

# Lecture 5: Lazy Evaluation and Infinite Data Structures

Søren Haagerup

Department of Mathematics and Computer Science  
University of Southern Denmark, Odense

November 21, 2016

# How does Haskell evaluate a program?

Since there are no side-effects in Haskell programs, the evaluation order of expressions is not specified by the order of the expressions:

$$\text{root } a \ b \ c = (-b + sd) / (2 * a)$$

**where**

$$sd = \text{sqrt } d$$

$$d = b * b - 4 * a * c$$

This definition is perfectly valid, even though the definition of  $d$  is written after  $sd$ .

```
> root 1 0 (-2)
1.4142135623730951
```

# Lazy evaluation

- The purity of Haskell functions (= absense of side-effects) gives more freedom to the compiler, when it decides when to evaluate each expression:
- The idea is to wait with evaluating an expression until it is *needed*.
- This concept is also known as **Lazy evaluation**.
- **Note:** Not all functional programming languages use Lazy evaluation as its evaluation model. This is just a design choice in Haskell.

# Advantages of Lazy evaluation

1. Avoids doing **unnecessary evaluation**
2. Allows programs to be **more modular**
3. Allows us to program with **infinite lists**

## Example 1

$a :: Bool$

$a = \neg a$

What happens if we evaluate  $a$ ?

$b :: Bool$

$b = False$

$c :: Bool$

$c = b \wedge a$

What happens if we evaluate  $c$ ?

$c' :: Bool$

$c' = a \wedge b$

What about  $c'$ ?

In other languages this behavior is called **short circuiting**, and is only possible in those languages because  $(\wedge)$  and  $(\vee)$  are built in to the language.

Note that  $(\wedge)$  and booleans *Bool* are *not* primitives in the Haskell language. They are just an ordinary function and an ordinary algebraic data type!

## Example 2

In Haskell we can define our own “if”-function by

$$\text{myIf} :: \text{Bool} \rightarrow a \rightarrow a \rightarrow a$$
$$\text{myIf True } v \_ = v$$
$$\text{myIf False } \_ v = v$$

What happens when we evaluate

$$\text{myIf True expr1 expr2}$$

? We get the result *expr1*, and *expr2* never gets evaluated.

**This is clearly different from the way other languages like Java and Python would evaluate a similar function.**

# Reducible expressions

- We know that Haskell evaluates expressions by succesively applying definitons, until no more simplifications are possible.
- Expressions which can be simplified are called “reducible expressions”.

## Examples:

|  |  |
|--|--|
| $(\lambda x \rightarrow x * 2)$          | Not reducible  |
| $(\lambda x \rightarrow x * 2) 5$        | Reducible to $5 * 2$ , which is reducible to 10  |
| $(\lambda x y \rightarrow 13 * x + y) 2$ | Reducible to $(\lambda y \rightarrow 13 * 2 + y)$ .<br>Then subexpression $13 * 2$ is reducible. |
| $(7, 5 + 3)$                             | Not reducible, but subexpression $5 + 3$ is.   |

Function application is also known as a  $\beta$ -reduction.

# Reduction strategies for function application

Consider the function  $sqr\ x = x * x$  and the expression  $sqr\ (4 + 2)$

- **Innermost reduction:** [call-by-value]

$$sqr\ (4 + 2) \equiv sqr\ 6 \equiv 6 * 6 \equiv 36$$

- **Outermost reduction:** [call-by-need]

$$sqr\ (4 + 2) \equiv (4 + 2) * (4 + 2) \equiv 6 * (4 + 2) \equiv 6 * 6 \equiv 36$$

- **Outermost reduction with sharing (= Graph reduction)**

$$sqr\ (4 + 2) \equiv \text{let } x = 4 + 2 \text{ in } x * x \equiv 6 * 6 \equiv 36$$

The general evaluation rule of Haskell can be described as

**Leftmost outermost reduction with sharing**



## One more example

*ackermann m n*

$$| m \equiv 0 \quad \quad \quad = n + 1$$

$$| m > 0 \wedge n \equiv 0 = \textit{ackermann} (m - 1) 1$$

$$| m > 0 \wedge n > 0 = \textit{ackermann} (m - 1) (\textit{ackermann} m (n - 1))$$

How does Haskell evaluate?

