Lazy Evaluation:

Infinite data structures, set comprehensions
(CTM Sections 1.8 and 4.5,
GIH 3.4: A Gentle Introduction to Haskell)

Carlos Varela

RPI

September 24, 2024

Adapted with permission from:

Seif Haridi

KTH

Peter Van Roy

UCL

Lazy evaluation

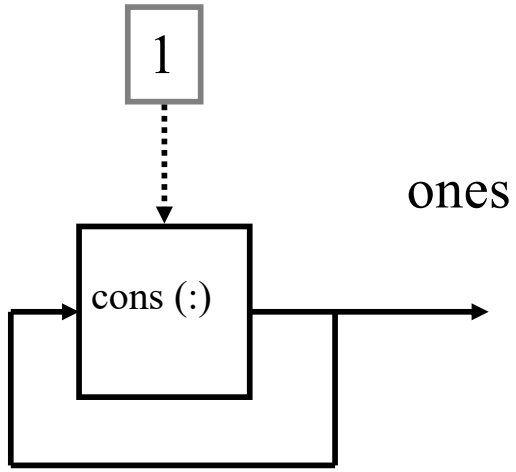
- Many (functional) programming languages evaluate functions eagerly (as soon as they are called)
- Another way is lazy evaluation where a computation is done only when the result is needed

```
ints n = n : ints (n+1)
```

- ints 0 denotes the infinite list:
  0 : 1 : 2 : 3 : ...

Define streams implicitly

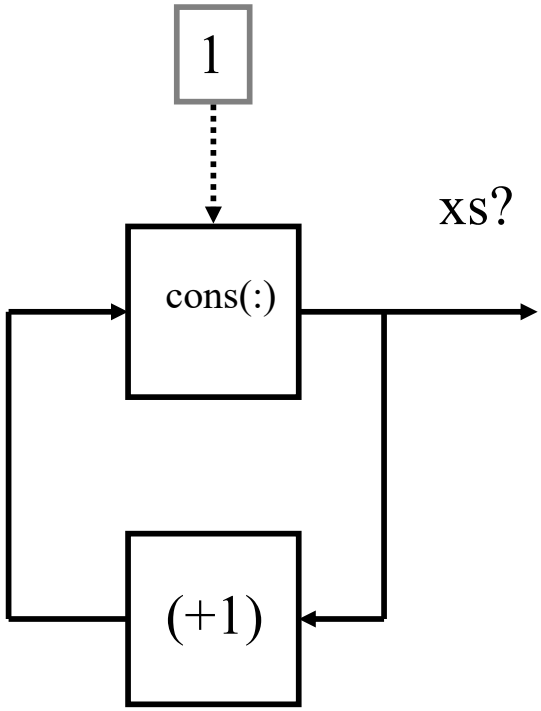
- ones = 1 : ones
- Infinite stream of ones



Define streams implicitly

```
xs = 1 : map (+1) xs
```

- What is xs?



Fibonacci sequence as an infinite list

```
fib = fib' 0 1
        where fib' f1 f2 = f1 : fib' f2 (f1+f2)
```

- fib denotes the infinite list:
  0 : 1 : 1 : 2 : 3 : 5 : 8 : ...

Map

```
map' :: (a -> b) -> [a] -> [b]
map' _ []    = []
map' f (h:t) = f h:map' f t
```

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

Sqrt using infinite lists

```
sqrt7 x = head (dropWhile (not . goodEnough) sqrtGuesses)
        where
            goodEnough guess = (abs (x – guess*guess))/x < 0.00001
            improve guess = (guess + x/guess)/2.0
            sqrtGuesses = 1:(map improve sqrtGuesses)
```

This sqrt example uses infinite lists enabled by lazy evaluation, and the map control abstraction.

iterate

```
iterate :: (a -> a) -> a -> [a]
iterate f s = s : iterate f (f s)
```

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

Sqrt using iterate abstraction

```
sqrt8 x = head (dropWhile (not . goodEnough) sqrtGuesses)
        where
            goodEnough guess = (abs (x – guess*guess))/x < 0.00001
            improve guess = (guess + x/guess)/2.0
            sqrtGuesses = iterate improve 1
```

This sqrt example uses infinite lists enabled by lazy evaluation, and the iterate control abstraction.

zipWith

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' f (x:xs) (y:ys)  = (f x y) : zipWith f xs ys
zipWith' _ xs     ys      = []
```

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

Fibonacci sequence using zipWith

```
fib' = 0 : 1 : zipWith (+) fib' (tail fib')
```

- fib' denotes the infinite list:
  0 : 1 : 1 : 2 : 3 : 5 : 8 : ...

Lazy evaluation (2)

```
pascal row = row : pascal (zipWith (+) (0:row) (row++[0]))
```

- 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 pascal is evaluated when needed

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

