Declarative Concurrency

Lazy Execution (VRH 4.5)

Carlos Varela
Rensselaer Polytechnic Institute

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

December 4, 2009

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 | ...

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


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

```
fun lazy {PascalList Row}
  Row | {PascalList
         {AddList
            Row
            {ShiftRight Row}}}
end
```


Lazy evaluation (3)

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

```
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)

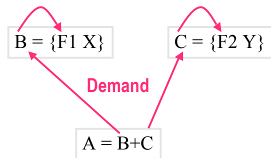
Example

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

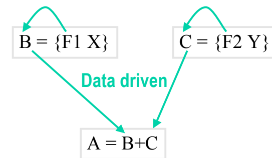
Example

- 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



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

```
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
```

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


How does it work?

```
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
```

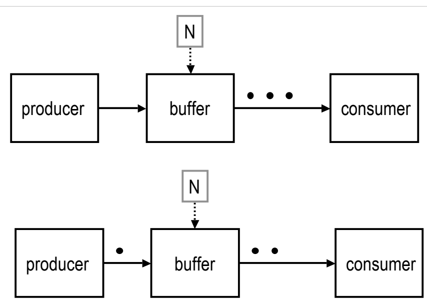
```
fun lazy {Ints N}
   N | {Ints N+1}
end

local 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



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

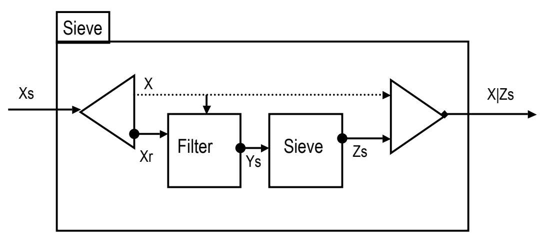
The buffer III

```
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
        fun {$ Y} Y mod X \= 0 end
       }}
end

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

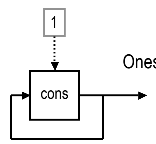

Lazy Filter

For the Sieve program we need a lazy filter

```
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
```

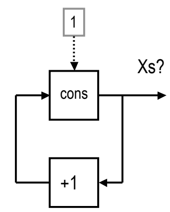
Define streams implicitly

- Ones = 1 | Ones
- Infinite stream of ones



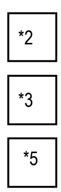
Define streams implicitly

- Xs = 1 | {LMap Xs
  fun {$ X} X+1 end}
- What is Xs ?



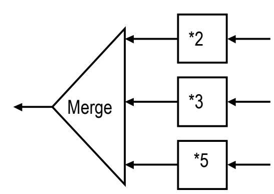
The Hamming problem

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



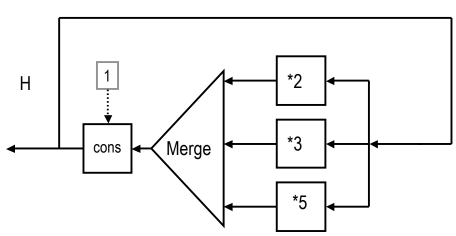
The Hamming problem

- Generate the first N elements of stream of integers of the form: $2^a 3^b 5^c$ with a,b,c ≥ 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 ≥ 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
  - $\{x*y \mid 1 \le x \le 10,\ 1 \le y \le x\}$
- 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) | x ← gen(a1,...,an) ; guard(x,...)
  y ← gen(x, a1,...,an) ; guard(y,x,...)
  ....
  ]
- No linguistic support in Mozart/Oz, but can be easily expressed

Example 1

- z = [x#x | x ← from(1,10)]
- Z = {LMap {LFrom 1 10} fun{$ X} X#X end}

- z = [x#y | x ← from(1,10), y ← from(1,x)]
- Z =

```
{LFlatten
   {LMap {LFrom 1 10}
    fun{$ X} {LMap {LFrom 1 X}
                fun {$ Y} X#Y end
             }
    end
   }
}
```


Example 2

- z = [x#y | x ← from(1,10), y ← from(1,x), x+y≤10]
- Z =

```
{LFilter
   {LFlatten
      {LMap {LFrom 1 10}
       fun{$ X} {LMap {LFrom 1 X}
                   fun {$ Y} X#Y end
                }
       end
      }
   }
   fun {$ X#Y} X+Y=<10 end} }
```

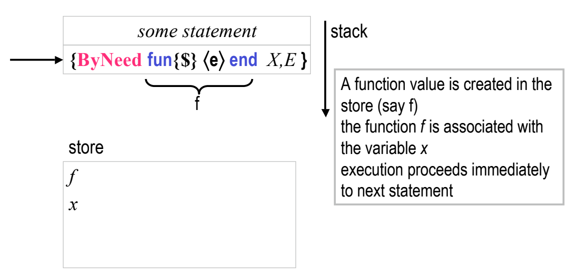

Implementation of lazy execution

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

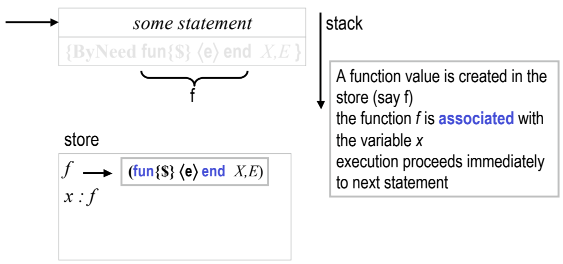
⟨s⟩::=	**skip**	*empty statement*
|	...	
|	**thread** ⟨s_1⟩ **end**	*thread creation*
|	{ByNeed **fun**{$} ⟨e⟩ **end** ⟨*x*⟩}	*by need statement*

(⟨e⟩ part: zero arity function; ⟨x⟩: variable)

Implementation



Implementation



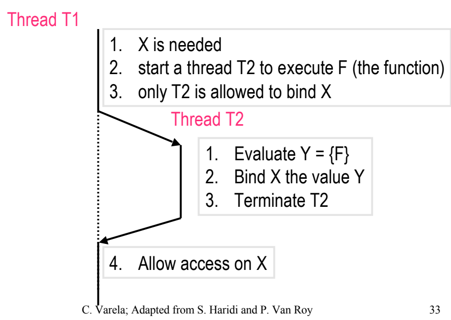
Accessing the ByNeed variable

- 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



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

90. Write a lazy append list operation `LazyAppend`. Can you also write `LazyFoldL`? Why or why not?
91. Exercise VRH 4.11.10 (pg 341)
92. Exercise VRH 4.11.13 (pg 342)
93. Exercise VRH 4.11.17 (pg 342)