Functional Programming

Christopher D. Clack

© 2020 Christopher D. Clack

FUNCTIONAL PROGRAMMING

Lecture 1

INTRODUCTION

© 2020 Christopher D. Clack

FUNCTIONAL PROGRAMMING OVERVIEW & INTRODUCTION

This lecture will start with an overview of the module, followed by an initial introduction to Functional Programming

CONTENTS

- Overview
- Introduction to Functional Programming

© 2020 Christopher D. Clack

FUNCTIONAL PROGRAMMING OVERVIEW & INTRODUCTION

OVERVIEW

© 2020 Christopher D. Clack

FUNCTIONAL PROGRAMMING OVERVIEW & INTRODUCTION

Module Objectives

This module explores the Functional Programming paradigm and the implementation technology for functional programming languages

It uses the functional language Miranda

It does not use Haskell (though we may discuss Haskell in passing)

OVERVIEW

Module objectives:

- Functional Programming paradigm and implementation technology
- Uses Miranda
- **Does not use Haskell**

© 2020 Christopher D. Clack

FUNCTIONAL PROGRAMMING OVERVIEW & INTRODUCTION

OVERVIEW

Student objectives:

- Basics of the lambda calculus and combinators, how they are used in programming and language implementation
- Main features of a lazy functional language
- Type checking, type-inference and the operation of the Hindley-Milner type system as implemented in Miranda
- Write, understand and analyse non-trivial high-quality functional programs in Miranda
- Computation and memory management issues affecting the sequential implementation of lazy functional languages
- Solve problems relating to all of the above, under examination conditions

Student Objectives

Your own objectives should be:

Understand the basics of the lambda calculus and combinators and how they are used in programming and in the implementation of functional languages

Understand the main features of a lazy functional language

Understand type checking, type-inference and the operation of the Hindley-Milner type system as implemented in Miranda

Write, understand and analyse non-trivial high-quality functional programs in Miranda

Understand the computation and memory management issues affecting the sequential implementation of lazy functional languages

Be able to solve problems relating to all of the above, under examination conditions

© 2020 Christopher D. Clack

FUNCTIONAL PROGRAMMING OVERVIEW & INTRODUCTION

Independent study:

Students are expected to improve their functional programming skills through independent study and practice

Students are expected to do extensive independent reading to augment the material covered in the lectures

OVERVIEW

Student objectives:

- Independent study:
  - Improve functional programming skills through independent study and practice
  - Extensive independent reading to augment the material covered in the lectures

© 2020 Christopher D. Clack

FUNCTIONAL PROGRAMMING OVERVIEW & INTRODUCTION

OVERVIEW

Schedule of lectures (Spring 2023):

- Mondays 13:05 – 13:55 MPEB 1.20
- Thursdays 14:05 – 14:55 Eng Front Exec Suite 1.03
- Fridays 12:05 – 12:55 Eng Front Exec Suite 1.03

No sessions during Reading Week. Each will comprise:

- A live lecture where you can ask questions at any time
- As time permits, a Q&A discussion at the end

Lecture slides are downloadable from Moodle. Lecturecast will be activated where possible and available via Moodle.

The module has 2 parts:

Part A: Lambda Calculus, Combinators and programming

Part B: advanced concepts and implementation

Here's the schedule of lectures for Spring 2023:

- Mondays 13:05 – 13:55
- Thursdays 14:05 – 14:55
- Fridays 12:05 – 12:55

There will be no sessions during Reading Week. Each session will comprise:

- A live lecture where you can ask questions at any time, and
- As time permits, a group Question & Answer session at the end

The slides for each lecture will be available for download from Moodle (with captions). Lecturecast will be activated where possible and available via Moodle (with some delay for editing).

The module has two parts: Part A covers Lambda Calculus, Combinators and Functional Programming (the whole of the first half of term, and then some of the second half of term), whereas Part B covers Language Implementation, including garbage collection, graph reduction and memory management (the remainder of the term)

© 2020 Christopher D. Clack

FUNCTIONAL PROGRAMMING OVERVIEW & INTRODUCTION

OVERVIEW

The **Moodle** page:

- essential resources
- including essential reading

There is no marked coursework

- formative exercises during and between lectures

The online Miranda book has simple self-study exercises and answers, and the Moodle page has more exercises

Assessment: 100% by exam

- writing Miranda code
- more than one “correct” answer
- **quality** of answers will be important

Past paper questions and feedback.

