

LiU-FP2016: Lecture 5

Lazy Functional Programming

Henrik Nilsson

University of Nottingham, UK

LiU-FP2016: Lecture 5 – p.1/46

Imperative vs. Declarative (1)

- **Imperative Languages:**
 - Implicit state.
 - Computation essentially a sequence of side-effecting actions.
 - Examples: Procedural and OO languages
- **Declarative Languages** (Lloyd 1994):
 - **No** implicit state.
 - A program can be regarded as a theory.
 - Computation can be seen as deduction from this theory.
 - Examples: Logic and Functional Languages.

LiU-FP2016: Lecture 5 – p.2/46

Imperative vs. Declarative (2)

Another perspective:

- **Algorithm = Logic + Control**
- Declarative programming emphasises the logic (“what”) rather than the control (“how”).
- Strategy needed for providing the “how”:
 - Resolution (logic programming languages)
 - Lazy evaluation (some functional and logic programming languages)
 - (Lazy) narrowing: (functional logic programming languages)

LiU-FP2016: Lecture 5 – p.3/46

No Control?

Declarative languages for practical use tend to be only **weakly declarative**; i.e., not totally free of control aspects. For example:

- Equations in functional languages are directed.
- Order of patterns often matters for pattern matching.
- Constructs for taking control over the order of evaluation. (E.g. `cut` in Prolog, `seq` in Haskell.)

LiU-FP2016: Lecture 5 – p.4/46

Relinquishing Control

Theme of this lecture: *relinquishing control by exploiting lazy evaluation.*

- Evaluation orders
- Strict vs. Non-strict semantics
- Lazy evaluation
- Applications of lazy evaluation:
 - Programming with infinite structures
 - Circular programming
 - Dynamic programming
 - Attribute grammars

LIU-FP2016: Lecture 5 – p.5/46

Evaluation Orders (1)

Consider:

```
sqr x = x * x
dbl x = x + x
main = sqr (dbl (2 + 3))
```

Roughly, any expression that can be evaluated or **reduced** by using the equations as rewrite rules is called a **reducible expression** or **redex**.

Assuming arithmetic, the redexes of the body of `main` are:

- `2 + 3`
- `dbl (2 + 3)`
- `sqr (dbl (2 + 3))`

LIU-FP2016: Lecture 5 – p.6/46

Evaluation Orders (2)

Thus, in general, many possible reduction orders. Innermost, leftmost redex first is called **Applicative Order Reduction** (AOR). Recall:

```
sqr x = x * x
dbl x = x + x
main = sqr (dbl (2 + 3))
```

Starting from `main`:

```
main ⇒ sqr (dbl (2 + 3)) ⇒ sqr (dbl 5)
⇒ sqr (5 + 5) ⇒ sqr 10 ⇒ 10 * 10 ⇒ 100
```

This is just **Call-By-Value**.

LIU-FP2016: Lecture 5 – p.7/46

Evaluation Orders (3)

Outermost, leftmost redex first is called **Normal Order Reduction** (NOR):

```
main ⇒ sqr (dbl (2 + 3))
⇒ dbl (2 + 3) * dbl (2 + 3)
⇒ ((2 + 3) + (2 + 3)) * dbl (2 + 3)
⇒ (5 + (2 + 3)) * dbl (2 + 3)
⇒ (5 + 5) * dbl (2 + 3) ⇒ 10 * dbl (2 + 3)
⇒ ... ⇒ 10 * 10 ⇒ 100
```

(Applications of arithmetic operations only considered redexes once arguments are numbers.) Demand-driven evaluation or **Call-By-Need**

LIU-FP2016: Lecture 5 – p.8/46

Why Normal Order Reduction? (1)

NOR seems rather inefficient. Any use?

- Best possible termination properties.

A pure functional languages is just the λ -calculus in disguise. Two central theorems:

- Church-Rosser Theorem I:
No term has more than one normal form.
- Church-Rosser Theorem II:
If a term has a normal form, then NOR will find it.

LIU-FP2016: Lecture 5 – p.9/46

Why Normal Order Reduction? (2)

- More expressive power; e.g.:
 - “Infinite” data structures
 - Circular programming
- More declarative code as control aspects (order of evaluation) left implicit.

