FizzBuzz In Haskell

Owen Lynch

April 25th 2018

Introducing Haskell

```haskell
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
```

Introducing Haskell

```haskell
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
```

Notable Features

Introducing Haskell

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
```

Notable Features

- Types

Introducing Haskell

```haskell
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
```

Notable Features

- Types
- Functions

Introducing Haskell

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
```

Notable Features

- Types
- Functions
- Pattern Matching

Introducing Haskell

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
```

Notable Features

- Types
- Functions
- Pattern Matching
- Recursion

Myths about Haskell

- Haskell is hard

Myths about Haskell

- Haskell is hard
  - *Learning* Haskell is hard, *programming* Haskell is easy!

Myths about Haskell

- Haskell is hard
  - *Learning* Haskell is hard, *programming* Haskell is easy!
  - Haskell lets you choose the level of abstraction that you are comfortable with, and you can slowly increase this while being productive the whole time

Myths about Haskell

- Haskell is hard
  - *Learning* Haskell is hard, *programming* Haskell is easy!
  - Haskell lets you choose the level of abstraction that you are comfortable with, and you can slowly increase this while being productive the whole time
- Haskell is for techbro superprogrammers

Myths about Haskell

- Haskell is hard
  - *Learning* Haskell is hard, *programming* Haskell is easy!
  - Haskell lets you choose the level of abstraction that you are comfortable with, and you can slowly increase this while being productive the whole time
- Haskell is for techbro superprogrammers
  - Haskell is for people who can't keep lots of stuff in their head at once and want the compiler to make sure they aren't messing things up

Myths about Haskell

- Haskell is hard
  - *Learning* Haskell is hard, *programming* Haskell is easy!
  - Haskell lets you choose the level of abstraction that you are comfortable with, and you can slowly increase this while being productive the whole time
- Haskell is for techbro superprogrammers
  - Haskell is for people who can't keep lots of stuff in their head at once and want the compiler to make sure they aren't messing things up
  - The Haskell community is one of the nicest and friendliest online communities I've been a part of, in part because there is so much to learn that everyone is comparatively a noob

Myths about Haskell

- Haskell is hard
  - *Learning* Haskell is hard, *programming* Haskell is easy!
  - Haskell lets you choose the level of abstraction that you are comfortable with, and you can slowly increase this while being productive the whole time
- Haskell is for techbro superprogrammers
  - Haskell is for people who can't keep lots of stuff in their head at once and want the compiler to make sure they aren't messing things up
  - The Haskell community is one of the nicest and friendliest online communities I've been a part of, in part because there is so much to learn that everyone is comparatively a noob
- Haskell is for academic ivory-towerists who do too much category theory for their own good

Myths about Haskell

- Haskell is hard
  - *Learning* Haskell is hard, *programming* Haskell is easy!
  - Haskell lets you choose the level of abstraction that you are comfortable with, and you can slowly increase this while being productive the whole time
- Haskell is for techbro superprogrammers
  - Haskell is for people who can't keep lots of stuff in their head at once and want the compiler to make sure they aren't messing things up
  - The Haskell community is one of the nicest and friendliest online communities I've been a part of, in part because there is so much to learn that everyone is comparatively a noob
- Haskell is for academic ivory-towerists who do too much category theory for their own good
  - Haskell is actually a very practical language for all sorts of tasks, and it has been battletested in industry for decades

However. . .

- In this talk we will not be doing anything practical.

However. . .

- In this talk we will not be doing anything practical.
- This is a silly tour of some neat features in Haskell.

However...

- In this talk we will not be doing anything practical.
- This is a silly tour of some neat features in Haskell.
- I am not expecting all of this talk to make sense to you right away.

However. . .

- In this talk we will not be doing anything practical.
- This is a silly tour of some neat features in Haskell.
- I am not expecting all of this talk to make sense to you right away.
- When you see something you don't understand, don't think "Agghhh Haskell is way to hard", think "Cool, I have something to figure out!"

Introducing FizzBuzz

- Make a list of the numbers from 1 to 100, except...

Introducing FizzBuzz

- Make a list of the numbers from 1 to 100, except. . .
  - For every number divisible by 3, put "Fizz" instead

Introducing FizzBuzz

- Make a list of the numbers from 1 to 100, except...
  - For every number divisible by 3, put "Fizz" instead
  - For every number divisible by 5, put "Buzz" instead

Introducing FizzBuzz

- Make a list of the numbers from 1 to 100, except. . .
  - For every number divisible by 3, put "Fizz" instead
  - For every number divisible by 5, put "Buzz" instead
  - For every number divisible by 5 and 3, put "FizzBuzz" instead

Introducing FizzBuzz

- Make a list of the numbers from 1 to 100, except...
  - For every number divisible by 3, put “Fizz” instead
  - For every number divisible by 5, put “Buzz” instead
  - For every number divisible by 5 and 3, put “FizzBuzz” instead
- How would you solve FizzBuzz?

Lists in Haskell

How can we make a list until we know what a list *is*?

