
  - Extremely useful tool for refactoring

Simple Types: Deriving

- **deriving**: ask compiler to auto-implement an interface

Simple Types: Deriving

- **deriving**: ask compiler to auto-implement an interface
  - For simple interfaces/typeclasses, this is possible

Simple Types: Deriving

- **deriving**: ask compiler to auto-implement an interface
  - For simple interfaces/typeclasses, this is possible
  - Simple includes: printing, equality, ordering, enumeration, ...

Simple Types: Records

```
data Person = Person
  { name :: String
  , age :: Int
  }
```

Simple Types: Records

```
> Person "Lantern" 27
Person "Lantern" 27
> name (Person "Lantern" 27)
"Lantern"
> let newPerson p = Person $ name p $ (age p) + 1
> newPerson $ Person "Lantern" 27
Person "Lantern" 28
```

Simple Types: Type Parameters

```haskell
-- A type that already exists
data Maybe a =
  Just a
  | Nothing deriving (Show, Eq)
```

Simple Types: Recursive Types

```haskell
data List' a =
  Nil
  | Cons a (List' a) deriving (Show, Eq)

data Tree a =
  EmptyTree
  | Node a (Tree a) (Tree a) deriving (Show, Eq)
```

Functions on Recursive Types: Lists

```haskell
head' :: List' a -> Maybe a
head' Nil = Nothing
head' (Cons x rest) = Just x
```

Functions on Recursive Types: Lists

```
> Cons 1 $ Cons 2 $ Nil
```

```
> Cons 1 $ Cons 2 $ Nil
Cons 1 (Cons 2 (Nil)) :: Num a => List' a
```

Functions on Recursive Types: Lists

```
> Nil
```

Functions on Recursive Types: Lists

```
> Nil
Nil :: List' a
```

Functions on Recursive Types: Lists

```
> head' Nil
```

```
> head' Nil
Nothing :: Maybe a
```

Functions on Recursive Types: Lists

```
> head' $ Cons 1 $ Nil
```

Functions on Recursive Types: Lists

```
> head' $ Cons 1 $ Nil
Cons 1 :: Num a => Maybe a
```

Functions on Recursive Types: Trees

```
insert :: Ord a => a -> Tree a -> Tree a
insert v EmptyTree = Node v EmptyTree EmptyTree
insert v (Node n l r)
  | v == n = Node v l r  -- create identical node in place
  | v < n = Node n (insert v l) r
  | v > n = Node n l (insert v r)
```

```
> EmptyTree
```

Functions on Recursive Types: Trees

```
> EmptyTree
EmptyTree :: Tree a
```

Functions on Recursive Types: Trees

```
> Node 2 EmptyTree EmptyTree
```

```
> Node 2 EmptyTree EmptyTree
Node 2 EmptyTree EmptyTree :: Num a => Tree a
```

Functions on Recursive Types: Trees

```
> let example = Node 2 EmptyTree EmptyTree
```

Functions on Recursive Types: Trees

```
> let example = Node 2 EmptyTree EmptyTree
> insert 1 example
```

Functions on Recursive Types: Trees

```
> let example = Node 2 EmptyTree EmptyTree
> insert 1 example
Node 2 (Node 1 EmptyTree EmptyTree) EmptyTree
```

Functions on Recursive Types: Trees

```
> let example = Node 2 EmptyTree EmptyTree
> insert 1 example
Node 2 (Node 1 EmptyTree EmptyTree) EmptyTree
> insert 2 example
```

Functions on Recursive Types: Trees

```
> let example = Node 2 EmptyTree EmptyTree
> insert 1 example
Node 2 (Node 1 EmptyTree EmptyTree) EmptyTree
> insert 2 example
Node 2 EmptyTree EmptyTree
```

Functions on Recursive Types: Trees

```
> let example = Node 2 EmptyTree EmptyTree
> insert 1 example
Node 2 (Node 1 EmptyTree EmptyTree) EmptyTree
> insert 2 example
Node 2 EmptyTree EmptyTree
> insert 3 example
```