*const 5 (ackermann 4 2)*

# Primitive arithmetic operations $(+)$ , $(-)$ , $(*)$ , $(/)$

Cannot be evaluated in outermost manner:

$$3 * 4 + 3 * 4$$

→ {first operand not done}

$$3 * 4$$

→ {both operands done, arithmetic}

$$12$$

$$\dots 12 + 3 * 4$$

→ {second operand not done}

$$3 * 4$$

→ {both operands done, arithmetic}

$$12$$

$$\dots 12 + 12$$

→ {both operands done, arithmetic}

$$24$$

We say that the functions are *strict* in both arguments.

## Another example

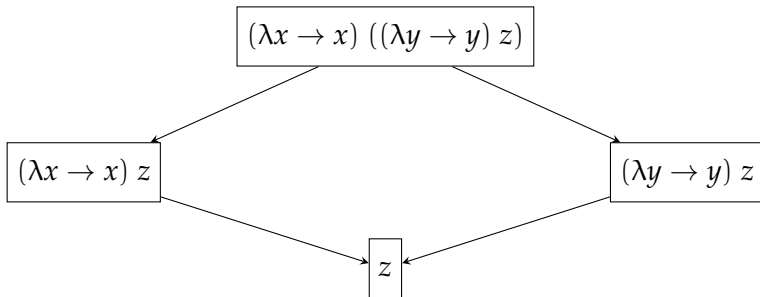
Consider again the function

$$\text{sqr } x = x * x$$

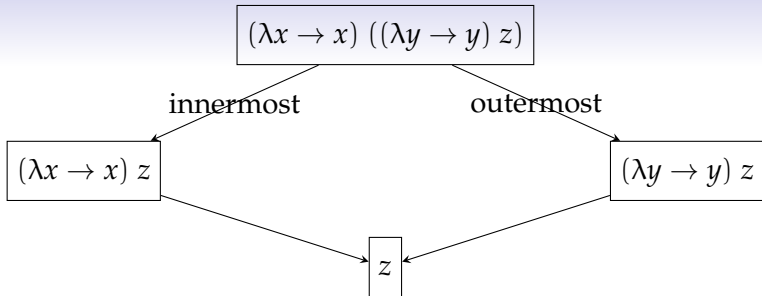
How does Haskell evaluate

$$\begin{aligned}\text{sqr } (\text{sqr } 2) &\equiv \text{let } x = \text{sqr } 2 \text{ in } x * x && \text{-- (apply outer } \text{sqr}) \\ &\equiv \text{let } x = 2 * 2 \text{ in } x * x && \text{-- (apply inner } \text{sqr}) \\ &\equiv \text{let } x = 4 \text{ in } x * x && \text{-- (apply inner } (*)) \\ &\equiv 16 && \text{-- (apply outer } (*))\end{aligned}$$

# Innermost and outermost reductions



Which is innermost and which is outermost?



- when an expression contains no reducible subexpression, it is in **normal form**
- some expressions might not have a normal form
- an expression has at most one normal form
- **outermost reduction** always reaches the normal form if there exists a reduction strategy that reaches the normal form
- **outermost reduction with sharing** uses at most as many reduction steps as innermost reduction

# Function application summarized

The evaluation rule for function application

1. Evaluate the function until it is a lambda term
2. Then plug in the arguments.
3. Multiple occurrences of a formal argument must all point to the same actual argument (sharing)

## Example (function which is not a lambda term)

$(\text{if } \text{True} \text{ then } (\lambda c \rightarrow (c, 0)) \text{ else } (\lambda c \rightarrow (c, 1))) \text{ blah}$   
 $\rightarrow \{\text{need function}\}$   
     $\text{if } \text{True} \text{ then } (\lambda c \rightarrow (c, 0)) \text{ else } (\lambda c \rightarrow (c, 1))$   
     $\rightarrow \{\text{if-then-else}\}$   
         $(\lambda c \rightarrow (c, 0))$   
...  $(\lambda c \rightarrow (c, 0)) \text{ blah}$   
 $\rightarrow \{\text{function application}\}$   
     $(\text{blah}, 0)$

## Example (function with multiple arguments)

$(\lambda c \_ \rightarrow c) \text{ True } (3 * 4 + 3 * 4)$   
 $\rightarrow \{\text{need function}\}$   
     $(\lambda c \_ \rightarrow c) \text{ True}$   
     $\rightarrow \{\text{function application}\}$   
         $(\backslash \_ \rightarrow \text{True})$   
...  $(\backslash \_ \rightarrow \text{True}) (3 * 4 + 3 * 4)$   
 $\rightarrow \{\text{function application}\}$   
     $\text{True}$



