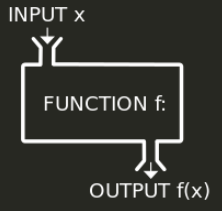
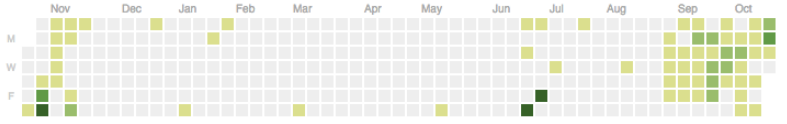
Strategies for Code Correctness

Inspired by Functional Frogramming



@chrismwendt / GitHub Engineering All-Hands Meeting 2015-10-22

Chris Wendt - @chrismwendt - @github/search



Haskell

Idris

`map`, `select`, and `reduce`

Imperative compared to Functional array processing

```ruby
def sum_of_even_squares(array)
  sum = 0
  for n in array
    square = n * n
    sum += square if square.even?
  end
  sum
end
```

```ruby
def sum_of_even_squares(array)
  array
    .map { |n| n * n }
    .select { |n| n.even? }
    .reduce(0) { |sum, n| sum + n }
end
```

Pure

- Math `+`, `cos`, `sqrt`
- String operations `split`, `MD5.hexdigest`, `reverse`

Impure

- IO `gets`, `File.write`, `HTTP::get`, `%x{rm -rf /}`
- Nondeterminism `rand`, `Time.now`
- `$global` mutable state

`map`, `select`, and `reduce`

Pure functions

Impure

```ruby
def truncate_lines
  n = ARGV[0].to_i
  puts($stdin.readlines.map { |line| line[0...n] })
end
```

Pure

```ruby
def truncate_lines(n, lines)
  lines.map { |line| line[0...n] })
end

def main
  puts(truncate_lines(ARGV[0].to_i, $stdin.readlines))
end
```

`map`, `select`, and `reduce`

Pure functions

Composition

```
def parse_query(string)
  filters = []
  until scanner.eos?
    if scanner.scan('([a-z]+):([a-z]+)')
      filters << {
        :key   => scanner[0],
        :value => scanner[1] }
    elsif ...
  end
  filters
end
```

```
class QueryParser < Parslet::Parser
  rule(:word) { match('[a-z]+') }
  rule(:filter) { (word.as(:key) >> str(':') >>
    word.as(:value)).as(:filter) }
  rule(:query) { (filter | ...).repeat }
  root(:query)
end
```

`map`, `select`, and `reduce`

Pure functions

Composition

Immutable values

Hypothetical production code

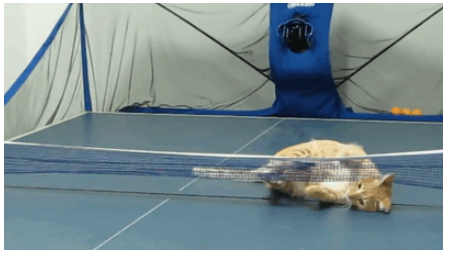
```
def index(commit)
  elasticsearch.index(commit)
  log(commit)
end

def log(commit)
  puts commit.keep_if { |k, v|
    ['author', 'id'].include? k }
end
```


Alex Shchepetilnikov
@irqed
99 little bugs in the code
99 little bugs in the code
Take one down, patch it around
117 little bugs in the code
RETWEETS
22,592
FAVORITES
10,457

More correctness with static types

- Banish null (`Optional` only when you want it)
- Use algebraic data types (fewer invalid representations of data)
- Limit effects in types (prevent a function from performing IO)



Banish null

1. Eliminate the ~~`null`~~ keyword

2. Introduce the type `Optional a`

```
data Optional a = None | Some a

divide : Float -> Float -> Optional Float
divide a 0 = None
divide a b = Some (a / b)

case (divide 3 0) of
  None -> "failed, due to division by 0"
  Some x -> "succeeded"
```

4. PROFIT!!!

Representing geometric shapes

```
data Point = MakePoint Float Float

data Shape = Circle Float
           | NGon Int Float
           | Polygon (List Point)

bad : Shape
bad = Polygon 3 -- COMPILE ERROR

radius : Shape -> Optional Float
radius (Circle r) = Some r
radius (NGon _ r) = Some r
radius _          = None
```

These functions CANNOT perform IO!

```
radius : Shape -> Optional Float
length : String -> Int
```

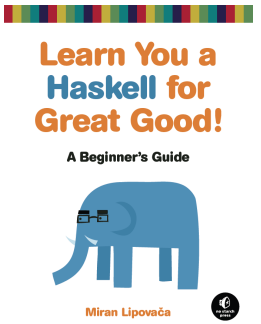
But these can

```
getLine : IO String
printLine : String -> IO ()
```

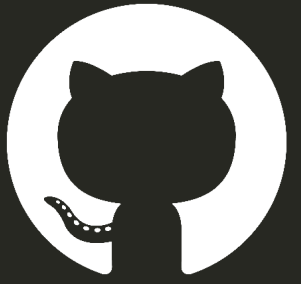
No function with this type signature exists

```
IO a -> a
```

Learn You a Haskell



More academic: https://github.com/bitemyapp/learnhaskell



@chrismwendt