Lazy Evaluation:
Infinite data structures, set comprehensions
(CTM Sections 1.8 and 4.5)

Carlos Varela
RPI
September 24, 2019

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 result is needed

- Calculates the infinite list: $0 \mid 1 \mid 2 \mid 3 \mid ...$

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

\[
\text{let } \text{sqrt } x = \text{head} \ (\text{dropWhile} \ (\text{not} \ . \ \text{goodEnough}) \ \text{sqrtGuesses}) \\
\quad \text{where} \\
\quad \text{goodEnough } \text{guess} = (\text{abs} \ (x - \text{guess}^2))/x < 0.00001 \\
\quad \text{improve } \text{guess} = (\text{guess} + x/\text{guess})/2.0 \\
\quad \text{sqrtGuesses} = 1:(\text{map} \ \text{improve} \ \text{sqrtGuesses})
\]

Infinite lists (sqrtGuesses) are enabled by lazy evaluation.
Map in Haskell

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

\[
\text{map}' \_ [] = []
\]

\[
\text{map}' f \ (h:t) = f \ h:\text{map}' \ f \ t
\]

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 10th row and later the 11th row
- The function continues where it left off

```plaintext
declare
L = {PascalList [1]}
{Browse L}
{Browse L.1}
{Browse L.2.1}

L<Future>
[1]
[1 1]
```
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.

\[ A = B + C \]

\[ B = \{ F1 \ X \} \]

\[ C = \{ F2 \ Y \} \]

Demand
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.

\[
\begin{align*}
B &= \{F1 \, X\} \\
C &= \{F2 \, Y\} \\
A &= B + C
\end{align*}
\]
Using Lazy Streams

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

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

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

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

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

• Use a lazy buffer
• It takes a lazy input stream $\text{In}$ and an integer $N$, and returns a lazy output stream $\text{Out}$
• 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
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 \{\text{Buffer3 In N}\}
\quad \text{End} = \text{thread}
\quad \{\text{List.drop In N}\}
\quad \text{end}
\text{fun lazy} \{\text{Loop In End}\}
\quad \text{E2} = \text{thread} \text{ End.2 end}
\quad \text{In.1|\{\text{Loop In.2} E2\}}
\quad \text{end}
\text{in}
\quad \{\text{Loop In End}\}
\text{end}

Traverse the In stream, forces the producer to emit 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

```plaintext
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
end
```
Primes in Haskell

```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

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

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

- Generate the first $N$ elements of stream of integers of the form: $2^a \cdot 3^b \cdot 5^c$ 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
        L
    end
    {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
  – \( \{x \cdot y \mid 1 \leq x \leq 10, 1 \leq y \leq x \} \)
• Equivalent List comprehension expression is
  – \([X \cdot Y \mid X = 1..10 ; Y = 1..X]\)
• Example:
  – \([1 \cdot 1 \ 2 \cdot 1 \ 2 \cdot 2 \ 3 \cdot 1 \ 3 \cdot 2 \ 3 \cdot 3 \ldots \ 10 \cdot 10]\)
List Comprehensions

• The general form is
• \[ f(x, y, ..., z) \mid x \leftarrow \text{gen}(a_1, ..., a_n) ; \text{guard}(x, ...)
\]
\[ y \leftarrow \text{gen}(x, a_1, ..., a_n) ; \text{guard}(y, x, ...)
\]
\[ ... \]
\[
\]
• No linguistic support in Mozart/Oz, but can be easily expressed
Example 1

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

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

C. Varela; Adapted from S. Haridi and P. Van Roy
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 \} \}
  \{ \text{fun} \{ X \} \{ \text{LMap} \{ \text{LFrom} 1 \ X \} \}
    \{ \text{fun} \{ Y \} \ X \# Y \ \text{end} \}
  \} \}
  \{ \text{end} \}
  \} \}
\{ \text{fun} \{ X \# Y \} \ X + Y \leq 10 \ \text{end} \} \}
List Comprehensions in Haskell

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

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

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

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.
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

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

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

The following defines the syntax of a statement, ⟨s⟩ denotes a statement:

\[
\langle s \rangle ::= \text{skip} \quad \text{empty statement}
\]

\[
\| \ldots
\]

\[
\text{thread } \langle s_1 \rangle \text{ end} \quad \text{thread creation}
\]

\[
\{ \text{ByNeed fun}\{\$\} \langle e \rangle \text{ end}\} \quad \langle x \rangle \{ \text{by need statement}\}
\]

zero arity function

variable
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
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
Accessing the ByNeed variable

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

• Access by some thread T1
  – if $X > 1000$ then $\{\text{Browse hello#}X\} \text{ end}$

  or

  – $\{\text{Wait X}\}$
  – 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

**Thread T2**

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

4. Allow access on X

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

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

fun {Ints N}
    fun {F} N | {Ints N+1} end
    in {ByNeed F}
end
Exercises

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.