# Quiz time!

What does the following expressions evaluate to?

$$(\lambda x \rightarrow x) \text{ True} = \text{True}$$

$$(\lambda x \rightarrow x) \perp = \perp$$

$$(\lambda x \rightarrow ()) \perp = ()$$

$$(\lambda x \rightarrow \perp) () = \perp$$

$$(\lambda x f \rightarrow f x) \perp = \lambda f \rightarrow f \perp$$

$$\text{length} (\text{map } \perp [1, 2]) = 2$$

# Pattern matching

**Rule:** For each pattern (in order), evaluate the expression “as much as needed” to be able to verify or refute a pattern.

**case** *const (Just a) b of {Just \_ → True; Nothing → False}*  
→ {evaluate argument for pattern Just \_}  
    *const (Just a) b*  
→ {function application}  
    *Just a*  
... **case** *Just a of {Just \_ → True; Nothing → False}*  
→ {match pattern}  
    *True*

## Pattern matching summarized

- Reduce the expression to be pattern matched to Weak Head Normal Form, and check for constructor match with outermost constructor in pattern.
- Repeat for corresponding subpatterns and subexpressions.

An expression is in **Weak Head Normal Form (WHNF)**, if it is either:

- a constructor (eventually applied to arguments) like *True*, *Just (square 42)* or  $(:) 1 []$
- a built-in function applied to too few arguments (perhaps none) like  $(+) 2$  or *sqrt*.
- or a lambda abstraction  $\lambda x \rightarrow expression$ .

## Ex 1/4: Pattern matching (Level 1) - Built-in primitive

**case**  $3 * 4$  **of**  $\{0 \rightarrow \text{True}; \_ \rightarrow \text{False}\}$   
→ {evaluate argument for pattern 0}  
     $3 * 4$   
→     {arithmetic}  
        12  
... **case** 12 **of**  $\{0 \rightarrow \text{True}; \_ \rightarrow \text{False}\}$   
→ {match pattern}  
    *False*

## Ex 2/4: Pattern matching (Level 0) - Variable or wildcard

**case** 3 \* 4 **of** { $x \rightarrow \text{True}$ }  
 $\rightarrow$  {match pattern}  
*True*

## Ex 3/4: Pattern matching (Level 1) - Variable in pattern

```
case Just (3 * 4) of {Just  $x \rightarrow (x, x)$ ; Nothing  $\rightarrow (0, 0)$ }  
 $\rightarrow$  {match pattern}  
let  $x = 3 * 4$  in  $(x, x)$ 
```



## Ex 4/4: Pattern matching in function definitions

*cond* :: *Bool* → *a* → *a* → *a*

*cond True x y* = *x*

*cond False x y* = *y*

Evaluate

*cond* (4 < 2) (5 \* 17) (2 \* 3)

→ {check if expression matches True}

4 < 2

→ {primitive comparison}

*False*

... *cond False* (5 \* 7) (2 \* 3)

→ {check if expression matches False}

2 \* 3

→ {primitive arithmetic}

6

How do we proceed when evaluating

$$\text{map } (1+) (\text{map } (2*) [1,2,3])$$

in the **leftmost outermost reduction with sharing?**

Recall that

$$\begin{aligned}\text{map } f [] &= [] \\ \text{map } f (x : xs) &= f\ x : \text{map } f\ xs\end{aligned}$$

See that we calculate elements of the resulting list one by one.  
So the list we are working on could potentially be infinite, if we only consume finitely many elements!



# Infinite lists

Lazy evaluation allows us to program with **infinite lists** of values!

Consider the recursive definition

$$ones :: [Int]$$
$$ones = 1 : ones$$

Unfolding the recursion a few times gives

$$ones = 1 : ones$$
$$= 1 : 1 : ones$$
$$= 1 : 1 : 1 : ones$$
$$= \dots$$

That is, *ones* is the **infinite list** of 1's.

## Innermost vs. outermost

1. Now consider evaluating *head ones* using innermost reduction:

$$\begin{aligned} \text{head ones} &= \text{head } (1 : \text{ones}) \\ &= \text{head } (1 : 1 : \text{ones}) \\ &= \text{head } (1 : 1 : 1 : \text{ones}) \\ &= \dots \end{aligned}$$

Does not terminate!