The Moodle page has lots of essential resources, including details of essential reading

There is no marked coursework, but students must complete formative exercises during and between lectures

The online Miranda book has simple self-study exercises and answers, and the Moodle page has more exercises

Assessment is 100% by exam – which will include the writing of Miranda code under exam conditions. Questions may have more than one “correct” answer, and the **quality** of answers will be an important criterion in marking.

Many past paper exam questions are available, without answers.

In the last week of term there will be a discussion of possible answers to exam questions.

© 2020 Christopher D. Clack

FUNCTIONAL PROGRAMMING OVERVIEW & INTRODUCTION

INTRODUCTION

to FUNCTIONAL PROGRAMMING

© 2020 Christopher D. Clack

FUNCTIONAL PROGRAMMING OVERVIEW & INTRODUCTION

INTRODUCTION *to* FUNCTIONAL PROGRAMMING

- A language whose syntax can be defined in just 20 characters, yet can compute anything that is computable
- The only two functions you'll ever need to compute anything that is computable
- Manipulating functions as data, and data as functions
- Programming with infinite data structures
- A completely different way of thinking about solving problems, and the technology that enables all of this

The story of Functional Programming is the story of a language whose syntax can be defined in just 20 characters, yet can compute anything that is computable

It's a story about the only two functions you'll ever need to compute anything that is computable

It's about manipulating functions as data, and data as functions

It's about programming with infinite data structures

It's about a completely different way of thinking about solving problems, and the technology that enables all of this

© 2020 Christopher D. Clack

FUNCTIONAL PROGRAMMING OVERVIEW & INTRODUCTION

Many people think of Alan Turing as "the father of computing", but there are many other important figures in computer science. In particular, Alonzo Church (Turing's PhD supervisor)

Here are some notable pioneers

INTRODUCTION *to* FUNCTIONAL PROGRAMMING
BEGINNINGS

Moses Schönfinkel (1889-1942) – combinatory logic

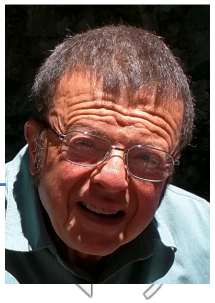
Haskell Curry (1900-1982) – combinatory logic

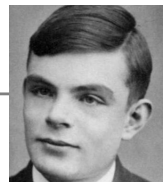
Alonzo Church (1903-1995) – the Lambda Calculus

- Stephen Kleene (1909-1994) – recursion theory

- John Rosser (1907-1989) - logician

- Dana Scott (1932-) – semantics
    - *(many PhD students)*
    - David Turner (1946-) – SASL, KRC, Miranda





- Alan Turing (1912-1954) – the Turing Machine

© 2020 Christopher D. Clack

FUNCTIONAL PROGRAMMING OVERVIEW & INTRODUCTION

INTRODUCTION *to* FUNCTIONAL PROGRAMMING
BEGINNINGS

Two different (and provably equivalent) approaches to programming:

Turing: using a small set of rules to control machines

Church: a general theory using **functions** (expressed as *rules* rather than as graphs) as a foundation for **logic** to express and solve problems in a precise way

These two approaches have led to two radically different styles of programming and programming languages

So what? Does it matter what language we use?

Yes. Because different languages **change the way we think**

Church and Turing promoted two different (and provably equivalent) approaches to programming:

Turing was interested in building machines and understanding how to control them using a small set of rules

Church was interested in developing a general theory using **functions** (expressed as *rules* rather than as graphs) as a foundation for **logic** and **mathematics** to express and solve problems in a precise way

These approaches have much in common, but they led to two radically different styles of programming and programming languages

If two languages are both ”Turing complete” why should we care what language we use?

They may differ in expressivity, abstraction, protection from common errors, etc.

