Languages and Compilers
Concurrency

Adam Sampson
School of Design and Informatics
Abertay University

Today's plan

- This is a **languages** lecture

- Concurrency as an idea
- A brief taster of a concurrent language
- Semantic analysis of concurrent languages
- Runtime support for concurrency

Concurrency

- A model of programming
  - There are concurrent programming languages, just as there are OO programming languages...
  - … and many languages have concurrent features
- Concurrency means composing your program out of **processes**, each of which has its own control flow, and which all execute in parallel
- Processes can **interact** in defined ways – e.g. by sending and receiving messages, or working on shared memory together

Concurrency and parallelism

- Why write a concurrent program?
- Because of **concurrency**: if you're modelling a system which has lots of activities going on at once, you can model each as a process
  - e.g. network servers, simulated worlds
  - that is, because it makes the program **simpler**
- Because of **parallelism**: modern computers have multiple CPU cores, so you need lots of parallel activities to use them efficiently
  - e.g. games, media processing, HPC
  - that is, because it makes the program **faster**

A concurrent programming language

- **occam** (Inmos, 1983)
- Originally intended for transputers: some of the first cheap, fast multiprocessor CPUs
- I'm using this because it's pretty simple and I've written compilers/runtimes for it; I'll link to some similar, more modern languages at the end!

occam: the normal stuff

```
PROC main ()
  INT x:
  SEQ
    x := 42
    trace.sin (“x is: ”, x)
:
```

- Indentation-based
- SEQ introduces a block
- Declarations end with :
- Names can contain . (no special meaning)

occam: the normal stuff

```
PROC increment (INT var)
  var := var + 1
:

PROC main ()
  INT x:
  SEQ
    x := 42
    increment (x)
    trace.sin (“x is: ”, x)
:
```

- PROC defines a procedure (also has FUNCTION)
- Formal arguments are by-reference normally

occam: the normal stuff

```
PROC increment (INT var)
  var := var + 1
:

PROC main ()
  INT x:
  SEQ
    x := 42
    SEQ
      increment (x)
      increment (x)
    trace.sin (“x is: ”, x)
:
```

- You can nest blocks however you like
  - (You can also have nested PROCs, which is nice)

SEQ and PAR

- Why does SEQ introduce a block?
  - Because it means “do the things inside the block SEQuentially”
- You can also have a block that starts with PAR
  - … which means “do the things inside the block in PARallel”

```
INT here, there:
SEQ
  PAR
    here := fetch.temperature (“dundee”)
    there := fetch.temperature (“perth”)
  trace.sisn (“Dundee is “, here – there,
              “ degrees warmer than Perth”)
```

SEQ and PAR

- Both SEQ and PAR can have a “replicator” attached to them – if you say:

```
SEQ i = 0 FOR 3
  do.something (i)
```

```
PAR i = 0 FOR 3
  do.something (i)
```

- … that's **semantically equivalent** to:

```
SEQ
  do.something (0)
  do.something (1)
  do.something (2)
```

```
PAR
  do.something (0)
  do.something (1)
  do.something (2)
```

- i.e. you could write out the iterations of the loop by hand, and it'd have the same effect

SEQ and PAR

- So this is quite convenient for parallelising existing bits of code – you can just turn SEQ into PAR and have the same behaviour, just with things now running in parallel
  - Some variants of other languages have similar ideas – e.g. OpenMP for C/Fortran lets you annotate “parallel for” loops

```
[480][640]PIXEL image:
PAR y = 0 FOR 480
  PAR x = 0 FOR 640
    image[y][x] := render.pixel (x, y)
```

Synchronisation

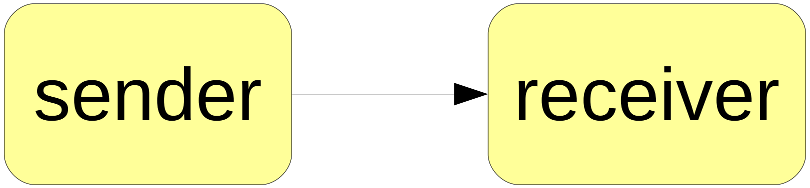
- How do you get processes to interact?
- occam's primitive is the **channel** –
  a data pipe which connects two processes
- One process sends (!), the other receives (?)

