# Lazy Evaluation in Haskell

Winter 2015-2016

# Table of Contents

# Simple definition: lazy evaluation

- Lazy evaluation means that an expression is evaluated only if it is required.
- In terms of Lambda calculus: evaluate the outer function application first.

$$\begin{aligned} (\lambda z.(z^2 + 3))\ (3+4) &= (3+4)^2 + 3 \\ &= (7^2 + 3) \\ &= 49 + 3 \\ &= 52 \end{aligned}$$

# How to understand it

- In the lambda calculus, everything is either a function, or a literal constant, i.e., a number, or other value.
- Expressions are made up of subexpressions, using literals and function calls.
- Evaluating an expression:
  1. A literal does not need to be evaluated.
  2. A function call is evaluated by using the application rule.
- Lazy evaluation: Perform the outer function call first.

$$
\begin{aligned}
(\lambda z.(z^2 + 3))\ (3 + 4) &= (3 + 4)^2 + 3 \\
&= (7^2 + 3) \\
&= 49 + 3 \\
&= 52
\end{aligned}
$$

- Evaluation finishes when you've reached a literal value.

# Operational Definition: Lazy evaluation

- Literal constants (functions, numbers, characters, Boolean values) are already evaluated. They don't need further evaluation.
- Haskell tries to be lazy about function calls.
- Lazy evaluation performs the outer application first.
- The result is either:
  1. A literal constant.
  2. Another function call.
- Sub-expressions are evaluated only when required.

# Simple Arithmetic Example

- Lazy evaluation means that it is not always necessary to evaluate every expression.
- Consider the simple function:

```
plus a b = a + b
```

$$
\begin{aligned}
&\mathrm{plus}\ (4 * 2)\ (5 - 3) \\
&\quad = \ (\lambda a.\lambda b.(a + b))\ (4 * 2)\ (5 - 3) \\
&\quad = \ (4 * 2) + (5 - 3) \\
&\quad = \ 8 + 2 \\
&\quad = \ 10
\end{aligned}
$$

- Note that the expressions given as arguments to plus were not evaluated before using the application rule.
- In a nut-shell, lazy evaluation is evaluating the outer-most application first.

# Another Simple Example

- Consider the following more complicated expression:

$$
\begin{aligned}
&\text{plus } (5+2) \ (\text{plus } 7 \ (8-6)) \\
&\quad = \ (5+2) + (\text{plus } 7 \ (8-6)) \\
&\quad = \ 7 + (\text{plus } 7 \ (8-6)) \\
&\quad = \ 7 + (7 + (8-6)) \\
&\quad = \ 7 + (7+2) \\
&\quad = \ 7 + 9 \\
&\quad = \ 16
\end{aligned}
$$

- Because of lazy evaluation, the outer application of plus was evaluated before the inner application was rewritten.
- The "harder" work was "postponed" until the very last minute.

# Yet another simple example

- Consider the function

```
seven :: a -> Int
seven x = 7
```

$$\begin{aligned} &\text{seven } (\text{plus } 7\ (8-6)) \\ =\ & 7 \end{aligned}$$

- Because of lazy evaluation, the inner application of plus is not evaluated at all.
- Evaluating the argument to seven is wasted work.
- Consider the following function, which is an infinite loop:

```
dumb x = 1 + dumb x
```

$$\text{seven } (\text{dumb } 1) = 7$$

# The tricky bits

- Haskell has syntax we've never seen in the Lambda Calculus:
  - Guarded expressions
  - Patterns
  - If-then-else
  - Recursion
  - etc. . .
- The key is to recognize when evaluation is required.

# Evaluation required on test in if-then-else

```
or :: Bool -> Bool -> Bool
or a b = if (a == True) then True else b
```

- The first argument is compared to the literal **True**
- The first argument must be evaluated.
- The second argument does not need to be evaluated (yet), so Haskell doesn't.
- If the first argument is **False**, the second argument is returned without evaluation.