2. And with outermost reduction:

$$\begin{aligned} \text{head ones} &= \text{head } (1 : \text{ones}) \\ &= 1 \end{aligned}$$

Terminates!

# Infinite lists

We now see that

$$ones = 1 : ones$$

really defines a **potentially infinite list** that is only evaluated as much as needed by the context in which it is used.

# Modular programming

We can generate **finite** lists by taking elements from infinite lists. For example

- *take 5 ones* = [1, 1, 1, 1, 1]
- *take 5 [1..]* = [1, 2, 3, 4, 5]

Lazy evaluation allows us to make programs **more modular**, by separating control from data:

$$\underbrace{\text{take } 5}_{\text{control}} \underbrace{[1..]}_{\text{data}}$$

Using lazy evaluation, the data is only evaluated as much as required by the control part.

## Modular programming: Example

Consider

$$\text{replicate} :: \text{Int} \rightarrow a \rightarrow [a]$$

In a non-lazy language, we would define it like:

$$\text{replicate } 0 \ v = []$$

$$\text{replicate } n \ v = v : \text{replicate } (n - 1) \ v$$

Since Haskell is lazy, we can make the even simpler function

$$\text{repeat} :: a \rightarrow [a]$$

$$\text{repeat } v = vs \textbf{ where } vs = v : vs$$

and define *replicate* by

$$\text{replicate } n = \text{take } n \circ \text{repeat}$$

# Primes: The seive of Eratosthenes

*primes* :: [Integer]

*primes* = *seive* [2..]

*seive* :: [Integer] → [Integer]

*seive* (*p* : *xs*) = *p* : *seive* [*x* | *x* ← *xs*, *x* 'mod' *p* ≠ 0]

*primes* = [2,3,5,7,11,13,17,19...]

# Fibonacci numbers

*fibs* :: [Integer]

*fibs* = 0 : 1 : zipWith (+) *fibs* (tail *fibs*)

# Cyclic structures

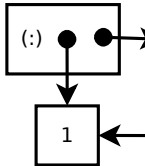
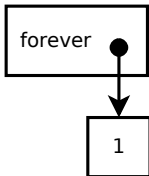
Two similar structures with different representation in the computer's memory:

$ones = forever\ 1$

$forever\ x = x : forever\ x$

$ones' = forever'\ 1$

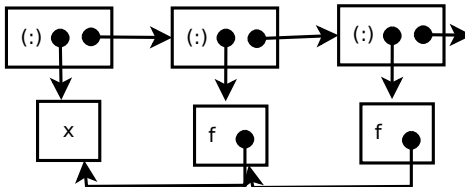
$forever'\ x = zs\ \mathbf{where}\ zs = x : zs$





# The efficiency of *iterate*

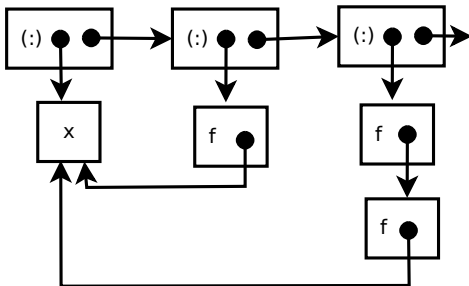
$iterate\ f\ x = x : iterate\ f\ (f\ x)$



# The efficiency of *iterate*

$$\text{iterate}' f x = x : \text{map } f (\text{iterate}' f x)$$

**Demo with :sprint.**



## Laziness for memoization

*fibs* = map *fib'* [0..]

*fib'* 0 = 0

*fib'* 1 = 1

*fib'* *n* = *fib* (*n* - 1) + *fib* (*n* - 2)

*fib* *n* = *fibs* !! *n*

```
> fib 20000
```

```
2531162323732361242240155...
```

```
(2.01 secs, 21312464 bytes)
```

```
> fib 20000
```

```
2531162323732361242240155...
```

```
(0.02 secs, 4668504 bytes)
```

## Laziness for memoization

*fibs* = *map fib'* [0..]