LIU-FP2016: Lecture 5 – p.10/46

Exercise 1

Consider:

```
f x = 1
g x = g x
main = f (g 0)
```

Attempt to evaluate `main` using both AOR and NOR. Which order is the more efficient in this case? (Count the number of reduction steps to normal form.)

LIU-FP2016: Lecture 5 – p.11/46

Strict vs. Non-strict Semantics (1)

- \perp , or “bottom”, the **undefined value**, representing **errors** and **non-termination**.
- A function f is **strict** iff:

$$f \perp = \perp$$

For example, $+$ is strict in both its arguments:

$$\begin{aligned}(0/0) + 1 &= \perp + 1 = \perp \\ 1 + (0/0) &= 1 + \perp = \perp\end{aligned}$$

LIU-FP2016: Lecture 5 – p.12/46

Strict vs. Non-strict Semantics (2)

Again, consider:

```
f x = 1
g x = g x
```

What is the value of $f (0/0)$? Or of $f (g 0)$?

- AOR: $f (0/0) \Rightarrow \perp$; $f (g 0) \Rightarrow \perp$
Conceptually, $f \perp = \perp$; i.e., f is strict.
- NOR: $f (0/0) \Rightarrow 1$; $f (g 0) \Rightarrow 1$
Conceptually, $f \circ \perp = 1$; i.e., $f \circ \circ$ is non-strict.

Thus, NOR results in non-strict semantics.

LIU-FP2016: Lecture 5 – p.13/46

Lazy Evaluation (1)

Lazy evaluation is a **technique for implementing NOR** more efficiently:

- A redex is evaluated **only if needed**.
- **Sharing** employed to avoid duplicating redexes.
- Once evaluated, a redex is **updated** with the result to avoid evaluating it more than once.

As a result, under lazy evaluation, any one redex is evaluated at most once.

LIU-FP2016: Lecture 5 – p.14/46

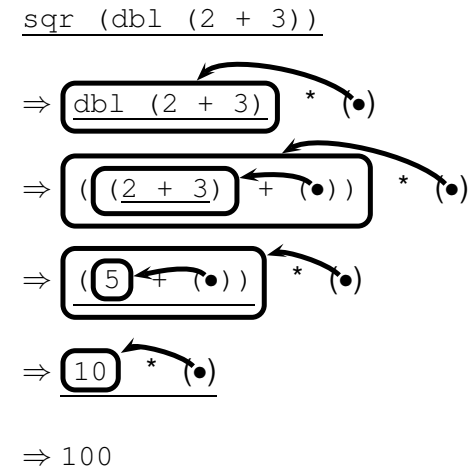
Lazy Evaluation (2)

Recall:

```
sqr x = x * x
dbl x = x + x
```

main =

```
sqr (dbl (2+3))
```



LIU-FP2016: Lecture 5 – p.15/46

Exercise 2

Evaluate `main` using AOR, NOR, and lazy evaluation:

```
f x y z = x * z
g x      = f (x * x) (x * 2) x
main     = g (1 + 2)
```

(Only consider an applications of an arithmetic operator a redex once the arguments are numbers.)

How many reduction steps in each case?

Answer: 7, 8, 6 respectively

LIU-FP2016: Lecture 5 – p.16/46

Infinite Data Structures (1)

```
take 0 xs      = []  
take n []     = []  
take n (x:xs) = x : take (n-1) xs
```