# Evaluation required on match against literal

```
and :: Bool -> Bool -> Bool
and False b = False
and a b = b
```

- The first argument is matched against the literal **False**
- To check the match, the first argument must be evaluated.
- The second argument is not matched against a literal.
- The second argument does not need to be evaluated (yet), so Haskell doesn't.
- If the first argument is **True**, the second argument is returned without evaluation.

# Evaluation required on test with guard

```
or :: Bool -> Bool -> Bool
or a b
   | a == True = True
   | otherwise = b
```

- The first argument is compared to the literal **True**
- The first argument must be evaluated.
- The second argument does not need to be evaluated (yet), so Haskell doesn't.
- If the first argument is **False**, the second argument is returned without evaluation.

# Lazy Evaluation of non-trivial functions

```
fact n
  | n == 0 = 1
  | n > 0  = n * (fact (n-1))
```

- Suppose we require Haskell to evaluate `fact (2 + 1)`.
- $(2+1)$ gets evaluated when the first guard `n == 0` is tried.
- $n$'s value, 3, is stored
- If the first guard is **True**, 1 is returned
- If the first guard is **False**, the next guard is tried.
- If $n > 0$, then the expression `3 * (fact (3 - 1))` is returned but not evaluated.
- If the value of `3 * (fact (3 - 1))` is needed, it may be evaluated later. For now, Haskell just returns the expression itself.

# Interesting behaviour

- The function errorB halts with an error if the first argument is negative.

```
errorB x y
  | x < 0     = error "negative"
  | otherwise = y
```

```
Main> errorB (10) 2
2
Main> errorB (-10) 2

Program error: negative
```

# Interesting behaviour

- The function foo adds 2 values together, unless one of the arguments is negative.

```
foo a b = (errorB a b) + (errorB b a)
```

```
Main> foo 1 2
3
Main> foo 2 1
3
Main> foo 2 (-3)

Program error: negative
```

# Interesting behaviour

- The function bar adds 5 to every element in a list, unless one of the elements is negative.

```
bar []     = []
bar (x:xs) = (foo x 5) : bar xs
```

```
Main> bar [1,2,3,5,10]
 [6,7,8,10,15]
Main> bar [1,2,-3,5,10]
 [6,7,
Program error: negative
```

- Notice that two values are displayed before the error occurs!

# Interesting behaviour

- Pay careful attention to the following example:

```
Main> length (bar  [1,2,3,5,10])
5
Main> length (bar [−1,−2,−3,−5,−10])
5
```

- Notice that no error was thrown. Why?
  - The function bar creates a list by applying (foo 5) to each element.
  - If foo were evaluated, an error would be thrown.
  - The function **length** does not evaluate the elements in the list, it just counts how many there are.

# Interesting behaviour

- **length** does not evaluate the elements of the list.

```
length []     = 0
length (x:xs) = 1 + length xs
```

- **length** does not use $x$ in any way; $x$ is never matched or compared to anything. If it has a value, **length** doesn't care.

# When are expressions evaluated?

- In pattern matching against a literal value.
- In guards when an expression is checked.
- In if-then-else when an expression is checked
- By the **show** function in Hugs.

# List Comprehensions: Introduction I

- A "comprehension" is the name for special syntax to create lists
- The syntax is similar to *set notation* in mathematics
- The syntax is designed to be convenient for the programmer, but it's nothing more than functions.
- Because of lazy evaluation:
    1. a list is only created if necessary
    2. the whole list may not be needed – Haskell creates only as much as is needed by other functions

# Simple Examples: lists of numbers

**Prelude**> [1 .. 5]
[1,2,3,4,5]

**Prelude**> [10 .. 20]
[10,11,12,13,14,15,16,17,18,19,20]

**Prelude**> [3.0 .. 7.0]
[3.0,4.0,5.0,6.0,7.0]

```
Prelude> [0.5 .. 5.5]
  [0.5,1.5,2.5,3.5,4.5,5.5]

Prelude> [0.5 .. 5]
  [0.5,1.5,2.5,3.5,4.5,5.5]

Prelude> [3.0 .. 7.0]
  [3.0,4.0,5.0,6.0,7.0]

Prelude> [3.1 .. 7]
  [3.1,4.1,5.1,6.1,7.1]
```

General case: $[n..m]$ evaluates to $[n, n+1, n+2, \cdots, n+k]$ where $n+k \leq m+1/2$.

