Introduction, history of FP and core concepts

Lecture 1

MPhil ACS & Part III course, Functional Programming: Implementation, Specification and Verification

Magnus Myreen
Michaelmas term, 2013

What is the course about?

Course title:

Functional Programming:
Implementation, Specification and Verification

What is the course about?

Course title:

Functional Programming:
Implementation, Specification and Verification

A more descriptive title:

Specification and Verification Applied to
Implementations of
Functional Programming Languages (Lisp and SML)

Aim

This course has two aims that will be addressed in parallel.

1. to teach formal specification and verification, and
2. to teach how functional languages are implemented.

Realisation:

> This course mostly teaches formal verification (1) by using running examples from FP implementation (2).

Exception: last three lectures concentrate on (2) and uses of FP.

Aim

This course has two aims that will be addressed in parallel.

1. to teach formal specification and verification, and
2. to teach how functional languages are implemented.

Realisation:

> This course mostly teaches formal verification (1) by using running examples from FP implementation (2).

Exception: last three lectures concentrate on (2) and uses of FP.

The course is based on recent research. Potential to get involved!

Prerequisites

This course is not about:

1. how to program using functional languages, nor
2. how to use a proof assistant.

(1) is a good to know, (2) is not at all necessary for this course.

It helps to have previous knowledge of:

- the lambda calculus
- the deductive system of classical logic (e.g. FOL)

Course organisation

Lectures:

16 lectures
4 guest lecturers

Location:

Room SW01, Computer Lab, JJ Thompson Avenue

Time:

9.05am, every Tuesday and Thursday, 10 Oct - 3 Dec

Assessment:

2 "tick exercises", this term (20 % of overall mark)
1 take-home test, beginning of next term (80 % of mark)

Tick deadlines: 28 Oct, 21 Nov, exercises will appear on website

Course material

What is examinable?

Everything that is lectured is examinable (unless explicitly stated otherwise).

Slides will be available on the course website.

http://www.cl.cam.ac.uk/teaching/1314/L26/

The website will also contain supplementary material that goes beyond the lectured (examinable) material.

People involved

Main lecturer:

Magnus Myreen

Faculty member:

Prof Mike Gordon

Guest lecturers:

Ramana Kumar – PhD on compiler verification
Scott Owens – expert on types and semantics
Jeremy Yallop – Ocaml expert, Ocaml Labs hacker
Anil Madhavapeddy – OS guru, Xen, Mirage etc.

Admins: Kate Cisek, Lise Gough

Feedback

This course is new.

Feedback will be appreciated.

Early feedback is most helpful for you and me.

Functional Programming:
What is it?

Functional Programming: What is it?

No single definite definition.

Functional Programming: What is it?

No single definite definition.

Functional languages strive to mimic mathematics:

> Every computation is a function (in the mathematical sense) of the inputs, i.e. does not interact with implicit state.

NB: few functional languages are strictly pure as above.

Functional Programming: What is it?

No single definite definition.

Functional languages strive to mimic mathematics:

> Every computation is a function (in the mathematical sense) of the inputs, i.e. does not interact with implicit state.

NB: few functional languages are strictly pure as above.

Nowadays "functional language" is often used to mean more:

- functions treated as first-class values
- loops written as recursion
- static typing is used
- data is (mostly) immutable, abstract and garbage collected

Referential Transparency

FP discourages use of side-effects.

Referential Transparency

FP discourages use of side-effects.

Gain: referential transparency and equational reasoning
"equals can be replaced by equals"

```
(... x + x ...) where x = f a
```

which is the same as:

```
(... f a + f a ...)
```

Referential Transparency

FP discourages use of side-effects.

Gain: referential transparency and equational reasoning
“equals can be replaced by equals”

```
(... x + x ...) where x = f a
```

which is the same as:

```
(... f a + f a ...)
```

Makes debugging & informal reasoning much simpler.

Referential Transparency

FP discourages use of side-effects.

Gain: referential transparency and equational reasoning
“equals can be replaced by equals”

```
(... x + x ...) where x = f a
```

which is the same as:

```
(... f a + f a ...)
```

Makes debugging & informal reasoning much simpler.

Impure languages support this only partially.

Brief history of FP

In 1930s, Alzono Church, Alan Turing, John von Neumann and Kurt Gödel lived in Princeton and thought about computation.