*fib'* 0 = 0

*fib'* 1 = 1

*fib'* *n* = *fib* (*n* - 1) + *fib* (*n* - 2)

*fib* *n* = *fibs* !! *n*

```
*Main> :sprint fibs
```

```
fibs = _
```

```
*Main> take 1 fibs
```

```
[0]
```

```
*Main> :sprint fibs
```

```
fibs = 0 : _
```

```
*Main> fib 10
```

```
55
```

```
*Main> :sprint fibs
```

```
fibs = 0 : 1 : 1 : 2 : 3 : 5 : 8 : 13 : 21 : 34 : 55 : _
```

```
*Main>
```

# The dark side of Lazy Evaluation

- We have seen, that *lazy evaluation* always uses the same or fewer number of reduction steps as *eager evaluation*.
- But: If you are not careful, your algorithm might use more **space** than needed. **The unevaluated expressions take up space!**

# Summing up numbers

$$\text{foldr } f \ z \ [] = z$$

$$\text{foldr } f \ z \ (x : xs) = f \ x \ (\text{foldr } f \ z \ xs)$$

$$\begin{aligned}
 \text{foldr } (+) \ 0 \ (1 : 2 : 3 : []) &= 1 + \text{foldr } (+) \ 0 \ (2 : 3 : []) \\
 &= 1 + (2 + \text{foldr } (+) \ 0 \ (3 : [])) \\
 &= 1 + (2 + (3 + \text{foldr } (+) \ 0 \ [])) \\
 &= 1 + (2 + (3 + 0)) \\
 &= 1 + (2 + 3) \\
 &= 1 + 5 \\
 &= 6
 \end{aligned}$$

$O(n)$  space! Overflow for large lists!

## Summing up numbers

$$\text{foldl } f \ z \ [] = z$$

$$\text{foldl } f \ z \ (x : xs) = \text{foldl } f \ (f \ z \ x) \ xs$$

$$\begin{aligned}\text{foldl } (+) \ 0 \ (1 : 2 : 3 : []) &= \text{foldl } (+) \ (0 + 1) \ (2 : 3 : []) \\ &= \text{foldl } (+) \ ((0 + 1) + 2) \ (3 : []) \\ &= \text{foldl } (+) \ (((0 + 1) + 2) + 3) \ ([]) \\ &= ((0 + 1) + 2) + 3 \\ &= (1 + 2) + 3 \\ &= 3 + 3 \\ &= 6\end{aligned}$$

$O(n)$  space again! What do we do?

## Forcing evaluation

Haskell has the following primitive function

$$seq :: a \rightarrow b \rightarrow b \quad \text{-- primitive}$$

The call

$$seq\ x\ y$$

evaluates  $x$  before returning  $y$ .

The function  $seq$  can be used to define strict function application:

$$(\$!) :: (a \rightarrow b) \rightarrow a \rightarrow b$$

$$f\ \$!\ x = x\ 'seq'\ f\ x$$



## Quiz time!

Forcing only evaluates to weak head normal form (i.e., a lambda abstraction, literal or constructor application)

What does the following expressions evaluate to?

$$(\lambda x \rightarrow ()) \$! \perp = \perp$$

$$\text{seq } (\perp, \perp) () = ()$$

$$\text{snd } \$! (\perp, \perp) = \perp$$

$$(\lambda x \rightarrow ()) \$! (\lambda x \rightarrow \perp) = ()$$

$$\text{length } \$! \text{ map } \perp [1,2] = 2$$

$$\text{seq } (\perp + \perp) () = \perp$$

$$\text{seq } (\text{foldr } \perp \perp) () = ()$$

$$\text{seq } (1 : \perp) () = ()$$

## Summing up numbers

$$\text{foldl}' f z [] = z$$

$$\text{foldl}' f z (x : xs) = \mathbf{let} \ z' = f \ z \ x \ \mathbf{in} \ z' \ \text{'seq'} \ (\text{foldl}' f z' xs)$$

$$\text{foldl}' (+) 0 (1:2:3:[])$$

$$= \mathbf{let} \ z' = (0 + 1) \ \mathbf{in} \ z' \ \text{'seq'} \ \text{foldl}' (+) z' (2:3:[])$$

$$= \text{foldl}' (+) 1 (2:3:[])$$

$$= \mathbf{let} \ z' = (1 + 2) \ \mathbf{in} \ z' \ \text{'seq'} \ \text{foldl}' (+) z' (3:[])$$

$$= \text{foldl}' (+) 3 (3:[])$$

$$= \mathbf{let} \ z' = (3 + 3) \ \mathbf{in} \ z' \ \text{'seq'} \ \text{foldl}' (+) z' (3:[])$$

$$= \text{foldl}' (+) 6 []$$

$$= 6$$

$O(1)$  space.