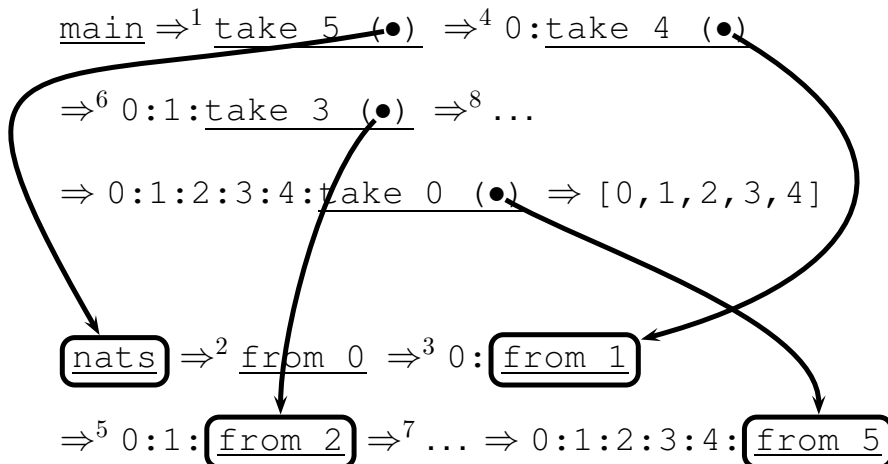
```
from n = n : from (n+1)
```

```
nats = from 0
```

```
main = take 5 nats
```

LIU-FP2016: Lecture 5 – p.17/46

Infinite Data Structures (2)



LIU-FP2016: Lecture 5 – p.18/46

Circular Data Structures (2)

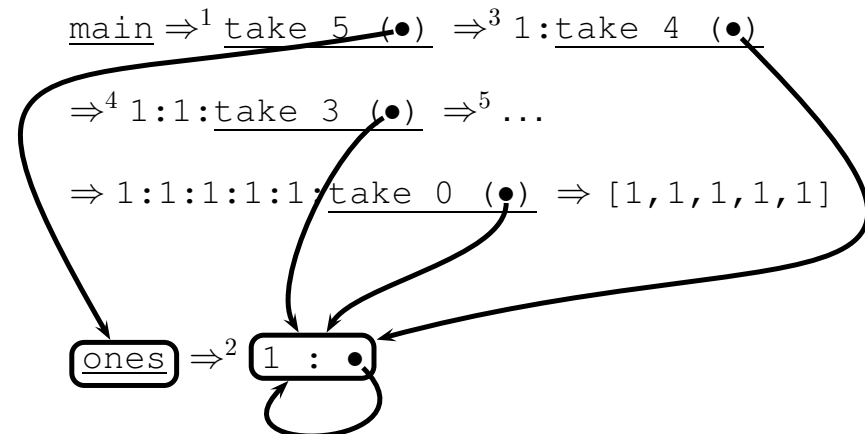
```
take 0 xs      = []  
take n []     = []  
take n (x:xs) = x : take (n-1) xs
```

```
ones = 1 : ones
```

```
main = take 5 ones
```

LIU-FP2016: Lecture 5 – p.19/46

Circular Data Structures (2)



LIU-FP2016: Lecture 5 – p.20/46

Exercise 3

Given the following tree type

```
data Tree = Empty
          | Node Tree Int Tree
```

define:

- An infinite tree where every node is labelled by 1.
- An infinite tree where every node is labelled by its depth from the root node.

LIU-FP2016: Lecture 5 – p.21/46

Exercise 3: Solution

```
treeOnes = Node treeOnes 1 treeOnes

treeFrom n = Node (treeFrom (n + 1))
                  n
                  (treeFrom (n + 1))

treeDepths = treeFrom 0
```

LIU-FP2016: Lecture 5 – p.22/46

Circular Programming (1)

A non-empty tree type:

```
data Tree = Leaf Int | Node Tree Tree
```

Suppose we would like to write a function that replaces each leaf integer in a given tree with the **smallest** integer in that tree.

How many passes over the tree are needed?

One!

LIU-FP2016: Lecture 5 – p.23/46

Circular Programming (2)

Write a function that replaces all leaf integers by a given integer, and returns the new tree along with the smallest integer of the given tree:

```
fmr :: Int -> Tree -> (Tree, Int)
fmr m (Leaf i) = (Leaf m, i)
fmr m (Node tl tr) =
  (Node tl' tr', min ml mr)
  where
    (tl', ml) = fmr m tl
    (tr', mr) = fmr m tr
```

LIU-FP2016: Lecture 5 – p.24/46

Circular Programming (3)

For a given tree t , the desired tree is now obtained as the **solution** to the equation:

$$(t', m) = \text{fmr } m \ t$$

Thus:

$$\begin{aligned} \text{findMinReplace } t &= t' \\ \text{where} \\ (t', m) &= \text{fmr } m \ t \end{aligned}$$

Intuitively, this works because `fmr` can compute its result without needing to know the **value** of m .