```haskell
data [a] = [] | a : [a]
-- data List a = Empty | Cons a (List a)
-- (:) x xs == x : xs
-- ([]) a = [a]
```

- New Stuff:

Lists in Haskell

How can we make a list until we know what a list *is*?

```
data [a] = [] | a : [a]
-- data List a = Empty | Cons a (List a)
-- (:) x xs == x : xs
-- ([]) a = [a]
```

- New Stuff:
  - Data Declarations

Lists in Haskell

How can we make a list until we know what a list *is*?

```
data [a] = [] | a : [a]
-- data List a = Empty | Cons a (List a)
-- (:) x xs == x : xs
-- ([]) a = [a]
```

- New Stuff:
  - Data Declarations
  - Parametric data declarations

Lists in Haskell

How can we make a list until we know what a list *is*?

```
data [a] = [] | a : [a]
-- data List a = Empty | Cons a (List a)
-- (:) x xs == x : xs
-- ([]) a = [a]
```

- New Stuff:
  - Data Declarations
  - Parametric data declarations
  - Special syntax for stuff

What does it look like?

```haskell
firstFourPrimes :: [Int]
firstFourPrimes = 2:(3:(5:(7:[])))

type String = [Char]
-- "abc" = ['a', 'b', 'c']

everFlavoredBeanFlavors :: [String]
everFlavoredBeanFlavors =
  ["earwax", "vomit", "marmalade", "spinach"]
```

How do we work with it?

Recursion!

Our first step is generating all of the numbers from 1 to 100.

```
range :: Int -> Int -> [Int]
range n m
  | n == m    = [n]
  | otherwise = n:(range (n+1) m)
-- range n m == [n..m]
```

How do we work with it?

Recursion!

Our first step is generating all of the numbers from 1 to 100.

```
range :: Int -> Int -> [Int]
range n m
  | n == m    = [n]
  | otherwise = n:(range (n+1) m)
-- range n m == [n..m]
```

- New stuff:

How do we work with it?

Recursion!

Our first step is generating all of the numbers from 1 to 100.

```
range :: Int -> Int -> [Int]
range n m
  | n == m    = [n]
  | otherwise = n:(range (n+1) m)
-- range n m == [n..m]
```

- New stuff:
  - Currying

How do we work with it?

Recursion!

Our first step is generating all of the numbers from 1 to 100.