Brief history of FP

In 1930s, Alzono Church, Alan Turing, John von Neumann and Kurt Gödel lived in Princeton and thought about computation.

Church's (pure untyped) lambda calculus most relevant to FP.

$$t \;::=\; v \;\mid\; \lambda v.\,t \;\mid\; t\;t$$

Brief history of FP

In 1930s, Alzono Church, Alan Turing, John von Neumann and Kurt Gödel lived in Princeton and thought about computation.

Church's (pure untyped) lambda calculus most relevant to FP.

$$t \ ::= \ v \ \mid \ \lambda v.\, t \ \mid \ t\ t$$

function application

abstraction, unnamed function

variable

Brief history of FP

In 1930s, Alzono Church, Alan Turing, John von Neumann and Kurt Gödel lived in Princeton and thought about computation.

Church's (pure untyped) lambda calculus most relevant to FP.

$$t \;::=\; v \;\mid\; \lambda v.\, t \;\mid\; t\; t$$

function application

abstraction, unnamed function

variable

- a calculus about functions (thus computation)
- functions can be applied to themselves
- originally developed as a foundation for all of mathematics

Recursion in lambda calculus

The *Y* combinator:

$$Y \equiv \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

Recursion in lambda calculus

The *Y* combinator:

$$Y \equiv \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

is a fixed-point combinator:

$$\begin{aligned} & Y\ e \\ = \quad & (\lambda x.\ e\ (x\ x))\ (\lambda x.\ e\ (x\ x)) \\ = \quad & e\ ((\lambda x.\ e\ (x\ x))\ (\lambda x.\ e\ (x\ x))) \\ = \quad & e\ (Y\ e) \end{aligned}$$

Recursion in lambda calculus

The *Y* combinator:

$$Y \;\equiv\; \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

is a fixed-point combinator:

$$\begin{array}{rl} & Y\ e \\ = & (\lambda x.\ e\ (x\ x))\ (\lambda x.\ e\ (x\ x)) \\ = & e\ ((\lambda x.\ e\ (x\ x))\ (\lambda x.\ e\ (x\ x))) \\ = & e\ (Y\ e) \end{array}$$

Recursive functions can be defined non-recursively!

Recursion in lambda calculus

The *Y* combinator:

$$Y \;\equiv\; \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

is a fixed-point combinator:

$$\begin{array}{cl} & Y\ e \\ = & (\lambda x.\ e\ (x\ x))\ (\lambda x.\ e\ (x\ x)) \\ = & e\ ((\lambda x.\ e\ (x\ x))\ (\lambda x.\ e\ (x\ x))) \\ = & e\ (Y\ e) \end{array}$$

Recursive functions can be defined non-recursively!

$$fac \;\equiv\; Y\ (\lambda f.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \times f(n{-}1))$$

Recursion in lambda calculus

The *Y* combinator:

$$Y \;\equiv\; \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

is a fixed-point combinator:

$$\begin{aligned} & Y\ e \\ = \;& (\lambda x.\ e\ (x\ x))\ (\lambda x.\ e\ (x\ x)) \\ = \;& e\ ((\lambda x.\ e\ (x\ x))\ (\lambda x.\ e\ (x\ x))) \\ = \;& e\ (Y\ e) \end{aligned}$$

