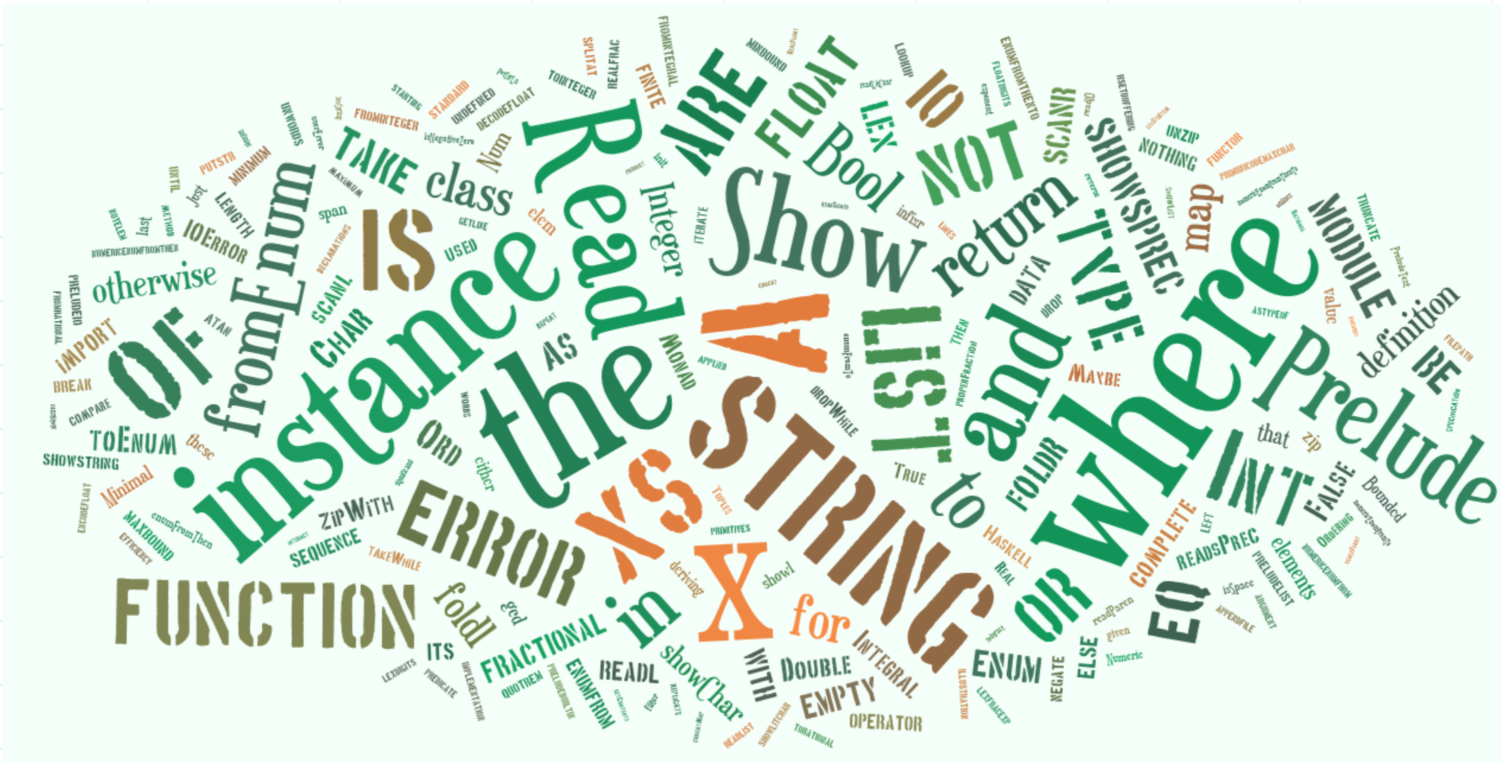
Functional Programming and λ Calculus



Amey Karkare
Dept of CSE, IIT Kanpur

Software Development Challenges

- Growing size and complexity of modern computer programs
- Complicated architectures
  - Massively parallel architectures, Memory hierarchy, distributed systems,...
- Fast and cost effective software development
- Above all: Correctness!
  - Proof that the program works for *all* cases

Well-structured Software

- Easy to write and debug
- Reusable modules
- Amenable to proofs
- Permit rapid prototyping

Solutions to the development challenges.

Programming style to support development of well-structured software.