LIU-FP2016: Lecture 5 – p.25/46

A Simple Spreadsheet Evaluator (1)

	a	b	c
1	c3 + c2		
2	a3 * b2	2	a2 + b2
3	7		a2 + a3

 \Rightarrow

	a	b	c
1	37		
2	14	2	16
3	7		21

```
s' = array (bounds s)
  [ (r, evalCell s' (s ! r))
  | r <- indices s ]
```

The evaluated sheet is again simply the **solution** to the stated equation. No need to worry about evaluation order. **Any caveats?**

LIU-FP2016: Lecture 5 – p.26/46

A Simple Spreadsheet Evaluator (2)

As it is quite instructive, let us develop this evaluator together. Some definitions to get us started:

```
type CellRef = (Char, Int)

type Sheet a = Array CellRef a

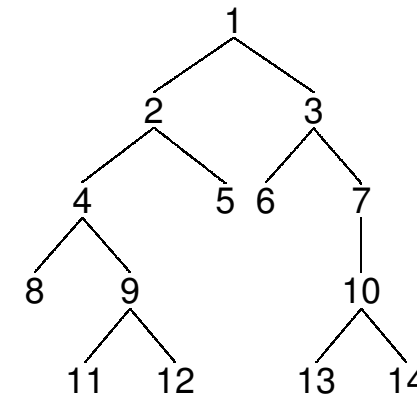
data BinOp = Add | Sub | Mul | Div

data Exp = Lit Double
         | Ref CellRef
         | App BinOp Exp Exp
```

LIU-FP2016: Lecture 5 – p.27/46

Breadth-first Numbering (1)

Consider the problem of numbering a possibly infinitely deep tree in breadth-first order:



LIU-FP2016: Lecture 5 – p.28/46

Breadth-first Numbering (2)

The following algorithm is due to G. Jones and J. Gibbons (1992), but the presentation differs.

Consider the following tree type:

```
data Tree a = Empty
            | Node (Tree a) a (Tree a)
```

Define:

$\text{width } t \ i$ The width of a tree t at level i (0 origin).

$\text{label } t \ i \ j$ The j th label at level i of a tree t (0 origin).

LIU-FP2016: Lecture 5 – p.29/46

Breadth-first Numbering (3)

The following system of equations defines breadth-first numbering:

$$\text{label } t \ 0 \ 0 = 1 \quad (1)$$

$$\text{label } t \ (i + 1) \ 0 = \text{label } t \ i \ 0 + \text{width } t \ i \quad (2)$$

$$\text{label } t \ i \ (j + 1) = \text{label } t \ i \ j + 1 \quad (3)$$

Note that $\text{label } t \ i \ 0$ is defined for **all** levels i (as long as the widths of all tree levels are finite).

LIU-FP2016: Lecture 5 – p.30/46

Breadth-first Numbering (4)

The code that follows sets up the defining system of equations:

- **Streams** (infinite lists) of labels are used as a **mediating data structure** to allow equations to be set up between adjacent nodes within levels and between the last node at one level and the first node at the next.
- Idea: the tree numbering function for a subtree takes a stream of labels for the **first node** at each level, and returns a stream of labels for the **node after the last node** at each level.

LIU-FP2016: Lecture 5 – p.31/46

Breadth-first Numbering (5)

- As there manifestly are **no cyclic dependences** among the equations, we can entrust the details of solving them to the lazy evaluation machinery in the safe knowledge that a solution will be found.

