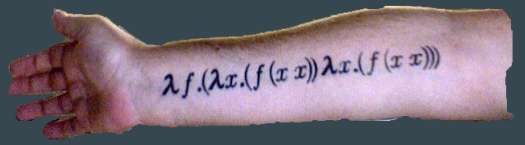
Pure Laziness: An Introduction to the Haskell Programming Language

Richard Townsend (Oberlin '13)
Columbia University

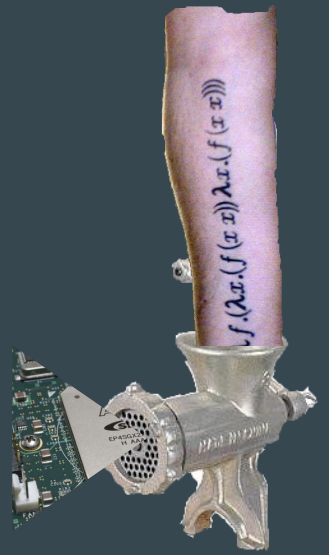
Richard's Route to Research



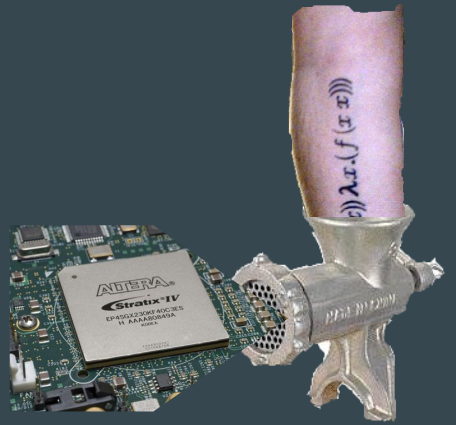
Richard's Route to Research

Richard's Route to Research

Richard's Route to Research



Richard's Route to Research



Richard's Route to Research

Bridging the abstraction gap

High-level

?

Programming Languages

Low-level

Python

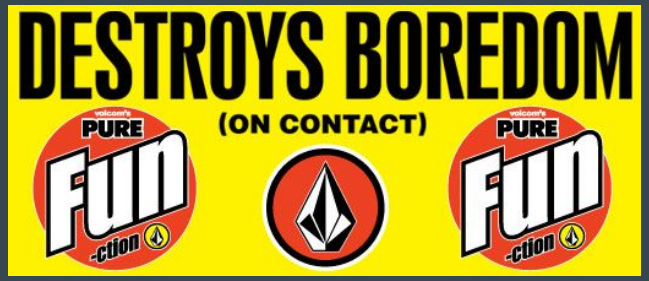
Java

C

Assembly

Haskell is Higher!

