

## 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

1

## 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
```

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

2

## 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
```

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

3

## Lazy evaluation (3)

- Lazy evaluation will avoid redoing work if you decide first you need the 10<sup>th</sup> row and later the 11<sup>th</sup> 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]
```

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

4

## 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)

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

5

## Example

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

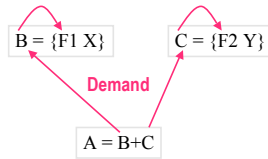
- Assume F1, F2 and F3 are lazy functions
- B = {F1 X} and C = {F2 Y} are executed only if and when their results are needed in A = B+C
- D = {F3 Z} is not executed since it is not needed

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

6

## 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

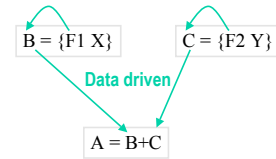


C. Varela; Adapted from S. Haridi and P. Van Roy

7

## 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



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

8

## 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
    
```

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

9

## 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
    
```

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

10

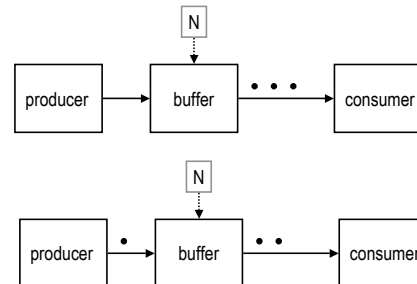
## Improving throughput

- 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

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

11

## The buffer example



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

12

## The buffer

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

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

Traversing the In stream, forces the producer to emit N elements

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

13

## 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
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**

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

14

## 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
end
in
  {Loop In End}
end
```

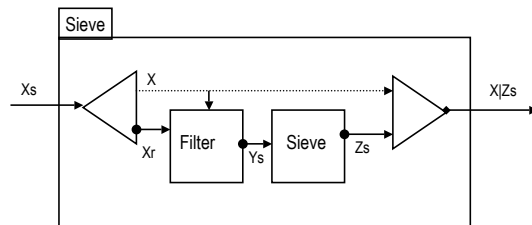
Traverse the In stream, forces the producer to emit N elements **and at the same time serves the consumer, and requests the next element ahead**

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

15

## Larger Example: The Sieve of Eratosthenes

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



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

16

## 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
```

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

17

## Lazy Filter

For the Sieve program we need a lazy filter

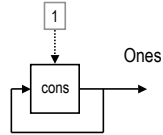
```
fun lazy {LFilter Xs F}
  case Xs
  of nil then nil
  [] X|Xr then
    if {F X} then X|{LFilter Xr F} else {LFilter Xr F} end
  end
end
```

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

18

## Define streams implicitly

- Ones = 1 | Ones
- Infinite stream of ones

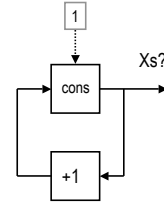


C. Varela; Adapted from S. Haridi and P. Van Roy

19

## Define streams implicitly

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

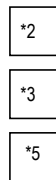


C. Varela; Adapted from S. Haridi and P. Van Roy

20

## 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)

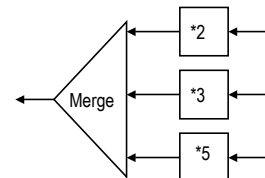


C. Varela; Adapted from S. Haridi and P. Van Roy

21

## 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)

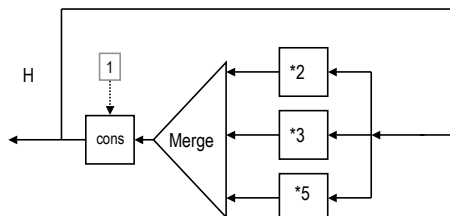


C. Varela; Adapted from S. Haridi and P. Van Roy

22

## 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)



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

23

## 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

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

24

## 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
  - $\{x*y \mid 1 \leq x \leq 10, 1 \leq y \leq x\}$
- Equivalent List comprehension expression is
  - $[X*Y \mid X = 1..10 ; Y = 1..X]$
- Example:
  - $[1*1 \ 2*1 \ 2*2 \ 3*1 \ 3*2 \ 3*3 \ \dots \ 10*10]$

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

25

## 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

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

26

## Example 1

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

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

27

## Example 2

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

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

28

## Implementation of lazy execution

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

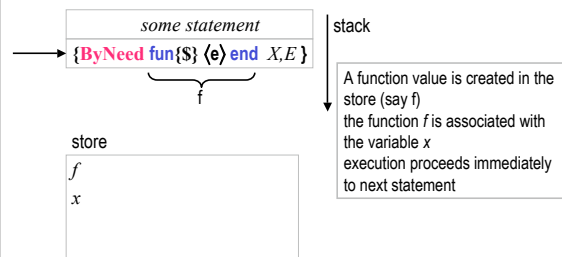
```

<s> ::= skip                                empty statement
      | ...
      | thread <s1> end                    thread creation
      | {ByNeed fun{S} {e} end } <x>      by need statement
                                       { }
                                       zero arity function    variable
    
```

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

29

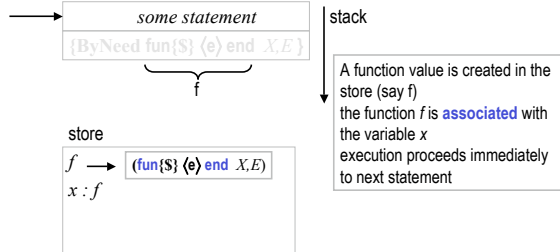
## Implementation



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

30

## Implementation



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

31

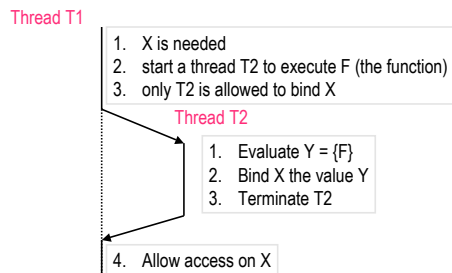
## Accessing the ByNeed variable

- $X = \{\text{ByNeed fun}\{S\} 111*111 \text{ end}\}$  (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

32

## Implementation



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

33

## 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
```

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

34

## Exercises

94. Write a lazy append list operation `LazyAppend`. Can you also write `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)

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

35