Functional Languages

- Fundamental operation is the application of functions to arguments.
- Main features to improve modularity:
  - No (almost none!!) side effects
  - Higher order functions
  - Lazy evaluation

Example

Summing the integers 1 to 10 in C:

```
int total = 0, i;
for (i = 1; i <= 10; ++i)
   total = total+i;
```

Values change for both total and i during program execution

Example

- Summing integers 1 to 10 in a pure functional language
  - No side effect => No assignments to variables!

```
sum (m, n) = if (m > n) 0
             else m + sum (m+1, n)


sum (1, 10) // main function
```

Historical Background

[source: http://www.cs.nott.ac.uk/~gmh/chapter1.ppt]

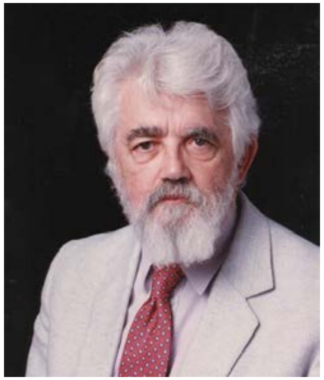
1930s:

Alonzo Church develops the <u>lambda calculus</u>, a simple but powerful theory of functions.

Historical Background

[source: http://www.cs.nott.ac.uk/~gmh/chapter1.ppt]

1950s:

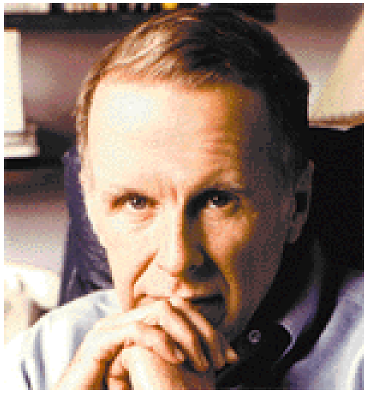


John McCarthy develops Lisp, the first functional language, with some influences from the lambda calculus, but retaining variable assignments.

Historical Background

[source: http://www.cs.nott.ac.uk/~gmh/chapter1.ppt]

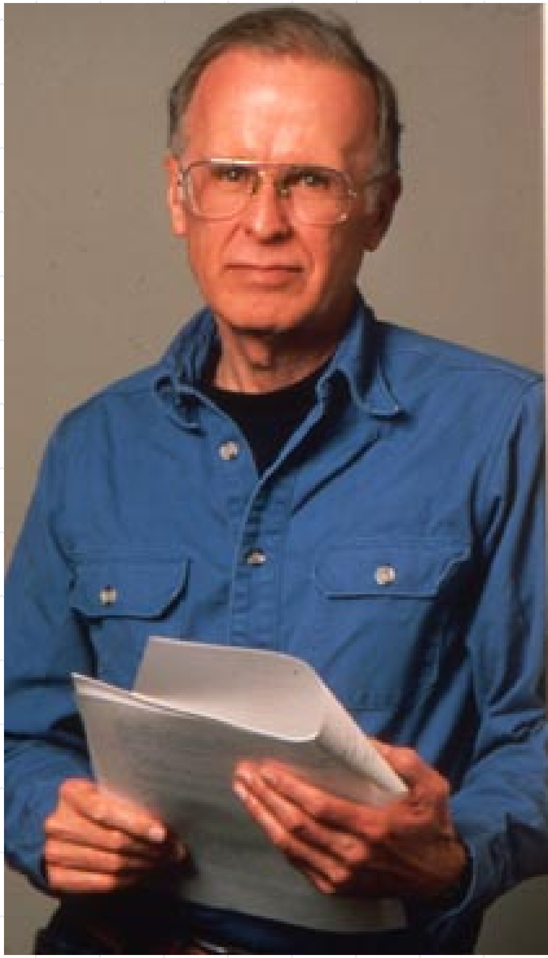
1970s:



John Backus develops FP, a functional language that emphasizes *higher-order functions* and *reasoning about programs*.

Trivia

- John Backus : Proposed (in 1954) a program that translated high level expressions into native machine code.
- Fortran I project (1954-1957): The first compiler was released



1977 ACM Turing Award

"for profound, influential, and lasting contributions to the design of practical high-level programming systems, notably through his work on FORTRAN, and for publication of formal procedures for the specification of programming languages."

Introduced FP in his Turing Award lecture

"Can Programming be Liberated from the von Neumann Style?".