Functions on Recursive Types: Trees

```
> let example = Node 2 EmptyTree EmptyTree
> insert 1 example
Node 2 (Node 1 EmptyTree EmptyTree) EmptyTree
> insert 2 example
Node 2 EmptyTree EmptyTree
> insert 3 example
Node 2 EmptyTree (Node 3 EmptyTree EmptyTree)
```

Functions on Recursive Types: Trees (Chained)

```
> let example = Node 2 EmptyTree EmptyTree
```

Functions on Recursive Types: Trees (Chained)

```
> let example = Node 2 EmptyTree EmptyTree
> insert 1 $ insert 3 $ insert 4 $ insert 5 example
```

Functions on Recursive Types: Trees (Chained)

```
> let example = Node 2 EmptyTree EmptyTree
> insert 1 $ insert 3 $ insert 4 $ insert 5 example
Node 2 (Node 1 EmptyTree EmptyTree)
       (Node 5
         (Node 4 (Node 3 EmptyTree EmptyTree)
          EmptyTree)
        EmptyTree)
> -- pretty-printing is my doing
> -- that it prints at all is because of 'deriving (Show)'
```

Typeclasses: Interfaces for Haskell

- Typeclasses allow one to:

Typeclasses: Interfaces for Haskell

- Typeclasses allow one to:
  - Define type constraints

Typeclasses: Interfaces for Haskell

- Typeclasses allow one to:
  - Define type constraints
  - Define what functions a type must implement

Example: What a Typeclass Looks Like

```haskell
class Eq' a where
    -- point-free default impls.
    -- provide one of (==') or (/=')
    (===) :: a -> a -> Bool
    (/==) :: a -> a -> Bool
    l === r = not $ l /== r
    l /== r = not $ l === r
```

Example: A Manual Typeclass Instance for Weekdays

```haskell
instance Eq' Weekday where
    Monday ==' Monday = True
    Tuesday ==' Tuesday = True
    -- ...
    Sunday ==' Sunday = True
    _ ==' _ = False
```

Modules at Last

```haskell
-- Geometry/Circle.hs
module Geometry.Circle
( area
, perimeter
) where

-- the most accurate; more accurate than Prelude.pi
pi' :: Float
pi' = 3.1415926

area :: Float -> Float
area r = pi' * r**2

perimeter :: Float -> Float
perimeter r = 2 * pi' * r
```

Using a Module

```
module Main where
import Geometry.Circle

main = print $ area 10
```

Using a Module

```haskell
module Main where

-- useful for avoiding name collisions
import qualified Geometry.Circle as GC

main = print $ GC.area 10
```

Mythology: Purity => Useless

- "Haskell is useless": link

Mythology: Purity => Useless

- “Haskell is useless”: link
- “Haskell is the world’s finest imperative language.” – SPJ

Mythology: GC + Functional => Slow

- Performance concerns?

Mythology: GC + Functional => Slow

- Performance concerns?
- The obvious toy benchmarks

Mythology: GC + Functional => Slow

- Performance concerns?
- The obvious toy benchmarks
- Haskell Warp vs. Nginx

Mythology: GC + Functional => Slow

- Performance concerns?
- The obvious toy benchmarks
- Haskell Warp vs. Nginx
- Haskell SDN Controller

Mythology: GC + Functional => Slow

- Performance concerns?
- The obvious toy benchmarks
- Haskell Warp vs. Nginx
- Haskell SDN Controller
- Haskell on a GPU

Mythology: GC + Functional => Slow

- Performance concerns?
- The obvious toy benchmarks
- Haskell Warp vs. Nginx
- Haskell SDN Controller
- Haskell on a GPU
- Haskell Parallel Arrays

Mythology: Haskell is Purely Academic

- Users in Industry

Mythology: Haskell is Purely Academic

- Users in Industry
- Projects in Haskell. Notably, for me:

Mythology: Haskell is Purely Academic

- Users in Industry
- Projects in Haskell. Notably, for me:
    - pandoc: Used to make this talk

Mythology: Haskell is Purely Academic

- Users in Industry
- Projects in Haskell. Notably, for me:
  - pandoc: Used to make this talk
  - hakyll: static blog generator

Mythology: Haskell is Purely Academic

- Users in Industry
- Projects in Haskell. Notably, for me:
    - pandoc: Used to make this talk
    - hakyll: static blog generator
    - ghcjs: haskell -> JS compiler

Mythology: Haskell is Purely Academic

- Users in Industry
- Projects in Haskell. Notably, for me:
    - pandoc: Used to make this talk
    - hakyll: static blog generator
    - ghcjs: haskell -> JS compiler
    - idris: dependently-typed FP language

Just Enough Haskell?

