v1.1
20140407

Programmazione Avanzata e Paradigmi

Ingegneria e Scienze Informatiche - UNIBO

a.a 2013/2014

Lecturer: Alessandro Ricci

[module 1.6]
FUNCTIONAL PROGRAMMING - FOUNDATIONS

SUMMARY

- FP foundation
  - λ-calculus - introduction
    - syntax and semantics
    - computation as reduction: β-reduction
    - confluence property & Church-Rosser theorem
    - λ-calculus as a computing modek
      - modeling data structures
      - modeling recursion - fixed-point theorem
  - Lisp & descendant

FOUNDATIONS

- The origin and foundations of functional languages can be traced back to two contributions, developed in different contexts and for different purposes
  - **Lambda calculus (λ-calculus)** (~1930s)
    - context: investigations about the foundations of mathematics
  - **Lisp language** (~1958)
    - context: Artificial Intelligence

LAMBDA CALCULUS

- Developed by **Alonzo Church** [1932-33,1941]
  - investigating the foundations of mathematics
- Formal calculus that aims at *capturing formally* one's intuition about the *behavior of functions*
  - calculus = a syntax for terms and a set of rewriting rules for transforming terms
- Modern functional languages can be thought as (non trivial) embellishments of the lambda calculus

Alonzo Church, Professor of Mathematics, whose students include Alan Turing, Leon Henkin, Stephen Kleene, Michael Rabin (Turing Award 1976), Dana Scott (Turing Award 1976), Barkley Rosser, Martin Davis, and whose PhD descendents include 3000 other mathematicians and computer scientists, including Robert Constable, Edmund Clarke (Turing Award 2007), Allen Emerson (Turing Award 2007), David Harel, Tom Mitchell, and Les Valiant (Turing Award 2010).

FUNCTIONS AS COMPUTATIONS

- Capturing the *computational* aspects of functions
  - to be contrasted with the classic idea/semantics of functions as sets (sets of argument/value pairs)
- Allows to express self-applications = functions applied to themselves
  - self-application leads to paradoxes in traditional theories based on e.g. sets
    - a set containing it self...
  - instead, Lambda calculus allows to gain the effect of recursion without explicitly writing a recursive definition
    - without inconsistencies or paradoxes.

λ CALCULUS - SYNTAX

- Syntax:

  `M ::= X | M M | λX.M | ( M )`

  where
  - `M` is a λ-terms (or simply terms)
  - `X` is a non-terminal representing a generic variable.
    - It can assume any symbol from a denumerable set `(x, y, z, w, ...)`
  - `()` are terminal symbols
- The term `(λX.M)` is called **abstraction**
  - it represents the definition of a function
    - `λ` is the abstraction operator
- The term `(M M)` is called **application**
  - it represents the application of a function to some args

EXAMPLES

- Valid expressions
  - λx.x
  - (λx.x) y
  - λx.λy.x y
  - (λx.λy.x y)(λx.a)
  - …
- Invalid terms
  - λ.x
  - λ(λ.x).y
  - …

REMARKS

- All functions in the lambda calculus are *anonymous* functions
  - having no names
- They only accept *one* input variable
  - *currying* can be used to implement functions with several variables

SYNTACTIC CONVENTIONS

- Some conventions have been introduced to further simplify the notation.
  - application associates to the left:

    `M1 M2 M3 M4 ... Mn`

    stand for

    `((((M1 M2) M3) M4) ... Mn)`
  - the scope of λ extends as far as possible to the right:

    `λx.x y`

    stand for

    `(λx.(x y))`

    and not `((λx.x) y)`
  - sequences of λs may be collapsed:

    `λxyz.M = λx.λy.λz.M`

FREE AND BOUND VARIABLES

- Abstraction operator binds the variable representing the *formal parameter* of the function such that
  - renaming of the bound variable does not modify the semantics of an expression
  - possible substitutions have no effect on bound variables
- Free variable = variables not bound by a λ operator
  - formally the set of free variables of M, Fv(M), can be defined inductively as:

