MANCHESTER 1824

The University of Manchester

Chip Multiprocessors COMP35112

Lecture 15 - Functional Programming Languages and Dataflow Principles

Graham Riley

The Final Lecture

- The difficulties (related to shared updateable state) associated with using "normal" programming languages have led some to reconsider earlier attempts to exploit parallelism
- This lecture talks about two of these attempts which are related to one another:
  - Functional Programming and Dataflow computing
- Still an active area of research: (see later)
  - The recent EU TERAFLUX project (Ian Watson, Chris Kirkham, Mikel Lujan in this School + Siena + Cyprus + Barcelona + …) undertook to add Transactional Memory to Dataflow => extensible multicore architecture

Functional Programming

- Many languages
  - Lisp, ML (SML, CAML, ….), Haskell
  - Java and Python now support “lambda” expressions
  - Based on lambda calculus, Alonzo Church, Princeton, 1930s(!)
- Based on the idea that programming need not be imperative; instead it can be about defining functions, and then evaluating these with some arguments
- Functional programming is a subset of **declarative programming**, which also includes logic programming languages, such as Prolog

Referential Transparency

- A key property of functional languages is **referential transparency** – the property that:
  - names mean the same thing wherever they occur
  - you can freely replace a name with its value without changing the meaning of the program
  - i.e. functions have no **side effects**
- So updateable state (e.g. variables) is out!
  - From the programmer's perspective
  - Compiler and run-time can optimize
  - No variables makes programs easier to reason about and functional languages expose much concurrency…

Example: The Fibonacci Series

```
let fun fib 0 = 1
      | fib 1 = 1
      | fib n = fib (n - 1) + fib (n – 2)
in fib 5
```

This is a program – and its execution consists of **rewriting**:

fib 5 => fib 4 + fib 3 => fib 4 + fib 2 + fib 1 =>

fib 4 + fib 1 + fib 0 + fib 1 => fib 4 + 1 + 1 + 1 => ….

Absence of variables means that the order of rewriting is not important – can even be done in parallel!

Church-Rosser Property (1936)

- This property, that the order of evaluation does not affect the result, is called the Church-Rosser property (after mathematicians Alonzo Church and J. Barkley Rosser)

Other Features

- Can still have sophisticated data structures – but they are (if **purely functional**) not updateable
  - e.g. can have a search tree – but adding a new item produces a new tree, rather than modifying the original
    - As far as the programmer is concerned
- Pattern matching of arguments (e.g. fib)
- Some functional programming languages derive type information for you!
- Some allow "infinite constructs" and use lazy evaluation to avoid building more than needed to evaluate (simple e.g. fib, fib n only evaluates up to n)

What are FPLs not so good at?

- I/O: OK if input is e.g. a String, and output is a String, but it is difficult to fit interactive behaviour into that form (but there are solutions, e.g. infinite streams)
- Updateable state sometimes is useful, either to avoid repeated evaluation (i.e. an optimisation – FPL runtime system can do this behind the scenes) or because it seems a natural way to do things
- Many FPLs "cheat" – they allow impure features
  - E,g, Haskell – solves interactive problem this way
  - With care, can write functional code in C, Java, Python

