Lazy Evaluation:
Infinite data structures, set comprehensions (CTM Section 4.5)

Carlos Varela
RPI
September 22, 2015

Adapted with permission from:
Seif Haridi
KTH
Peter Van Roy
UCL
Lazy evaluation

• The functions written so far are evaluated eagerly (as soon as they are called)
• Another way is lazy evaluation where a computation is done only when the results is needed

• Calculates the infinite list: 0 | 1 | 2 | 3 | ...

```plaintext
declare
fun lazy {Ints N}
  N|{Ints N+1}
end
```
Sqrt using an infinite list

\[
\text{let } \sqrt{x} = \text{head} \ (\text{dropWhile} \ (\text{not} \ . \ \text{goodEnough}) \ \sqrt\text{Guesses}) \\
\text{where} \\
\text{goodEnough guess} = \frac{|x - \text{guess} \times \text{guess}|}{x} < 0.00001 \\
\text{improve guess} = \frac{\text{guess} + x/\text{guess}}{2.0} \\
\sqrt\text{Guesses} = 1:(\text{map} \ \text{improve} \ \sqrt\text{Guesses})
\]

Infinite lists (\sqrt\text{Guesses}) are enabled by lazy evaluation.
Map in Haskell

\[ \text{map}' :: (\text{a} \rightarrow \text{b}) \rightarrow \text{[a]} \rightarrow \text{[b]} \]

\[
\begin{align*}
\text{map}'_\_ \text{[]} & = [] \\
\text{map}' f \text{ (h:t)} & = f \text{ h} : \text{map}' f \text{ t}
\end{align*}
\]

Functions in Haskell are lazy by default. That is, they can act on infinite data structures by delaying evaluation until needed.
Lazy evaluation (2)

- Write a function that computes as many rows of Pascal’s triangle as needed
- We do not know how many beforehand
- A function is *lazy* if it is evaluated only when its result is needed
- The function `PascalList` is evaluated when needed

```plaintext
fun lazy {PascalList Row}
  Row | {PascalList
       {AddList
        {ShiftLeft Row}
        {ShiftRight Row})}
end
```
Lazy evaluation (3)

- Lazy evaluation will avoid redoing work if you decide first you need the 10\textsuperscript{th} row and later the 11\textsuperscript{th} row
- The function continues where it left off

\begin{verbatim}
declare
L = {PascalList [1]}
{Browse L}
{Browse L.1}
{Browse L.2.1}
L<Future>
[1]
[1 1]
\end{verbatim}
Lazy execution

• Without lazyness, the execution order of each thread follows textual order, i.e., when a statement comes as the first in a sequence it will execute, whether or not its results are needed later
• This execution scheme is called *eager execution*, or *supply-driven* execution
• Another execution order is that a statement is executed only if its results are needed somewhere in the program
• This scheme is called *lazy evaluation*, or *demand-driven* evaluation (some languages use lazy evaluation by default, e.g., Haskell)
Example

B = \{F1 X\}
C = \{F2 Y\}
D = \{F3 Z\}
A = B+C

• Assume F1, F2 and F3 are lazy functions
• B = \{F1 X\} and C = \{F2 Y\} are executed only if and when their results are needed in A = B+C
• D = \{F3 Z\} is not executed since it is not needed
Example

- In lazy execution, an operation suspends until its result is needed
- The suspended operation is triggered when another operation needs the value for its arguments
- In general multiple suspended operations could start concurrently

B = \{F1 \ X\}
C = \{F2 \ Y\}

\[ A = B + C \]
Example II

- In data-driven execution, an operation suspends until the values of its arguments results are available
- In general the suspended computation could start concurrently

\[ A = B + C \]

\[ B = \{F1 \ X\} \]
\[ C = \{F2 \ Y\} \]