But mostly we should care because different languages **change the way we think**

© 2020 Christopher D. Clack

FUNCTIONAL PROGRAMMING OVERVIEW & INTRODUCTION

INTRODUCTION *to* FUNCTIONAL PROGRAMMING
PROGRAMMING STYLES

What do we mean by different programming styles?

- The “imperative” languages (typically based on giving instructions to a machine): e.g. Java, C++, Fortran, C
- The ”declarative” languages (typically based on the more abstract concept of expressing and solving a problem): e.g. SASL, KRC, Miranda, Haskell, Prolog
  - Note: contains both “functional” languages and “logic” languages

On the next slide we’ll see examples of these two styles

What do we mean by different programming styles?

We can create ways to classify programming languages. One way is to identify two classes:

1. The “imperative” languages (typically based on giving instructions to a machine, as proposed by Turing): e.g. Java, C++, Fortran, C
2. The ”declarative” languages (typically based on the more abstract concept of expressing and solving a problem, as proposed by Church): e.g. SASL, KRC, Miranda, Haskell, Prolog

Note that the class of declarative languages traditionally contains both “functional” languages (based on Church’s Lambda Calculus) and “logic” languages (based on first-order logic – i.e. predicate logic)

On the next slide we’ll see simple examples of the “imperative” and the “functional” styles of programming

© 2020 Christopher D. Clack

FUNCTIONAL PROGRAMMING OVERVIEW & INTRODUCTION

INTRODUCTION *to* FUNCTIONAL PROGRAMMING
PROGRAMMING STYLE EXAMPLE

Variable "results" contains a sequence of 100 integers: write code to select all those with a value less than 10

Imperative style:

```
int  small[100];
int  j,k;
for  (j=0,k=0;j<100;j++)
        if(results[j]<10){small[k]=results[j];k++;}
return (small);
```

Functional style (1):

```
filter (< 10) results
```

Functional style (2):

```
filter (< 10) results
where     filter f []       = []
          filter f (x:xs) = x: (filter f xs), if (f x)
                          = filter f xs, otherwise
```

Here's a simple example

Assume "results" is the name of a variable that contains a sequence of 100 integers, and write code to select all those with a value less than 10

The first example here gives the "imperative" style. It doesn't matter exactly what imperative language is used here: notice instead how names are given to variables to store intermediate results, including counters for the loop

The second example gives the "functional" style, which is very much shorter

We can criticise the second example by saying that it is *hiding* code inside the definition of the function "filter", to which the answer would be (i) this is exactly the point, because pure functions make this easy (!), and (ii) the third example includes the full definition of filter and it is still shorter!

An industry comparison of imperative and functional programs concluded that for large projects the functional style is typically about five times shorter and five times faster to code

Notice how the functional style is concise, with low syntactic "clutter", and a reduced need to manage the storage of intermediate results

© 2020 Christopher D. Clack

FUNCTIONAL PROGRAMMING OVERVIEW & INTRODUCTION

INTRODUCTION *to* FUNCTIONAL PROGRAMMING
PROGRAMMING STYLE EXAMPLE

Thus for any single phase m and component i,

$$\zeta_{m,i} = \frac{\xi_i}{\sum_{r \in phases} s_r K_i^{r,m}} \tag{9.7}$$

```
> zeta m i k xi phases s =
>     xi i / sum [s r * k r m i | r <- phases]
```

We can now derive a partial derivative for ζ with respect to s. This will be useful later when we construct and then solve some residual functions.

$$\frac{\partial \zeta_{m,i}}{\partial s_n} = -\frac{\xi_i \sum_{r \in phases} \frac{\partial s_r}{\partial s_n} K_i^{r,m}}{\left(\sum_{r \in phases} s_r K_i^{r,m} \right)^2} \tag{9.8}$$

$$\frac{\partial \zeta_{m,i}}{\partial s_n} = -\zeta_{m,i} \frac{\sum_{r \in phases} \frac{\partial s_r}{\partial s_n} K_i^{r,m}}{\sum_{r \in phases} s_r K_i^{r,m}} \tag{9.9}$$

```
> dzeta_ds m i n k xi dep_phase phases s =
>     neg (zeta m i k xi phases s)
>     * sum [ds_ds r n dep_phase * k r m i | r <- phases]
>     / sum [s r * k r m i | r <- phases]
```

Some care must be taken when formulating the partial derivatives with respect to s, since not all of these variables are independent. From equation 9.1, it is clear that all but one of the s variables are indepenent, the last being one minus the sum of the rest. The choice of which variable is dependent is arbitrary, but care should be taken that one is consistent once the choice is made.