Channels

```
PROC sender (CHAN INT c!)
  WHILE TRUE
    c ! 42
:
PROC receiver (CHAN INT c?)
  WHILE TRUE
    INT x:
    SEQ
      c ? x
      trace.sin (“received: “, x)
:
PROC main ()
  CHAN INT c:
  PAR
    sender (c!)
    receiver (c?)
:
```

Channels

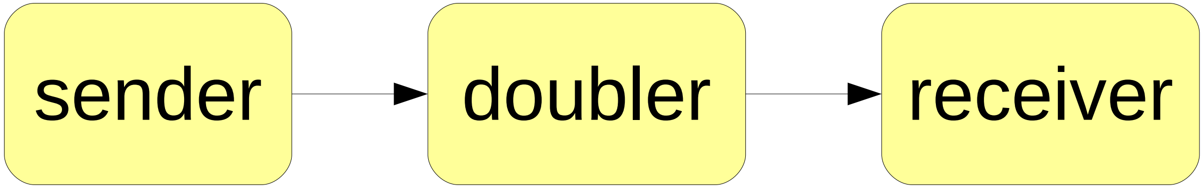
```
PROC sender (CHAN INT c!)
  WHILE TRUE
    c ! 42
:
PROC receiver (CHAN INT c?)
  WHILE TRUE
    INT x:
    SEQ
      c ? x
      trace.sin (“received: “, x)
:
PROC main ()
  CHAN INT c:
  PAR
    sender (c!)
    receiver (c?)
:
```