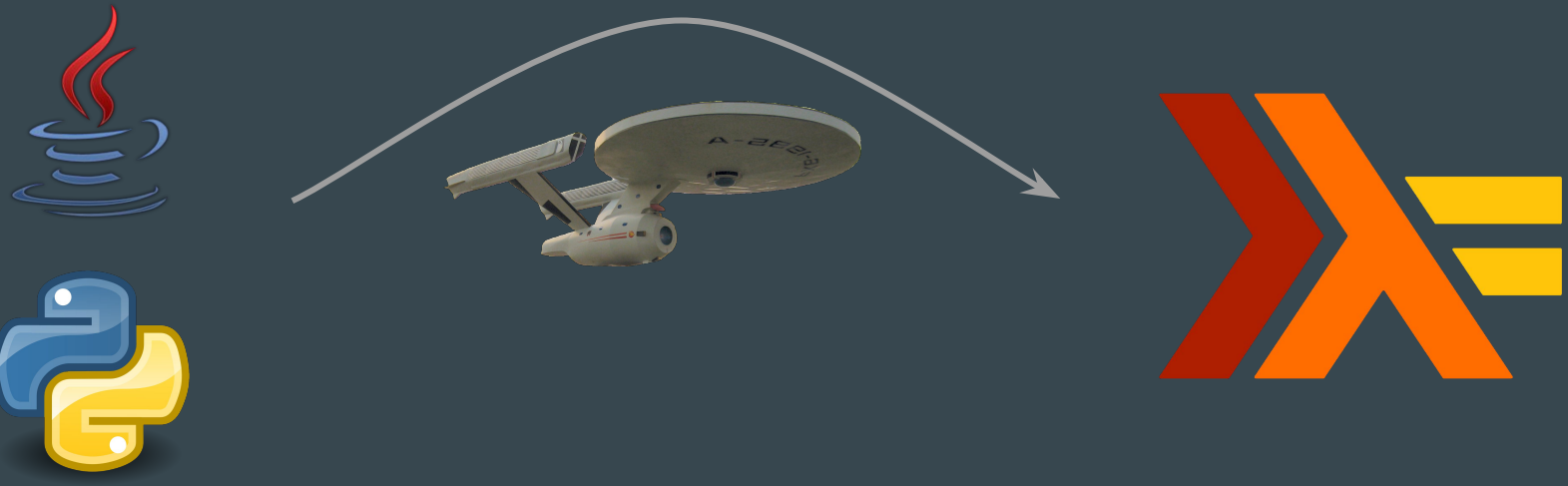
Pure functions



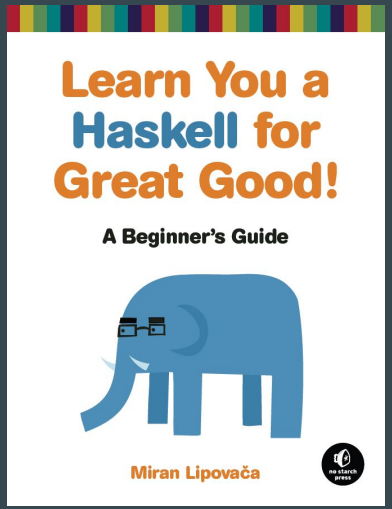
Powerful Type System

Lazy Evaluation

Exploring the Strange New World of Haskell



Exploring the Strange New World of Haskell



learnyouahaskell.com

Pure Functional Style

$f(x) = x^2 - 5$

$g(y) = f(4) + y$

```
f x = x^2 - 5

g y = f 4 + y
```

Pure Functional Style: Example

```
top3 doc = result
```

Input: A String representing a text document.

Output: A list of the 3 words that appear with highest frequency.

“In computer science functional programming is a programming paradigm a style of building the structure and elements of computer programs that treats computation as the evaluation of mathematical functions
and avoids changing state and mutable data It is a declarative programming paradigm which means programming is done with expressions”

http://www.onlamp.com/pub/a/onlamp/2007/07/12/introduction-to-haskell-pure-functions.html?page=

Pure Functional Style: Example

Input: A String representing a text document.

Output: A list of the 3 words that appear with highest frequency.

```
top3 doc = result
  where
    listOfWords = words doc
```

["In","computer","science","functional",
"programming","is","a","programming",
"paradigm","a","style","of","building","the",
"structure","and","elements","of",
"computer","programs","that","treats",
"computation","as","the","evaluation","of",
"mathematical","functions","and","avoids",
"changing","state","and","mutable","data",
"It","is","a","declarative","programming",
"paradigm","which","means",
"programming","is","done","with",
"expressions"]

http://www.onlamp.com/pub/a/onlamp/2007/07/12/introduction-to-haskell-pure-functions.html?page=

Pure Functional Style: Example

```
import Data.Char



top3 doc = result
  where
    listOfWords = words doc
    lowercase str = map toLower str
```

Input: A String representing a text document.

Output: A list of the 3 words that appear with highest frequency.

["In","computer","science","functional",
"programming","is","a","programming",
"paradigm","a","style","of","building","the",
"structure","and","elements","of",
"computer","programs","that","treats",
"computation","as","the","evaluation","of",
"mathematical","functions","and","avoids",
"changing","state","and","mutable","data",
"It","is","a","declarative","programming",
"paradigm","which","means",
"programming","is","done","with",
"expressions"]

http://www.onlamp.com/pub/a/onlamp/2007/07/12/introduction-to-haskell-pure-functions.html?page=

Pure Functional Style: Example

```
import Data.Char



top3 doc = result
  where
    listOfWords = words (lowercase doc)
    lowercase str = map toLower str
```

Input: A String representing a text document.