Quicksort: English description

1. Empty list is already sorted.
2. For a non empty list
    a. Pick the first element, **pivot**, from the array.
    b. Recursively quicksort the array of elements with values less than the pivot. Call it S.
    c. Recursively quicksort the array of elements with values greater than or equal to the pivot, except the pivot. Call it G.
    d. The final sorted array is: the elements of S followed by pivot, followed by the elements of G.

Quicksort: Functional (Haskell) description*

```
quicksort [] = []
quicksort (x:xs) =
        quicksort [y | y <- xs, y<x]
        ++ [x]
        ++ quicksort [y | y <- xs, y>=x]
```

* source: https://www.haskell.org/tutorial/haskell-98-tutorial.pdf

Higher order function

```
add x y = x + y
inc = add 1

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

- map is a higher order function. It takes a function as argument.
- Functional programming treats functions as first-class citizens. There is no discrimination between function and data.

```
map inc [1, 2, 3]  =>    [2, 3, 4]
```

Lazy evaluation

- Do not evaluate an expression unless it is needed
- Never evaluate an expression more than once

```
length [1/1, 2/2, 0/0, 4/4]
                                        =>   4

numsFrom n = n : numsFrom (n+1)
squares = map (^2) (numsfrom 0)
take 5 squares
                    => [0,1,4,9,16]
```

Lambda calculus

The “assembly language” of functional programming

The Abstract Syntax

- A really **tiny** language of expressions

// An expression can be a

$$e :: x$$ // Variable

$$\mid \lambda x.\, e_1$$ // Function Definition

$$\mid e_1\ e_2$$ // Function Application

$$\mid (e_1)$$

That's all the Syntax!!

Conventions

- $\lambda x.\, e_1 e_2 e_3$ is an abbreviation for $\lambda x.\,(e_1 e_2 e_3)$, i.e., the scope of x is as far to the right as possible until it is
  - terminated by a) whose matching (occurs to the left of the λ, or
  - terminated by the end of the term
- Application associates to the left: $e_1 e_2 e_3$ is to be read as $(e_1 e_2) e_3$ and not as $e_1 (e_2 e_3)$
- $\lambda xyz.\, e$ is an abbreviation for $\lambda x \lambda y \lambda z.\, e$ which in turn is actually $\lambda x.\,(\lambda y.\,(\lambda z.\, e))$

α-renaming

- The name of a bound variable has no meaning except for its use to identify the bounding λ.
- Renaming a λ variable including all its bound occurrences does not change the meaning of an expression.
- For example, $\lambda x. x\ x\ y$ is equivalent to $\lambda u. u\ u\ y$
  - But it is not same as $\lambda x. x\ x\ w$
  - Can not change free variable!

β-reduction(Execution)

- if an abstraction $\lambda x. e_1$ is applied to a term e_2 then the result of the application is
  - the body of the abstraction e_1 with all free occurrences of the formal parameter x replaced with e_2.
- For example,

$$(\lambda f \lambda x. f\ (f\ x))\ twice \rightarrow^{\beta}$$

$$twice\ (twice\ x)$$

Caution

- During β-reduction, make sure a free variable is not captured inadvertently.
- The following reduction is WRONG

$$\lambda x.\lambda y.x\ \ (\lambda x.y) \rightarrow \lambda y.\lambda x.y$$

- Use α-renaming to avoid variable capture

$$\lambda x.\lambda y.x\ \ (\lambda x.y) \rightarrow \lambda u \lambda v.u\ (\lambda x.y)$$

$$\rightarrow \lambda v.\lambda x.y$$

Exercise

- Apply β-reduction as far as possible

1. $(\lambda x\ y\ z.\ x\ z\ (y\ z))\ (\lambda x\ y.\ x)\ (\lambda y.\ y)$

2. $(\lambda\ x.\ x\ x)\ (\lambda x.\ x\ x)$

3. $(\lambda x\ y\ z.\ x\ z\ (y\ z))\ (\lambda x\ y.\ x)((\lambda x.\ x\ x)(\lambda x.\ x\ x))$

Church-Rosser Theorem

- Multiple ways to apply β-reduction
- Some may not terminate
- However, if two different reduction sequences terminate then **they always terminate in the same term**

$$\begin{tikzcd} & e \arrow[dl] \arrow[dr] & \\ e_1 \arrow[dr] & & e_2 \arrow[dl] \\ & e' & \end{tikzcd}$$