```
pt = pascal [1]
take 10 pt
nth pt 11
```

```
> take 10 pt
[[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1],[1,5,10,10,5,1],[1,6,15,20,15,6,1],[1,7,21,35,35,21,7,1],[1,8,28,56,70,56,28,8,1],[1,9,36,84,126,126,84,36,9,1]]
> nth pt 11
[1,10,45,120,210,252,210,120,45,10,1]
```

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)

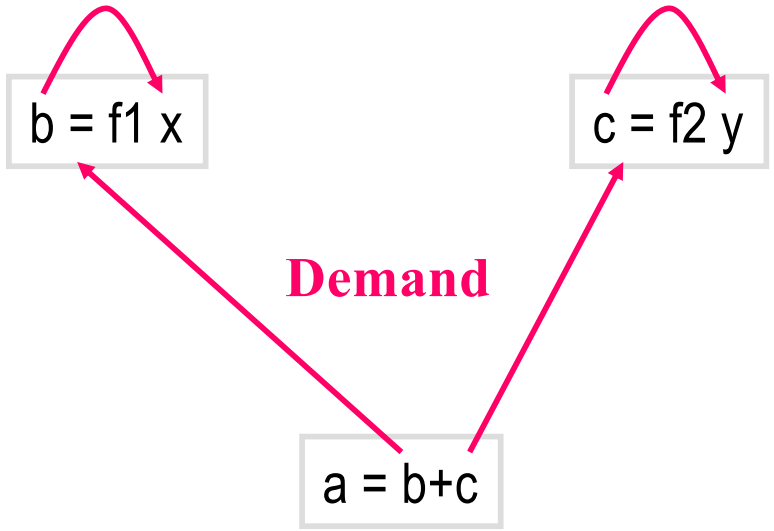
Example

```
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

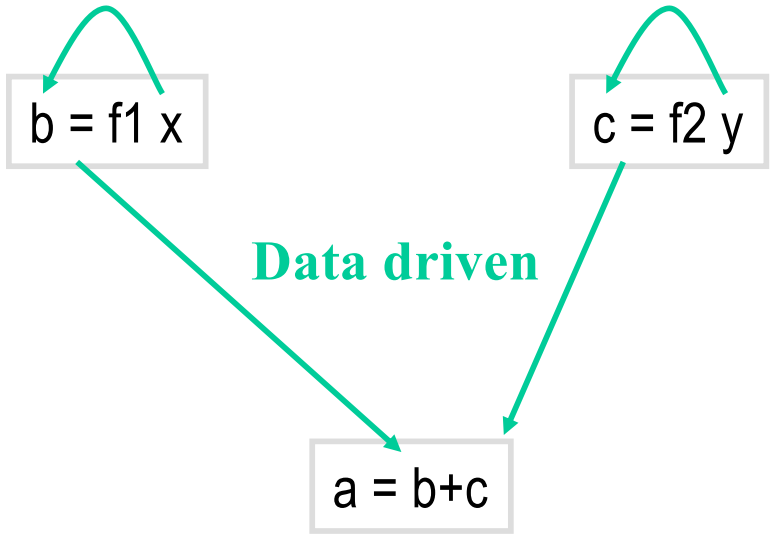
Example

- 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

```
nth :: [a] -> Integer -> a
nth (h:t) 1 = h
nth (h:t) n = nth t (n-1)
```