# Adjusting the interval between elements

**Prelude**> [1,3 .. 10]
[1,3,5,7,9]

**Prelude**> [1.0, 1.1 .. 2.0]
[1.0,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8,1.9,2.0]

**Prelude**> [1.1, 1.6 .. 3.0]
[1.1,1.6,2.1,2.6,3.1]

**Prelude**>[10, 9 .. 0]
[10,9,8,7,6,5,4,3,2,1,0]

General case: $[n, p..m]$ evaluates to
$[n, n+(p-n), n+2(p-n), \cdots, n+k(p-n)]$ where
$n+k(p-n) \leq m+(p-n)/2$.

# Extension to other Enum types

**Prelude**> ['a' .. 'f']
"abcdef"

**Prelude**> ['a', 'e' .. 'z']
"aeimquy"

# Using comprehensions in functions

- A comprehension can be defined using variables:

```
product []     = 1
product (x:xs) = x * product xs

factorial n = product [1 .. n]
```

- You might think this is a waste of effort, to build a list only to count to *n*.
- Because of lazy evaluation, and the way **product** is defined, very little extra memory is needed.

# A glimpse under the hood

- A comprehension is a syntactic convenience; the notation is mapped to a function expression involving a function called **enumFromTo** or **enumFromThenTo**, eg

```
Prelude> [1 .. 5]
 [1,2,3,4,5]
Prelude> enumFromTo 1 5
 [1,2,3,4,5]
Prelude> [1,3 .. 10]
 [1,3,5,7,9]
Prelude> enumFromThenTo 1 3 10
 [1,3,5,7,9]
```

  Informally, we call this kind of convenience syntactic sugar.
- This notation is available to any **Enum** type.

# Comprehensions and Lazy Evaluation

- The comprehension technique is still based on functions, but these functions are hidden by "syntactic sugar"
- If the functions are lazy, then the lists they define are constructed lazily, i.e., only as needed.
- Example:

```
Main> take 3 [1 .. 100000]
 [1,2,3]
```

- The comprehension defines a very long list.
- **take** only needs 3 of them.
- The rest of the list is not needed, and not created!

# Generators

- Haskell allows a programmer to construct lists from other lists, using notation even more similar to set notation:

```
Prelude> [x*x | x <- [1 .. 10]]
 [1,4,9,16,25,36,49,64,81,100]
```

- You can read the `<-` in two ways:
  1. takes the value of each of (like an assignment)
  2. an element in (like set theory's $\in$ relation)

# Qualifiers

- You can also restrict the values by adding a qualifier:

```
Prelude> [x | x <- [1 .. 10], x `mod` 2 == 0]
[2,4,6,8,10]
```

The qualifier is a way to restrict the elements in the set, to those that satisfy the condition

```
Prelude> [x | x <- [0, 5 .. 100], 23 < x && x < 61 ]
[25,30,35,40,45,50,55,60]
```

every elements is checked

# The general form: $[e|q_1, \cdots, q_k]$

Text

- $e$ is any expression, using names in the scope, and names defined by `<-` in any of the $q_i$
- the $q_i$ are called qualifiers, and can be
  1. a generator, `pat <- list` The scope of `pat` is the comprehension
  2. `pat` can be a pattern matching the contents of the list (e.g., a tuple for a list of tuples)
  3. a qualifier any expression with a Boolean value.
     The expression can use names in any of $q_1, \cdots, q_{i-1}$.
- We read the expression left-to-right;
- Names defined by a generator can be used in any expression to its right

