Functional Programming
An Introduction

Walter Cazzola

Dipartimento di Informatica
Università degli Studi di Milano
e-mail: cazzola@di.unimi.it
twitter: @w_cazzola

What is functional programming?

- Functions are first class (objects).
  - That is, everything you can do with "data" can be done with functions themselves (such as passing a function to another function).
- Recursion is used as a primary control structure.
  - In some languages, no other "loop" construct exists.
- There is a focus on list processing.
  - Lists are often used with recursion on sub-lists as a substitute for loops.
- "Pure" functional languages eschew side-effects.
  - This excludes assignments to track the program state.
  - This discourages the use of statements in favor of expression evaluations.

Whys

- All these characteristics make for more rapidly developed, shorter, and less bug-prone code.
- A lot easier to prove formal properties of functional languages and programs than of imperative languages and programs.

The basic idea is to model everything as a «mathematical function».

There are only two linguistic constructs:

- abstraction, used to define the function;
- application, used to call it.

No state concept

- this means no assignments are allowed
- variables are just names.

E.g., in $f(x) = x + 1$ the name f is irrelevant,

- the function $g(x) = x + 1$ represents the same function;
- it can be referred as $x \mapsto x + 1$.

λ-expressions are made of constants, variables, λ, . and parenthesis

1. if x is a variable or a constant then x is a λ-expression;
2. if x is a variable and M is a λ-expression then $\lambda x.M$ is a λ-expression;
3. if M, N are λ-expressions then (MN) is a λ-expression.

Abstraction & Application

λ-calculus provides only two basic operations: abstraction and application

- $\lambda x.x+1$ is an example of abstraction that defines the successor;
- $(\lambda x.x+1)7$ is an example of application that calculates the successor of 7;
  - application is left-associative, i.e., $MNP \equiv (MN)P$.

Binding, Free and Bound Variables

- in $\lambda x.xy$ x is a bound variable whereas y is unbound (free)
- in $\lambda x.\lambda y.xy$ (for short $\lambda xy.xy$) both variables are bound;
- in $(\lambda x.M)y$, all the occurrences of x in M are replaced by y (denoted as $M[x/y]$) and brings to $M[x/y]$ as a result
  - e.g., $(\lambda x.x+1)7 \to x+1[x/7] \to 7+1 \to 8$.

ML is a general-purpose functional programming language developed by Robin Milner et al. in the 70ies.

- ML is the acronym for metalanguage, since it is an abstraction on polymorphic λ-calculus.

Features of ML include:

- a call-by-value evaluation strategy, first-class functions, parametric polymorphism,
- static typing, type inference, algebraic data types, pattern matching, and exception handling.

ML uses eager evaluation, which means that all sub-expressions are always evaluated.

- lazy evaluation can be achieved through the use of closures.

We will use OCaML (`http://caml.inria.fr`).

OCaML is an implementation of ML with extra functionality (object-orientation, modules, imperative statements, . . .).

OCaML comes with

- an interpreter (`ocaml`) and
- a compiler (`ocamlc`).

```
let main() = print_string("Hello World in ML Style\n");;
main();;
```

```
[12:28]cazzola@surtur:~/lp/ml>ocamlc -o helloworld helloworld.ml
[12:28]cazzola@surtur:~/lp/ml>ls
helloworld*  helloworld.cmi  helloworld.cmo  helloworld.ml
[12:28]cazzola@surtur:~/lp/ml>helloworld
Hello World in ML Style.
[12:28]cazzola@surtur:~/lp/ml>rlwrap ocaml
        Objective Caml version 4.12.0

# let main() =  print_string("Hello World in ML Style.\n");;
val main : unit -> unit = <fun>
# main();;
Hello World in ML Style.
- : unit = ()
# ^D
[12:29]cazzola@surtur:~/lp/ml>
```

ML derives directly from λ-calculus:

- functions are defined independently of their name

```
let succ = fun x -> x+1;;
let succ x = x+1;;
```

functions can be aliased

```
let succ' = succ;;
```

- calls are simply the application of the arguments to the function

```
succ 2;;
(fun x -> x+1) 2;;
```

```
[16:19]cazzola@surtur:~/lp/ml>ocaml
        Objective Caml version 4.12.0

# let succ = fun x -> x+1;;
val succ : int -> int = <fun>
# succ 7;;
- : int = 8
# succ -1;;
Error: This expression has type int -> int
       but an expression was expected of type int
```