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


let i0 = ints 0
nth i0 11
```

How does it work?

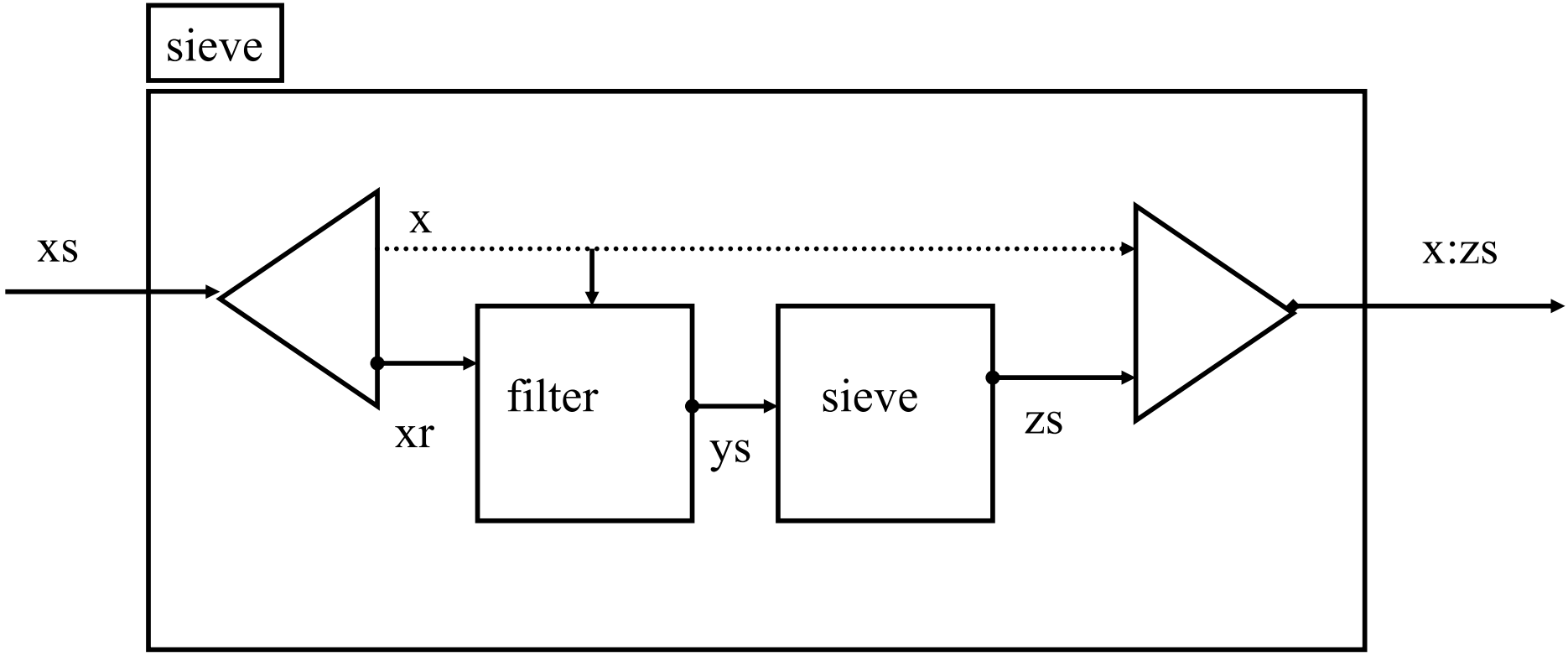
```
nth :: [a] -> Integer -> a
nth (h:t) 1 = h
nth (h:t) n = nth t (n-1)
```

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

```
let i0 = ints 0
nth i0 11
```

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



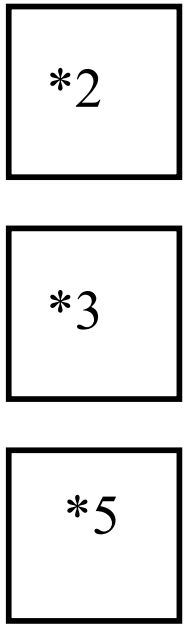
Sieve of Eratosthenes

```
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.

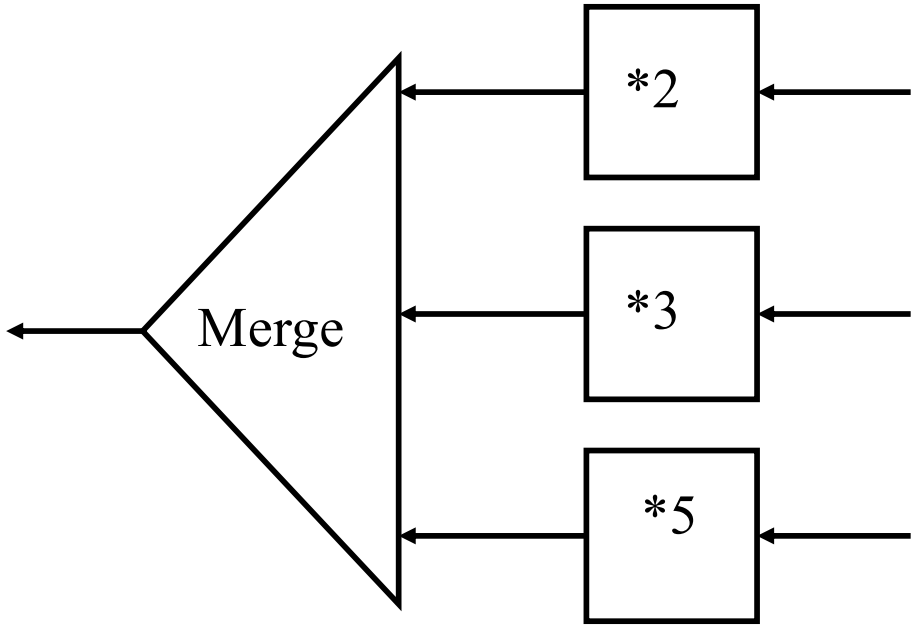
The Hamming problem

- 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)



