

# MGS 2011: FUN Lecture 1

## *Lazy Functional Programming*

Henrik Nilsson

University of Nottingham, UK

MGS 2011: FUN Lecture 1 – p.1/40

## What Is a Functional Language? (1)

- **Imperative Languages:**
  - Implicit state.
  - Computation essentially a sequence of side-effecting actions.
- **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.

MGS 2011: FUN Lecture 1 – p.2/40

## What Is a Functional Language? (2)

Another perspective:

- **Algorithm = Logic + Control**
- Declarative programming emphasises the logic (“what”) rather than the control (“how”).
- Examples:
  - Resolution (logic programming)
  - Lazy evaluation (found in some functional and logic languages)

MGS 2011: FUN Lecture 1 – p.3/40

## What Is a Functional Language? (3)

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.)

MGS 2011: FUN Lecture 1 – p.4/40

## What Is a Functional Language? (4)

Exactly what constitute a functional language is somewhat contentious.

Pragmatically, a functional language is one that encourages a mostly declarative, **functional style** of programming.

Typical features/characteristics:

- Functions are first-class entities.
- Computation expressed through function application.
- Recursive (and co-recursive) definitions.

MGS 2011: FUN Lecture 1 – p.5/40

## What Is a Functional Language? (5)

This “definition” covers both:

- **Pure** functional languages: no side effects
  - (Weakly) declarative: equational reasoning valid (with care); **referentially transparent**.
  - Example: Haskell
- **Mostly** functional languages: some side effects, e.g. for I/O.
  - Equational reasoning with care.
  - Examples: ML, OCaml, Scheme, Erlang

(Real purists would point out that non-termination is also a side effect.)

MGS 2011: FUN Lecture 1 – p.6/40

## This and the Following Lectures

- In this and the following lectures we will explore **Purely Functional Programming** through the use of **Haskell**.
- Theme of today: **Relinquishing control: exploiting lazy evaluation**

Will assume some familiarity with functional programming in a language like Haskell or ML. Will explain Haskell syntax and other points as needed: **Just ask!**

MGS 2011: FUN Lecture 1 – p.7/40

## Evaluation Orders (1)

Consider:

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

Many possible reduction orders. Innermost, leftmost **redex** first is called **Applicative Order Reduction** (AOR):

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

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

MGS 2011: FUN Lecture 1 – p.8/40

## Evaluation Orders (2)

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**

MGS 2011: FUN Lecture 1 – p.9/40

## Why Normal Order Reduction? (1)

NOR seems rather inefficient. Any use?

- Best possible termination properties. Two important theorems from the  $\lambda$ -calculus:
  - 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.

MGS 2011: FUN Lecture 1 – p.10/40

## 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.

MGS 2011: FUN Lecture 1 – p.11/40

## 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:

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

MGS 2011: FUN Lecture 1 – p.12/40

## Strict vs. Non-strict Semantics (2)

Consider:

`foo x = 1`

What is the value of `foo (0/0)`?

- AOR:  $\text{foo } (0/0) \Rightarrow \perp$   
Conceptually,  $\text{foo } \perp = \perp$ ; i.e., `foo` is strict.
- NOR:  $\text{foo } (0/0) \Rightarrow 1$   
Conceptually,  $\text{foo } \perp = 1$ ; i.e., `foo` is non-strict.

Thus, NOR results in non-strict semantics.

Note: NOR gave well-defined result, AOR did not.

MGS 2011: FUN Lecture 1 – p.13/40

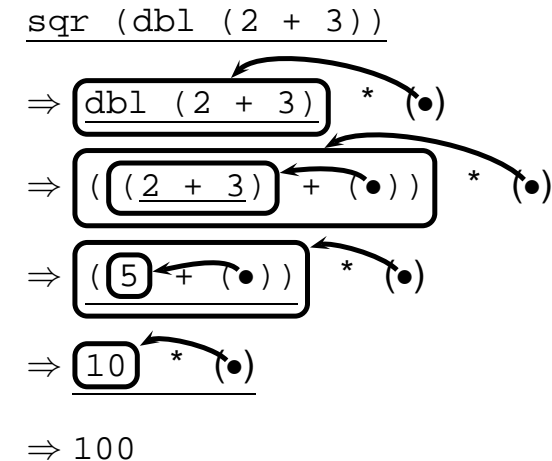
## Lazy Evaluation (1)

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

- An expression is evaluated **only if needed**.
- **Sharing** employed to ensure any one expression evaluated at most once.