LIU-FP2016: Lecture 5 – p.32/46

Breadth-first Numbering (6)

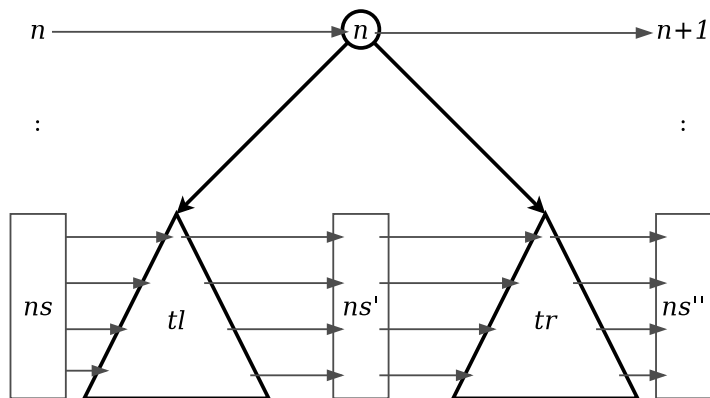
```

bfn :: Tree a -> Tree Integer
bfn t = t'
  where
    (ns, t') = bfnAux (1 : ns) t

bfnAux :: [Integer] -> Tree a
        -> ([Integer], Tree Integer)
bfnAux ns Empty = (ns, Empty)
bfnAux (n : ns) (Node tl _ tr) = ((n + 1) : ns'',
                                  Node tl' n tr')
  where
    (ns', tl') = bfnAux ns tl
    (ns'', tr') = bfnAux ns' tr
  
```

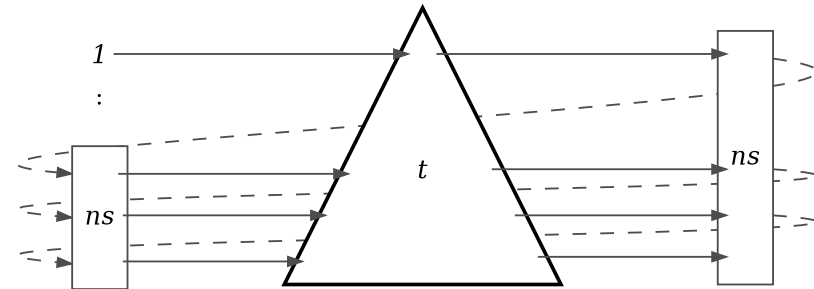
LIU-FP2016: Lecture 5 – p.33/46

Breadth-first Numbering (7)



LIU-FP2016: Lecture 5 – p.34/46

Breadth-first Numbering (8)



LIU-FP2016: Lecture 5 – p.35/46

Dynamic Programming

Dynamic Programming:

- Create a **table** of all subproblems that ever will have to be solved.
- Fill in table without regard to whether the solution to that particular subproblem will be needed.
- Combine solutions to form overall solution.

Lazy Evaluation is a perfect match as saves us from having to worry about finding a suitable evaluation order.

LIU-FP2016: Lecture 5 – p.36/46

The Triangulation Problem (1)

Select a set of **chords** that divides a convex polygon into triangles such that:

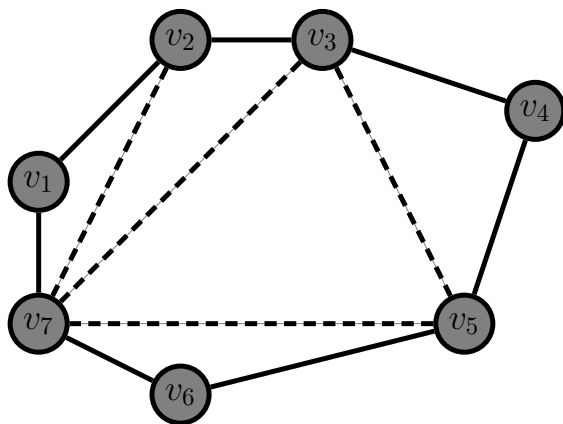
- no two chords cross each other
- the sum of their length is minimal.

We will only consider computing the minimal length.

See Aho, Hopcroft, Ullman (1983) for details.