Scoping

- a new binding to a name hides the old bind;
- static binding is used in function definition (closure).
  - i.e., a triplet: args list, function body and environment (x, x+y, [5/y]).

```
[17:01]cazzola@surtur:~/lp/ml>ocaml
        OCaML version 4.12.0
# let f x = 5;;
val f : 'a -> int = <fun>
# let f x = 7;;
val f : 'a -> int = <fun>
# f 1;;
- : int = 7
# let y = 5;;
val y : int = 5
# let addy = fun x -> x+y;;
val addy : int -> int = <fun>
# addy 8;;
- : int = 13
# let y=10;;
val y : int = 10
# addy 8;;
- : int = 13
# (fun x -> x+y) 8;;
- : int = 18
[17:57]cazzola@surtur:~/lp/ml>
```

In ML functions are first class citizens

- i.e., they can be used as values;
- when passed to a function this is an high-order function.

```
let compose f g x = f (g x);;
let compose' (f, g) x = f (g x);;
```

```
[15:30]cazzola@surtur:~/lp/ml>ocaml
# let compose f g x = f (g x);;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# let compose' (f,g) x = f (g x);;
val compose' : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b = <fun>
# let succ = fun x -> x +1;;
val succ : int -> int = <fun>
# let plus1 = compose succ;;
val plus1 : ('_a -> int) -> '_a -> int = <fun>
# let plus1' = compose' succ;;
Error: This expression has type int -> int
       but an expression was expected of type ('a -> 'b) * ('c -> 'a)
# let plus2 = plus1 succ;;
val plus2 : int -> int = <fun>
# let plus2'= compose'(succ, succ);;
val plus2' : int -> int = <fun>
# plus2 7;;
- : int = 9
# plus2' 7;;
- : int = 9
```

Functions can be defined by pattern matching.

```
match expression with
  | pattern when boolean expression -> expression
  | pattern when boolean expression -> expression
```

Patterns can contain

- constants, tuples, records, variant constructors and variable names;
- a catchall pattern denoted _ that matches any value; and
- sub-patterns containing alternatives, denoted **pat_1 | pat_2**.

When a pattern matches

- the corresponding expression is returned.
- the (optional) when clause is a guard on the matching; it filters out undesired matchings.

```
let invert x =
  match x with
    | true -> false
    | false -> true ;;

let invert' = function
  true -> false | false -> true ;;
```

Functional Programming

Walter Cazzola

A function is called recursive when it is defined through itself.

Example: Factorial.

- $5! = 5 * 4 * 3 * 2 * 1$
- Note that: $5! = 5 * 4!$, $4! = 4 * 3!$ and so on.

Potentially a recursive computation.

From the mathematical definition:

$$n! = \begin{cases} 1 & \text{if } n=0, \\ n*(n-1)! & \text{otherwise.} \end{cases}$$

When $n=0$ is the base of the recursive computation (axiom) whereas the second step is the inductive step.

Recursion
What in ML?

Still, a function is recursive when its execution implies another invocation to itself.

- directly, i.e. in the function body there is an explicit call to itself;
- indirectly, i.e. the function calls another function that calls the function itself (mutual recursion).

```
let rec fact(n) = if n<=1 then 1 else n*fact(n-1);;

let main() =
  print_endline("fact(  5) : - "^string_of_int(fact(5)));
  print_endline("fact(  7) : - "^string_of_int(fact(7)));
  print_endline("fact( 15) : - "^string_of_int(fact(15)));
  print_endline("the largest admissible integer is ;- "^string_of_int(max_int));
  print_endline("fact( 25) : - "^string_of_int(fact(25)));;

main();;
```

```
[11:31]cazzola@surtur:~/lp/ml>ocamlc -o fact fact.ml
[11:31]cazzola@surtur:~/lp/ml>fact
fact(  5) : - 120
fact(  7) : - 5040
fact( 15) : - 1307674368000
the largest admissible integer is ;- 4611686018427387903
fact( 25) : - -2188836759280812032
[11:31]cazzola@surtur:~/lp/ml>
```

Recursion
Execution: What's Happen?

```
[11:45]cazzola@surtur:~/lp/ml>ocaml
        OCaML version 4.12.0
# let rec fact(n) =
    if n<=1
    then 1
    else n*fact(n-1);;
val fact : int -> int = <fun>
# fact 4;;
- : int = 24
[11:46]cazzola@surtur:~/lp/ml>
```