# More examples

```
allpairs :: [a]->[b]->[(a,b)]
allpairs xs ys = [(x,y) | x <- xs, y <- ys]
```

```
Prelude> allpairs [1 .. 3] "abc"
[(1,'a'),(1,'b'),(1,'c'),(2,'a'),(2,'b'),
(2,'c'),(3,'a'),(3,'b'),(3,'c')]
```

```
somepairs :: Num a => a -> [(a,a)]
somepairs n = [(x,y) | x <- [1 .. n], y <- [x+1 .. n]]
```

```
Prelude> somepairs 3
 [(1,2),(1,3),(2,3)]
```

- In many cases, simple comprehensions can be written a number of ways.
- A comprehension replaces the (tedious) design of a function to generate lists
- The comprehension technique is still based on functions, but these functions are hidden by "syntactic sugar"

# Graph search

- Assume a directed graph is represented by a list of pairs as edges.
- E.g., here's a simple "chain graph"

```
simpleEdgeSet = [(1,2),(2,3),(3,4)]
```

- Write a program that performs depth-first search (DFS) in a graph, given:
  - A starting node
  - A goal node
  - Returns True if the goal node can be reached by DFS.
  - Variation 1: return a path from start to goal
  - Variation 2: return all paths from start to goal

# Implementing Search

- Key concept: Given a list of nodes to explore, pick one.
- To explore a node means to follow an edge out of it, to another node.
- Algorithm Search
  1. If you've reached the goal, return true
  2. If there are no more choices, return false
  3. Otherwise, pick a node, then:
     - 3.1 Collect all nodes adjacent to the current node
     - 3.2 Add them to the list of choices
     - 3.3 Search using the new list of choices

# Implementing DFS

- A specialization of the search algorithm
- Use a LIFO queue to store the choices
- Pick the first node in the queue
- Add the new nodes to the front of the queue
- In Haskell, we can use a list for our FIFO queue.
- A draft of the program:

```
dfs edges goal []  = False
dfs edges goal (c:choices)
  | goal == c    = True
  | otherwise = dfs goal (newchoices ++ choices)
      where newchoices = ...
```

# Generating new choices

- The edges are stored in a list
- Given a node *c*, filter the list for the ones that start at *c*
- Need to have list of edges as input.
- A revised draft of the program:

```
dfs edges goal []  = False
dfs edges goal (c:choices)
   | goal == c    = True
   | otherwise    = dfs edges goal (newchoices ++ choices)
     where newchoices = [ v | (u,v) <- edges, u == c]
```

# Demonstration

```
Main> dfs [(1,2),(2,3),(3,4)] 4 [1]
True
Main> dfs [(1,2),(2,3),(3,4)] 1 [4]
False
Main> dfs [(1,2),(1,3),(1,4),(2,5)] 5 [1]
True
Main> dfs [(1,2),(1,3),(1,4),(2,5)] 5 [3]
False
Main> dfs [(1,2),(1,3),(1,4),(2,1),(2,5)] 5 [1]
^CInterrupted.
```

# The Maybe type

- For search tasks, you want to return a value if you find one
- Or return some signal that the search failed
- Haskell defines the **Maybe** type for this purpose.

```
data Maybe a = Nothing | Just a
  deriving (Show)
```

- The container **Just** <value> should be used to indicate a successful search, and to contain the object that you found.
- The value Nothing can be used to indicate that the search was unsuccessful. It contains nothing.
- This solves a problem that sometimes arises when there is no value that you can force into service to signal failure.
  - In other languages, we sometimes force a value to serve as a signal of this kind, e.g., `return −1;`, or `return null;`
  - This only works if the value you choose is not one of the possible answers.
- Predefined for you in the Haskell Prelude.