```
range :: Int -> Int -> [Int]
range n m
  | n == m    = [n]
  | otherwise = n:(range (n+1) m)
-- range n m == [n..m]
```

- New stuff:
  - Currying
  - Guard Notation

List Transformations

Now, we need to do something to each of those numbers.

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

List Transformations

Now, we need to do something to each of those numbers.

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

- New stuff:

List Transformations

Now, we need to do something to each of those numbers.

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

- New stuff:
  - Functions applied to functions!

The first solution

```haskell
fizzbuzz1 :: Int -> String
fizzbuzz1 n
  | rem n 15 == 0 = "FizzBuzz"
  | rem n  5 == 0 = "Buzz"
  | rem n  3 == 0 = "Fizz"
  | otherwise     = show n

sol1 :: [String]
sol1 = map fizzbuzz1 [1..100]
```

Discuss first solution

- What did we like?

Discuss first solution

- What did we like?
  - Recursion!

Discuss first solution

- What did we like?
  - Recursion!
  - Combinators that allowed us to not do recursion!

Discuss first solution

- What did we like?
  - Recursion!
  - Combinators that allowed us to not do recursion!
  - Case syntax!

Discuss first solution

- What did we like?
  - Recursion!
  - Combinators that allowed us to not do recursion!
  - Case syntax!
- What did we not like?

Discuss first solution

- What did we like?
  - Recursion!
  - Combinators that allowed us to not do recursion!
  - Case syntax!
- What did we not like?
  - Not extensible enough! (What about Bazz??)

FizzBuzzBazz

```haskell
fizzbuzzbazz1 :: Int -> String
fizzbuzzbazz1 n
  | rem n 105 == 0 = "FizzBuzzBazz"
  | rem n  35 == 0 = "BuzzBazz"
  | rem n  21 == 0 = "FizzBazz"
  | rem n  15 == 0 = "FizzBuzz"
  | rem n   7 == 0 = "Bazz"
  | rem n   5 == 0 = "Buzz"
  | rem n   3 == 0 = "Fizz"
  | otherwise      = show n
```

FizzBuzzBazz

```haskell
fizzbuzzbazz1 :: Int -> String
fizzbuzzbazz1 n
  | rem n 105 == 0 = "FizzBuzzBazz"
  | rem n  35 == 0 = "BuzzBazz"
  | rem n  21 == 0 = "FizzBazz"
  | rem n  15 == 0 = "FizzBuzz"
  | rem n   7 == 0 = "Bazz"
  | rem n   5 == 0 = "Buzz"
  | rem n   3 == 0 = "Fizz"
  | otherwise      = show n
```

This is terrible!

How should we solve this?

- We need to be able to compose different “zz”s

How should we solve this?

- We need to be able to compose different “zz”s
- To do this, we will use a very common structure… monoids!

Monoid Hype

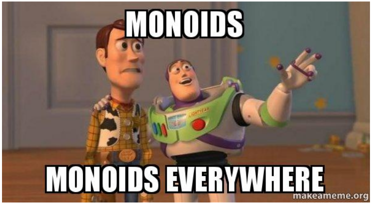
"Monoids are ubiquitous throughout programming. The difference is that in Haskell we recognize and talk about them."

– Real World Haskell

Monoid Hype

"Monoids are ubiquitous throughout programming. The difference is that in Haskell we recognize and talk about them."

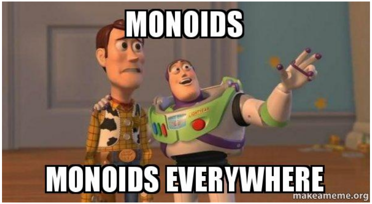
– Real World Haskell

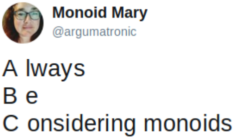


Monoid Hype

"Monoids are ubiquitous throughout programming. The difference is that in Haskell we recognize and talk about them."

– Real World Haskell





OK, yeah, but what are these things?

- A monoid is a data type `m` with

OK, yeah, but what are these things?

- A monoid is a data type `m` with
  - a function `(<>) :: m -> m -> m`. . .

OK, yeah, but what are these things?

- A monoid is a data type `m` with
  - a function `(<>) :: m -> m -> m`...
  - a special element `mempty :: m`

OK, yeah, but what are these things?

- A monoid is a data type `m` with
  - a function `(<>) :: m -> m -> m`...
  - a special element `mempty :: m`
- It should also be "nice" – it should satisfy

OK, yeah, but what are these things?

- A monoid is a data type `m` with
  - a function `(<>) :: m -> m -> m`...
  - a special element `mempty :: m`
- It should also be "nice" – it should satisfy
  - `a <> (b <> c) == (a <> b) <> c`

OK, yeah, but what are these things?

- A monoid is a data type `m` with
  - a function `(<>) :: m -> m -> m`...
  - a special element `mempty :: m`
- It should also be "nice" – it should satisfy
  - `a <> (b <> c) == (a <> b) <> c`
  - `a <> mempty == a == mempty <> a`

You already know lots of monoids!

Take a couple minutes and think of monoids

- Any type of number ($\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$) along with addition

You already know lots of monoids!

Take a couple minutes and think of monoids

- Any type of number ($\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$) along with addition
- Any type of number along with multiplication

You already know lots of monoids!

Take a couple minutes and think of monoids

- Any type of number ($\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$) along with addition
- Any type of number along with multiplication
- Matrices with addition

You already know lots of monoids!

Take a couple minutes and think of monoids

- Any type of number ($\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$) along with addition
- Any type of number along with multiplication
- Matrices with addition
- Matrices with multiplication (non-commutative!)

You already know lots of monoids!

Take a couple minutes and think of monoids

- Any type of number ($\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$) along with addition
- Any type of number along with multiplication
- Matrices with addition
- Matrices with multiplication (non-commutative!)
- Any ordered set along with max and a "negative infinity" element as identity

You already know lots of monoids!

Take a couple minutes and think of monoids

- Any type of number ($\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$) along with addition
- Any type of number along with multiplication
- Matrices with addition
- Matrices with multiplication (non-commutative!)
- Any ordered set along with max and a “negative infinity” element as identity
- Any ordered set along with min and a “positive infinity” element as identity

Some nitty-gritties. . .

In Haskell, we write the definition of a Monoid like this