It runs fact(4):

- a new frame with n = 4 is pushed on the stack;
- n is greater than 1;
- it calculates 4*fact(3)6, it returns 24

It runs fact(3):

- a new frame with n = 3 is pushed on the stack;
- n is greater than 1;
- it calculates 3*fact(2)2, it returns 6

It runs fact(2):

- a new frame with n = 2 is pushed on the stack;
- n is greater than 1;
- it calculates 2*fact(1)1, it returns 2

It runs fact(1):

- a new frame with n = 1 is pushed on the stack;
- n is equal to 1;
- it returns 1

At any invocations the run-time environment creates an **activation record** or **frame** used to store the current values of:

- local variables, parameters and the location for the return value.

To have a frame for any invocation permits to:

- trace the execution flow;
- store the current state and restore it after the execution;
- avoid interferences on the local calculated values.

Warning:

Without any stopping rule, the inductive step will be applied "forever".

- Actually, the inductive step is applied until the memory reserved by the virtual machine is full.

Leonardo Pisano, known as Fibonacci, in 1202 in his book "Liber Abaci" faced the (quite unrealistic) problem of determining:

> «how many pairs of rabbits can be produced from a single pair if each pair begets a new pair each month and every new pair becomes productive from the second month on, supposing that no pair dies»

To introduce a sequence whose i-th member is the sum of the 2 previous elements in the sequence. The sequence will be soon known as the Fibonacci numbers.

Recursion
Case Study: Fibonacci Numbers (Cont'd)

Fibonacci numbers are recursively defined:

$$f(n) = \begin{cases} 0 & \text{if } n=0, \\ 1 & \text{if } n=1 \text{ or } n=2, \\ f(n-1) + f(n-2) & \text{otherwise.} \end{cases}$$

The implementation comes forth from the definition:

```
open List;;
let rec fibo(n) = if n<=1 then n else fibo(n-1) + fibo(n-2);;

let main() =
  let in's = [5; 7; 15; 25; 30] in
    for i=0 to List.length in's -1 do
      print_endline(
        "fibo("^string_of_int(nth in's i)^") :- "^string_of_int(fibo(nth in's i)));
    done;;
main();;
```

```
[16:08]cazzola@surtur:~/lp/ml>ocamlc -o fibo fibo.ml
[16:14]cazzola@surtur:~/lp/ml>fibo
fibo(5) :- 5
fibo(7) :- 13
fibo(15) :- 610
fibo(25) :- 75025
fibo(30) :- 832040
[16:14]cazzola@surtur:~/lp/ml>
```

Recursion
Recursion Easier & More Elegant

The recursive solution is more intuitive:

```
let rec fibo(n) = if n<=1 then n else fibo(n-1) + fibo(n-2);;
```

The iterative solution is more cryptic:

```
let fibo(n) =
  let fib' = ref 0 and fib''= ref 1 and fib = ref 1 in
    if n<=1 then n
    else
      (for i=2 to n do
        fib := !fib' + !fib'';
        fib' := !fib'';
        fib'' := !fib;
      done;
      !fib);;
```

But ...

The iterative implementation is more efficient:

```
[18:22]cazzola@surtur:~/lp/ml>time time_ifibo 50
fibo(50) :- 12586269025
0.000u 0.006s 0:00.00 0.0%      0+0k 0+0io 0pf+0w
[18:22]cazzola@surtur:~/ml/lp>time time_rfibo 50
fibo(50) :- 12586269025
1605.211u 1.688s 26:48.62 99.8% 0+0k 0+0io 0pf+0w
[18:49]cazzola@surtur:~/lp/ml>
```

The overhead is mainly due to the creation of the frame but this also affects the occupied memory.