Functional Programming: the Advert

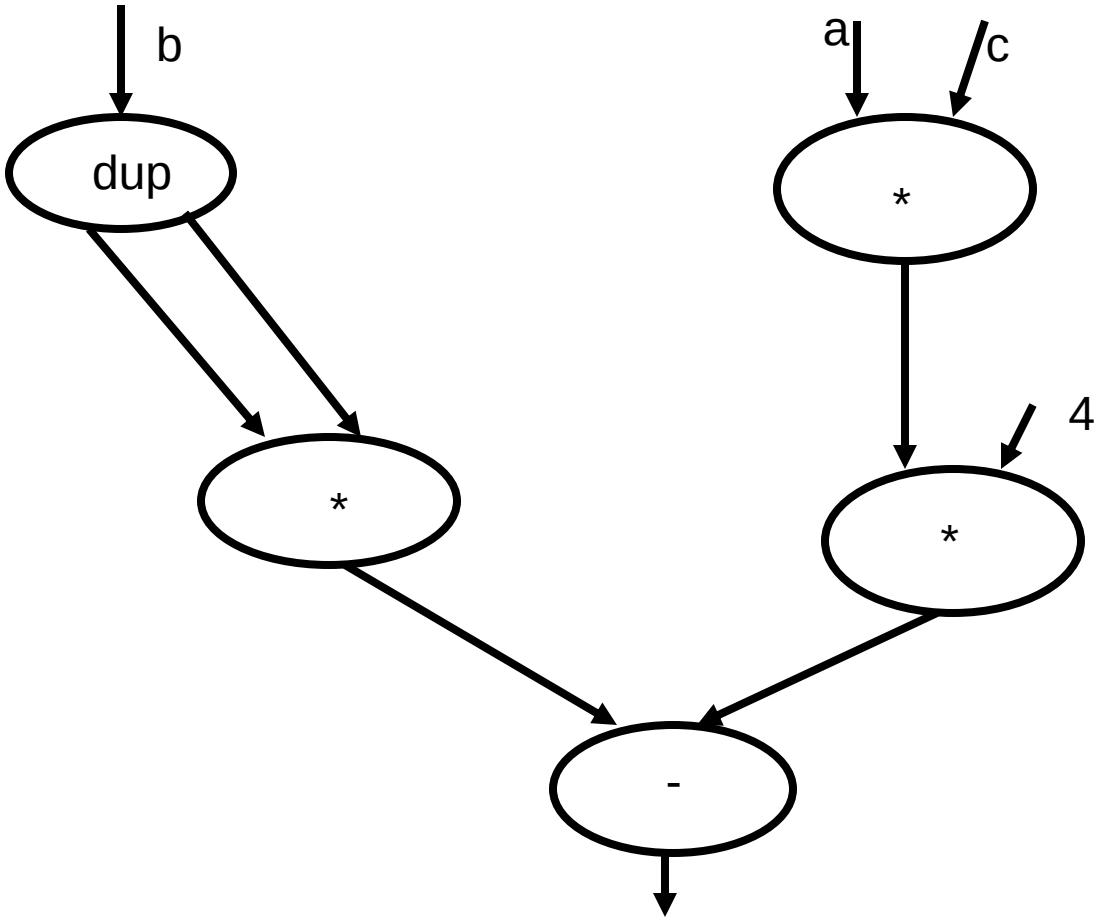
- Allows programmers to concentrate on the computation required at a higher level than imperative programming
- Communicates this to the compiler without superfluous constraints which would impede parallelisation
- Allows compiler/run-time system to make best use of the resources available to evaluate the program
  - A difficult problem…

- Functional languages can be compiled to dataflow graphs which led to research into dataflow machines…

Dataflow

- A very simple principle of execution
  - Data values *flow* to the instructions which need them (in *tokens*) – *Not* Von Neumann (IF, ID, E, MA, WB)
  - When an instruction has all the operands it needs, it can execute (on any processor) and eventually generate a result – another token
    - Values are generated and consumed by instructions
  - Program is a graph in which:
    - Nodes are instructions
    - Arcs are paths along which data tokens flow from node to node
  - Overall inputs are values flowing in to the graph
  - Overall results are values flowing out of the graph

Example: Computing the Discriminant ($b^2 - 4ac$)