- We've covered:

Just Enough Haskell?

- We've covered:
  - Core syntax

Just Enough Haskell?

- We've covered:
  - Core syntax
  - Defining functions

Just Enough Haskell?

- We've covered:
  - Core syntax
  - Defining functions
  - Defining own types (of many *kinds*)

Just Enough Haskell?

- We've covered:
  - Core syntax
  - Defining functions
  - Defining own types (of many *kinds*)
    - (pun intended)

Just Enough Haskell?

- We've covered:
  - Core syntax
  - Defining functions
  - Defining own types (of many *kinds*)
    - (pun intended)
  - Defining type classes

Just Enough Haskell?

- We've covered:
  - Core syntax
  - Defining functions
  - Defining own types (of many *kinds*)
    - (pun intended)
  - Defining type classes
  - Some myth-smashing

Leveraging the Wisdom of Functional Programming

- Higher-order operations

- Higher-order operations
- Equational reasoning

Leveraging the Wisdom of Functional Programming

- Higher-order operations
- Equational reasoning
- Lambda calculus

- Higher-order operations
- Equational reasoning
- Lambda calculus
- Going further

Higher-Order Thinking

"Can programming be liberated from the von-Neumann Bottleneck?" – John Backus

Higher-Order Thinking

- As a fundamental notion, we can elevate the way we iterate over data

- As a fundamental notion, we can elevate the way we iterate over data
- These are the functions: map, filter, fold/reduce

Higher-Order Thinking

- As a fundamental notion, we can elevate the way we iterate over data
- These are the functions: map, filter, fold/reduce
- They take a function and a collection to perform what they do

Higher-Order Thinking

- As a fundamental notion, we can elevate the way we iterate over data
- These are the functions: map, filter, fold/reduce
- They take a function and a collection to perform what they do
    - Tim Sweeney on: FP and higher-order ops 2006, pg. 35

Iteration Pattern: Map

```python
def map(f, xs):
    result = []
    for x in xs:
        result.append(f(x))

    return result
```

Iteration Pattern: Map

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

Iteration Pattern: Filter

```python
def filter(f, xs):
    result = []
    for x in xs:
        if f(x):
            result.append(x)

    return result
```

Iteration Pattern: Filter

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter f (x:xs)
  | f x = x : filter f xs
  | otherwise = filter f xs
```

Iteration Pattern: Reduce/Fold

```
def fold(f, init, xs):
    result = init
    for x in xs:
        result = f(result, x)
    return result
```

Iteration Pattern: Reduce/Fold

```haskell
-- note: this impl. not tail recursive
-- overflows stack for large [a]
fold :: (a -> b -> b) -> b -> [a] -> b
fold _ init [] = init
fold f init (x:xs) = f x $ fold f init xs
```

Higher-Order Functions: Why?

- map, filter, fold: powerful iteration primitives

Higher-Order Functions: Why?

- map, filter, fold: powerful iteration primitives
- Communicates intent clearly

Higher-Order Functions: Why?

- map, filter, fold: powerful iteration primitives
- Communicates intent clearly
    - Reader need only find primitives to determine intent

Higher-Order Functions: Why?

- map, filter, fold: powerful iteration primitives
- Communicates intent clearly
  - Reader need only find primitives to determine intent
- Also, composable and versatile

Higher-Order Functions: Composed

```
> let xs = [1..5]
```

Higher-Order Functions: Composed

```
> let xs = [1..5]
> map (+1) xs
```

Higher-Order Functions: Composed

```
> let xs = [1..5]
> map (+1) xs
[2, 3, 4, 5, 6]
```

Higher-Order Functions: Composed