MGS 2011: FUN Lecture 1 – p.14/40

## Lazy Evaluation (2)



MGS 2011: FUN Lecture 1 – p.15/40

## Infinite Data Structures (1)

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

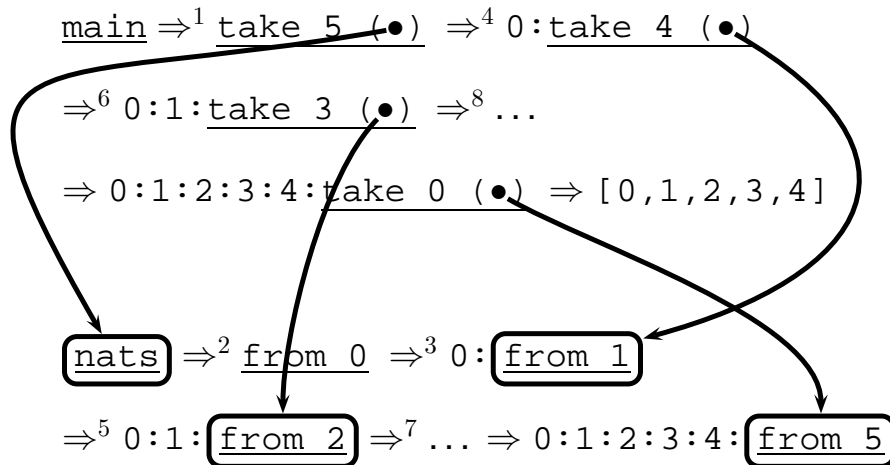
```
from n = n : from (n+1)
```

```
nats = from 0
```

```
main = take 5 nats
```

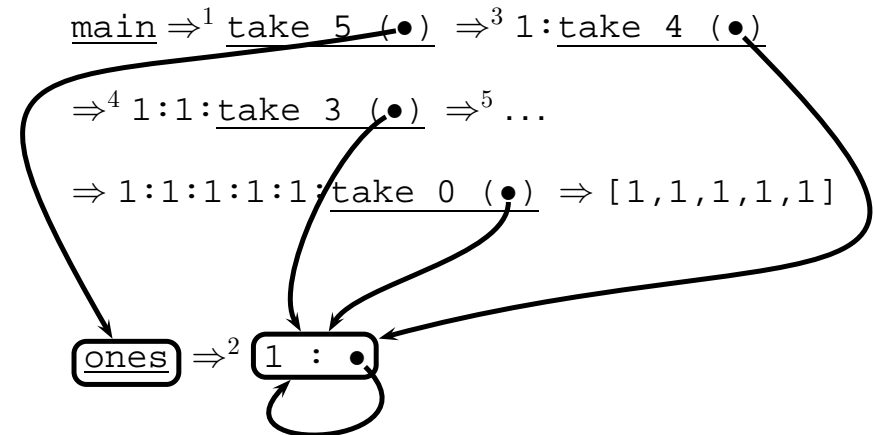
MGS 2011: FUN Lecture 1 – p.16/40

## Infinite Data Structures (2)



MGS 2011: FUN Lecture 1 – p.17/40

## Circular Data Structures (2)



MGS 2011: FUN Lecture 1 – p.19/40

## 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
```

MGS 2011: FUN Lecture 1 – p.18/40

## 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!**

MGS 2011: FUN Lecture 1 – p.20/40

## 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
```

MGS 2011: FUN Lecture 1 – p.21/40

## Circular Programming (3)

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

```
(t', m) = fmr m t
```

Thus:

```
findMinReplace t = t'
  where
    (t', m) = fmr m t
```

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

MGS 2011: FUN Lecture 1 – p.22/40

## A Simple Spreadsheet Evaluator

	a	b	c		a	b	c
1	c3 + c2			⇒	1	37	
2	a3 * b2	2	a2 + b2		2	14	16
3	7		a2 + a3		3	7	21
	s				r		