Because functional programming code is so concise it is ideally suited to the "literate" style of programming, where the documentation takes priority with pieces of code interspersed throughout the documentation

The language Miranda has specific support for this style, permitting Miranda code to be embedded in LaTeX documents, as shown in this example

For example, the mathematical definition for $\zeta_{m,i}$ is given by the function definition "zeta" that takes arguments m, i, k, xi, phases and s – these being the free variables in the mathematical definition

© 2020 Christopher D. Clack

FUNCTIONAL PROGRAMMING OVERVIEW & INTRODUCTION

INTRODUCTION *to* FUNCTIONAL PROGRAMMING
A PRACTICAL EXAMPLE

Functional programming languages are “elegant” – like Haiku? or like Karate?

A practical example: world’s largest IT consultancy, a large mission-critical project for in international bank

Prototyping: discrete-event simulation of an object-oriented design (a network of components communicating via streams of data). Especially to prototype multiple optimisation and approximation algorithms

Client wanted C++, but technically “not viable” – would take too long

Consultancy considered Smalltalk, but commercially “not viable” (didn’t want to insult the client)

A practical example

Functional Programming languages are renowned for their “elegance”

- But are they like Japanese Haiku poetry (elegant, but not very practical)?
- Or are they like Karate (elegant, and useful in a fight)?

Here we will discuss (at a cursory level) a practical example where UCL helped the world’s largest IT consultancy in a large mission-critical project (with over 100 developers) for an international settlement bank

The aim was to **prototype** a large object-oriented design. The prototype was required to be a discrete-event simulation of a network of components communicating via streams of data, with one or more system inputs and one system output

A key task was to prototype the central optimisation and approximation algorithms

The main constraint was that the client required the production code to use C++, which was “not viable” for rapidly prototyping a large number of algorithms – it would take too long to develop the underlying components!

Smalltalk was considered, but because it was known to the client it was rejected (the consultancy did not want to “insult” the client).

© 2020 Christopher D. Clack

FUNCTIONAL PROGRAMMING OVERVIEW & INTRODUCTION

INTRODUCTION *to* FUNCTIONAL PROGRAMMING
A PRACTICAL EXAMPLE

Alternative approach — use a functional language: Miranda

- Higher Order, Statically Typed, Lazy, with Garbage Collection, no pointers, no assignment
- Unknown to client (!)

Selling points:

- Speed and Clarity with which algorithms can be (i) expressed and (ii) validated
- Can simulate key object-oriented designs in detail
- With minimal detail for other components!
- Access to expertise: a “champion”

Note : Speed of execution was almost totally irrelevant!

[see Braine et al (1998) “Simulating an object-oriented financial system in a functional language” http://arxiv.org/abs/2011.11593]

Alternative approach — use a functional language (Miranda)

- Higher Order (where functions can be passed as arguments and results of functions), Statically Typed (giving compile-time error checking), Lazy (which optimises performance), with Garbage Collection (for automatic memory management), no pointers (which minimises programmer errors), and single assignment (within scope, a name always has the same value, minimising errors)
- For this project, it was entirely unknown to client (!)