```
> let xs = [1..5]
> map ((*2) . (+1)) xs
```

Higher-Order Functions: Composed

```
> let xs = [1..5]
> map ((*2) . (+1)) xs
[6, 8, 10, 12, 14]
```

Higher-Order Functions: Composed

```
> let xs = [1..5]
> filter (odd) $ map ((*2) . (+1)) xs
```

Higher-Order Functions: Composed

```
> let xs = [1..5]
> filter (odd) $ map ((*2) . (+1)) xs
[]
```

Higher-Order Functions: Composed

```
> let xs = [1..5]
> fold (*) 1 $ filter (odd) $ map (^2) xs
```

Higher-Order Functions: Composed

```
> let xs = [1..5]
> fold (*) 1 $ filter (odd) $ map (^2) xs
225
```

Higher-Order Functions: Composed

```
> let xs = [1..5]
> fold (*) 1 $ filter (odd) $ map (^2) xs
225
> -- product of odd numbered squares
```

Higher-Order Vocabulary

- Communicate intent, not details

Higher-Order Vocabulary

- Communicate intent, not details
- Compose smaller pieces to build larger systems

Higher-Order Vocabulary

- Communicate intent, not details
- Compose smaller pieces to build larger systems
- Taken to the end: embedded domain-specific languages (EDSLs)

Equational Reasoning

- In the absence of stateful modification, one can:

Equational Reasoning

- In the absence of stateful modification, one can:
  - Substitute invocation of function for next step

Equational Reasoning

- In the absence of stateful modification, one can:
  - Substitute invocation of function for next step
- Result:

Equational Reasoning

- In the absence of stateful modification, one can:
    - Substitute invocation of function for next step
- Result:
    - Clear separation of concerns

Equational Reasoning

- In the absence of stateful modification, one can:
  - Substitute invocation of function for next step
- Result:
  - Clear separation of concerns
  - Leak-free abstractions

Equational Reasoning

- In the absence of stateful modification, one can:
  - Substitute invocation of function for next step
- Result:
  - Clear separation of concerns
  - Leak-free abstractions
- Learn more: Equational reasoning

Going Further: Lambda Calculus

- Basis for functional programming languages

Going Further: Lambda Calculus

- Basis for functional programming languages
- Known to be Turing machine equivalent

Going Further: Lambda Calculus

- Basis for functional programming languages
- Known to be Turing machine equivalent
- Three primitives to express all computation:

Going Further: Lambda Calculus

- Basis for functional programming languages
- Known to be Turing machine equivalent
- Three primitives to express all computation:
  - Variable: `x`

Going Further: Lambda Calculus

- Basis for functional programming languages
- Known to be Turing machine equivalent
- Three primitives to express all computation:
  - Variable: x
  - Abstraction: \f.x x

Going Further: Lambda Calculus

- Basis for functional programming languages
- Known to be Turing machine equivalent
- Three primitives to express all computation:
  - Variable: `x`
  - Abstraction: \f.x x
  - Application: f x

Going Further: Lambda Calculus

- Basis for functional programming languages
- Known to be Turing machine equivalent
- Three primitives to express all computation:
  - Variable: `x`
  - Abstraction: \f.x x
  - Application: f x
- Can be used to craft type-safe EDSLs

Going Further: Lambda Calculus

- Basis for functional programming languages
- Known to be Turing machine equivalent
- Three primitives to express all computation:
  - Variable: x
  - Abstraction: \f.x x
  - Application: f x
- Can be used to craft type-safe EDSLs
- Learn more: Type-safe EDSLs

Going Further: More Higher-Order Primitives

- There's a few more primitives of interest

Going Further: More Higher-Order Primitives

- There's a few more primitives of interest
- There's also a mathematical vocabulary

Going Further: More Higher-Order Primitives

- There's a few more primitives of interest
- There's also a mathematical vocabulary
- Learn more: Bananas, Lenses, Envelopes, and Barbed Wire

Just Enough Functional Programming?

- Think: higher-order

Just Enough Functional Programming?

- Think: higher-order
  - Map, fold, filter; not for and while

Just Enough Functional Programming?

- Think: higher-order
  - Map, fold, filter; not for and while
- Small pieces -> clean abstractions