Output: A list of the 3 words that appear with highest frequency.

["in","computer","science","functional",
"programming","is","a","programming",
"paradigm","a","style","of","building","the",
"structure","and","elements","of",
"computer","programs","that","treats",
"computation","as","the","evaluation","of",
"mathematical","functions","and","avoids",
"changing","state","and","mutable","data",
"it","is","a","declarative","programming",
"paradigm","which","means",
"programming","is","done","with",
"expressions"]

http://www.onlamp.com/pub/a/onlamp/2007/07/12/introduction-to-haskell-pure-functions.html?page=

Pure Functional Style: Example

```
import Data.Char
import Data.List


top3 doc = result
  where
    listOfWords = words (lowercase doc)
    lowercase str = map toLower str
    wordGroups = sort listOfWords
```

Input: A String representing a text document.

Output: A list of the 3 words that appear with highest frequency.

["a","a","a","and","and","and","as","avoids","building","changing","computation","computer","computer","data","declarative","done","elements","evaluation","expressions","functional","functions","in","is","is","is","it","mathematical","means","mutable","of","of","of","paradigm","paradigm","programming","programming","programming","programming","programs","science","state","structure","style","that","the","the","treats","which","with"]

http://www.onlamp.com/pub/a/onlamp/2007/07/12/introduction-to-haskell-pure-functions.html?page=

Pure Functional Style: Example

```
import Data.Char
import Data.List


top3 doc = result
  where
    listOfWords = words (lowercase doc)
    lowercase str = map toLower str
    wordGroups = group (sort listOfWords)
```

Input: A String representing a text document.

Output: A list of the 3 words that appear with highest frequency.

```
[["a","a","a"],["and","and","and"],["as"],
["avoids"],["building"],["changing"],
["computation"],["computer","computer"],
["data"],["declarative"],["done"],
["elements"],["evaluation"],["expressions"],
["functional"],["functions"],["in"],
["is","is","is"],["it"],["mathematical"],
["means"],["mutable"],["of","of","of"],
["paradigm","paradigm"],
["programming","programming",
"programming","programming"],
["programs"],["science"],["state"],
["structure"],["style"],["that"],["the","the"],
["treats"],["which"],["with"]]
```

http://www.onlamp.com/pub/a/onlamp/2007/07/12/introduction-to-haskell-pure-functions.html?page=

Pure Functional Style: Example

```
import Data.Char
import Data.List
import Data.Ord

top3 doc = result
  where
    listOfWords = words (lowercase doc)
    lowercase str = map toLower str
    wordGroups = group (sort listOfWords)
    largestGroups = sortBy (comparing length) wordGroups
```

Input: A String representing a text document.

Output: A list of the 3 words that appear with highest frequency.

[["as"],["avoids"],["building"],["changing"],
["computation"],["data"],["declarative"],
["done"],["elements"],["evaluation"],
["expressions"],["functional"],["functions"],
["in"],["it"],["mathematical"],["means"],
["mutable"],["programs"],["science"],
["state"],["structure"],["style"],["that"],
["treats"],["which"],["with"],["computer",
"computer"],["paradigm","paradigm"],
["the","the"],["a","a","a"],["and","and",
"and"],["is","is","is"],["of","of","of"],
["programming","programming",
"programming","programming"]]

http://www.onlamp.com/pub/a/onlamp/2007/07/12/introduction-to-haskell-pure-functions.html?page=

Pure Functional Style: Example

```
import Data.Char
import Data.List
import Data.Ord

top3 doc = result
  where
    listOfWords = words (lowercase doc)
    lowercase str = map toLower str
    wordGroups = group (sort listOfWords)
    largestGroups = reverse (sortBy (comparing length) wordGroups)
```

Input: A String representing a text document.

Output: A list of the 3 words that appear with highest frequency.