```
class Monoid m where
  mempty :: m
  (<>) :: m -> m -> m
```

Unfortunately, we can't write down the laws. . . we just have to trust that whenever someone implements a Monoid they make sure they satisfy them!

List Monoid

```
instance Monoid [a] where
  mempty = []
  [] <> ys = ys
  (x:xs) <> ys = x:(xs <> ys)
```

Maybe Monoid

```
data Maybe a = Just a | Nothing

instance (Monoid a) => Monoid (Maybe a) where
  mempty = Nothing
  Nothing <> Nothing = Nothing
  (Just a) <> Nothing = Just a
  Nothing <> (Just b) = Just b
  (Just a) <> (Just b) = Just (a <> b)
```

Working with Maybes

```
fromMaybe :: a -> Maybe a -> a
fromMaybe def Nothing = def
fromMaybe _ (Just y) = y
```

Combining lists

```haskell
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f acc [] = acc
foldl f acc (x:xs) = foldl f (f acc x) xs
-- sum == foldl (+) 0
-- length == foldl (\n _ -> n + 1) 0

mconcat :: (Monoid m) => [m] -> m
mconcat = foldl (<>) mempty
-- mconcat ["tweedle", "dee"] == "tweedledee"
```

FizzBuzz Revisited

```haskell
fizzbuzz2 :: [(Int, String)] -> Int -> String
fizzbuzz2 conds n = fromMaybe (show n) mdesc
  where
    mdesc = mconcat (map maybeWord conds)
    maybeWord (k, word)
      | rem n k == 0 = Just word
      | otherwise    = Nothing