Selling points:

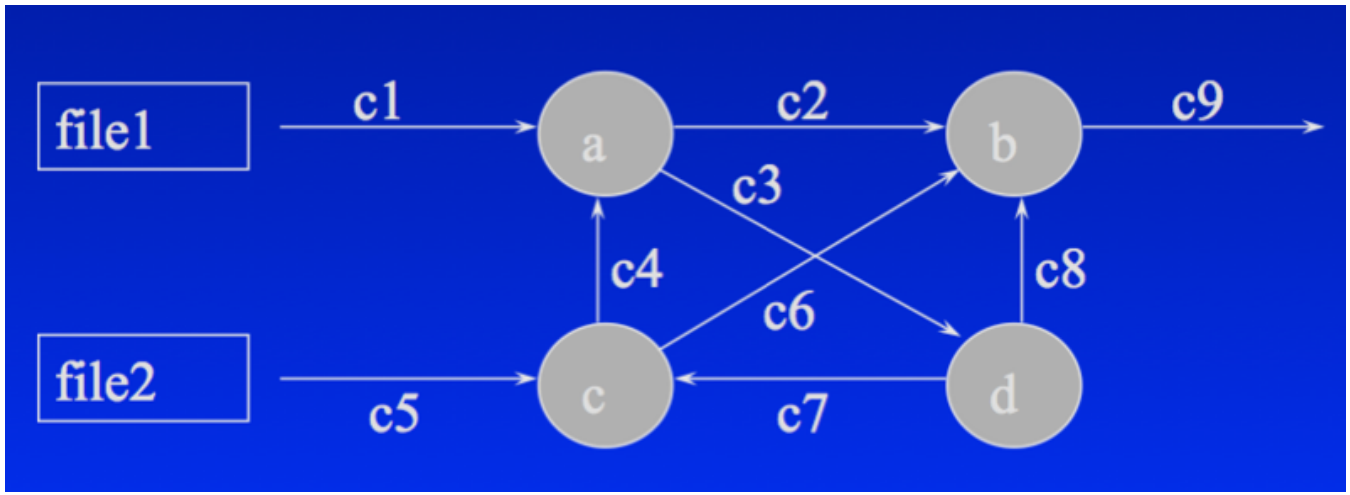
1. The speed and clarity with which algorithms could be (i) expressed and (ii) validated
2. The ability to simulate key object-oriented designs in detail
3. The ability to give minimal detail for other components
4. Access to expertise: a “champion” within the IT consultancy

Note: Speed of execution was almost totally irrelevant!

© 2020 Christopher D. Clack

FUNCTIONAL PROGRAMMING OVERVIEW & INTRODUCTION

INTRODUCTION *to* FUNCTIONAL PROGRAMMING
A PRACTICAL EXAMPLE



c1 = *read "file1"*
*c*5 = read "file2"
*c*9 = b (c2, c6, c8)
(*c*2, *c*3) = a (c1, c4)
(c4, c6) = c (c5, c7)
(c7, c8) = d (c3)

[for similar workflow modelling also see Kelly et al (2018) "Lambda Calculus as a workflow model"]

To demonstrate the FP approach, this diagram shows a simple model of a network of components (nodes) sending streams of data (edges) to each other

The nodes were implemented as recursive (looping) functions a, b, c and d, and the edges c1, c2 etc were implemented as potentially-infinite streams of (time, value) events

The interesting aspect of a FP approach to prototyping this network is how straightforward it is to model the topology of interaction between the nodes

For example, if functions a, b, c and d are defined elsewhere we can simply give definitions for the edges as shown in the Miranda code below the diagram

In the code:

c1 = read "file1"

- the edge c1 is the result of reading a stream of (time, value) data from "file1", and similarly for c5

c9 = b (c2, c6, c8)

- the edge c9 is the data output by function b when it is applied to inputs c2, c6 and c8

(c2, c3) = a (c1, c4)

- the two edges c2 and c3 are the two outputs from the function a when it is applied to inputs c1 and c4, and similar definitions are given for the edges c4, c6 and c7, c8

© 2020 Christopher D. Clack

FUNCTIONAL PROGRAMMING OVERVIEW & INTRODUCTION

INTRODUCTION *to* FUNCTIONAL PROGRAMMING
A PRACTICAL EXAMPLE

The Miranda code provided a simple, expression-based, behavioural specification that was executable

Synthetic data was generated by some components, otherwise real data could be used in the simulation

Miranda algorithms provided a specification for implementation in C++

- Rapid development – about 5x faster than C++
- Concise expression – 4-5 times shorter than C++
- Design optimised early in the software lifecycle, with confidence increased through validation on real data
- Almost NO errors in prototype code, very few errors in C++ code
- Viewed as a commercial advantage
- “champion” promoted!