Just Enough Functional Programming?

- Think: higher-order
  - Map, fold, filter; not for and while
- Small pieces -> clean abstractions
- Preserve simplicity at each layer

Just Enough Functional Programming?

- Think: higher-order
  - Map, fold, filter; not for and while
- Small pieces -> clean abstractions
- Preserve simplicity at each layer
- Learn more: Why FP Matters

Leveraging Type Theory

- What is type theory?

Leveraging Type Theory

- What is type theory?
- What is type safety?

Leveraging Type Theory

- What is type theory?
- What is type safety?
- Why do types matter?

Leveraging Type Theory

- What is type theory?
- What is type safety?
- Why do types matter?
- Software development with rich types

What is Type Theory?

- A formal system of reasoning

What is Type Theory?

- A formal system of reasoning
- Sometimes proposed as an alternative to set theory

What is Type Theory?

- A formal system of reasoning
- Sometimes proposed as an alternative to set theory
  - I call this refactoring the foundations of math

What is Type Theory?

- A formal system of reasoning
- Sometimes proposed as an alternative to set theory
  - I call this refactoring the foundations of math
- Direct connection to logic (Curry-Howard isomorphism)

What is Type Theory?

- A formal system of reasoning
- Sometimes proposed as an alternative to set theory
  - I call this refactoring the foundations of math
- Direct connection to logic (Curry-Howard isomorphism)
- Influences type systems

What is Type Theory?

- A formal system of reasoning
- Sometimes proposed as an alternative to set theory
  - I call this refactoring the foundations of math
- Direct connection to logic (Curry-Howard isomorphism)
- Influences type systems
  - System F: (Girard–Reynolds) polymorphic lambda-calculus

What is Type Theory?

- A formal system of reasoning
- Sometimes proposed as an alternative to set theory
    - I call this refactoring the foundations of math
- Direct connection to logic (Curry-Howard isomorphism)
- Influences type systems
    - System F: (Girard–Reynolds) polymorphic lambda-calculus
    - Hindley–Milner type system

What is Type Safety?

- A property of a type system that guarantees that:

What is Type Safety?

- A property of a type system that guarantees that:
  - If the program compiles

What is Type Safety?

- A property of a type system that guarantees that:
  - If the program compiles
  - It will never “get stuck”

What is Type Safety?

- A property of a type system that guarantees that:
  - If the program compiles
  - It will never "get stuck"
  - No undefined states

What is Type Safety?

- A property of a type system that guarantees that:
  - If the program compiles
  - It will never “get stuck”
  - No undefined states
- Safety = Preservation + Progress

What is Type Safety?

- A property of a type system that guarantees that:
  - If the program compiles
  - It will never "get stuck"
  - No undefined states
- Safety = Preservation + Progress
  - **Progress**: a well-typed term t is either:

What is Type Safety?

- A property of a type system that guarantees that:
  - If the program compiles
  - It will never "get stuck"
  - No undefined states
- Safety = Preservation + Progress
  - **Progress**: a well-typed term t is either:
    - a value, t

What is Type Safety?

- A property of a type system that guarantees that:
  - If the program compiles
  - It will never "get stuck"
  - No undefined states
- Safety = Preservation + Progress
  - **Progress**: a well-typed term t is either:
    - a value, t
    - a term t:T with a path t => t'

What is Type Safety?

- A property of a type system that guarantees that:
  - If the program compiles
  - It will never “get stuck”
  - No undefined states
- Safety = Preservation + Progress
  - **Progress**: a well-typed term t is either:
    - a value, t
    - a term t:T with a path t => t’
  - **Preservation**: types are preserved

What is Type Safety?

- A property of a type system that guarantees that:
  - If the program compiles
  - It will never “get stuck”
  - No undefined states
- Safety = Preservation + Progress
  - **Progress**: a well-typed term t is either:
    - a value, t
    - a term t:T with a path t => t’
  - **Preservation**: types are preserved