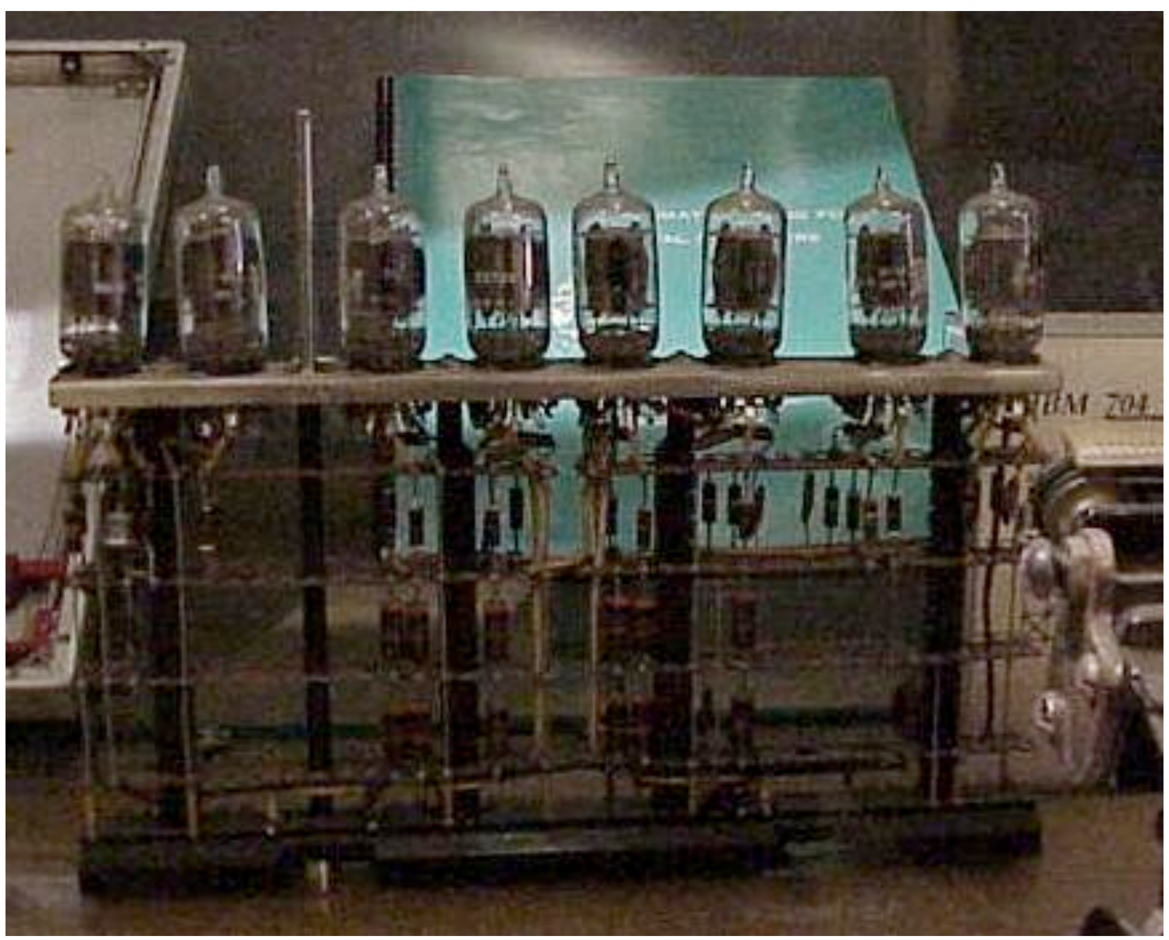
Recursive functions can be defined non-recursively!

$$fac \;\equiv\; Y\ (\lambda f.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \times f(n{-}1))$$

$$fac\ n \;=\; \text{if } n = 0 \text{ then } 1 \text{ else } n \times fac(n{-}1)$$

late 1950s, McCarthy: LISP

Computers started to be available.