sol2 :: [String]
sol2 = map (fizzbuzz2 conds) [1..100]
  where conds = [(3, "Fizz"), (5, "Buzz")]
```

Reflections on take 2

- What do we like?

Reflections on take 2

- What do we like?
  - Monoids!

Reflections on take 2

- What do we like?
  - Monoids!
  - Currying!

Reflections on take 2

- What do we like?
  - Monoids!
  - Currying!
  - Much more extensible!

Reflections on take 2

- What do we like?
  - Monoids!
  - Currying!
  - Much more extensible!
- What do we not like?

Reflections on take 2

- What do we like?
  - Monoids!
  - Currying!
  - Much more extensible!
- What do we not like?
  - Still not extensible enough!

FizzBuzz: Hard Mode

- FizzBuzz, but…

FizzBuzz: Hard Mode

- FizzBuzz, but...
  - For every prime number, put “Buzz” instead

FizzBuzz: Hard Mode

- FizzBuzz, but...
  - For every prime number, put "Buzz" instead
  - For every fibonacci number, put "Fizz" instead

FizzBuzz: Hard Mode

- FizzBuzz, but...
  - For every prime number, put “Buzz” instead
  - For every fibonacci number, put “Fizz” instead
  - For every number that is a fibonacci number and a prime number, put “FizzBuzz” instead

Infinity and Beyond!

```
repeat :: a -> [a]
repeat x = x:(repeat x)
-- take 3 (repeat 1) = [1,1,1]

primes :: [Int]
primes = from 2
  where
    from k = k:(filter (ndiv k) (from (k+1)))
    ndiv k n = rem n k /= 0
-- take 4 primes == [2,3,5,7]
```

Zipping Lists

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

fibs :: [Int]
fibs = 1:2:(zipWith (+) fibs (tail fibs))
-- This one's a braintwister!
```

Lists as Monoids Revisited

```
newtype Stream a = S [a]

unS :: Stream a -> [a]
unS (S xs) = xs

instance Monoid a => Monoid (Stream a) where
  mempty = S []
  (S []) <> (S ys) = S ys
  (S xs) <> (S []) = S xs
  (S (x:xs)) <> (S (y:ys)) = S ((x <> y):rest)
    where (S rest) = (S xs) <> (S ys)
```

Conceptually, when the list ends, we treat the rest as an infinite stream of `mempty`s (this might make more sense in the context of the next slide)

Thoughts on Streams

`Stream Double` is a vector space, where Double is the type of real numbers (OK, floating point numbers) in Haskell.

```
scalarMult :: Double -> Stream Double -> Stream Double
scalarMult r (S xs) = S (map (r*) xs)

addV :: Stream Double -> Stream Double -> Stream Double
addV = (<>)
-- The monoid instance for Double uses addition

negV :: Stream Double -> Stream Double
negV = scalarMult (-1)
```

This is the same thing as the vector space of polynomials of one variable over $\mathbb{R}$.

$[1, 0, \pi] \approx 1 + \pi x^2$

FizzBuzz: The Final Showdown

```haskell
spacer :: [Int] -> a -> [Maybe a]
spacer (n:ns) s = loop (n-1) (n:ns)
  where
    loop 0 (k1:k2:ns) =
      (Just s):(loop (k2-k1-1) (k2:ns))
    loop k ns = Nothing:(loop (k-1) ns)

combine :: (Monoid a) => [([Int], a)] -> [Maybe a]
combine = unS . mconcat . map (S . uncurry spacer)

sol3 :: [String]
sol3 = zipWith fromMaybe (map show [1..100]) descs
  where descs = combine [(primes,"Fizz"), (fibs,"Buzz")]
```

Reflecting on the Final Showdown

- What we liked

Reflecting on the Final Showdown

- What we liked
  - Really really extensible!

Reflecting on the Final Showdown

- What we liked
  - Really really extensible!
  - Argument-free function definition!

Reflecting on the Final Showdown

- What we liked
  - Really really extensible!
  - Argument-free function definition!
- What we didn't like

Reflecting on the Final Showdown

- What we liked
  - Really really extensible!
  - Argument-free function definition!
- What we didn't like
  - Really hard to understand!