    - t:T and t -> t’ => t’:T

Why Do Types Matter?

"Program testing can be used to show the presence of bugs, but never there absence." – Edsger W. Dijkstra

Why Do Types Matter?

- It is not enough to reach 100% test coverage

Why Do Types Matter?

- It is not enough to reach 100% test coverage
- Must prove that certain states cannot be reached

Why Do Types Matter?

- It is not enough to reach 100% test coverage
- Must prove that certain states cannot be reached
  - "Make illegal states unrepresentable": video – Yaron Minsky

Why Do Types Matter?

- It is not enough to reach 100% test coverage
- Must prove that certain states cannot be reached
  - "Make illegal states unrepresentable": video – Yaron Minsky
- Communicate design

Why Do Types Matter?

- It is not enough to reach 100% test coverage
- Must prove that certain states cannot be reached
  - "Make illegal states unrepresentable": video – Yaron Minsky
- Communicate design
  - Compiler enforces assumptions and abstractions

Why Do Types Matter?

- It is not enough to reach 100% test coverage
- Must prove that certain states cannot be reached
  - “Make illegal states unrepresentable”: video – Yaron Minsky
- Communicate design
  - Compiler enforces assumptions and abstractions
  - Turn “don’t do that” -> “can’t do that”: video

Software Development with Rich Types

- Encode enough representation in type system

Software Development with Rich Types

- Encode enough representation in type system
  - Abstract data types + type constraints

- Encode enough representation in type system
  - Abstract data types + type constraints
  - Transitions as functions

Software Development with Rich Types

- Encode enough representation in type system
  - Abstract data types + type constraints
  - Transitions as functions
- Iteratively refine the assumptions

Software Development with Rich Types

- Encode enough representation in type system
  - Abstract data types + type constraints
  - Transitions as functions
- Iteratively refine the assumptions
- Compile the code

Software Development with Rich Types

- Encode enough representation in type system
  - Abstract data types + type constraints
  - Transitions as functions
- Iteratively refine the assumptions
- Compile the code
- Repeat as needed with refactorings

Software Development with Rich Types

- Encode enough representation in type system
  - Abstract data types + type constraints
  - Transitions as functions
- Iteratively refine the assumptions
- Compile the code
- Repeat as needed with refactorings
- Learn more: type-driven development

Types: Learning Even More

- Start here: Types and Programming Languages

Types: Learning Even More

- Start here: Types and Programming Languages
- Deeper dives:

Types: Learning Even More

- Start here: Types and Programming Languages
- Deeper dives:
  - Practical Foundations for Programming Languages

Types: Learning Even More

- Start here: Types and Programming Languages
- Deeper dives:
  - Practical Foundations for Programming Languages
  - Category Theory

- Start here: Types and Programming Languages
- Deeper dives:
  - Practical Foundations for Programming Languages
  - Category Theory
  - Why Dependent Types Matter

Types: Learning Even More

- Start here: Types and Programming Languages
- Deeper dives:
  - Practical Foundations for Programming Languages
  - Category Theory
  - Why Dependent Types Matter
  - Certified Programming With Dependent Types

Types: Learning Even More

- Start here: Types and Programming Languages
- Deeper dives:
  - Practical Foundations for Programming Languages
  - Category Theory
  - Why Dependent Types Matter
  - Certified Programming With Dependent Types
- Specific applications:

Types: Learning Even More

- Start here: Types and Programming Languages
- Deeper dives:
  - Practical Foundations for Programming Languages
  - Category Theory
  - Why Dependent Types Matter
  - Certified Programming With Dependent Types
- Specific applications:
  - Verified TLS

Types: Learning Even More

- Start here: Types and Programming Languages
- Deeper dives:
  - Practical Foundations for Programming Languages
  - Category Theory
  - Why Dependent Types Matter
  - Certified Programming With Dependent Types
