Functional and Concurrent Programming

Simon Thompson
s.j.thompson@kent.ac.uk

CO545 Lecture 1
CO545: functional and concurrent programming
CO545: functional and concurrent programming

Lectures
22 lectures: introduction, functional programming, concurrent programming, going further
CO545: functional and concurrent programming

Lectures
22 lectures: introduction, functional programming, concurrent programming, going further

Classes
10 seminars: from next week.
11 terminal sessions: from this week
CO545: functional and concurrent programming

**Lectures**
22 lectures: introduction, functional programming, concurrent programming, going further

**Classes**
10 seminars: from next week.
11 terminal sessions: from this week

**Lecturers**
Simon Thompson  
`s.j.thompson@kent.ac.uk`
Dominic Orchard  
`d.a.orchard@kent.ac.uk`
CO545: functional and concurrent programming

**Lectures**
22 lectures: introduction, functional programming, concurrent programming, going further

**Classes**
10 seminars: from next week.
11 terminal sessions: from this week

**Lecturers**
Simon Thompson  
s.j.thompson@kent.ac.uk
Dominic Orchard  
d.a.orchard@kent.ac.uk

**Resources**
Moodle for slides, lecture recordings, programs, class and seminar resources
What will I learn?
What will I learn?

**Functional ideas**
Values, names, evaluation, structured types, lists, higher-order functions, recursion, PBT.
What will I learn?

**Functional ideas**
Values, names, evaluation, structured types, lists, higher-order functions, recursion, PBT.

**Concurrent ideas**
Processes and messages, process ids and spawn, asynchrony and mailboxes, fail-safe and exits, …
What will I learn?

**Functional ideas**
Values, names, evaluation, structured types, lists, higher-order functions, recursion, PBT.

**Concurrent ideas**
Processes and messages, process ids and spawn, asynchrony and mailboxes, fail-safe and exits, …

**Put it into practice**
Using these ideas in practice in the Erlang programming language.
What will I learn?

**Functional ideas**
Values, names, evaluation, structured types, lists, higher-order functions, recursion, PBT.

**Concurrent ideas**
Processes and messages, process ids and spawn, asynchrony and mailboxes, fail-safe and exits, …

**Put it into practice**
Using these ideas in practice in the Erlang programming language.

**Context**
Haskell for “lazy”, typed programming. Other concurrency models; scaling up Erlang systems.
How will I learn?
How will I be assessed?

Four assessments: two functional, two concurrent.
How will I be assessed?

Four assessments: two functional, two concurrent.

Four assessments: two in-class, two take-home.
Programming means making this …
Intel Xeon Phi 'Knight's Corner': 1+ teraFLOPS with double precision; 62 x86 processor cores and a 512 bit GDDR5 memory controller
... do this
A list of programming languages fetched from Wikipedia in CSV format – [https://github.com/jamhall/programming-languages-csv](https://github.com/jamhall/programming-languages-csv)
Hardware

Low-level but …

… fiendishly complicated.
The Turing machine

http://www.math.uri.edu/~kulenm/mth381pr/comput/fig.jpg
The von Neumann model
The von Neumann model

Computation is …

… performed in a sequence of steps

… through changing values stored in memory
Scaling up the von Neumann model

- object-oriented
- object-based
- structured
- procedural
- assembler
- machine code
Scaling up the von Neumann model
Levels of abstraction

Instructions to store and fetch values from memory …

... and to perform arithmetical operations on these values and registers.

Linear sequence of code, with (un)conditional jumps.
Levels of abstraction

Instructions to store and fetch values from memory …

… and to perform arithmetical operations on these values and registers.

Linear sequence of code, with (un)conditional jumps.

Plus symbolic names, library routines.
Levels of abstraction

Symbolic variables, simple control structures …

… reuse of sections of code, using global variables.
Levels of abstraction

Symbolic variables, simple control structures …

… reuse of sections of code, w/ parameters, return values, scopes.

Defined composite data types.
Levels of abstraction

Group together variables with the procedures and functions that operate on them.
Levels of abstraction

Group together variables with the procedures and functions that operate on them.

Inheritance allows objects to extend / modify other objects.
Examples

- object-oriented
- object-based
- structured
- procedural
- assembler
- machine code
Examples

- object-oriented
- object-based
- structured
- procedural
- assembler
- machine code

010010...
Examples

object-oriented
object-based
structured
procedural
assembler
machine code

SPARC, 86x, ...
010010...
Examples

- object-oriented
- object-based
- structured
- procedural
- assembler
- machine code

Examples:

- BASIC
- SPARC, 86x, ...
- 010010...
Examples

- Pascal, Modula 2
- BASIC
- SPARC, 86x, ...
- 010010...