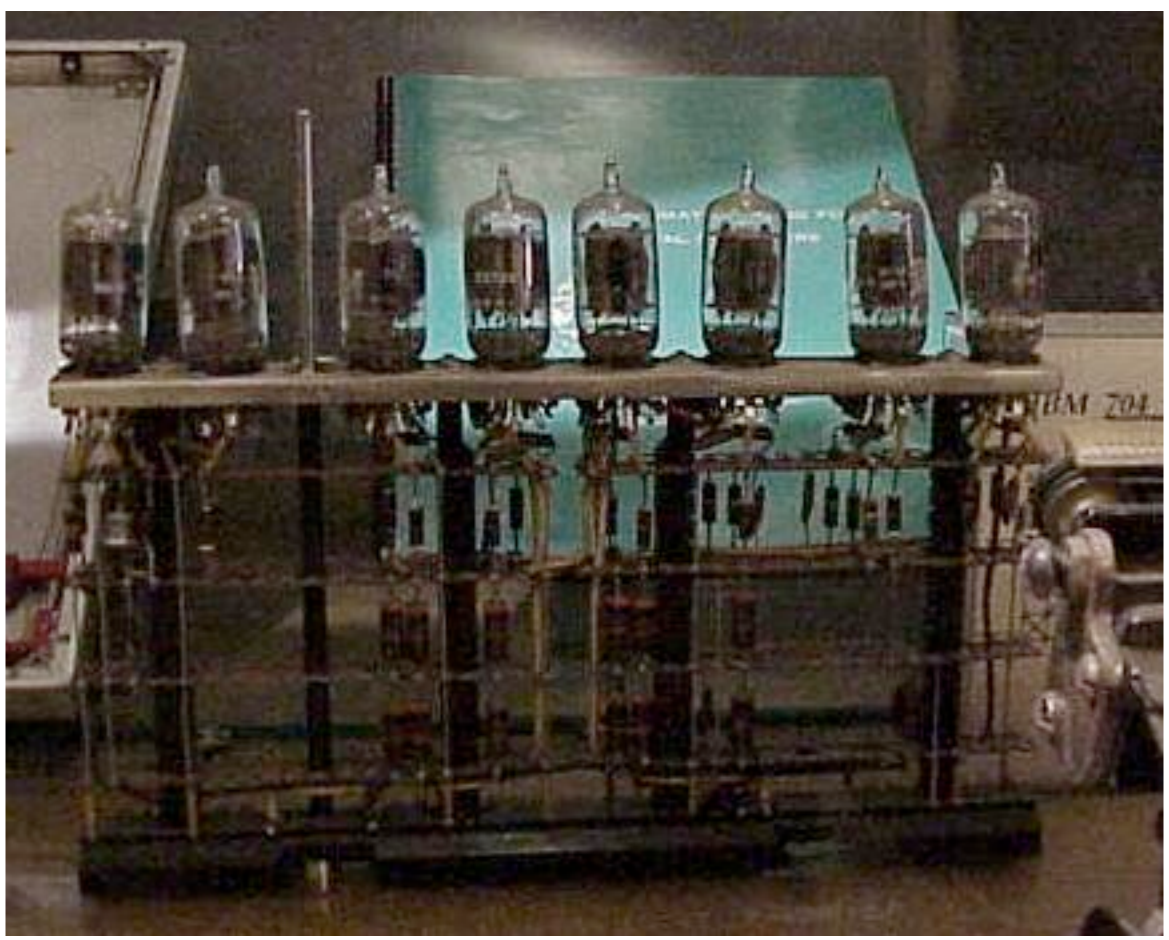


One of about 4000 Logic Modules for an IBM 704 Computer (1954). [photo: http://infolab.stanford.edu/]

late 1950s, McCarthy: LISP

Computers started to be available.

McCarthy developed LISP as a list processing language, originally for the IBM 704 (vacuum tube computer)