    $Fv(x) = \{ x \}$

    $Fv(M\ N) = Fv(M) \cup Fv(N)$

    $Fv(\lambda x.M) = Fv(M) - \{x\}$
  - bound variables Bv(M):

    $Bv(x) = \{\}$

    $Bv(M\ N) = Bv(M) \cup Bv(N)$

    $Bv(\lambda x.M) = Bv(M) \cup \{x\}$

COMPUTATION: BETA REDUCTION

- Computation proceeds by rewriting according to the prescriptions of β-reduction:

  $(\lambda x.M)N \rightarrow M[N/x]$

- M β-reduces to N - this is written: $(M \rightarrow N)$ - when N is the result of the application of one step of β-reduction to some subterm of M
  - $(\lambda x.M)N$ is called *redex*
  - $M[N/x]$ is called *reductum*
- Not deterministic
  - multiple redexes can be chosen

SUBSTITUTION

- Substitution M[N/x] *without variable capture* can be defined as:

```
x[N/x] = N
y[N/x] = y,                                  when x != y
(M1 M2)[N/x] = (M1[N/x])(M2[N/x])
(λy.M)[N/x] = (λy.M[N/x])                    when x!=y, y not in Fv(N)
(λy.M)[N/x] = (λy.M)                         when x = y (*)
```

(*) this means that substitution does not apply to bound occurrences of the variable

β-REDUCTION: EXAMPLE

- Evaluating: `(λx.x)((λx.x)(λz.(λx.x) z))`
  - 3 possible redexes:
    - `(λx.x)((λx.x)(λz.(λx.x) z))`
    - `((λx.x)(λz.(λx.x) z))`
    - `(λx.x) z`
- A reduction:

```
(λx.x)((λx.x)(λz.(λx.x) z)) ->
(λx.x)(λz.(λx.x) z) ->
(λz.(λx.x) z) ->
(λz.z)
```

α-EQUIVALENCE

- Expressions are equivalent when they differ only in the names of their free variables.
- This is formalized by the α-equivalence:

$\lambda x.M \equiv \lambda y.M[y/x]$ y fresh, i.e. y not in Fv(M)

β-EQUIVALENCE

- β-reduction is not symmetric
  - M → N, it is not the case that N → M'
- **β-equivalence**: symmetric relation defined as the *reflexive and transitive closure* of the β-reduction (=β)
  - inductive definition:
    - If M→M′ then M =β M′.
    - For all terms M, M =β M holds.
    - If M =β M′, then M′ =β M.
    - If M =β M′ and M′ =β M″, then M =β M″.
  - =β is the smallest relation satisfying these conditions
- M =β N means that M and N are connected through a sequence of β-reductions
  - not necessarily in the same direction

NORMAL FORMS

- A normal form is as λ-term that does not contain a redex
  - so β-reduction terminates
- For instance: (λx.(λy.y) x) → λx.x
  - where λx.x is a normal form.
- Not every term has a normal form
  - there are terms which reduces an infinite number of times without producing a normal form
  - example: Ω = (λx.x x)(λx.x x)
    (λx.x x)(λx.x x) → (x x)[(λx.xx)/x]
    = (λx.x x)(λx.x x) → …

CONFLUENCE (CHURCH-ROSSER PROPERTY)

- Inherent non-determinism has no dangerous effect: whatever redex we chose to reduce first, the final result (normal form) of a sequence of reductions in always the same
- Fundamental property called ***confluence***
  - that is: λ-calculus is confluent under α- and β- reductions
- **Theorem** (Chuch-Rosser)

  If M reduces to N1 in a number of reduction steps, and M also reduces also to N2 in a number of reduction steps, then there exist a term P such that both N1 and N2 both reduce to P in a number of steps

CONFLUENCE (CHURCH-ROSSER PROPERTY)

