COMP4075/G54RFP: Lecture 1

Administrative Details and Introduction

Henrik Nilsson

University of Nottingham, UK

Finding People and Information (1)

- Henrik Nilsson
  Room A08, Computer Science Building
  e-mail: `nhn@cs.nott.ac.uk`

Finding People and Information (1)

- Henrik Nilsson
  Room A08, Computer Science Building
  e-mail: `nhn@cs.nott.ac.uk`
- Main module web page:
  `www.cs.nott.ac.uk/~psznhn/G54RFP`
  `www.cs.nott.ac.uk/~psznhn/COMP4075`

Finding People and Information (1)

- Henrik Nilsson
  Room A08, Computer Science Building
  e-mail: `nhn@cs.nott.ac.uk`
- Main module web page:
  `www.cs.nott.ac.uk/~psznhn/G54RFP`
  `www.cs.nott.ac.uk/~psznhn/COMP4075`
- Moodle (COMP4075):
  `moodle.nottingham.ac.uk/course/view.php?id=94617`

Finding People and Information (2)

- Direct questions concerning lectures and coursework to the ***Moodle COMP4075/G54RFP Forum***.

Finding People and Information (2)

- Direct questions concerning lectures and coursework to the ***Moodle COMP4075/G54RFP Forum***.

  Anyone can ask and answer questions, but you must not post exact solutions to the coursework.

Aims and Motivation (1)

Aims and Motivation (1)

- Why did you opt to take this module?

Aims and Motivation (1)

- Why did you opt to take this module?
- What are good reasons to take this module?

Aims and Motivation (2)

- To introduces tools, techniques, and theory needed for programming real-world applications functionally.

Aims and Motivation (2)

- To introduces tools, techniques, and theory needed for programming real-world applications functionally.
- Particular emphasis on the inherent benefits of functional programming and strong typing for:
  - reuse
  - maintenance
  - concurrency
  - distribution
  - scalability
  - high availability

Aims and Motivation (3)

- Such aspects have:
  - contributed to the popularity of functional programming for demanding applications e.g. in the finance industry
  - have had a significant impact on the design of many languages and frameworks such as Java, C#, and Rust, MapReduce, React

Aims and Motivation (4)

We will use Haskell as medium of instruction, but:

- What is covered has broad applicability.
- Guest lectures and coursework provide opportunities to branch out beyond Haskell.

Content

The module will cover a range of topics, some more foundational, some applied, such as:

- Lazy functional programming
- Purely functional data structures
- Key libraries
- Functional design patterns
- Concurrency
- Web programming
- GUIs

Guest Lectures

- In the process of organising 3 to 4 guest lectures and/or tutorials.
- Time frame: November–December.
- To allow lecturers to travel on the day, these will likely take place in the afternoon slot or the lab slot. Possibly in an ad hoc slot if a lecturer cannot make the Friday.

Literature (1)

No main reference. The following two will be useful, though, both freely available online:

- *Haskell*, Wikibooks
- *Real World Haskell*, by Bryan O'Sullivan, John Goerzen, and Don Stewart

We will also use tutorials, research papers, videos, etc. References given on module web page or as we go along.

Lecture Notes

- Come prepared to take notes.

Lecture Notes

- Come prepared to take notes.
- All ***electronic*** slides, program code, and other supporting material in ***electronic*** form used during the lectures, will be made available on the course web page.

Lecture Notes

- Come prepared to take notes.
- All ***electronic*** slides, program code, and other supporting material in ***electronic*** form used during the lectures, will be made available on the course web page.
- ***However!*** The electronic record of the lectures is neither guaranteed to be complete nor self-contained!

Assessment

- 50 % unseen written examination (1.5 h),
  50 % coursework

Assessment

- 50 % unseen written examination (1.5 h), 50 % coursework
- Coursework, 2 parts:
  - Part I: Basics; 15 h
  - Part II: Advanced topics and applications; 35 h

Assessment

- 50 % unseen written examination (1.5 h), 50 % coursework
- Coursework, 2 parts:
  - Part I: Basics; 15 h
  - Part II: Advanced topics and applications; 35 h
- Coursework support from 18 October.

Assessment

- 50 % unseen written examination (1.5 h),
  50 % coursework
- Coursework, 2 parts:
  - Part I: Basics; 15 h
  - Part II: Advanced topics and applications; 35 h
- Coursework support from 18 October.
- Use time until then to get up to speed on Haskell.

Coursework Timeline

Preliminary timeline (TBC):

- Part I:
  - Release: Wednesday 16 October
  - Deadline: Wednesday 6 November
- Part II:
  - Release: Wednesday 6 November
  - Deadline: Wednesday 11 December

Coursework Timeline

Preliminary timeline (TBC):

- Part I:
  - Release: Wednesday 16 October
  - Deadline: Wednesday 6 November
- Part II:
  - Release: Wednesday 6 November
  - Deadline: Wednesday 11 December

Start early! It is ***not*** possible to do this coursework at the last minute.

COMP4095: Optional Project (1)

- 10 credits (100 hours)

COMP4095: Optional Project (1)

- 10 credits (100 hours)
- Opportunity to learn in depth about aspects of functional programming at scale.

COMP4095: Optional Project (1)

- 10 credits (100 hours)
- Opportunity to learn in depth about aspects of functional programming at scale.
- Project must be clearly related to what is covered in COMP4075/G54RFP, but “functional” interpreted in a broad sense.

COMP4095: Optional Project (1)

- 10 credits (100 hours)
- Opportunity to learn in depth about aspects of functional programming at scale.
- Project must be clearly related to what is covered in COMP4075/G54RFP, but “functional” interpreted in a broad sense.
- Project defined through a “pitch” that must be discussed and agreed. Needs to clarify:
  - The relevance of the project to COMP4075
  - Size appropriate for 10 credits

COMP4095: Optional Project (2)

Preliminary timeline (TBC):

- Release of project criteria: Wednesday 13 November
- Pitch deadline: Wednesday 4 December (but earlier is better)
- Submission deadline (code and report): 15 January

Imperative vs. Declarative (1)

- ***Imperative Languages***:
  - Implicit state.
  - Computation essentially a sequence of side-effecting actions.
  - Examples: Procedural and OO languages

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***

Imperative vs. Declarative (2)

Another perspective:

- ***Algorithm = Logic + Control***
- Declarative programming emphasises the logic (“what”) rather than the control (“how”).

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)

No Control?

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

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.

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.

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:
  - Programming with infinite structures
  - Circular programming
  - Dynamic programming
  - Attribute grammars

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

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)

- $\perp$, 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:

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

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: $\texttt{f}\ (\underline{\texttt{0/0}}) \Rightarrow \bot$; $\texttt{f}\ (\underline{\texttt{g 0}}) \Rightarrow \bot$
  Conceptually, $\texttt{f}\ \bot = \bot$; i.e., `f` is strict.
- NOR: $\underline{\texttt{f (0/0)}} \Rightarrow 1$; $\underline{\texttt{f (g 0)}} \Rightarrow 1$
  Conceptually, $\texttt{f}\ \bot = 1$; 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:

Lazy Evaluation (1)

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

- A redex is evaluated ***only if needed***.

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.

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.

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))
```

Lazy Evaluation (2)

Recall:

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

$\underline{\texttt{sqr (dbl (2 + 3))}}$

$\Rightarrow \boxed{\underline{\texttt{dbl (2 + 3)}}} \ * \ (\bullet)$

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) + (•)) * (•)
```

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 + (•)) * (•)
```

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 * (•)
```

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***.

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.

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.

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?

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

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.