Data driven
Using Lazy Streams

```
fun \{\text{Sum Xs A Limit}\}
  \text{if Limit>0 then}
    \text{case Xs of X|Xr then}
      \{\text{Sum Xr A+X Limit-1}\}
    \text{end}
  \text{else A end}
end

local \text{Xs S in}
  \text{Xs={Ints 0}}
  \text{S={Sum Xs 0 1500}}
  \{\text{Browse S}\}
end
```
How does it work?

```haskell
fun \{\text{Sum Xs A Limit}\}  
  \text{if Limit}>0 \text{ then}  
  \text{case } \text{Xs of } X|Xr \text{ then}  
  \{\text{Sum Xr A+X Limit-1}\}  
  \text{end}  
  \text{else A end}  
\text{end}

fun \text{lazy } \{\text{Ints N}\}  
  N | \{\text{Ints N+1}\}  
\text{end}

\text{local } Xs \text{ S in}  
  Xs = \{\text{Ints 0}\}  
  S = \{\text{Sum Xs 0 1500}\}  
  \{\text{Browse S}\}  
\text{end}
```
Improving throughput

• Use a lazy buffer
• It takes a lazy input stream In and an integer N, and returns a lazy output stream Out
• When it is first called, it first fills itself with N elements by asking the producer
• The buffer now has N elements filled
• Whenever the consumer asks for an element, the buffer in turn asks the producer for another element
The buffer example

C. Varela; Adapted from S. Haridi and P. Van Roy
The buffer

fun \{Buffer1 \ In \ N\} 
    End=\{List.drop \ In \ N\}

    fun lazy \{Loop \ In \ End\} 
        In.1|\{Loop \ In.2 \ End.2\} 
        end 
    in 
    \{Loop \ In \ End\} 
    end

Traversing the In stream, forces the producer to emit N elements
The buffer II

fun \{Buffer2 In N\}
    End = thread
        \{List.drop In N\}
    end

fun lazy \{Loop In End\}
    In.1|\{Loop In.2 End.2\}
end

in
    \{Loop In End\}
end

Traversing the In stream, forces the producer to emit N elements and at the same time serves the consumer
fun \{Buffer3\ \text{In N}\}
\text{End} = \text{thread}
\{\text{List.drop In N}\}
\text{end}

fun \text{lazy}\ \{\text{Loop In End}\}
\text{E}2 = \text{thread End.2 end}
\text{In.1|}\{\text{Loop In.2 E2}\}
\text{end}

Traverse the \text{In} stream, forces the producer to emit \text{N} elements and at the same time serves the consumer, and requests the next element ahead.
Larger Example: The Sieve of Eratosthenes

- Produces prime numbers
- It takes a stream 2...N, peals off 2 from the rest of the stream
- Delivers the rest to the next sieve
Lazy Sieve

```
fun lazy {Sieve Xs}
  X|Xr = Xs in
  X | {Sieve {LFilter
    Xr
    fun {Y} Y mod X \= 0 end
  }}
end

fun {Primes} {Sieve {Ints 2}} end
```
Lazy Filter

For the Sieve program we need a lazy filter

```haskell
fun lazy {LFilter Xs F}
  case Xs
  of nil then nil
    [] X|Xr then
      if {F X} then X|{LFilter Xr F} else {LFilter Xr F} end
  end
end
```
Primes in Haskell

```
ints :: (Num a) => a -> [a]
ints n = n : ints (n+1)

sieve :: (Integral a) => [a] -> [a]
sieve (x:xr) = x : sieve (filter (\y -> (y `mod` x /= 0)) xr)

primes :: (Integral a) => [a]
primes = sieve (ints 2)
```

Functions in Haskell are lazy by default. You can use `take 20 primes` to get the first 20 elements of the list.
Define streams implicitly

- Ones = 1 | Ones
- Infinite stream of ones
Define streams implicitly

- \( Xs = 1 \mid \{ \text{LMap } Xs \}
  \begin{array}{l}
  \quad \text{fun } \{ X \} \ X+1 \text{ end}
  \end{array} \\
- \text{What is } Xs?
The Hamming problem