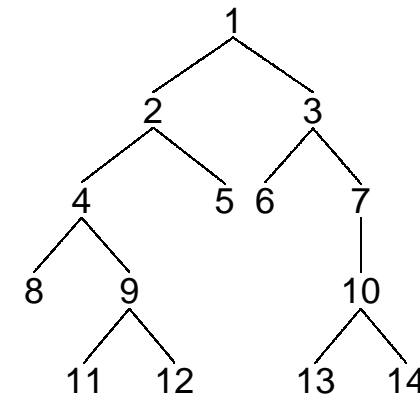
```
r = array (bounds s)
  [ ((i,j), eval r (s!(i,j)))
  | (i,j) <- indices s ]
```

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

MGS 2011: FUN Lecture 1 – p.23/40

## Breadth-first Numbering (1)

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



MGS 2011: FUN Lecture 1 – p.24/40

## 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).

MGS 2011: FUN Lecture 1 – p.25/40

## 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).

MGS 2011: FUN Lecture 1 – p.26/40

## 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.

MGS 2011: FUN Lecture 1 – p.27/40

## 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.

MGS 2011: FUN Lecture 1 – p.28/40

## Breadth-first Numbering (6)

```

bfm :: Tree a -> Tree Integer
bfm t = t'
  where
    (ns, t') = bfmAux (1 : ns) t

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

```

*Eqns (1) & (2)*

*Eqn (3)*

MGS 2011: FUN Lecture 1 – p.29/40

## 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.

MGS 2011: FUN Lecture 1 – p.30/40

## The Triangulation Problem (1)

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

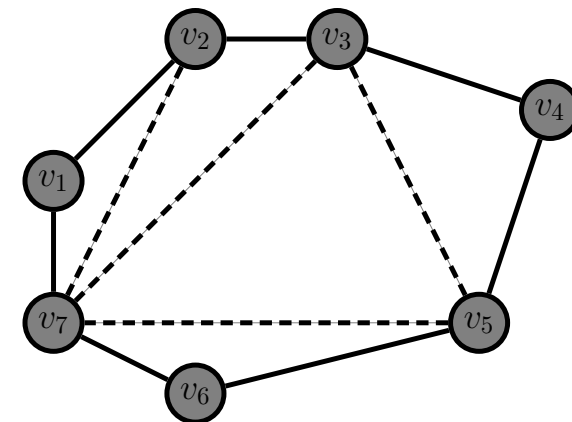
- no two chords cross each other
- the sum of their length is minimal.

We will only consider computing the minimal length.

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

MGS 2011: FUN Lecture 1 – p.31/40

## The Triangulation Problem (2)



MGS 2011: FUN Lecture 1 – p.32/40

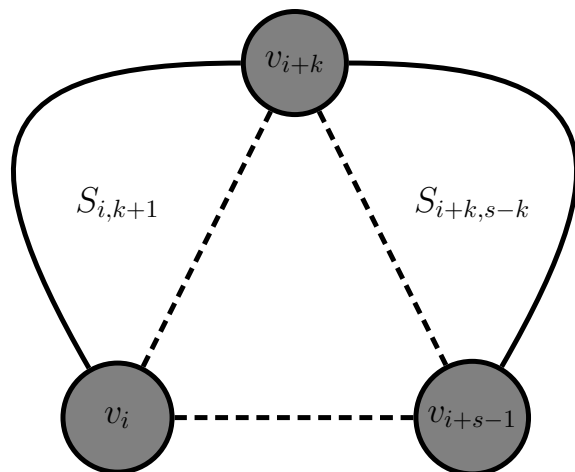


## The Triangulation Problem (3)

- 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!

MGS 2011: FUN Lecture 1 – p.33/40

## The Triangulation Problem (4)



MGS 2011: FUN Lecture 1 – p.34/40

## 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  $s \geq 4$ :

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

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

MGS 2011: FUN Lecture 1 – p.35/40

## 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
```

MGS 2011: FUN Lecture 1 – p.36/40

## 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.

MGS 2011: FUN Lecture 1 – p.37/40

## Attribute Grammars (2)

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

MGS 2011: FUN Lecture 1 – p.38/40

## 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

MGS 2011: FUN Lecture 1 – p.39/40

## 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.

MGS 2011: FUN Lecture 1 – p.40/40