- Corollary
  - if a term can be reduced a normal form, this normal form is unique and independent of the path followed to reach it
  - normal forms are unique up to α-equivalence
- Note
  - it is still possible for a reduction sequence not to terminate even if the term has a normal form:
    $(\lambda xy.y)\Omega \rightarrow (\lambda xy.y)\Omega \rightarrow \ldots$
  - the same term has a terminating reduction sequence:
    $(\lambda x.\lambda y.y)\Omega \rightarrow \lambda y.y$

EVALUATION STRATEGY

- Two common valuation strategies for λ-calculus
  - **normal** order - choosing the *leftmost*, *outermost* redex first
    - ***non strict*** semantics: the application is evaluated first
    - i.e. if e1 and e2 are redexes in a term and e1 is a sub term of e2, the e1 will not be reduced next
  - **applicative** order - choosing the leftmost, innermost redex first
    - ***strict*** semantics: the argument is evaluated first

- **Theorem** (Church-Rosser II)
  If a term has a normal form, then the normal order reduction can reduce it

EVALUATION STRATEGY IN FUNCTIONAL LANGUAGES

- **Call-by-name**
  - *normal-order* + further restriction that no reductions are allowed inside abstractions
- **Call-by-value**
  - *applicative order* + further restriction that no reductions are allowed inside abstractions

EXAMPLE

- Evaluating: `(λx.x)((λx.x)(λz.(λx.x) z))`

- Normal order:

```
(λx.x)((λx.x)(λz.(λx.x) z)) ->
(λx.x)(λz.(λx.x) z) ->
(λz.(λx.x) z) ->
(λz.z)
```

- Call by name

```
(λx.x)((λx.x)(λz.(λx.x) z)) ->
(λx.x)(λz.(λx.x) z) ->
(λz.(λx.x) z)
```

EXAMPLE

- Applicative order:

```
(λx.x)((λx.x)(λz.(λx.x) z)) ->
(λx.x)((λx.x)(λz.z)) ->
(λx.x)(λz.z) ->
(λz.z)
```

- Call by value

```
(λx.x)((λx.x)(λz.(λx.x) z)) ->
(λx.x)(λz.(λx.x) z) ->
(λz.(λx.x) z)
```

LAMBDA-CALCULUS AS A MODEL OF COMPUTATION

- Lambda calculus failed to be used as a foundation for all of mathematics (like set theory, for instance)
- ...but it became a reference model for describing how computation works extensively investigated and extended
  - with a very strong powerful consistency result given by the Church-Rosser theorem
  - large body of literature and results
  - today used by computer scientists for studying rigorously languages and their properties

ENCODING DATA TYPES: BOOLEAN

- Any kind of data structure can be modelled with λ-calculus and λ-terms
- For instance, Boolean data type
  - constants
    `TRUE ≡ (λxy.x)`
    `FALSE ≡ (λxy.y)`
  - conditional expressions
    `IF ≡ (λbte.bte)`
  - operators
    `AND ≡ λ b1 b2.IF b1 b2 FALSE`
    `OR ≡ λ b1 b2.IF b1 TRUE b2`
    `NOT ≡ λ b1.IF b1 FALSE TRUE`

ENCODING NATURAL NUMBERS

- Church numerals:
  n ≡ λsz.sss…z, i.e. s composed n times
- Representing the values:
  0 ≡ λsz.z
  1 ≡ λsz.sz
  2 ≡ λsz.s(sz)
  3 ≡ λsz.s(s(sz))
  …
- Representing operations:
  SUCC ≡ λn.λsz.s(nsz)
  ADD ≡ λm.λn.λsz.ms(nsz)
  IFZERO ≡ λm.m (λx.FALSE) TRUE
- Alternative definitions:
  ADD ≡ λm.λn.λsz.m SUCC n
  MUL ≡ λm.λn.λsz.m (ADD n) 0
  EXP ≡ λm.λn.λsz.m (MUL n) 1

PAIRS (2-TUPLES)