# Finding a path

- Need to store the path to each choice from the starting point
- When the goal is reached, the path is right there!

```
dfsp edges goal [] = Nothing
dfsp edges goal ((c,p):choices)
   | goal == c    = Just (c:p)
   | otherwise    = dfsp edges goal (new ++ choices)
          where new = [ (v,c:p) | (u,v) <- edges, u == c]
```

# Demonstration

```
Main> dfsp [(1,2),(2,3),(3,4)] 4 [(1,[])]
Just [4,3,2,1]
Main> dfsp [(1,2),(2,3),(3,4)] 1 [(4,[])]
Nothing
Main> dfsp [(1,2),(1,3),(1,4),(2,5)] 5 [(1,[])]
Just [5,2,1]
Main> dfsp [(1,2),(1,3),(1,4),(2,5)] 5 [(3,[])]
Nothing
```

The paths are reversed. The input is clumsy.

# Tidying up

```
dfsp2 edges goal start = dfsloop [(a,[]) | a <- start]
  where dfsloop [] = Nothing
        dfsloop ((c,p):choices)
          | goal == c    = Just (c:p)
          | otherwise    = dfsloop (new ++ choices)
              where new = [ (v,c:p) | (u,v) <- edges, u == c]
```

# Finding all paths

- When you find one path, don't stop, look for more!
- Put any answers you find in a list.

```
dfspall edges goal start = dfsloop [(a ,[]) | a <- start]
  where dfsloop [] = []
        dfsloop ((c,p):choices)
          | goal == c   = (Just (c:p)) : (dfsloop choices)
          | otherwise   = dfsloop (new ++ choices)
              where new = [ (v,c:p) | (u,v) <- edges, u == c]
```

# Demonstration

```
Main> dfspall [(1,2),(1,3),(1,4),(2,5)] 5 [1]
[Just [5,2,1]]
Main> dfspall [(1,2),(1,3),(1,4),(2,5),(3,5)] 5 [1]
[Just [5,2,1], Just [5,3,1]]
Main> dfspall [(1,2),(1,3),(1,4),(2,5),(3,5)] 1 [5]
[]
```

# Breadth first search

- In BFS, explore old nodes first, new nodes later.
- Very simple change!

```
bfspall edges goal start = loop [(a,[]) | a <- start]
  where loop [] = []
        loop ((c,p):choices)
          | goal == c  = (Just (c:p)) : (loop choices)
          | otherwise  = loop (choices ++ new)
             where new = [ (v,c:p) | (u,v) <- edges, u == c]
```

# Infinite Lists

- Because of lazy evaluation, list comprehensions can specify infinitely long lists
- You can write expressions that are lazy, and do not require the entire list to be constructed first
- Such functions process the list, one by one, and just keep going
- Much like a `while(true)` loop in C or Java
- The simplest infinite list is constructed by

```
ifny :: [Integer]
ifny = [1 ..]
```

- This is a comprehension with no end point

# The function iterate

- Defined in Haskell:

```
iterate f x = x : iterate f (f x)
```

- Can create infinite lists which are based on a given function
- Examples:

```
Main> iterate (+1) 0
  [0,1,2,3,4,5,6,7,8,9,{  Interrupted !}
Main> take 10 (iterate (\ x -> x + x) 2)
 [2,4,8,16,32,64,128,256,512,1024]
```

# What good are infinite lists?

- Any use of an infinite list could be replaced by an expression that specifies the number of items exactly
- But specifying the number ties the use of the data to the data itself
- Because of lazy evaluation, we can separate the use from the context:
  - A lazy generator builds the elements
  - A lazy function consumes them one by one until finished

# What good are infinite lists? (II)

- Application: random number generator
  - set up an infinite list of random numbers
  - they are generated one by one as needed
  - without an explicit loop
- Application: File I/O
  - treat the file as a string of unknown length
  - substrings and characters can be read from the stream
  - a file is (usually finite) but treating as an infinite list simplifies the model
  - (this is what most languages' file I/O does anyway)
- Application: GUI events:
  - treat the sequence of events as a list of unknown length
  - treat each element one by one

# What good are infinite lists? (III)

- Application: Implementing non-deterministic programs
  - for problems solved by search
  - treat the search space as a list of possibilities to try
  - future choices appear later in the list
  - How long is it?
  - Unknown, just generate one possibility at a time
- In general, an infinite list represents a stream of data whose length is unknown. Being able to model this data as a list "generated" one at a time is a very flexible approach to solving many kinds of problems.