- object-oriented
- object-based
- structured
- procedural
- assembler
- machine code
Examples

- JavaScript, Haxe
- Pascal, Modula 2
- BASIC
- SPARC, 86x, ...
- 010010...
Examples

Java, C#, C++
JavaScript, Haxe
Pascal, Modula 2
BASIC
SPARC, 86x, ...
010010...

object-oriented
object-based
structured
procedural
assembler
machine code
The von Neumann model

By Kapooht - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=25789639
Calculation

\[ 4x + 3y \quad \text{when} \quad \begin{align*} x &= 2 \\ y &= 6 \end{align*} \]

\[ = 4(2) + 3(6) \]

\[ = 8 + 18 \]

\[ = 26 \]

http://www.wikihow.com/Evaluate-an-Algebraic-Expression
Calculation

Computation = calculation.

Programming = defining your own functions

\[ f(x, y) = 4x + 3y \]

? f 2 6
---> 4*2 + 3*6
---> 8 + 18
---> 26
Calculation

No variables / storage locations.

Just values of expressions …
… and names for values.

\[ f(x, y) = 4x + 3y \]

? f 2 6
---\[ 4*2 + 3*6 \]
---\[ 8 + 18 \]
---\[ 26 \]
Abstraction hiding the VNM

No variables / storage locations.

Just values of expressions …

… and names for values.
Abstraction hiding the VNM

No variables / storage locations.

Just values of expressions …
… and names for values.
Abstraction hiding the VNM

No variables / storage locations.

Just values of expressions …
… and names for values.
## The variety of functional languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Haskell</strong></td>
<td>Leading “lazy” language: calculate on demand. Strongly typed. Side effects only via “monads”</td>
</tr>
<tr>
<td><strong>LISP</strong></td>
<td>Earliest language with funs. Symbolic computing: eval. Weak types. AI applications</td>
</tr>
<tr>
<td><strong>OCaml</strong></td>
<td>Strongly typed but “strict”. Side effects but controlled. Systems programming e.g. Xen. Basis of F# (in .NET fwd)</td>
</tr>
<tr>
<td><strong>Erlang</strong></td>
<td>Weakly typed, strict. Concurrency at heart. Side effects controlled. Use? telecoms, WhatsApp, …</td>
</tr>
</tbody>
</table>
The variety of functional languages

**Haskell**
Leading “lazy” language: calculate on demand.
Strongly typed.
Side effects only via “monads”

**LISP**
Earliest language with funs.
Symbolic computing: eval.
Weak types.
AI applications

**OCaml**
Strongly typed but “strict”.
Side effects but controlled.
Systems programming e.g Xen
Basis of F# (in .NET fwd)

**Erlang**
Weakly typed, strict.
Concurrency at heart.
Side effects controlled.
Use? telecoms, WhatsApp, …
Interpreter

source -> program -> results
Compiler

source program → target program
Compiler

source program \rightarrow \text{front end} \rightarrow \text{IR} \rightarrow \text{back end} \rightarrow \text{target program}
Compiler

source program → front end → optimiser → back end → target program
Compiler

Uncover the structure in the text

Lexical analysis: find the words.
Parsing: find the structure.
Analysis: find aspects of the meaning.
Generate an intermediate representation.
Compiler

IR: e.g. SSA format
Optimisation = Analysis + Transformation
Optimiser will combine many small optimisations e.g ...
... data flow: float out calculations from loops.

Uncover the structure in the text
Lexical analysis: find the words.
Parsing: find the structure.
Analysis: find aspects of the meaning.
Generate an intermediate representation.
Uncover the structure in the text
Lexical analysis: find the words.
Parsing: find the structure.
Analysis: find aspects of the meaning.
Generate an intermediate representation.

Generate instructions for the target machine.
Allocating registers to symbolic variables.
Scheduling instructions.

Compiler

source program -> front end
IR -> optimiser
IR -> back end
target program

IR: e.g. SSA format
Optimisation = Analysis + Transformation
Optimiser will combine many small optimisations e.g …
… data flow: float out calculations from loops.
Concurrency
Concurrency
Concurrency

Threads = multiple modifiers + one memory
Concurrency

Threads = multiple modifiers + one memory

Processes = multiple modifiers + their own memory
Fight over scarce resources

Message-passing concurrency

Each process has its own memory …

… garbage collected separately, too.

Communication is via message passing …

… no shared memory at all.

Concurrency is a design artefact …

… but maps onto hardware parallelism.
Don’t forget …

Lecture tomorrow 10:00 in Woolf Lecture Theatre …

… and terminal sessions start this week.