Overview

- Call-by-value vs Call-by-name vs Call-by-need
- Introduction and Elimination Forms
- Applications to software engineering
Haskell is Lazy

• Haskell does not evaluate things until it absolutely must
  • What does that mean?

• Haskell does not evaluate things more than once
  • Why would it?
Call-by-Value

\[
\begin{aligned}
f &:: \text{Int} \to \text{Int} \\
f \; x &= x - 1
\end{aligned}
\]

\[
\begin{aligned}
\text{>>> } f \; 5 \\
4
\end{aligned}
\]
Call-by-Value

\[ f :: \text{Int} \to \text{Int} \to \text{Int} \]
\[ f \ x \ y = \text{if} \ x = 0 \ \text{then} \ 0 \ \text{else} \ y \]

\[ g :: \text{Int} \to \text{Int} \]
\[ g \ 0 = 0 \]
\[ g \ n = n + g(n-1) \]

First computes \( g \) down to a \textbf{value} then it evaluates \( f \)

Replaces the variable with the value computed
Call-by-Name

\[ f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[ f \ x \ y = \text{if } x = 0 \text{ then } 0 \text{ else } y \]

\[ g :: \text{Int} \rightarrow \text{Int} \]
\[ g \ 0 = 0 \]
\[ g \ n = n + g(n-1) \]

\[ f \ 0 \ (g(1000000000)) \]

Does not compute \( g \) until it is needed to be computed

Replaces the variable with the means of computing the variable
Call-by-Name isn’t always better…

\[
f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
f \ x \ y = \text{if } x = 0 \text{ then } 0 \text{ else } y+y
\]

\[
g :: \text{Int} \rightarrow \text{Int} \\
g \ 0 = 0 \\
g \ n = n+g(n-1)
\]

\[
f \ 0 \ (g(1000000000))
\]

Does not compute \(g\) until it is needed to be computed
Replaces the variable with the means of computing the variable
Recomputes \(y\)!
Call-by-Need

\[
\begin{align*}
\text{f} &:: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
\text{f} \ x \ y &= \text{if } x = 0 \text{ then } 0 \text{ else } y + y \\
\text{g} &:: \text{Int} \rightarrow \text{Int} \\
\text{g} \ 0 &= 0 \\
\text{g} \ n &= n + \text{g}(n-1) \\
\text{f} \ 0 \ (\text{g}(1000000000)) \\
\end{align*}
\]

Does not compute g until it is needed to be computed
Replaces the variable with the means of computing the variable
Computations are shared across occurrences
Functions aren’t the only delayed evaluation

```haskell
g :: Int -> Int
g 0 = 0
g n = n + g(n-1)
```

```haskell
x = (g 100000000):(g 100000000):[]
```

```haskell
length x
>>> 2
```

Constructors also don’t need to evaluate!
So what does require evaluation?

And what does not?
Introduction Forms & Function Arguments Do Not Require Evaluation

• Introduction forms are syntax that build up bigger things

• Essentially, when building bigger things, you don’t need to evaluate the smaller things

• So you have big things that have non-computed values within them
Introduction Forms & Function Arguments Do Not Require Evaluation

\[
g :: \text{Int} \to \text{Int}
g \ 0 = 0
g \ n = n + g(n - 1)
\]

\[
x = (g \ 100000000) : (g \ 100000000) : []
\]

\[
y = \text{Node} (\text{Leaf}, (g \ 100000000), \text{Leaf})
\]

\[
z = (g \ 100000000, g \ 100000000)
\]
Elimination Forms and Do Require Evaluation

- What are elimination forms?

- Things that break big things down into little things

- In Haskell, all elimination forms except one can be addressed in pattern matching
Pattern Matching Elimination

- Basically evaluates as little as possible to know whether it matches a pattern or not

```haskell
isEmpty :: [a] -> Bool
isEmpty [] = True
isEmpty _  = False

isEmpty (repeat (g 100000000))
>>> False
```
Other form of Elimination

Function Calls (for the function)!
Function Calls Elimination

\[
f : \text{Bool} \rightarrow \text{Int} \rightarrow \text{Int}
f \text{ True } i = i 
f \text{ False } i = i + 1
\]

Evaluate until the function is a lambda, then apply the lambda

\[
f \text{ True}
f \text{ True } 2
\]
Function Calls Elimination

\[ f : \text{Bool} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[ f \ b = \text{if } b \text{ == True then } \lambda \ i \rightarrow i \text{ else } \lambda \ i \rightarrow i+1 \]

Evaluate until the function is a lambda, then apply the lambda

\[ f \ \text{True} \]
\[ f \ \text{True 2} \]
Cool Uses in Haskell

```haskell
fib :: [Int]
fib = 0 : nxt
    where nxt = 1 : zipWith (+) fib nxt

fibN :: Int -> [Int]
fibN i = take i fib

fibVal :: Int -> Int
fibVal i = fib!!i
```

https://stackoverflow.com/questions/50101409/fibonacci-infinite-list-in-haskell
fib :: [Int]
fib = 0 : nxt
    where nxt = 1 : zipWith (+) fib nxt

fibN :: Int -> [Int]
fibN i = take i fib

fibVal :: Int -> Int
fibVal i = fib!!i