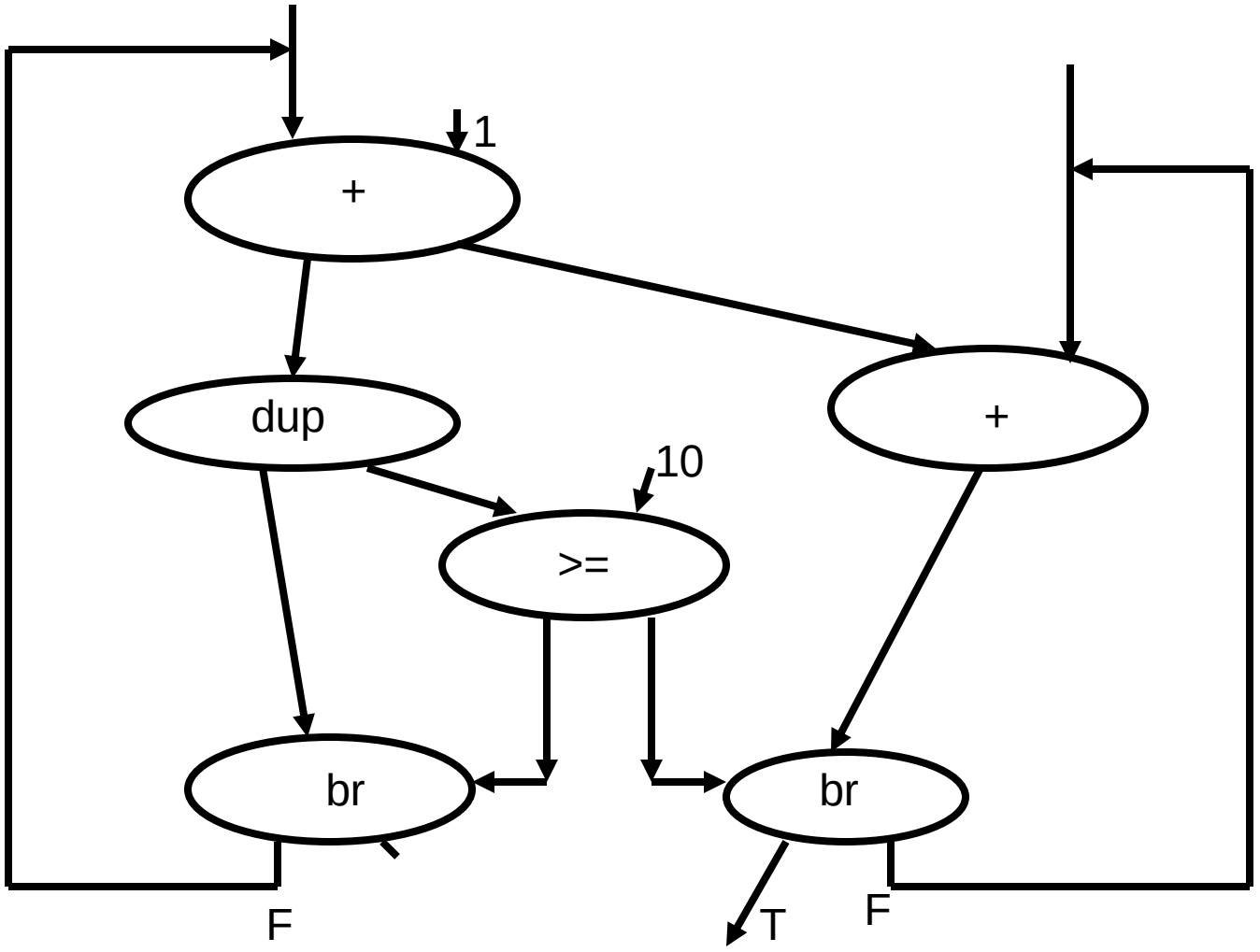


Loops

- Straight-through programs are not very interesting ….
- Can create loops by having backward arcs
- Need control nodes (gates, switches, or **branches**) with boolean inputs that determine direction of flow