Reflecting on the Final Showdown

- What we liked
  - Really really extensible!
  - Argument-free function definition!
- What we didn't like
  - Really hard to understand!
- There should always be a balance, no matter what your programming language is. The difference is, in Haskell you have many more choices about how you balance.

Resources

- Resources for learning haskell

Resources

- Resources for learning haskell
  - Learn You a Haskell for Great Good

Resources

- Resources for learning haskell
  - Learn You a Haskell for Great Good
    - FREE online at learnyouahaskell.com

Resources

- Resources for learning haskell
  - Learn You a Haskell for Great Good
    - FREE online at learnyouahaskell.com
  - Haskell Programming From First Principles

Resources

- Resources for learning haskell
  - Learn You a Haskell for Great Good
    - FREE online at learnyouahaskell.com
  - Haskell Programming From First Principles
    - REALLY GOOD, NOT FREE, online at haskellbook.com

Resources

- Resources for learning haskell
  - Learn You a Haskell for Great Good
    - FREE online at learnyouahaskell.com
  - Haskell Programming From First Principles
    - REALLY GOOD, NOT FREE, online at haskellbook.com
  - Real World Haskell

Resources

- Resources for learning haskell
  - Learn You a Haskell for Great Good
    - FREE online at learnyouahaskell.com
  - Haskell Programming From First Principles
    - REALLY GOOD, NOT FREE, online at haskellbook.com
  - Real World Haskell
    - PRETTY GOOD, FREE, google for pdf or email me

Resources

- Resources for learning haskell
  - Learn You a Haskell for Great Good
    - FREE online at learnyouahaskell.com
  - Haskell Programming From First Principles
    - REALLY GOOD, NOT FREE, online at haskellbook.com
  - Real World Haskell
    - PRETTY GOOD, FREE, google for pdf or email me
- Resources for using Haskell

Resources

- Resources for learning haskell
  - Learn You a Haskell for Great Good
    - FREE online at learnyouahaskell.com
  - Haskell Programming From First Principles
    - REALLY GOOD, NOT FREE, online at haskellbook.com
  - Real World Haskell
    - PRETTY GOOD, FREE, google for pdf or email me
- Resources for using Haskell
  - stack, a package manager for Haskell

Resources

- Resources for learning haskell
  - Learn You a Haskell for Great Good
    - FREE online at learnyouahaskell.com
  - Haskell Programming From First Principles
    - REALLY GOOD, NOT FREE, online at haskellbook.com
  - Real World Haskell
    - PRETTY GOOD, FREE, google for pdf or email me
- Resources for using Haskell
  - stack, a package manager for Haskell
    - haskellstack.org

Resources

- Resources for learning haskell
  - Learn You a Haskell for Great Good
    - FREE online at learnyouahaskell.com
  - Haskell Programming From First Principles
    - REALLY GOOD, NOT FREE, online at haskellbook.com
  - Real World Haskell
    - PRETTY GOOD, FREE, google for pdf or email me
- Resources for using Haskell
  - stack, a package manager for Haskell
    - haskellstack.org
  - haskell.org, the center of all things Haskell

Resources

- Resources for learning haskell
  - Learn You a Haskell for Great Good
    - FREE online at learnyouahaskell.com
  - Haskell Programming From First Principles
    - REALLY GOOD, NOT FREE, online at haskellbook.com
  - Real World Haskell
    - PRETTY GOOD, FREE, google for pdf or email me
- Resources for using Haskell
  - stack, a package manager for Haskell
    - haskellstack.org
  - haskell.org, the center of all things Haskell
  - hoogle, a search engine for functions, data types, etc. (REALLY USEFUL)

Thank You

Thank You

Shout out to my beta testers Max, Megan and Shelley, who (will) give me valuable feedback.

Thank You

Shout out to my beta testers Max, Megan and Shelley, who (will) give me valuable feedback.

Thank you all for coming and being curious!