[["programming","programming","programming","programming"],["of","of","of"],["is","is","is"],["and","and","and"],["a","a","a"],["the","the"],["paradigm","paradigm"],["computer","computer"],["with"],["which"],["treats"],["that"],["style"],["structure"],["state"],["science"],["programs"],["mutable"],["means"],["mathematical"],["it"],["in"],["functions"],["functional"],["expressions"],["evaluation"],["elements"],["done"],["declarative"],["data"],["computation"],["changing"],["building"],["avoids"],["as"]]

http://www.onlamp.com/pub/a/onlamp/2007/07/12/introduction-to-haskell-pure-functions.html?page=

Pure Functional Style: Example

```
import Data.Char
import Data.List
import Data.Ord

top3 doc = result
  where
    listOfWords = words (lowercase doc)
    lowercase str = map toLower str
    wordGroups = group (sort listOfWords)
    largestGroups = take 3 (reverse (sortBy (comparing length) wordGroups))
```

Input: A String representing a text document.

Output: A list of the 3 words that appear with highest frequency.

```
[["programming","programming",
"programming","programming"],["of","
of","of"],["is","is","is"]]
```

http://www.onlamp.com/pub/a/onlamp/2007/07/12/introduction-to-haskell-pure-functions.html?page=

Pure Functional Style: Example

```
import Data.Char
import Data.List
import Data.Ord

top3 doc = result
  where
    listOfWords = words (lowercase doc)
    lowercase str = map toLower str
    wordGroups = group (sort listOfWords)
    largestGroups = take 3 (reverse (sortBy (comparing length) wordGroups))
    result = map head largestGroups
```

Input: A String representing a text document.

Output: A list of the 3 words that appear with highest frequency.

["programming","of","is"]

http://www.onlamp.com/pub/a/onlamp/2007/07/12/introduction-to-haskell-pure-functions.html?page=

Powerful Type System: Explicit Types

```
listOfWords :: [String]
listOfWords = words (lowercase doc)
```

```
pythag :: Int -> Int -> Int -> Bool
pythag a b c = a^2 + b^2 == c^2
```

```
length :: [a] -> Int
length [] = 0
length (x:rest) = 1 + length rest
```

Powerful Type System: Define Your Own!

```
data Bool = True | False

data Shape = Square Float
           | Circle Float
           | Triangle Float Float Float

data Op = Add | Sub | Mul | Div

data Expr = Binop Expr Op Expr | Lit Int
```

Powerful Type System: Using your data types

```haskell
data Op = Add | Sub | Mul | Div
data Expr = Binop Expr Op Expr | Lit Int

eval :: Expr -> Int
eval (Lit x) = x
eval (Binop e1 Add e2) = eval e1 + eval e2
eval (Binop e1 Mul e2) = eval e1 * eval e2
eval (Binop e1 Sub e2) = eval e1 - eval e2
eval (Binop e1 Div e2) = eval e1 `div` eval e2
```

```haskell
eval (Lit 3)
=> 3
```

```haskell
eval (Binop (Lit 3) Add (Lit 4))
=> eval (Lit 3) + eval (Lit 4)
=> 3 + eval (Lit 4)
=> 3 + 4
=> 7
```

Executing Code

```
f :: Int -> Int -> Int
f x y = x + 1


f (27 + 2) (sum [1..10000000])
=> f 29 (sum [1..10000000])
=> ... computing sum ...
=> f 29 50000005000000
=> 29 + 1
=> 30
```

Laziness: I'll do it later...

```
f :: Int -> Int -> Int
f x y = x + 1


f (27 + 2) (sum [1..10000000])
=> (27 + 2) + 1
=> 29 + 1
=> 30
```

Laziness: To Infinity...

```
[1..10000000]
=> enumFromTo 1 10000000
=> 1 : enumFromTo (1 + 1) 10000000
=> 1 : 2 : enumFromTo (2 + 1) 10000000
=> … lots of calls …
=>  1 : 2 : … : 10000000 : []
```

```
enumFromTo :: Int -> Int -> [Int]
enumFromTo x y =
    if x > y
      then []
      else x : enumFromTo (x+1) y
```

```
[1..]
=> enumFrom 1
=> 1 : enumFrom (1 + 1)
=> 1 : 2 : enumFrom (2 + 1)
=> … lots of calls …
=>  1 : 2 : … : 10000000 : enumFrom (10000000 + 1)
=> … infinitely more calls …
```

```
enumFrom :: Int -> [Int]
enumFrom x = x : enumFrom (x+1)
```

Laziness: To Infinity...

```
take 2 [1..]
=> take 2 (enumFrom 1)
=> take 2 (1 : enumFrom 2)
=> 1 : take 1 (enumFrom 2)
=> 1 : take 1 (2 : enumFrom 3)
=> 1 : 2 : take 0 (enumFrom 3)
=> 1 : 2 : []
```

```
enumFrom :: Int -> [Int]
enumFrom x = x : enumFrom (x+1)

take :: Int -> [a] -> [a]
take 0 _  = []
take n [] = []
take n (x:xs) = x : take (n-1) xs
```

Laziness: ...and beyond!