- Church encoding for pairs
  - PAIR ≡ λx.λy.λf.f x y
  - FIRST ≡ λp.p TRUE
  - SECOND ≡ λp.p FALSE
- Example
  - the pair containing the symbols a and b can be represent by the function PAIR a b
  - then
    - FIRST (PAIR *a* b) → *a*
    - SECOND (PAIR *a* b) → b
- Pairs are the simplest example of composite data structure
  - the approach can be generalised into any possible composition

LISTS

- Pairs can be used to model lists. A list can be either NIL or a CONS where:
  - `NIL ≡ λx.TRUE` --- the empty list
  - `CONS ≡ λx y z.(z x y)` --- similar to pair
- Functions retrieving the head/tail of a list:
  - `CAR ≡ λx.x TRUE` --- getting the head of a list x
  - `CDR ≡ λx.x FALSE` --- getting the tail

LOCAL VARIABLES

- Let expressions:

  ```
  let x = e1 in e2
  ```

  can be represented by:

  ```
  (λx.e2) e1
  ```

RECURSION - FIXED POINT THEOREM

- **Fixed point** theorem
  every lambda expression e has a fixpoint e' such that
  (e e') =β e'
- Proof
  - take e' to be (Y e), where Y known as **Y combinator:**

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

  - then, by applying Y to an expression e we have

    (Y e) = (λf.(λx.f(x x))(λx.f(x x)) e
    = (λx. e (x x)) (λx. e (x x))
    = e ((λx. e (x x)) (λx. e (x x))
    = e (Y e)

RECURSION - FIXED POINT THEOREM

- Remark: self-replicating behaviour

  `Yf = f(Yf) = f(f(Yf)) = f(f(f(Yf))) = …`

  – “unfolding effect”

RECURSION - FIXED POINT THEOREM

- The property suggested by the fixpoint theorem can be used to model *recursive function in terms of non-recursive ones*
  - consider the recursive function:

    `f ≡ ... f ....`
  - this could be rewritten as:

    `f ≡ (λf. ... f ...) f`

    where the inner occurrence of f is now bound

    => f is a fixpoint of the lambda expression (λf. ... f ...)
  - that is, it is what we can compute also with Y, so fix point `f = e' = (Y e) = (Y (λf. ... f ...))`, so:

    `f ≡ Y (λf. ... f ...)`

    which is the *nonrecursive* definition for f.

EXAMPLE: FACTORIAL

- The factorial function

  $fac \equiv \lambda n.\ if\ (n = 0)\ then\ 1\ else\ (n * fac(n - 1))$

  can be written non recursively as:

  $fac = Y\ (\lambda\ fac.\ \lambda n.\ if\ (n = 0)\ then\ 1\ else\ (n * fac(n - 1)))$

EXAMPLE OF COMPUTATION

- Being `G = λf.λn.(1, if n = 0; else n × (f (n−1)))` (*)

```
(Y G) 4
G (Y G) 4
(λf.λn.(1, if n = 0; else n × (f (n−1)))) (Y G) 4
(λn.(1, if n = 0; else n × ((Y G) (n−1)))) 4
1, if 4 = 0; else 4 × ((Y G) (4−1))
4 × (G (Y G) (4−1))
4 × ((λn.(1, if n = 0; else n × ((Y G) (n−1)))) (4−1))
4 × (1, if 3 = 0; else 3 × ((Y G) (3−1)))
4 × (3 × (G (Y G) (3−1)))
4 × (3 × ((λn.(1, if n = 0; else n × ((Y G) (n−1)))) (3−1)))
4 × (3 × (1, if 2 = 0; else 2 × ((Y G) (2−1))))
4 × (3 × (2 × (G (Y G) (2−1))))
4 × (3 × (2 × ((λn.(1, if n = 0; else n × ((Y G) (n−1)))) (2−1))))
4 × (3 × (2 × (1, if 1 = 0; else 1 × ((Y G) (1−1)))))
4 × (3 × (2 × (1 × (G (Y G) (1−1)))))
4 × (3 × (2 × (1 × ((λn.(1, if n = 0; else n × ((Y G) (n−1)))) (1−1)))))
4 × (3 × (2 × (1 × (1, if 0 = 0; else 0 × ((Y G) (0−1))))))
4 × (3 × (2 × (1 × (1))))
24
```