# A simple example: Prime numbers

- We can combine functions and comprehensions in many ways
- Consider the list of all prime numbers:

```
primes :: [Integer]
primes = [ x | x <- [2 ..], isAprime x]
```

- Simple enough, requires suitable definition for isAprime
- What is a prime number? No divisor except 1 and itself

```
isAprime :: Integer -> Bool
isAprime x = nodivisor x [2 .. x-1]
  where nodivisor _ [] = True
        nodivisor x (y:ys) = not (x `mod` y == 0)
                             && nodivisor x ys
```

Very simple, but does too much work

# The Sieve of Eratosthenes

*Sieve the twos, and sieve the threes*
*The Sieve of Eratosthenes*
*When the multiples sublime*
*The numbers that are left are prime!*

- Build a list of primes by removing non-primes
  1. Start with all integers `[2 ..]`
  2. Keep the 2
  3. Remove every number greater than 2 but divisible by 2
  4. The next number is prime: 3 (keep it)
  5. Remove every number divisible by 3
  6. The next number is 5 (because 4 was eliminated), keep it
  7. remove all numbers divisible by 5
  8. ...
- Using lazy evaluation, we will build the list one by one
- Each prime number found causes other numbers to be eliminated

# The Implementation of sieve

```
primes :: [Integer]

primes =
```

Where to start? The first prime number is 2

# The Implementation of sieve

$$\text{primes} :: [\textbf{Integer}]$$

$$\text{primes} = 2 :$$

Generate a list of numbers not divisible by 2

# The Implementation of sieve

```
primes :: [Integer]

primes = 2 : [x | x <- [3 ..], x `mod` 2 /= 0]
```

Getting there... Now eliminate factors of 3...

# The Implementation of sieve

```
primes :: [Integer]

primes = 2 : 3 : [y|y <- [x|x <- [2 ..], x `mod` 2 /= 0],
                   y `mod` 3 /= 0]
```

Let's redesign.

# The Implementation of sieve

```
primes :: [Integer]

primes = sieve [2 ..]

sieve :: [Integer] -> [Integer]

sieve (x:xs) =
```

The relationship here is: keep *x*, remove multiples of *x* from *xs*

# The Implementation of sieve

```
primes :: [Integer]

primes = sieve [2 ..]

sieve :: [Integer] -> [Integer]

sieve (x:xs) = x :
```

Keep $x$

# The Implementation of sieve

```
primes :: [Integer]

primes = sieve [2 ..]

sieve :: [Integer] -> [Integer]

sieve (x:xs) = x : [y | y <- xs, y `mod` x /= 0]
```

Eliminate multiples of *x* from *xs*
but, then eliminate multiples of the first y. How?

# The Implementation of sieve

```
primes :: [Integer]

primes = sieve [2 ..]

sieve :: [Integer] -> [Integer]

sieve (x:xs) = x : sieve [y | y <- xs, y `mod` x /= 0]
```

Sieve the tail!

# Notes

- sieve has no base case.
- primes is an infinite list
- We can ask for any subset, using **take** or **takeWhile**

```
Main> take 5 primes
 [2,3,5,7,11]

Main> takeWhile (<25) primes
 [2,3,5,7,11,13,17,19,23]
```

- Only as many primes as necessary are computed (lazy)
- Each number *i* is only divided by prime numbers once (efficient)