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f [] _ = []
zipWith f _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

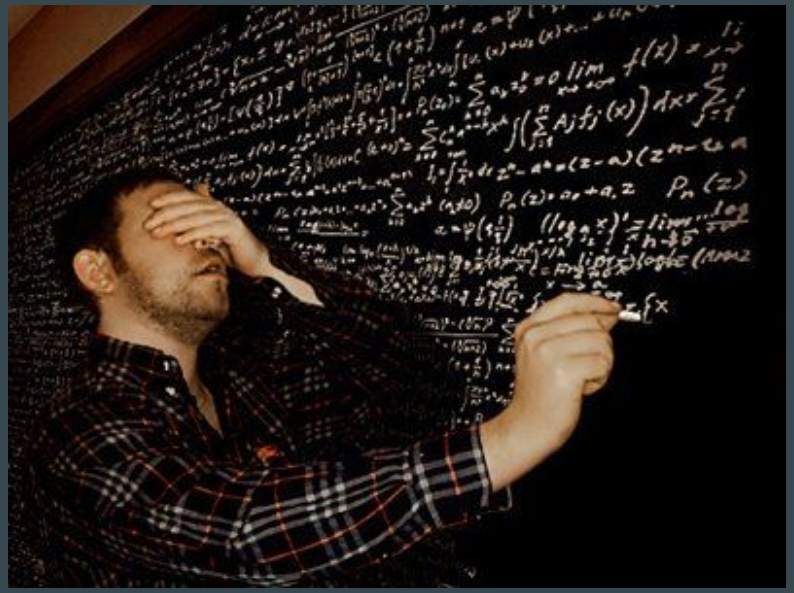
```
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
=> 1 : 1 : zipWith (+) (1:1:?) (1:?)
=> 1 : 1 : (1 + 1) : zipWith (+) (1:(1 + 1):?) ((1 + 1):?)
=> 1 : 1 : 2 : zipWith (+) (1:2:?) (2:?)
=> 1 : 1 : 2 : 3 : zipWith (+) (2:3:?) (3:?)
=> ...
```

```
fibs !! 0
=> 1
```

```
fibs !! 6
=> 13
```

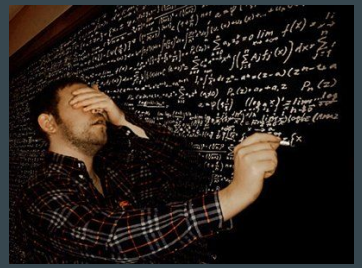
```
fibs !! 100
=> 573147844013817084101
```