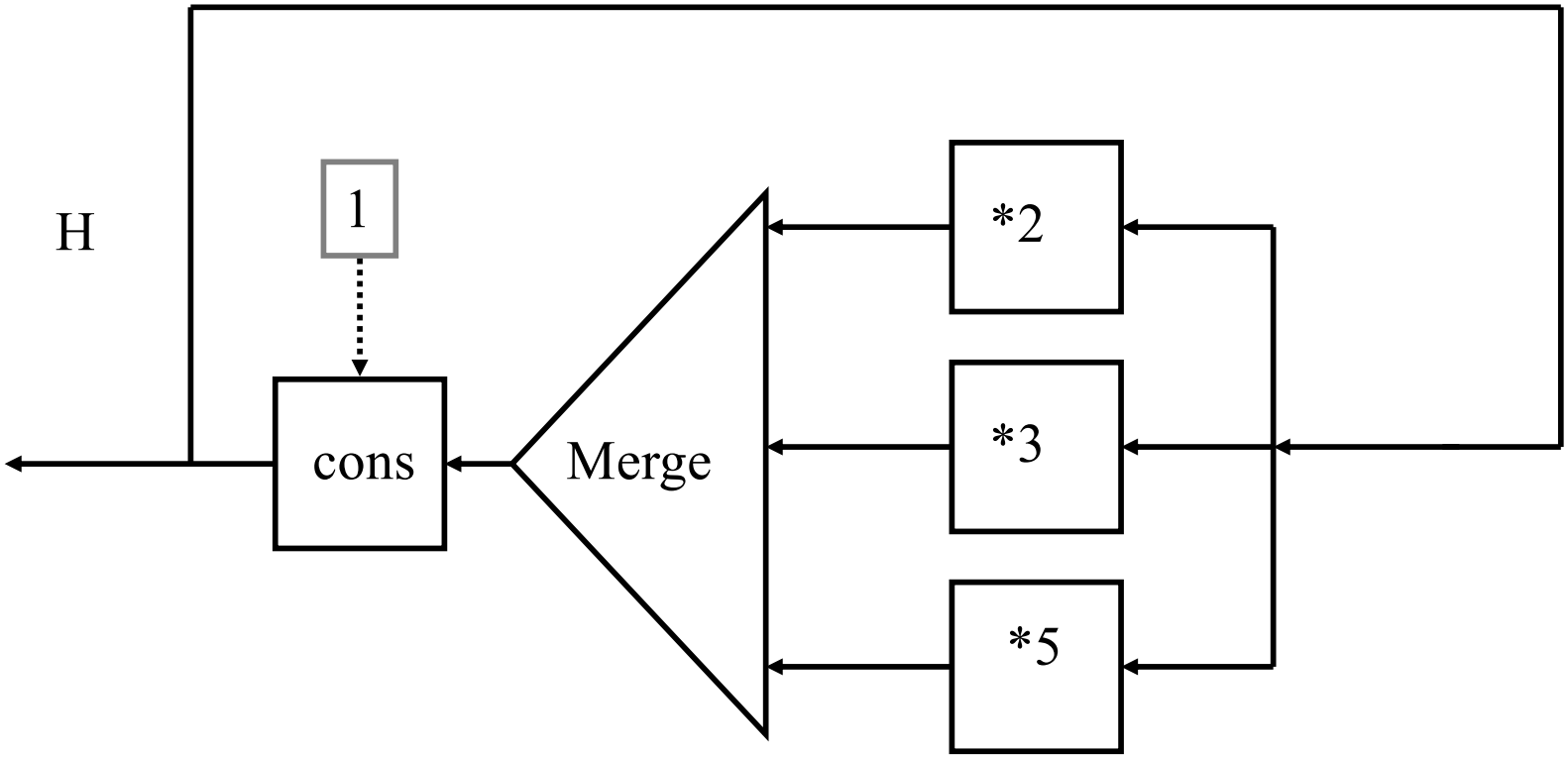
The Hamming problem

- 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)



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 | 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,...)
               ...]
```

- *Arithmetic sequences* can be used as generators:

```
[1..10]      =>  [1,2,3,4,5,6,7,8,9,10]
[1,3..10]    =>  [1,3,5,7,9]
[1,3..]      =>  [1,3,5,7,9,… (infinite sequence)
```

List Comprehensions

```
lc1 = [(x,y) | x <- [1..10], y <- [1..x]]

lc2 = filter (\(x,y)->(x+y<=10)) lc1

lc3 = [(x,y) | x <- [1..10], y <- [1..x], x+y<= 10]
```

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

Map using list comprehensions

```
map :: (a -> b) -> [a] -> [b]
map f l = [ f x | x <- l]
```

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]
```

Exercises

26. Is lazy evaluation slower than eager evaluation? 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.