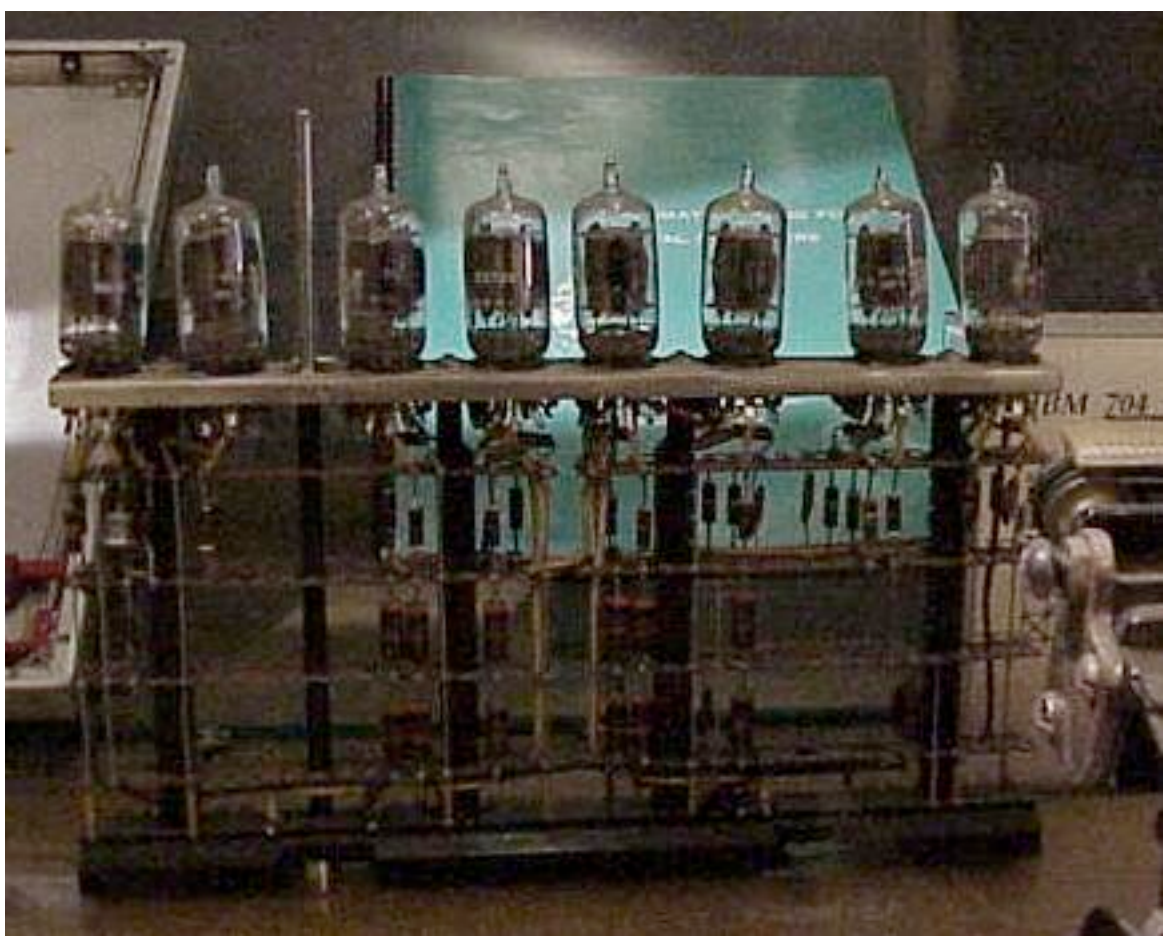


One of about 4000 Logic Modules for an IBM 704 Computer (1954). [photo: http://infolab.stanford.edu/]

late 1950s, McCarthy: LISP

Computers started to be available.

McCarthy developed LISP as a list processing language, originally for the IBM 704 (vacuum tube computer)



One of about 4000 Logic Modules for an IBM 704 Computer (1954). [photo: http://infolab.stanford.edu/]

LISP was significantly more abstract than other contemporary languages:

- FORTRAN (1957)
- COBOL (1959)

Assembly was previously used.

late 1950s, McCarthy: LISP

Contributions:

- `if`-expression and its use in definition of rec. functions
- functions as values
- abstract data: cons-cells, lists and garbage collection
- abstract syntax: s-expressions for data and code

late 1950s, McCarthy: LISP

Contributions:

- `if`-expression and its use in definition of rec. functions
- functions as values
- abstract data: cons-cells, lists and garbage collection
- abstract syntax: s-expressions for data and code

Example:

```
(define map (f list)
   (if (null list)
       nil
     (cons (f (car list)) (map f (cdr list)))))
```

late 1950s, McCarthy: LISP

Contributions:

- `if`-expression and its use in definition of rec. functions
- functions as values
- abstract data: cons-cells, lists and garbage collection
- abstract syntax: s-expressions for data and code

Example:

```
(define map (f list)
   (if (null list)
       nil
     (cons (f (car list)) (map f (cdr list)))))
```

Pragmatic goal: developed to make his AI research easier.

McCarthy [1979] writes that the lambda calculus played a small role in design of the first LISP, but did use a `lambda` keyword.

1960s, Peter Landin: SECD etc.

Influenced by Church, Curry, LISP and Algol 60, Landin developed the SECD machine and ISWIM.

SECD: an abstract machine that mechanises expression evaluation (more details in later lectures)

ISWIM: "If you See What I Mean" a family of FP languages.

1960s, Peter Landin: SECD etc.

Influenced by Church, Curry, LISP and Algol 60, Landin developed the SECD machine and ISWIM.

SECD: an abstract machine that mechanises expression evaluation (more details in later lectures)

ISWIM: "If you See What I Mean" a family of FP languages.

