CMPT 383
Haskell’s Laziness

Anders Miltner
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

f :: Int -> Int
f x = x-1

>>> f 5
4

>>> f (f 5)
3
Call-by-Value

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

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

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

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

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

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

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…

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

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

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

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

\[ f \; 1 \; (g \; 100) \]

\[ \text{if} \; 1 = 0 \; \text{then} \; 0 \; \text{else} \; (g \; 100) + (g \; 100) \]

\[ (g \; 100) + (g \; 100) \]

\[ 5050 + 5050 \]

\[ 10100 \]
Call-by-Need

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

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

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

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

\[ f \ 1 \ (g \ 100) \]
\[ \text{if} \ 1 = 0 \ \text{then} \ 0 \ \text{else} \ (g \ 100) + (g \ 100) \]
\[ (g \ 100) + (g \ 100) \]
\[ 5050 + 5050 \]
\[ 10100 \]
Functions aren’t the only delayed evaluation

```haskell
ghci> g :: Int -> Int
ghci> g 0 = 0
ghci> g n = n+g(n-1)
ghci> x = (g 100000000):(g 100000000):[]
ghci> 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.
- Specifically: Lambda creation, Tuple creation, Constructors.
Introduction Forms & Function Arguments Do Not Require Evaluation

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

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

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

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

- What are elimination forms?
- Things that break big things down into little things
- Tuple component extraction, pattern matching
- 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

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

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

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

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

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

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

\[
f \ \text{True} \\
f \ \text{True} \ 2
\]
Example 1

\[
f :: \text{Int} \rightarrow \text{Int}
\]
\[
x = [f\ 1, f\ 2, f\ 3]
\]

\[
\text{case}\ x\ \text{of}
\]
\[
\text{[]} \rightarrow \text{“Empty”}
\]
\[
h : t \rightarrow \text{show}\ h
\]

\[f\ 1\ \text{gets evaluated, and nothing else}\]
Example 2

```haskell
f :: Int -> Int
g :: Int -> [Int]
g i = i:(g (i+1))
x = fmap (\i -> f i) (g 0)
take 0 l = []
take i [] = []
take i h:t = h:(take (i-1) t)
take 3 x
```

`g 0, g 1, g 2` gets evaluated, and nothing else
Example 3

```
f :: Int -> Int

g :: Int -> [Int]
g i = i::(g (i+1))

x = fmap (\i -> f i) (g 0)

take 0 l = []
take i [] = []
take i h:t = h:(take (i-1) t)

y = fmap show x

take 3 y
```

`g 0, g 1, g 2, f 0, f 1, f 2` gets evaluated, and nothing else
Example 4

\[ f \colon \text{Int} \rightarrow \text{Int} \]

\[ g \colon \text{Int} \rightarrow \left[ \text{Int} \right] \]
\[ g\, i = i :: (g\, (i + 1)) \]

\[ x = \text{fmap}\, (\lambda i \rightarrow f\, i) (g\, 0) \]

\[ \text{take}\, i\, \left[ \right] = \left[ \right] \]
\[ \text{take}\, 0\, l = \left[ \right] \]
\[ \text{take}\, i\, h : t = h : (\text{take}\, (i - 1)\, t) \]

\[ \text{take}\, 3\, x \]

\( g\, 0,\ g\, 1,\ g\, 2,\ g\, 3 \) gets evaluated, and nothing else
Example 5

\[
f :: \text{Int} \to \text{Int} \\
g :: \text{Int} \to [\text{Int}] \\
g i = i :: (g (i + 1)) \\
x = \text{fmap} \ (\lambda i \to f i) \ (g \ 0) \\
take i \ [] = [] \\
take 0 \ l = [] \\
take i \ h : t = h : (\text{take} \ (i - 1) \ t) \\
y = \text{fmap} \ \text{show} \ x \\
take 3 \ y
\]

\text{g 0, g 1, g 2, g 3, f 0, f 1, f 2 gets evaluated, and nothing else}
Example 6

```
f :: Int -> Int

g = \x -> if x == 0 then \y -> y else \y -> y+1

g (f 0)
```

nothing gets evaluated
Example 7

\[ f :: \text{Int} \rightarrow \text{Int} \]

\[ g = \lambda x \rightarrow \text{if } x = 0 \text{ then } \lambda y \rightarrow y \text{ else } \lambda y \rightarrow y + 1 \]

\[ g \ (f \ 0) \ (f \ 1) \]

\[ f \ 0 \text{ gets evaluated, and nothing else} \]
Example 7

\[ f :: \text{Int} \rightarrow \text{Int} \]

\[ \text{myfun} = \lambda x \rightarrow \text{if } x = 0 \text{ then } \lambda y \rightarrow y \text{ else } \lambda y \rightarrow y+1 \]

\[ \text{show } (g \ (f\ 0) \ (f\ 1)) \]

\[ f\ 0,\ f\ 1 \text{ gets evaluated, and nothing else} \]
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