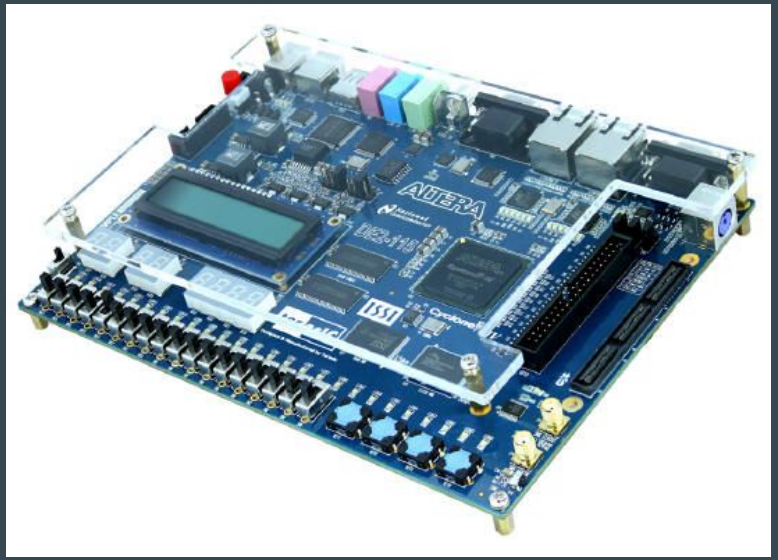
Proofs Are Trivial



$$
\begin{aligned}
&\texttt{unpack(pack(v))} \\
&= \texttt{unpack(pack(R p (R y z}^{k}\texttt{)}^{k}\texttt{))} \\
&= \texttt{unpack(P p (y (pack z)}^{k}\texttt{)}^{k}\texttt{)} \\
&= \texttt{R p (R y (unpack (pack z))}^{k}\texttt{)}^{k} \\
&= \texttt{R p (R y z}^{k}\texttt{)}^{k} \\
&= \texttt{v}
\end{aligned}
$$

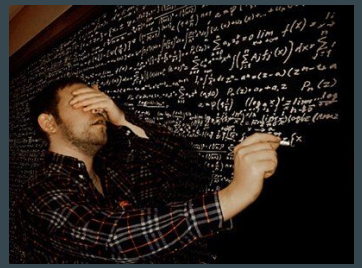
Embedding Languages

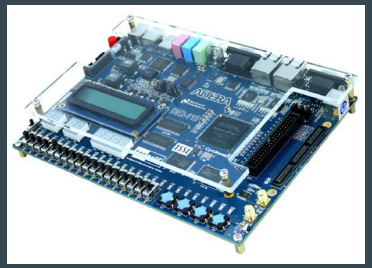


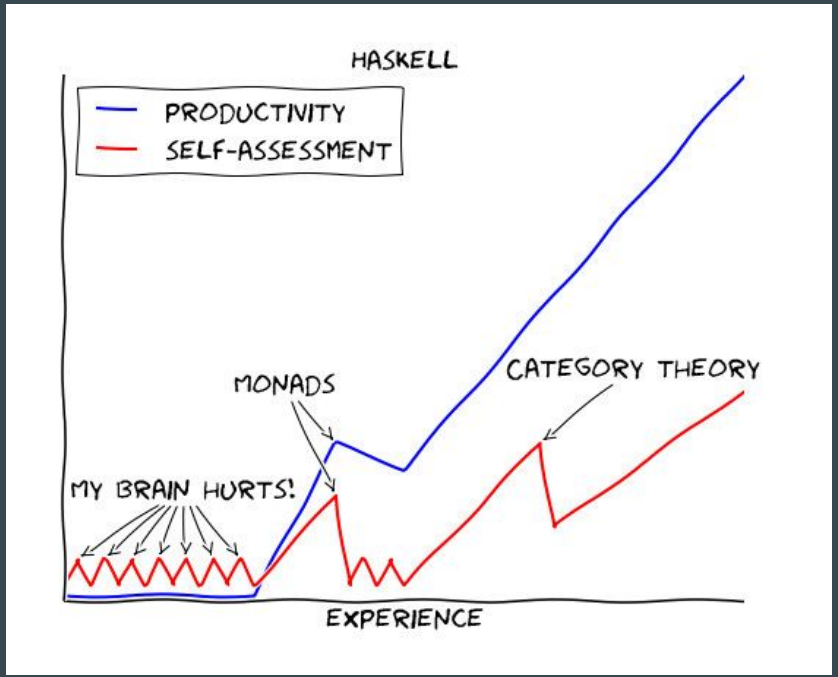



```
data Stream a = a :> (Stream a)

data MemOp addr word = MemRead addr
                     | MemWrite addr word


zipWith :: (a -> b -> c) ->
           Stream a -> Stream b -> Stream c
zipWith f (a :> as) (b :> bs) =
    f a b :> zipWith f as bs
```

Category Theory








```
class Applicative m => Monad m where

    (>>=)       :: forall a b. m a -> (a -> m b) -> m b

    (>>)        :: forall a b. m a -> m b -> m b
    m >> k = m >>= \_ -> k

    return      :: a -> m a
    return      = pure

    fail        :: String -> m a
    fail s      = error s
```

Thank you!