Contributions:

- lexical scoping (c.f. LISP's dynamic scoping)
- FP based on the lambda calculus
- emphasis on generality (hoped to be "the next 700 languages")
- emphasis on equational reasoning
- emphasis on writing programs to show what is computed rather than how

1970-80s, Gordon, Milner et al. ML

Gordon, Milner and Wadsworth (among others) developed ML

- originally as the "meta-language" of their LCF proof assistant
- higher-order functions, pattern-matching, module system, exceptions, references (side-effects, thus impure)

1970-80s, Gordon, Milner et al. ML

references are pointers to mutable cells

among others) developed ML

- originally as the “meta-language” of their LCF proof assistant
- higher-order functions, pattern-matching, module system, exceptions, references (side-effects, thus impure)

1970-80s, Gordon, Milner et al. ML

from Hope by Burstall et al.

references are pointers to mutable cells

- ... ong others) developed ML
- originally as the "meta-language" of their LCF proof assistant
- higher-order functions, pattern-matching, module system, exceptions, references (side-effects, thus impure)

1970-80s, Gordon, Milner et al. ML

from Hope by Burstall et al.

references are pointers to mutable cells

ong others) developed ML

- originally as the "meta-language" of their LCF proof assistant
- higher-order functions, pattern-matching, module system, exceptions, references (side-effects, thus impure)
- major contribution:
  - strongly and statically typed
  - uses type inference (doesn't require explicit type annotations)
  - allows polymorphism
  - user-defined concrete and abstract datatypes

  (more about this Hindley-Milner type system in later lectures.)

1970-80s, Gordon, Milner et al. ML

from Hope by Burstall et al.

references are pointers to mutable cells

…ong others) developed ML

- originally as the "meta-language" of their LCF proof assistant
- higher-order functions, pattern-matching, module system, exceptions, references (side-effects, thus impure)
- major contribution:
  - strongly and statically typed
  - uses type inference (doesn't require explicit type annotations)
  - allows polymorphism
  - user-defined concrete and abstract datatypes

  (more about this Hindley-Milner type system in later lectures.)

A program that passes type inference is guaranteed to not have any type errors!

1970-80s, Gordon, Milner et al. ML

from Hope by Burstall et al.

references are pointers to mutable cells

…ong others) developed ML

- originally as the "meta-language" of their LCF proof assistant
- higher-order functions, pattern-matching, module system, exceptions, references (side-effects, thus impure)
- major contribution:
  - strongly and statically typed
  - uses type inference (doesn't require explicit type annotations)
  - allows polymorphism
  - user-defined concrete and abstract datatypes

  (more about this Hindley-Milner type system in later lectures.)

A program that passes type inference is guaranteed to not have any type errors!

In 1997, formal semantics defined for Standard ML (SML)

ML continued

Example of ML program:

```
fun map f [] = []
  | map f (x::xs) = f x :: map f xs
```

ML continued

Curious fact: evaluation of any ML-typed lambda term terminates.

ML continued

Curious fact: evaluation of any ML-typed lambda term terminates.

How is recursion justified in typed ML?

ML continued

Curious fact: evaluation of any ML-typed lambda term terminates.

How is recursion justified in typed ML?

Lambda calculus uses *Y* combinator for recursion:

$$Y \;\equiv\; \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

ML continued

Curious fact: evaluation of any ML-typed lambda term terminates.

How is recursion justified in typed ML?

Lambda calculus uses *Y* combinator for recursion:

$$Y \equiv \lambda f.\ (\lambda x.\ f\ \underbrace{(x\ x)}_{\text{problem: this cannot be typed in ML's type system}})\ (\lambda x.\ f\ (x\ x))$$

ML continued

Curious fact: evaluation of any ML-typed lambda term terminates.

How is recursion justified in typed ML?

Lambda calculus uses *Y* combinator for recursion:

$$Y \equiv \lambda f.\ (\lambda x.\ f\ \underbrace{(x\ x)}_{\text{problem: this cannot be typed in ML's type system}})\ (\lambda x.\ f\ (x\ x))$$

Solution: use untyped lambda calculus to justify

$$Y\ e \ = \ e\ (Y\ e)$$

and use this equation typed $e : \alpha \to \alpha\ ,\ Y : (\alpha \to \alpha) \to \alpha$

ML continued

Curious fact: evaluation of any ML-typed lambda term terminates.

How is recursion justified in typed ML?