(*) in strict λ syntax: G = λf . λn. IF (ISZERO n) c1 (MUL n (f (PRED n)))

OTHER WELL-KNOWN COMBINATORS

- Besides Y, *combinators* are - in general - expressions that contain *no free variables*
- Main examples
  - Identity **I**
    - **I** ≡ λx.x
  - Constant functions **K**
    - **K** ≡ λx.λy.x
  - Application **S**
    - **S** ≡ λx.λy.λz.(x z (y z))
  - Composition **B**
    - **B** ≡ λg.λf.λx.g (f x)
  - Self-application Ω
    - Ω ≡ (λx.x x)(λx.x x)

S-K COMPLETENESS

- It can be shown that any lambda term can be expressed using only the **S** and **K** combinators
  - also the other combinators can be defined using S and K
- Examples:
  - $I \equiv S\ K\ K$
  - $TRUE \equiv K$
  - $FALSE \equiv S\ K$
  - …

CHURCH'S THESIS (~1936)

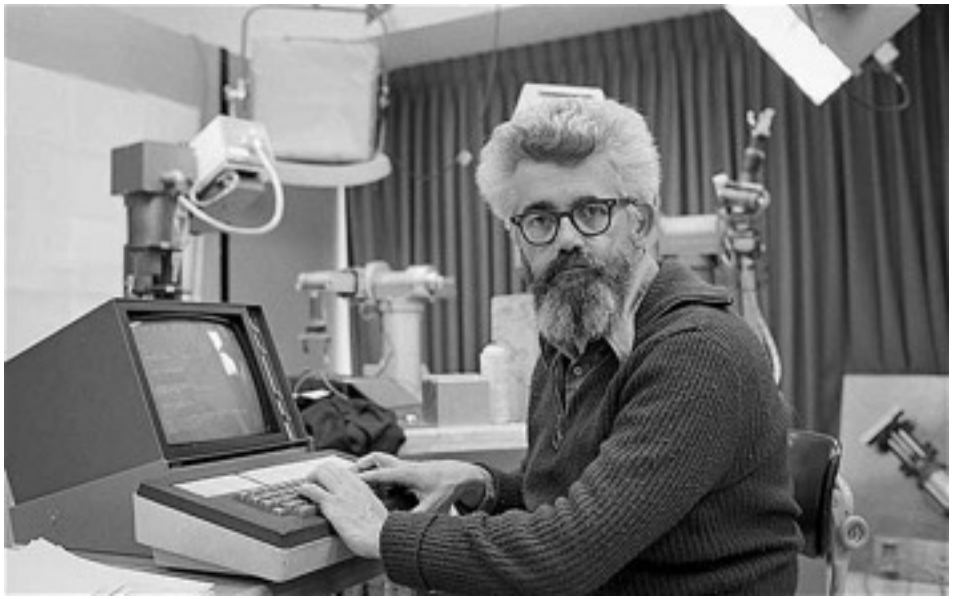
- The capability of simulating recursion is one of the strongest feature of the lambda calculus as a model of computation
- Church recognized this power by expressing his famous thesis:

> *Effectively computable functions from positive integers to positive integers are just those definable in the lambda calculus.*

... but the notion of "effectively computable" cannot be formalized as the pure notion of function, so no proofs can be given to this thesis.