The Miranda code provided a simple behavioural specification that was executable

The specification in Miranda was expression-based (NB re-ordering sub-expressions is easier than re-ordering commands)

Synthetic (statistical) data was generated by some components, otherwise real data could be used in the simulation

Miranda algorithms could be expressed in detail in a way that provided a specification for implementation in C++ (and C++ was semi-automatically generated from Miranda)

Results

- Rapid development – about 5x faster than C++
- Concise expression – 4 or 5 times shorter than C++
- Simulation and specification of complex processes – the design was optimised early in the software lifecycle, with confidence increased through validation on real data
- Almost NO errors in prototype code
- Vast reduction in errors in C++ code
- Viewed as a commercial advantage
- “champion” promoted!

© 2020 Christopher D. Clack

FUNCTIONAL PROGRAMMING OVERVIEW & INTRODUCTION

In summary, this lecture has provided an overview of the module, followed by an initial introduction to functional programming (including a practical example of using the language Miranda to prototype an object-oriented design)

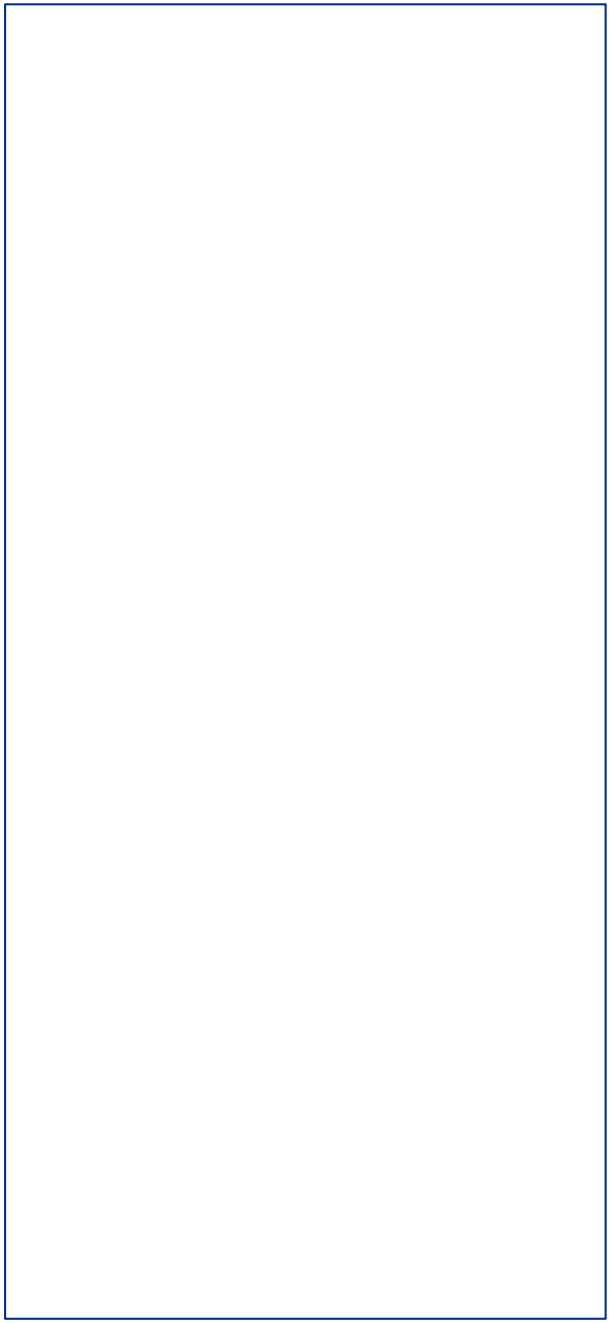
The next lecture will introduce the Lambda Calculus

SUMMARY

- Overview
- Introduction to Functional Programming

© 2020 Christopher D. Clack

FUNCTIONAL PROGRAMMING OVERVIEW & INTRODUCTION



© 2020 Christopher D. Clack