COMP4075: Lecture 3

Pure Functional Programming: Introduction

Henrik Nilsson

University of Nottingham, UK

Pure Functional Programming (1)

The main focus of this module is on ***pure*** functional programming to:

- help you learn how to solve problems purely
- help you understand the pros and cons of doing so
- ultimately allow you to chose the right language/paradigm/techniques, or mix, for the task at hand.

Pure Functional Programming (2)

- Using Haskell as a medium of instruction as it is:
  - the leading pure functional language
  - familiar to many of you from previous modules.
- But the module is not primarily about Haskell: look for the underlying principles!
- The use of Haskell here does not imply it is the only good (functional) language: there are many good languages out there. But grasping pure functional programming will make you a better programmer irrespective of which language you choose/have to use.

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.

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)

Imperative vs. Declarative (3)

- Declarative programming has many benefits; e.g., facilitates formal reasoning, program transformations, etc.
- Immediate payoff of declarative programming permeating ***all*** code is that it allows intent to be stated much more clearly: what not how does matter!
- However, implicit control and unconstrained effects do not mix well: purity is prerequisite.
- ***Disciplined*** use of effects still possible in a pure setting.

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.)

Relinquishing Control

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

- Evaluation orders
- Strict vs. Non-strict semantics
- Lazy evaluation
- Applications of lazy evaluation:
  - Writing clear and concise code
  - Programming with infinite structures
  - Circular programming
  - Dynamic programming

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))

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***.

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***

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.

Why Normal Order Reduction? (2)

- More declarative code as control aspects (order of evaluation) left implicit.
- More reusable components as usage implies control flow
- Better compositionality
- More expressive power; e.g.:
  - "Infinite" data structures
  - Circular programming

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.)

Strict vs. Non-strict Semantics (1)

- $\bot$, or "bottom", the ***undefined value***, representing ***errors*** and ***non-termination***.
- A function f is ***strict*** iff:

$$f\ \bot = \bot$$

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

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

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)` ⇒ $\bot$; `f (g 0)` ⇒ $\bot$
  Conceptually, `f` $\bot = \bot$; i.e., `f` is strict.
- NOR: `f (0/0)` ⇒ `1`; `f (g 0)` ⇒ `1`
  Conceptually, `f` $\bot = 1$; i.e., `f` is non-strict.

Thus, NOR results in non-strict semantics.

Lazy Evaluation (1)

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

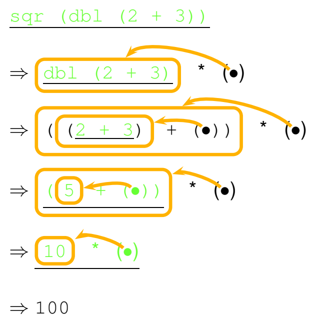
- A redex is evaluated ***only if needed***.
- ***Sharing*** employed to avoid duplicating redexes.
- Once evaluated, a redex is ***updated*** with the result to avoid evaluating it more than once.

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

Lazy Evaluation (2)

Recall:

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



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

Lazy Evaluation (3)

“Evaluated at most once” needs to be interpreted with care: it referes to individual redex ***instances***.

For example:

- `(1 + 2) * (1 + 2)`

  `1 + 2` evaluated twice as ***not the same*** redex.
- `f x = x + y where y = 6 * 7`

  `6 * 7` evaluated whenever `f` is called.

A good compiler will rearrange such computations to avoid duplication of effort, but this has nothing to do with laziness.

Lazy Evaluation (4)

Memoization means caching function results to avoid re-computing them. Also distinct from laziness.

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

Implicit Control Flow (1)

- Leaving the control flow implicit often allows for succinct, to-the-point definitions.
- While not a “game changer”, the improvement over explicit control flow can be substantial.

Implicit Control Flow (2)

Consider:

```
foo x y z
    | x < 0  = (a + b, a * b)
    | x == 0 = (b + c, b * c)
    | x > 0  = (c + a, c * a)
    where
        a = <exprA[y,z]>
        b = <exprB[y,z]>
        c = <exprC[y,z]>
```

Lazy evaluation ensures that only two of `a`, `b`, `c` are evaluated, depending on which ones are needed in the case determined by `x`.

Implicit Control Flow (3)

Avoiding duplication of code and computation in a strict language:

```
foo x y z
    | x < 0  = let a = f y z
                   b = g y z
               in (a + b, a * b)
    | x == 0 = let b = g y z
                   c = g y z
               in (b + c, b * c)
    | x > 0  = let c = g y z
                   a = f y z
               in (c + a, c * a)
```

Implicit Control Flow (4)

```
where
    f y z = <exprA[y,z]>
    g y z = <exprB[y,z]>
    h y z = <exprC[y,z]>
```

(Syntax still Haskell-like to facilitate comparison with previous version.)

Infinite Data Structures (1)

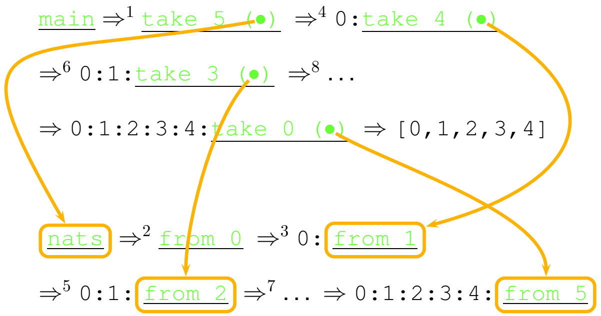
```
take 0 _        = []
take n []       = []
take n (x:xs)   = x : take (n-1) xs

from n = n : from (n+1)

nats = from 0

main = take 5 nats
```

Infinite Data Structures (2)



$\texttt{main} \Rightarrow^{1} \texttt{take 5 (•)} \Rightarrow^{4} \texttt{0:take 4 (•)}$

$\Rightarrow^{6} \texttt{0:1:take 3 (•)} \Rightarrow^{8} \ldots$

$\Rightarrow \texttt{0:1:2:3:4:take 0 (•)} \Rightarrow \texttt{[0,1,2,3,4]}$

$\boxed{\texttt{nats}} \Rightarrow^{2} \texttt{from 0} \Rightarrow^{3} \texttt{0:}\boxed{\texttt{from 1}}$

$\Rightarrow^{5} \texttt{0:1:}\boxed{\texttt{from 2}} \Rightarrow^{7} \ldots \Rightarrow \texttt{0:1:2:3:4:}\boxed{\texttt{from 5}}$

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.