LISP

- First "high-level" functional language introduced by **John McCarthy** in late 1950s
- Original motivation:
  - list-processing language for use in AI research
  - symbolic processing
- An earliest attempt was FLPL (Fortran-compiled list processing language) implemented in 1958 on top of FORTRAN on the IBM 704 end then Lisp was introduced few years later



LISP - MAIN CONTRIBUTIONS

- *Conditional expression* and its use in writing recursive functions
- The use of *lists* and *high-order operations* over lists
  - such as *mapcar*
- The central idea of a *cons* cell and the use of garbage collection as a method of reclaiming unused cells
- The use of **S-expression** (Symbolic expressions) to represent uniformly both program and data.

S-EXPRESSIONS

- An s-expression can be inductively defined as:
  - an *atom*, or
  - an expression of the form *(x . y)*

    where
    - x and y are s-expressions (ordered pair)
    - the definition of an atom varies per context
      - they are constants and symbols
- Abbreviated notation: omitting dots..
  - (x y z) stands for (x . (y . (z . NIL)))
  - NIL is the special end-of-list symbol
    - written '() in Scheme

S-EXPRESSIONS EXAMPLES IS LISP

- Lists: `(1 2 foo)`
- Expressions/function call:

```
(+ 1 2)
(append '(1 2) '(3 4))
```

- Lambda expressions:

```
(lambda (arg) (+ arg 1))
```

- Example of application:

```
((lambda (arg) (+ arg 1)) 5)
```

- Definitions:

```
(define factorial (n)
   (if (<= n 1)
       1
       (* n (factorial (- n 1)))))
```

LIST MODEL IN PARTICULAR

- They are implemented as linked list
  - each cell of this list is called a ***cons***
  - cons is composed of two pointers, called the **car** and **cdr**
- List built-in functions
  - `(cons h t)`
    - it returns a new cons made by h as car and t as cdr
  - `(car c)`
    - it returns the car value of the cons c
  - `(cdr c)`
    - it returns the cdr value of the cons c

CONDITIONAL EXPRESSION AND RECURSIVE FUNCTIONS

```
(define fac
     (lambda(n)
       (if (= n 0)
          1
          (* n (fac (- n 1))))))
```

HIGH-ORDER FUNCTIONS: THE mapcar EXAMPLE

```
(define mapcar
   (lambda (fun lst)
         (if (null? lst)
             '()
             (cons (fun (car lst))
                   (mapcar fun (cdr lst))))))
```

LISP DIALECTS: COMMON LISP

- Designed to be efficiently implementable on any personal computer or workstation
- Large language standard including many built-in data types, functions, macros and other language elements, as well as an object system (Common Lisp Object System or shorter CLOS).
- Borrowed certain features from Scheme such as lexical scoping and lexical closures

LISP DIALECTS: SCHEME

- Statically (lexically) scoped and properly tail-recursive dialect of the Lisp programming language
  - invented by Guy Lewis Steele Jr. and Gerald Jay Sussman at MIT
- Designed to have a cleaner and simpler semantics wrt Lisp and to patch some Lisp problems (e.g. dynamic scoping and dynamic closures) and to capture Hewitt's idea about actors and message passing
  - more close to lambda calculus
- One of the most used languages in education
  - it can express quite effectively different programming paradigms - not only the functional one

RECENT LISP DIALECTS: RACKET AND CLOJURE

- Racket
  - Recent evolution of Scheme, with features useful for programming in the large and software development
- Clojure
  - designed to be a *pragmatic* general-purpose language
  - strongly influenced by Haskell
  - it is a compiled language, compiling directly to JVM bytecode
  - it makes it possible to interact with the Java Environment

SUMMARY

- FP foundation
  - λ-calculus - introduction
    - syntax and semantics
    - computation as reduction: β-reduction
    - confluence property & Church-Rosser theorem
    - λ-calculus as a computing modek
      - modeling data structures
      - modeling recursion - fixpoint theorem
  - Lisp & descendant

BIBLIOGRAPHY

- Gabbrielli and Martini. “Programming Principles and Paradigms” (ch 11.6)
- Paul Hudak. *“Concepts, Evolution, and Application of Functional Programming Languages”*. ACM Computing Surveys, Vol. 21, No. 3, Sept 1989