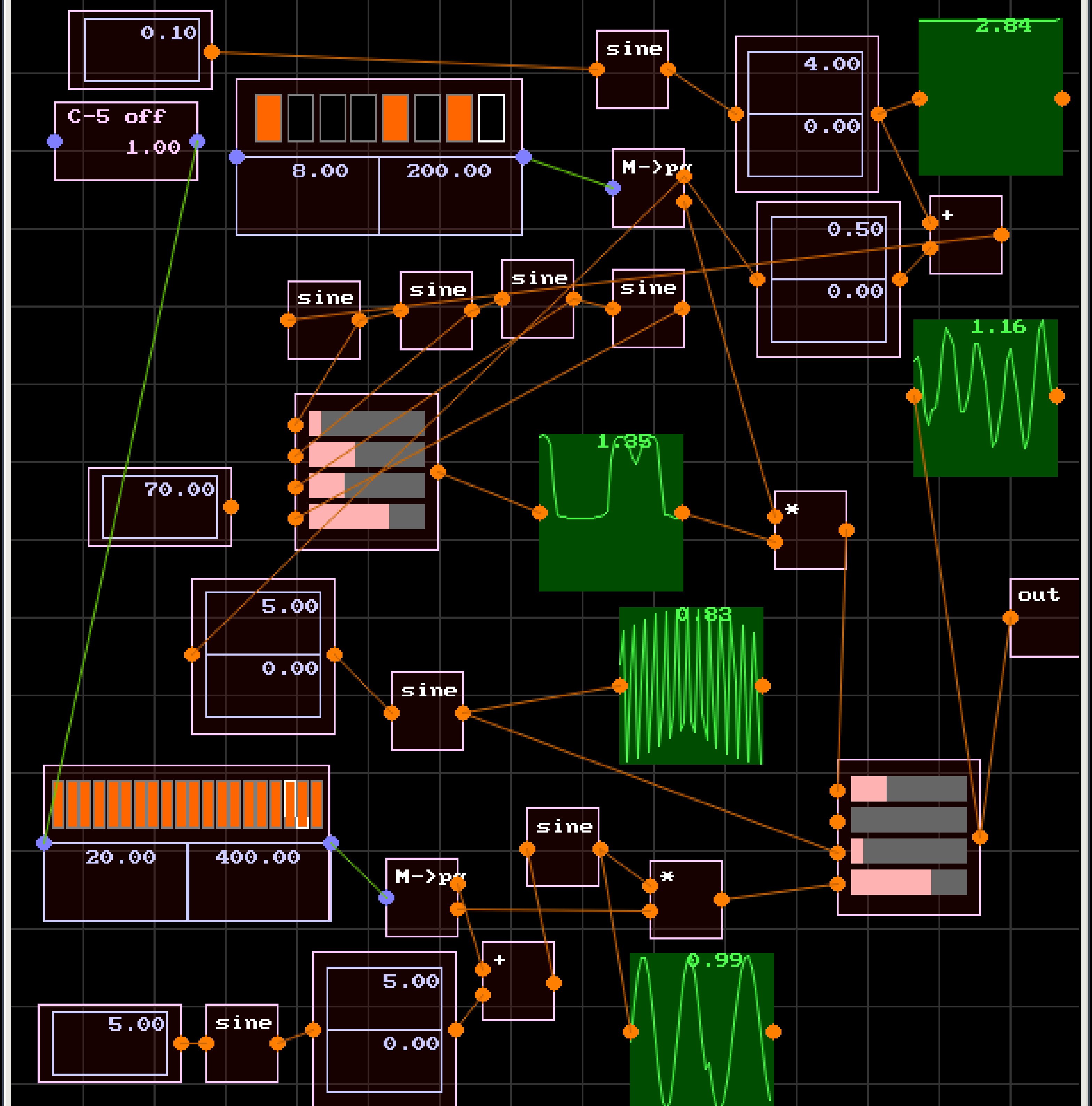


More complex networks

```
PROC doubler (CHAN INT in?, out!)
  WHILE TRUE
    INT x:
    SEQ
      in ? x
      out ! x * 2
:

PROC main ()
  CHAN INT a, b:
  PAR
    sender (a!)
    doubler (a?, b!)
    receiver (b?)
:
```





0.10
sine
4.00
0.00
2.84
C-5 off
1.00
8.00
200.00
M->pq
0.50
0.00
+
sine
sine
sine
sine
1.16
70.00
1.35
*
5.00
0.00
0.83
out
sine
20.00
400.00
M->pq
sine
*
+
5.00
0.00
0.99
5.00
sine

Process-oriented programming

- This model, with processes and channels, is called **process-oriented programming**
- Each process is isolated – it has its own private variables (in much the way objects do in OO)
- Each process has its own flow of control
- Processes only communicate using messages sent over channels (or other mechanisms)

- This has some nice properties: you can reason about and test processes in isolation, and processes compose in predictable ways

Where did this come from?

- occam is based on a **process calculus**: Communicating Sequential Processes (CSP)
- This is a mathematical notation for describing networks of processes interacting via channels
- The “calculus” part: there are laws for reasoning about semantics, rules for equivalence and rewriting (think rearranging equations)...
- … which gives lots of power to programmers and compiler writers for reasoning about the behaviour of programs
- Tools allow mathematical proofs about CSP

Why reason about programs?

- Because it's quite easy to write **incorrect** concurrent programs (in most languages!)

```
PROC increment (INT var)
  var := var + 1
:

PROC main ()
  INT x:
  SEQ
    x := 42
    SEQ
      increment (x)
      increment (x)
    trace.sin (“x is: ”, x)
:
```

Why reason about programs?

- Because it's quite easy to write **incorrect** concurrent programs (in most languages!)

```
PROC increment (INT var)
  var := var + 1
:

PROC main ()
  INT x:
  SEQ
    x := 42
    PAR
      increment (x)
      increment (x)
    trace.sin (“x is: ”, x)
:
```

What does this program print?