Simple Loop Example

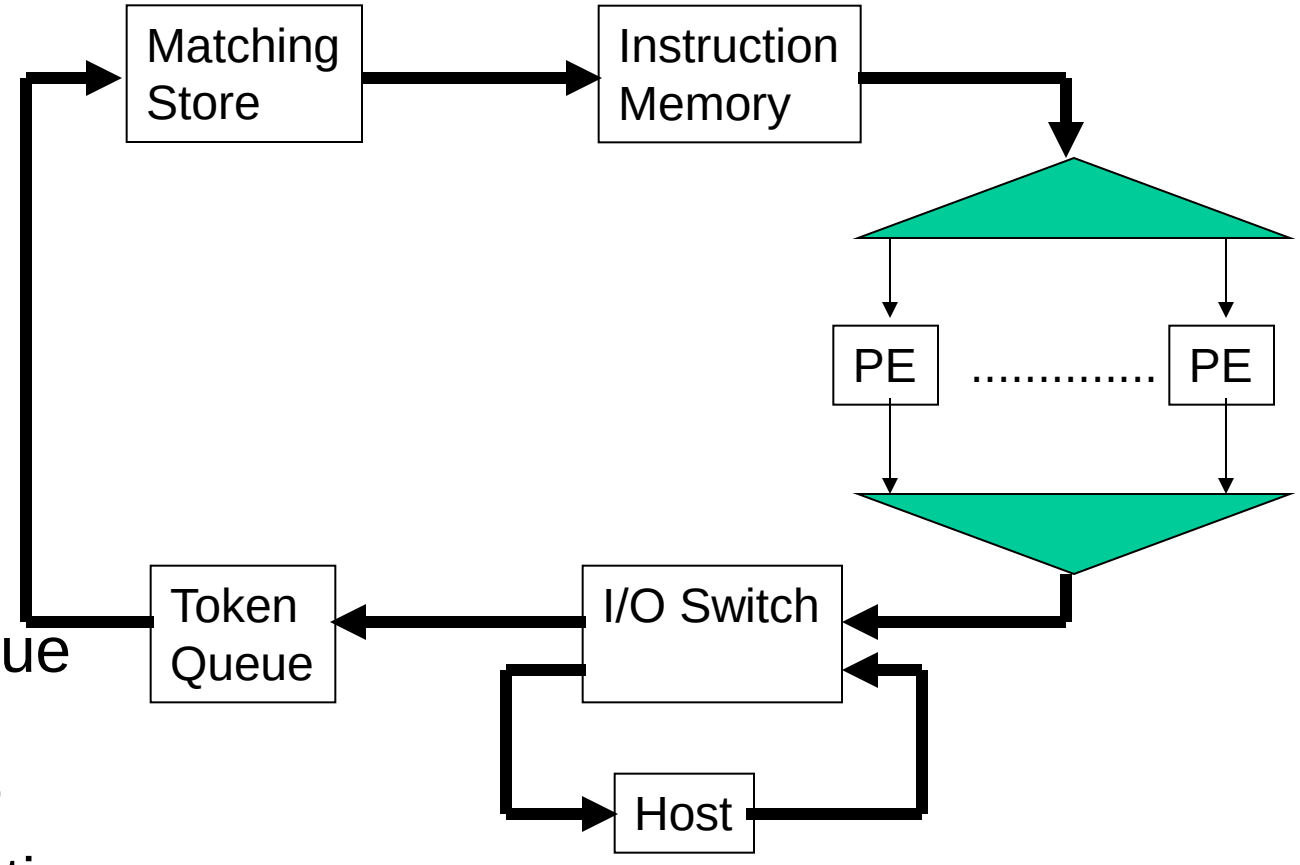
Two inputs, e.g. (3,12); one output (61 – Check!)



Functions

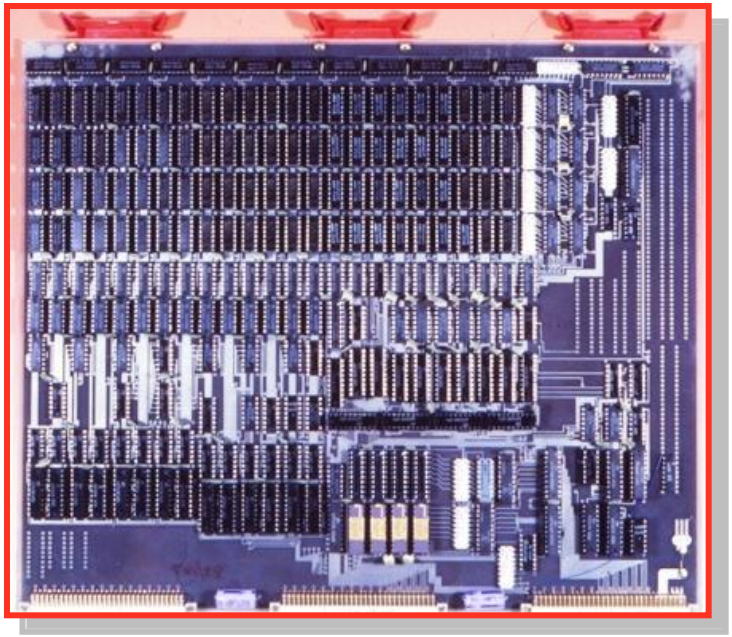
- Handling multiple calls to a function?
  - Function graphs avoid confusion from multiple calls by **colouring** tokens at call
- Can also use token colouring to distinguish different elements of an array (for example) flowing along a single arc
- Make all graphs self-cleaning, i.e. when function evaluation completes, no tokens remain in it
  - e.g. token after left branch is T in above