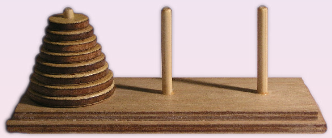
This can be avoided with a tail recursive solution:

```ocaml
let rec trfiboaux n m fib_m' fib_m =
  if (n=m) then fib_m
  else (trfiboaux n (m+1) fib_m (fib_m'+fib_m));;
let fibo n = if n<=1 then 1 else trfiboaux n 1 0 1;;
```

```
[16:59]cazzola@surtur:~/lp/ml>time trfibo 50
fibo(50) :- 12586269025
0.000u 0.005s 0:00.00 0.0%      0+0k 0+0io 0pf+0w
[16:59]cazzola@surtur:~/lp/ml>
```

Problem Description

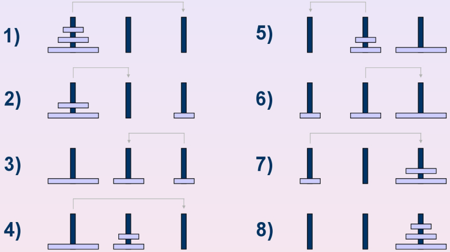
There are 3 available pegs and several holed disks that should be stacked on the pegs. The diameter of the disks differs from disk to disk each disk can be stacked only on a larger disk.



The goal of the game is to move all the disks, one by one, from the first peg to the last one without ever violate the rules.

The Towers of Hanoi
The Recursive Algorithm

3-Disks Algorithm



n-Disks Algorithm

Base: n=1, move the disk from the source (S) to the target (T);

Step: move n-1 disks from S to the first free peg (F), move the last disk to the target peg (T), finally move the n-1 disks from F to T.

The Towers of Hanoi

ML/OCaML Implementation

```
type peg = string*string*string ;;
type pegs = {mutable src: peg; mutable trg: peg; mutable aux:peg} ;;

let nth(x,y,z) n = match n with 1 -> x | 2 -> y | 3 -> z ;;
let set_nth(x,y,z) w n = match n with 1 -> (w,y,z) | 2 -> (x,w,z) | 3 -> (x,y,w) ;;

let set_nth_peg ps p n =
  match n with 1 -> ps.src <- p| 2 -> ps.trg <- p | 3 -> ps.aux <- p ;;
let nth_peg ps n = match n with 1 -> ps.src | 2 -> ps.trg | 3 -> ps.aux ;;

let top(x,y,z) =
  match x,y,z with "0","0","0" -> 3 | "0","0", _ -> 2 | "0", _, _ -> 1 | _, _, _ -> 0 ;;

let p:pegs={src=("1","2","3"); trg=("0","0","0"); aux=("0","0","0")} in
  let rec display ps n =
    if n <4 then (
      print_endline("  "^nth ps.src n^"    "^nth ps.trg n^"    "^nth ps.aux n);
      display ps (n+1);)
  and move ps source target =
    let s=(top (nth_peg ps source))+1 and t= top (nth_peg ps target) in (
      set_nth_peg ps (set_nth (nth_peg ps target) (nth (nth_peg ps source) s) t) target;
      set_nth_peg ps (set_nth (nth_peg ps source) "0" s) source;
      display ps 1;)
  and move_disks ps disks source target aux =
    if disks <=1 then (
      print_endline("moving from "^string_of_int(source)^" to "^string_of_int(target));
      move ps source target;)
    else (
      move_disks ps (disks-1) source aux target;
      print_endline("moving from "^string_of_int(source)^" to "^string_of_int(target));
      move ps source target;
      move_disks ps (disks-1) aux target source;
    );
  in (print_endline("Start!!!");display p 1; move_disks p 3 1 3 2;) ;;
```

The Towers of Hanoi
3-Disks Run

```
[16:21]cazzola@surtur:~/lp/ml>ocamlc -o hanoi2 hanoi2.ml
[16:21]cazzola@surtur:~/lp/ml>hanoi2
Start!!!                moving from 1 to 2   moving from 1 to 3   moving from 2 to 3
  1    0    0             0    0    0          0    0    0          0    0    0
  2    0    0             0    0    0          0    1    0          0    0    2
  3    0    0             3    2    1          0    2    3          1    0    3
moving from 1 to 3      moving from 3 to 2   moving from 2 to 1   moving from 1 to 3
  0    0    0             0    0    0          0    0    0          0    0    1
  2    0    0             0    1    0          0    0    0          0    0    2
  3    0    1             3    2    0          1    2    3          0    0    3
[16:21]cazzola@surtur:~/lp/ml>
```

References

- Davide Ancona, Giovanni Lagorio, and Elena Zucca.
  Linguaggi di Programmazione.
  Città Studi Edizioni, 2007.
- Greg Michaelson.
  An Introduction to Functional Programming through λ-Calculus.
  Addison-Wesley, 1989.
- Larry c. Paulson.
  ML for the Working Programmer.
  Cambridge University Press, 1996.