- Leftmost, outermost reduction will find the *normal form* if it exists

But what about other stuff?

- Constants ?
  - Numbers
  - Booleans
- Complex Types ?
  - Lists
  - Arrays
- Don’t we need “data”?

Recall: functions are first-class citizens!
Function is data and data is function.

Numbers

- We need a “Zero”
  - “Absence of item”
- And something to count
  - “Presence of item”
- Intuition: Whiteboard and Marker
  - Blank board represents Zero
  - Each mark by marker represents a count.
  - However, other pairs of objects will work as well
- Lets translate this intuition into λ-expr

Numbers

- Zero = $\lambda m.\ \lambda w.\ w$
  - No mark on whiteboard
- One = $\lambda m.\ \lambda w.\ m\ w$
- Two = $\lambda m.\ \lambda w.\ m\ (m\ w)$
- ...

- What about operations?
  - add, multiply, subtract, divide ...

Operations on Numbers

- succ = $\lambda xmw.\ m\ (x\ m\ w)$
  - Verify that $succ\ N = N + 1$
- add = $\lambda xymw.\ x\ m\ (y\ m\ w)$
  - Verify that $add\ N\ M = N + M$
- mult = $\lambda xymw.\ x\ (y\ m)\ w$
  - Verify that $mult\ N\ M = N * M$

called **Church Numerals.**

Booleans

- True and False
- Intuition: Select one out of two possible choices.
- λ-expressions
  - True = $\lambda x\, \lambda y.\, x$
  - False = $\lambda x\, \lambda y.\, y$

Operations on Booleans

- Logical operations

$$and = \lambda p\, q.\ p\ q\ p$$

$$not = \lambda p\ t\ f.\ p\ f\ t$$

...

- The conditional function if
  - $if\ c\ \ e_1\ \ e_2$ reduces to e_1 if c reduces to True and e_2 if c reduces to False

$$if = \lambda c\ e_t\ e_f\ .(c\ e_t\ e_f)$$

More...

- More such types can be found at
  - https://en.wikipedia.org/wiki/Church_encoding

- It is fun to come up with your own definitions for constants and operations over different types
  - or to develop understanding for existing definitions.

We are missing something!!

- The machinery described so far does not allow us to define Recursive functions
  - factorial, Fibonacci ...
- There is no concept of “named” functions
  - So no way to refer to a function “recursively”?
- Fix-point computation comes to rescue

Fix-point and Y-combinator

- A fix-point of a function f is a value p such that $f\ p = p$
- Assume existence of a *magic* expression, called Y-combinator, that when applied to a λ-expression, gives its fixed point
$$Y\ f = f\ (Y\ f)$$
- Y–combinator gives us a way to apply a function recursively

Factorial

fact =

    λn. if (isZero n) One (mult n (fact (pred n)))

= (λfλn. if (isZero n) One (mult n (f (pred n)))) fact

fact = g fact

fact is a fixed point of function

    g = λfλn. if (isZero n) One (mult n (f (pred n)))

Using Y-combinator,

  fact = Y (λfλn. if (isZero n) One (mult n (f (pred n))))

        = Y g

Verify

fact 2

= (Y g) 2 = g (Y g) 2

// Y f = f (Y f), definition of Y-combinator

= (λfλn. if (is0 n) 1 (* n (f (pred n)))) (Y g) 2

= (λn. if (is0 n) 1 (* n ((Y g) (pred n)))) 2

= if (is0 2) 1 (* 2 ((Y g) (pred 2)))

= (* 2 ((Y g) 1))

...

= (* 2 (* 1 (if (is0 0) 1 (* 0 ((Y g) (pred 0))))))

= (* 2 (* 1 1)) = 2

Recursion

- Y-combinator allows to unroll the body of loop once – similar to one unfolding of recursive call
- Sequence of Y-combinator applications allow complete unfolding of recursive calls
- BUT, what about the existence of Y-combinator?

Y-combinators

- Many candidates exist

$$Y_1 = \lambda f\big(\lambda x.\ f(x\ x)\big)\big(\lambda x. f(x\ x)\big)$$

$$T = \lambda abcdefghijklmnopqstuvwxyzr. \\ r(thisisafixedpointcombinator)$$

$$Y_{funny} = TTTTTTTTTTTTTTTTTTTTTTTTTT$$

- Verify that (Y f) = f (Y f) for each

Summary

- A cursory look at λ-calculus to understand how Functional Programming works
  - How it is different from imperative programming
- Functions are data, and Data are functions!
- Church Turing Thesis => The power of λ calculus equivalent to that of Turing Machine


THANK
YOU