• Generate the first N elements of stream of integers of the form: $2^a 3^b 5^c$ with $a, b, c \geq 0$ (in ascending order)
The Hamming problem

- Generate the first N elements of stream of integers of the form: $2^a \ 3^b \ 5^c$ with $a, b, c \geq 0$ (in ascending order)
The Hamming problem

- Generate the first N elements of stream of integers of the form: $2^a 3^b 5^c$ with $a, b, c \geq 0$ (in ascending order)
Lazy File Reading

fun {ToList FO}
  fun lazy {LRead} L T in
    if {File.readBlock FO L T} then
      T = {LRead}
    else
      T = nil {File.close FO} end
  end
L
{LRead}
end

• This avoids reading the whole file in memory
List Comprehensions

• Abstraction provided in lazy functional languages that allows writing higher level set-like expressions
• In our context we produce lazy lists instead of sets
• The mathematical set expression
  – \{x*y | 1\leq x \leq 10, 1\leq y \leq x\}
• Equivalent List comprehension expression is
  – \[X*Y | X = 1..10 ; Y = 1..X\]
• Example:
  – \[1*1 2*1 2*2 3*1 3*2 3*3 ... 10*10\]
List Comprehensions

• The general form is
  
  \[ f(x,y, ...,z) \mid x \leftarrow \text{gen}(a_1,...,a_n) ; \text{guard}(x,...) \\
  y \leftarrow \text{gen}(x, a_1,...,a_n) ; \text{guard}(y,x,...) \\
  \ldots \]

• No linguistic support in Mozart/Oz, but can be easily expressed
Example 1

• $z = [x \# x \mid x \leftarrow \text{from}(1,10)]$
• $Z = \{\text{LMap } \{\text{LFrom 1 10}\} \ \text{fun}\{\$X\} \ X\#X \ \text{end}\}$

• $z = [x \# y \mid x \leftarrow \text{from}(1,10), y \leftarrow \text{from}(1,x)]$
• $Z = \{\text{LFlatten}
\{\text{LMap } \{\text{LFrom 1 10}\}
\ \text{fun}\{\$X\} \ \{\text{LMap } \{\text{LFrom 1 } X\}
\ \text{fun}\{\$Y\} \ X\#Y \ \text{end}\}
\ \text{end}\}$
Example 2

• \( z = [x\#y \mid x \leftarrow \text{from}(1,10), \ y \leftarrow \text{from}(1,x), \ x+y \leq 10] \)

• \( Z = \{ \text{LFilter} \}
  \{ \text{LFlatten} \}
  \{ \text{LMap} \ \{ \text{LFrom} 1 \ 10 \} \}
  \left. \begin{array}{c}
  \text{fun} \ \{ \text{$X$} \} \ \{ \text{LMap} \ \{ \text{LFrom} 1 \ X \} \}
  \\quad \left. \begin{array}{c}
  \text{fun} \ \{ \text{$Y$} \} \ X\#Y \ \text{end} \end{array} \right. \\
  \text{end} \\
  \text{end} \\
  \end{array} \right\} \}
  \text{fun} \ \{ \text{$X\#Y$} \} \ X+Y=<10 \ \text{end} \} \} \)
List Comprehensions in Haskell

\[ \text{lc1} = [(x,y) \mid x \leftarrow [1..10], y \leftarrow [1..x]] \]

\[ \text{lc2} = \text{filter (\((x,y)\rightarrow (x+y\leq 10)) lc1} \]

\[ \text{lc3} = [(x,y) \mid x \leftarrow [1..10], y \leftarrow [1..x], x+y\leq 10] \]

Haskell provides syntactic support for list comprehensions. List comprehensions are implemented using a built-in list monad.
Quicksort using list comprehensions

quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (h:t) = quicksort [x | x <- t, x < h] ++
[h] ++
quicksort [x | x <-t, x >= h]
Higher-order programming

• Higher-order programming = the set of programming techniques that are possible with procedure values (lexically-scoped closures)