Lambda calculus uses *Y* combinator for recursion:

$$Y \;\equiv\; \lambda f.\ (\lambda x.\ f\ \underbrace{(x\ x)}_{})\ (\lambda x.\ f\ (x\ x))$$

problem: this cannot be typed in ML's type system

Solution: use untyped lambda calculus to justify

$$Y\ e \;=\; e\ (Y\ e)$$

and use this equation typed $e : \alpha \to \alpha\ ,\ Y : (\alpha \to \alpha) \to \alpha$

Justification:

ML $\approx$ typed lambda calculus with special *Y*-combinator constant.

1980: Turner and lazy FP

At the same time as ML was developed, David Turner developed influential FP languages (SASL, KRC, Miranda) with emphasis:

- pure FP (referential transparency)
- lazy evaluation
- use of rec. equations as syntactic sugar for lambda calculus
- convenient syntax (for the programmer)

1980: Turner and lazy FP

At the same time as ML was developed, David Turner developed influential FP languages (SASL, KRC, Miranda) with emphasis:

- pure FP (referential transparency)
- lazy evaluation
- use of rec. equations as syntactic sugar for lambda calculus
- convenient syntax (for the programmer)

1980s: an overall surge in interest of functional languages.

Haskell

Late 1980s:
"There was a strong consensus that the general use of modern, non-strict (lazy) functional languages was being hampered by the lack of a common language." -- Hudak

Haskell

Late 1980s:
"There was a strong consensus that the general use of modern, non-strict (lazy) functional languages was being hampered by the lack of a common language." -- Hudak

A committee was formed to design a language that provides:

- faster communication of new ideas,
- a stable foundation for applications development, and
- a vehicle for learning and using functional languages.

Haskell

Late 1980s:
"There was a strong consensus that the general use of modern, non-strict (lazy) functional languages was being hampered by the lack of a common language." -- Hudak

A committee was formed to design a language that provides:

- faster communication of new ideas,
- a stable foundation for applications development, and
- a vehicle for learning and using functional languages.

Result: Haskell - pure, lazy, statically typed - aims to be practical

Haskell

Late 1980s:
"There was a strong consensus that the general use of modern, non-strict (lazy) functional languages was being hampered by the lack of a common language." -- Hudak

A committee was formed to design a language that provides:

- faster communication of new ideas,
- a stable foundation for applications development, and
- a vehicle for learning and using functional languages.

Result: Haskell - pure, lazy, statically typed - aims to be practical
Noteworthy features in Haskell:

- purely functional monads for I/O
- typeclasses
- significant support for overloading

Examples of lazy evaluation

```
> let numbers = 1 : map (+1) numbers

> take 10 numbers
[1,2,3,4,5,6,7,8,9,10]

> numbers
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,
25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,
46, ...
```

Examples of lazy evaluation

```
> let numbers = 1 : map (+1) numbers

> take 10 numbers
[1,2,3,4,5,6,7,8,9,10]

> numbers
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,
25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,
46, ...
```

```
> let fib = f 0 1 where f m n = m : f n (m+n)

> take 10 fib
[0,1,1,2,3,5,8,13,21,34]
```

Influential people

Church (lambda calculus)

McCarthy (LISP, recursive functions using 'if', garbage collection, symbolic expressions, programs as data)

Landin (ISWIM, lexical scoping, higher-order function, SECD)

Milner, Gordon et al. (ML, type inference, polymorphism)

Steele & Sussman (Scheme, tail-call elimination)

Turner (lazy, pattern-matching, pure)

Burstall (algebraic datatypes)

Milner, Harper, Tofte (formal definition of SML, module system)

Hudak, Wadler, Peyton-Jones, et al. (Haskell, type classes, monads)

Leroy et al. (Ocaml)

This list does not attempt to be complete! Clearly, these people were influenced and aided by many others...

Summary

Course is about:

- formal verification
- implementation of FP

Summary

Course is about:

- formal verification
- implementation of FP

Functional languages:

- based on lambda calculus
- discourage side-effects (for referential transparency)

Summary

Course is about:

- formal verification
- implementation of FP

Functional languages:

- based on lambda calculus
- discourage side-effects (for referential transparency)

Three kinds of FP language:

- Lisp: untyped, s-expression based
- ML: statically typed, impure, strict
- Haskell: statically typed, pure, lazy