- Specific applications:
  - Verified TLS
  - Verified OS

What We Didn't Talk About

- Monads, Functors, and Category Theory

What We Didn't Talk About

- Monads, Functors, and Category Theory
- Haskell ecosystem

What We Didn't Talk About

- Monads, Functors, and Category Theory
- Haskell ecosystem
    - Libraries

What We Didn't Talk About

- Monads, Functors, and Category Theory
- Haskell ecosystem
  - Libraries
  - Editors

What We Didn't Talk About

- Monads, Functors, and Category Theory
- Haskell ecosystem
  - Libraries
  - Editors
  - QuickCheck, and property-based testing

What We Didn't Talk About

- Monads, Functors, and Category Theory
- Haskell ecosystem
  - Libraries
  - Editors
  - QuickCheck, and property-based testing
    - Deprecates unit testing, for the most part

What We Didn't Talk About

- Monads, Functors, and Category Theory
- Haskell ecosystem
  - Libraries
  - Editors
  - QuickCheck, and property-based testing
    - Deprecates unit testing, for the most part
  - cabal

What We Didn't Talk About

- Monads, Functors, and Category Theory
- Haskell ecosystem
  - Libraries
  - Editors
  - QuickCheck, and property-based testing
    - Deprecates unit testing, for the most part
  - cabal
  - setup and flow

What We Didn't Talk About

- Monads, Functors, and Category Theory
- Haskell ecosystem
  - Libraries
  - Editors
  - QuickCheck, and property-based testing
    - Deprecates unit testing, for the most part
  - cabal
  - setup and flow
- Type theory

What We Didn't Talk About

- Monads, Functors, and Category Theory
- Haskell ecosystem
  - Libraries
  - Editors
  - QuickCheck, and property-based testing
    - Deprecates unit testing, for the most part
  - cabal
  - setup and flow
- Type theory
  - Haskell limitations here

What We Didn't Talk About

- Monads, Functors, and Category Theory
- Haskell ecosystem
  - Libraries
  - Editors
  - QuickCheck, and property-based testing
    - Deprecates unit testing, for the most part
  - cabal
  - setup and flow
- Type theory
  - Haskell limitations here
  - Higher-kinded programming

What We Didn't Talk About

- Monads, Functors, and Category Theory
- Haskell ecosystem
  - Libraries
  - Editors
  - QuickCheck, and property-based testing
    - Deprecates unit testing, for the most part
  - cabal
  - setup and flow
- Type theory
  - Haskell limitations here
  - Higher-kinded programming
  - GADTs

What We Didn't Talk About

- Monads, Functors, and Category Theory
- Haskell ecosystem
  - Libraries
  - Editors
  - QuickCheck, and property-based testing
    - Deprecates unit testing, for the most part
  - cabal
  - setup and flow
- Type theory
  - Haskell limitations here
  - Higher-kinded programming
  - GADTs
- Areas of active research in all of the above

Closing Words

- Leveraging functions,

Closing Words

- Leveraging functions,
- . . . leveraging types,

Closing Words

- Leveraging functions,
- . . . leveraging types,
- . . . enlisting several decades of research in programming languages,

Closing Words

- Leveraging functions,
- ...leveraging types,
- ...enlisting several decades of research in programming languages,
- ...and several more decades of research in math and logic

Closing Words

- Leveraging functions,
- ... leveraging types,
- ... enlisting several decades of research in programming languages,
- ... and several more decades of research in math and logic
- ... We can strive for a **functional** future!

Additional Resources

- Learn You a Haskell

Additional Resources

- Learn You a Haskell
- Real World Haskell

Additional Resources

- Learn You a Haskell
- Real World Haskell
- Stephen Diel's Essential Haskell

Additional Resources

- Learn You a Haskell
- Real World Haskell
- Stephen Diel's Essential Haskell
- Several friendly posts by Eric Rasmussen

Additional Resources

- Learn You a Haskell
- Real World Haskell
- Stephen Diel's Essential Haskell
- Several friendly posts by Eric Rasmussen
- So many good resources to really learn from!

Additional Resources

- Learn You a Haskell
- Real World Haskell
- Stephen Diel's Essential Haskell
- Several friendly posts by Eric Rasmussen
- So many good resources to really learn from!
    - Each word is a link – have fun!