Why reason about programs?

- Because it's quite easy to write **incorrect** concurrent programs (in most languages!)

```
PROC increment (INT var)
  var := var + 1
:

PROC main ()
  INT x:
  SEQ
    x := 42
    PAR
      increment (x)
      increment (x)
    trace.sin (“x is: ”, x)
:
```

What does this program **mean** now?
(i.e. what are its semantics?)

Concurrency problems

- Does the program print 43? 44? 283?
- This program has a **race condition**:
we have two processes, running in parallel, trying to update the same shared variable
  - Most programming languages say you aren't allowed to do that – the program will compile, but it may produce a strange result or crash at runtime
  - … or it may silently work if the processes happen to run in the right order
- There are several classes of problems like this regarding how processes safely interact

Concurrent semantic analysis

- If I compile my example program with an occam compiler, it actually says:

  ```
  Error-occ21-cmp409.occ(13)- variable 'x'
  is assigned to in parallel
  ```

- … i.e. the occam compiler detects the error as part of its semantic analysis of the program
- It does this by tracking **ownership** of the variables in the program
  - Each variable must either be owned by a single process, or be read-only

Concurrent semantic analysis

- Suppose I write:

```
[10]INT array:
PAR
  PAR i = 0 FOR 10
    array[i] := i
  array[4] := 42
```

```
Error-occ21-
cmp409b.occ(3)-
variable 'array' is
assigned to in parallel
```

- This has the same error – array[4] is written by two processes – but the compiler has to expand the replicator to spot this
- This is very inefficient if it's FOR 10000000!
- … or impossible if it's FOR n

Concurrent semantic analysis

- A better way to handle this kind of analysis is to make it into a **constraint satisfaction problem**
  - (… which also has the acronym CSP – there are lots of these in computer science)
- Convert the code into a set of equations – $0 < i < 10$ and $i = 4$ in this case – and use a CSP solver to see if there are any solutions for i
  - If there are, then more than one process is using the same resource
- Equivalent techniques can be used to detect many concurrency problems
  - … but this makes semantic analysis complex!

Runtime support

- Once we've compiled a concurrent program (and hopefully checked it's safe), we also need some help from the runtime system to execute it

- The simple way: run each process as a **thread**
  - Multiple parallel flows of control within a single operating system process, sharing memory
  - Each thread has its own stack and registers
  - Downside: you usually can't have very many threads – inefficient and there's often an upper limit
  - A real-world complex concurrent program may have hundreds of thousands of processes running...

Runtime support

- The better way: **lightweight threads**
  - Simulate threads in userspace
  - Allocate stack frames for each process (often not contiguous, to avoid wasting space)
  - Use a limited number of OS threads to execute processes as they become ready to run
  - Switch between processes cooperatively: e.g. when a process does ! or ?
  - Allows much greater numbers of processes (millions on a typical machine)
- Most languages with concurrent facilities use this approach – more flexible than plain OS threads

Transactions

- Channels are only one possible communication mechanism – mailboxes, barriers...
- **Transactional memory** is another popular approach – use shared memory, but control access to it using database-like transactions
  - Begin, commit, roll back
- Offers good scalability – but difficult to implement efficiently – CPUs are starting to gain support for transactional operations
  - e.g. Intel TSX – although they got it wrong first time!

Concurrent languages to check out

- **Go** – distant cousin of occam – ex-Bell-Labs team now at Google – very similar to what we've seen
- **Rust** – also CSP-inspired, but with a powerful encoding of ownership in its type system
- **Erlang** – Actor model rather than CSP – originally designed for high-reliability telephony applications and has some neat process monitoring features
- **Haskell** – pure functional language but also has excellent lightweight concurrency features built in, with safety enabled by its type system
- Any questions?