Dataflow Processor Ring Architecture



- Token Input
- Insert in Queue
- Extract from Queue
- Try to Match
- Wait if no partner
- Else fetch instruction
- Allocate to free PE

Manchester Dataflow machine (1980s)



Matching store

Components

- Instruction Store: holds coding of the dataflow graph (equivalent to object code/machine code)
- Processor Bank: a number of processors – all holding NO state from one execution to next!
- Token Queue: buffering (and the place to insert any input data)
- Matching Store: where tokens meet up with (same coloured) tokens going to the same instruction (matching operation is effectively associative)

Programming

- Machine code level: very tedious (graphical or otherwise, e.g. assembly language)
- Higher level
  - Functional Programming Languages
  - Single Assignment Languages (a subset of FPLs)
    - "names" can be assigned to only once

SISAL

- Stream and Iteration in a Single Assignment Language
- Developed by LLNL, CSU, DEC and University of Manchester in 1980s
- To give “imperative” programmers a language which they could use easily
- Easy compilation to Dataflow – and all the other parallel machines (supercomputers) at LLNL and elsewhere

Example Loop in SISAL

```
Define Main
function Main(j, s : integer returns integer)
   for initial i := j;
                  tot := s;
    while (i < 10) repeat
        i := old i + 1 ;
        tot := old tot + i ;
   returns value of tot
   end for
end function
```

Reduction Operators in Loops

```
Define Main
function Main(j, s : integer returns integer)
   let tot := for initial i := j;
                  while (i < 10) repeat
                      i := old i + 1 ;
                  returns sum of i
                  end for
      in s + tot
   end let
end function
```

Python near equivalent… Not (quite) single assignment!

```
#!/usr/bin/python

import sys

def sumvalsOffsetSISAL(j,s):
  i = j
  tot = s
  while (i<10):
    i=i+1
    tot=tot+i
  return tot

x=int(sys.argv[1])
y=int(sys.argv[2])

print sumvalsOffsetSISAL(x,y)
```

Python: a recursive functional version…

```
#!/usr/bin/python

import sys

def sumvalsOffset(j,s):
  p = j+1
  q = s+p
  if not(p>=10):
    return sumvalsOffset(p,q)
  return q

x=int(sys.argv[1])
y=int(sys.argv[2])

print sumvalsOffset(x,y)
```

Why dataflow computers didn't (totally) succeed

- Moving tokens round a large network, and matching tokens and instructions together, turned out to be more costly than executing the instructions!
  - Can be thought of as not being able to exploit locality effectively
- Dataflow has been successful in signal processing and other areas
- Many techniques in modern Out-of-Order cores are based on dataflow research

Dataflow lives on today!

- Current resurgence of interest in "Coarse-grained" dataflow
  - DAG-based (Directed Acyclic Graph) or Task-based computing
  - Systems such as PLASMA, MAGMA in the Linear Algebra world
  - StarPU, OmpSS etc., OpenMP tasking
  - ** OpenStream (Antoniu Pop et al here in Manchester)
  - APT's EuroExa project – started October 2017 (FPGAs)
- A lively research area currently as we move to exascale computing with, potentially, billions of threads…

Wrap-up

- Exam – Do check yourselves too!
  - Date Friday 17th May, 14:00
  - 2 hours – answer ALL 3 questions (no options!)
  - NB: past paper from 2011 – COMP35111
- Revision session
  - Monday13th May 1400 – 1500+, IT401
  - Bring questions about attempted answers
- Back-up reading (list on course materials webpage)