Declarative Concurrency
Lazy Execution (VRH 4.5)

Carlos Varela
Rensselaer Polytechnic Institute

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

November 29, 2007

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

Lazy evaluation

• The default functions in Oz 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 | 1 | 2 | 3 | ...

\[
\begin{align*}
\text{declare} \\
\text{fun lazy \{Ints N\} N[\{Ints N+1\}]} \\
\text{end}
\end{align*}
\]

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

\[
\begin{align*}
\text{fun lazy \{PascalList Row\}} \\
\text{Row \{PascalList \}} \\
\text{AddList Row \{ShiftRight Row\}} \\
\text{end}
\end{align*}
\]

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

\[
\begin{align*}
\text{declare} \\
\text{L \{PascalList \{1\}\}} \\
\text{Browse L} \\
\text{Browse L \{1\}} \\
\text{Browse L \{2,1\}} \\
\text{L<Future \{1\}} \\
\text{\{1\}}
\end{align*}
\]

Lazy execution

• Without laziness, 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

\[
\begin{align*}
B \{F1 X\} \\
C \{F2 Y\} \\
D \{F3 Z\} \\
A = B+C \\
\end{align*}
\]

• 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.

**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.

**Using Lazy Streams**

```plaintext
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
define Xs S in
  Xs={Ints 0}
  S={Sum Xs 0 1500}
  {Browse S}
end
```

**How does it work?**

```plaintext
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
define Xs S in
  Xs={Ints 0}
  S={Sum Xs 0 1500}
  {Browse S}
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**

```
producer  buffer  consumer
```

```
producer  buffer  consumer
```
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

---

The buffer III

```
fun {Buffer3 In N}
  End = thread {List.drop In N}
  end
  fun lazy {Loop In End}
    E2 = thread End.2 end
    In.1 | {Loop In.2 E2}
  end
  in {Loop In End}
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}
    Xr
    fun {S Y} Y mod X != 0 end
  } end
  fun {Primes} {Sieve {Ints 2}} end
```

---

Lazy Filter

For the Sieve program we need a lazy filter

```
fun lazy {LFilter Xs F}
  case Xs
  of nil then nil
  [] Xr then
    if {F X} then X | {LFilter Xr F} else {LFilter Xr F} end
  end
end
```
Define streams implicitly

- Ones = 1 \mid \text{Ones}
- Infinite stream of ones

Define streams implicitly

- Xs = 1 \mid \text{LMap Xs} \{ \text{fun} \{ X \} \ X + 1 \} \end{text}
- 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)

Lazy File Reading

fun \{ToList FO\}
fun lazy \{LRead\} L T in
if \{File.readBlock FO L T\} then
  T = \{LRead\}
else
  T = \text{nil} \{File.close FO\}
end
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
  \[ \{ x \times y | 1 \leq x \leq 10, 1 \leq y \leq x \} \]
- Equivalent List comprehension expression is
  \[ [x \times y | X = 1..10 ; Y = 1..X] \]
- Example:
  \[ [1 \times 1, 2 \times 1, 2 \times 2, 3 \times 1, 3 \times 2, 3 \times 3, ..., 10 \times 10] \]

Example 1

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

Example 2

- \( z = [x \# y | x \leftarrow \text{from}(1,10), y \leftarrow \text{from}(1,x), x + y \leq 10] \)
- \( Z = \{\text{LFilter} \{\text{LFlatten} \{\text{LMap} \{\text{LFrom} 1 10\} \text{fun}(S) X \# X \text{end} \}
  \quad \text{fun} \{S Y\} X \# Y \text{end} \}
  \quad \text{fun} \{S X \# Y\} X + Y \leq 10 \text{end} \} \)

Implementation of lazy execution

The following defines the syntax of a statement, \( (s) \) denotes a statement

\[ (s) := \begin{cases} 
\text{skip} & \text{empty statement} \\
\text{thread} (s) & \text{thread creation} \\
\{\text{ByNeed fun}(s) \} & \text{by need statement} \\
\end{cases} \]

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
Implementation

some statement

stack

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

store

f ← \{fun \$(x) end X.E\}

x : f

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

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)

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

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

\begin{align*}
\text{fun} & \text{ lazy } \{\text{Ints } N\} \\
& N \{\text{Ints } N+1\} \\
\text{end}
\end{align*}

\begin{align*}
\text{fun} & \{\text{Ints } N\} \\
& \text{fun} \{F\} N \{\text{Ints } N+1\} \text{ end} \\
in & \{\text{ByNeed } F\} \\
\text{end}
\end{align*}

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

Exercises

94. Write a lazy append list operation \texttt{LazyAppend}. Can you also write \texttt{LazyFoldL}? Why or why not?
95. Exercise VRH 4.11.10 (pg 341)
96. *Exercise VRH 4.11.13 (pg 342)
97. *Exercise VRH 4.11.17 (pg 342)