LIU-FP2016: Lecture 5 – p.37/46

The Triangulation Problem (2)



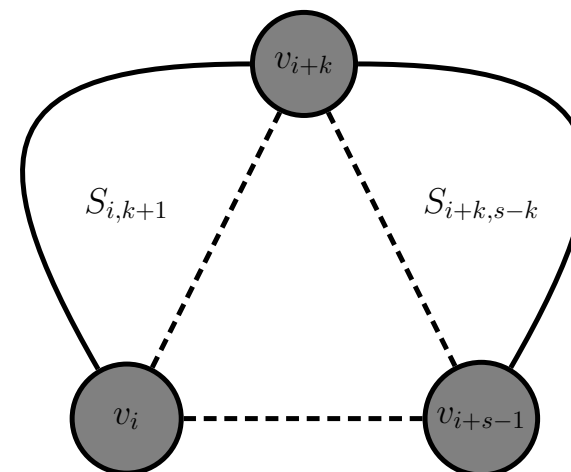
LIU-FP2016: Lecture 5 – p.38/46

The Triangulation Problem (3)

- Let S_{is} denote the subproblem of size s starting at vertex v_i of finding the minimum triangulation of the polygon $v_i, v_{i+1}, \dots, v_{i+s-1}$ (counting modulo the number of vertices).
- Subproblems of size less than 4 are trivial.
- Solving S_{is} is done by solving $S_{i,k+1}$ and $S_{i+k,s-k}$ for all $k, 1 \leq k \leq s-2$
- The obvious recursive formulation results in 3^{s-4} (non-trivial) calls.
- But for $n \geq 4$ vertices there are only $n(n-3)$ non-trivial subproblems!

LIU-FP2016: Lecture 5 – p.39/46

The Triangulation Problem (4)



LIU-FP2016: Lecture 5 – p.40/46

The Triangulation Problem (5)

- Let C_{is} denote the minimal triangulation cost of S_{is} .
- Let $D(v_p, v_q)$ denote the length of a chord between v_p and v_q (length is 0 for non-chords; i.e. adjacent v_p and v_q).
- For $s \geq 4$:

$$C_{is} = \min_{k \in [1, s-2]} \left\{ \begin{array}{l} C_{i, k+1} + C_{i+k, s-k} \\ + D(v_i, v_{i+k}) + D(v_{i+k}, v_{i+s-1}) \end{array} \right\}$$

- For $s < 4$, $S_{is} = 0$.

LIU-FP2016: Lecture 5 – p.41/46

The Triangulation Problem (6)

These equations can be transliterated straight into Haskell:

```
triCost :: Polygon -> Double
triCost p = cost!(0,n) where
  cost = array ((0,0), (n-1,n))
    [ ( (i,s),
        minimum [ cost!(i, k+1)
                  + cost!((i+k) `mod` n, s-k)
                  + dist p i ((i+k) `mod` n)
                  + dist p ((i+k) `mod` n)
                    ((i+s-1) `mod` n)
                  | k <- [1..s-2] ])
      | i <- [0..n-1], s <- [4..n] ] ++
    [ ((i,s), 0.0)
      | i <- [0..n-1], s <- [0..3] ]
  n = snd (bounds b) + 1
```

LIU-FP2016: Lecture 5 – p.42/46

Attribute Grammars (1)

Lazy evaluation is also very useful for evaluation of **Attribute Grammars**:

- The attribution function is defined recursively over the tree:
 - takes inherited attributes as extra arguments;
 - returns a tuple of all synthesised attributes.
- As long as there exists **some** possible attribution order, lazy evaluation will take care of the attribute evaluation.

LIU-FP2016: Lecture 5 – p.43/46

Attribute Grammars (2)

- The earlier examples on Circular Programming and Breadth-first Numbering can be seen as instances of this idea.

LIU-FP2016: Lecture 5 – p.44/46

Reading

- John W. Lloyd. Practical advantages of declarative programming. In *Joint Conference on Declarative Programming, GULP-PRODE'94*, 1994.
- John Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–197, April 1989.
- Thomas Johnsson. Attribute Grammars as a Functional Programming Paradigm. In *Functional Programming Languages and Computer Architecture, FPCA'87*, 1987

LIU-FP2016: Lecture 5 – p.45/46

Reading

- Geraint Jones and Jeremy Gibbons. *Linear-time breadth-first tree algorithms: An exercise in the arithmetic of folds and zips*. Technical Report TR-31-92, Oxford University Computing Laboratory, 1992.
- Alfred Aho, John Hopcroft, Jeffrey Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

LIU-FP2016: Lecture 5 – p.46/46