• Basic operations
  – **Procedural abstraction**: creating procedure values with lexical scoping
  – **Genericity**: procedure values as arguments
  – **Instantiation**: procedure values as return values
  – **Embedding**: procedure values in data structures

• Higher-order programming is the foundation of component-based programming and object-oriented programming
Embedding

- Embedding is when procedure values are put in data structures
- Embedding has many uses:
  - **Modules**: a module is a record that groups together a set of related operations
  - **Software components**: a software component is a generic function that takes a set of modules as its arguments and returns a new module. It can be seen as specifying a module in terms of the modules it needs.
  - **Delayed evaluation** (also called explicit lazy evaluation): build just a small part of a data structure, with functions at the extremities that can be called to build more. The consumer can control explicitly how much of the data structure is built.

C. Varela; Adapted from S. Haridi and P. Van Roy
Explicit lazy evaluation

• Supply-driven evaluation. (e.g. The list is completely calculated independent of whether the elements are needed or not. )

• Demand-driven execution. (e.g. The consumer of the list structure asks for new list elements when they are needed.)

• Technique: a programmed trigger.

• How to do it with higher-order programming? The consumer has a function that it calls when it needs a new list element. The function call returns a pair: the list element and a new function. The new function is the new trigger: calling it returns the next data item and another new function. And so forth.
Explicit lazy functions

```plaintext
fun lazy {From N} N | {From N+1} end
```

```plaintext
fun {From N}  
  fun {$} N | {From N+1} end
end
```
Implementation of lazy execution

The following defines the syntax of a statement, $\langle s \rangle$ denotes a statement

\[
\langle s \rangle ::= \text{skip} \quad \text{empty statement}
\]
\[
\quad \text{...}
\]
\[
\quad\text{thread } \langle s_1 \rangle \text{ end} \quad \text{thread creation}
\]
\[
\quad \{ \text{ByNeed fun }\{\$\} \langle e \rangle \text{ end} \} \quad \langle x \rangle \text{ by need statement}
\]

C. Varela; Adapted from S. Haridi and P. Van Roy
Implementation

A function value is created in the store (say f) the function f is associated with the variable x execution proceeds immediately to next statement

\{ByNeed fun\$\} \langle e \rangle \text{end} \; X,E \}

ByNeed
\{fun\}
\{$\}
\langle e \rangle
\text{end}
X,E

some statement

stack

store

f

x
Implementation

A function value is created in the store (say f) the function f is **associated** with the variable x execution proceeds immediately to next statement

---

some statement

\{ByNeed \text{fun}\{\}$\langle e\rangle \text{end } X,E \}\n
store

\[ f \quad \rightarrow \quad \text{(fun}\{\}$\langle e\rangle \text{end } X,E) \]\n
\[ x : f \]
Accessing the **ByNeed** variable

- **X = \{ByNeed fun\{\$\} 111*111 end\}** (by thread T0)

- **Access by some thread T1**
  - if $X > 1000$ then \{Browse hello\#X\} end

  or

  - \{Wait X\}
  - Causes X to be bound to 12321 (i.e. 111*111)
Implementation

Thread T1
1. X is needed
2. start a thread T2 to execute F (the function)
3. only T2 is allowed to bind X

4. Allow access on X

Thread T2
1. Evaluate Y = \{F\}
2. Bind X the value Y
3. Terminate T2
Lazy functions

fun lazy \{\text{Ints } N\}
   \ N \mid \{\text{Ints } N+1\}
end

fun \{\text{Ints } N\}
   \frac{F}{\text{N}} \mid \{\text{Ints } N+1\} \end
in \{\text{ByNeed } F\}
end
26. Write a lazy append list operation \texttt{LazyAppend}. Can you also write \texttt{LazyFoldL}? Why or why not?

27. CTM Exercise 4.11.10 (pg 341)

28. CTM Exercise 4.11.13 (pg 342)

29. CTM Exercise 4.11.17 (pg 342)

30. Solve exercise 29 (Hamming problem) in Haskell.