Functional Programming
An Introduction

Walter Cazzola

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

Functional Programming
Walter Cazzola
FP Introduction λ-calculus ML/OCaML Introduction functions scope high-order functions Pattern Matching recursion tail recursion Hanoi's Towers References

Functional Programming
Overview

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.

Functional Programming
Walter Cazzola
FP Introduction λ-calculus ML/OCaML Introduction functions scope high-order functions Pattern Matching recursion tail recursion Hanoi's Towers References

Functional Programming
Overview

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 $x \mapsto x + 1$.

Functional Programming
Walter Cazzola
FP Introduction λ-calculus ML/OCaML Introduction functions scope high-order functions Pattern Matching recursion tail recursion Hanoi's Towers References

Functional Programming
λ-Calculus [Church and Kleene ~1930].

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., $(\lambda x.x + 1)7 \to x + 1[x/7] \to 7 + 1 \to 8$.

Functional Programming
ML [Milner et al. ∼1970]

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).

Functional Programming
ML/OCaML [Leroy et al. ∼1980]

Functional Programming
Walter Cazzola
FP
Introduction
λ-calculus
ML/OCaML
Introduction
functions
scope
high-order functions
Pattern Matching
recursion
tail recursion
Hanoi's Towers
References

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>
```


Functional Programming
ML Functions

Functional Programming
Walter Cazzola
FP
Introduction
λ-calculus
ML/OCaML
Introduction
functions
scope
high-order functions
Pattern Matching
recursion
tail recursion
Hanoi's Towers
References

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
```


Functional Programming
Name Scope

Functional Programming
Walter Cazzola
FP
Introduction
λ-calculus
ML/OCaML
Introduction
functions
scope
high-order functions
Pattern Matching
recursion
tail recursion
Hanoi's Towers
References

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>
```

Functional Programming

High-Order Functions

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
```


Functional Programming
Walter Cazzola

FP
Introduction
λ-calculus
ML/OCaML
Introduction
Functions
Scope
High-order functions
Pattern Matching
Recursion
Tail recursion
Hanoi's Towers
References

Functional Programming

Functions & Pattern Matching

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

FP
Introduction
λ-calculus
ML/OCaML
Introduction
Functions
Scope
High-order functions
Pattern Matching
Recursion
Tail recursion
Hanoi's Towers
References

Recursion

Definition: Recursive Function

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.

Functional Programming
Walter Cazzola

FP
Introduction
λ-calculus
ML/OCaML
Introduction
Functions
Scope
High-order functions
Pattern Matching
Recursion
Tail recursion
Hanoi's Towers
References

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

Recursion
Side Notes on the Execution

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.

Recursion
Case Study: Fibonacci Numbers

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 ...

Recursion
Tail Recursion

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:

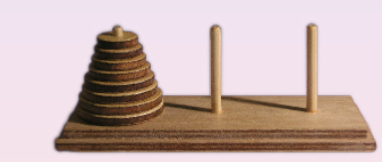
```
let rec trfiboaux n m fib_m' fib_m =
  if (n=m) then fib_m
  else (trfiboaux n (m+1) fib_m (fib_m'+fib_m));;

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>
```


The Towers of Hanoi
Definition (Édouard Lucas, 1883)

Problem Description

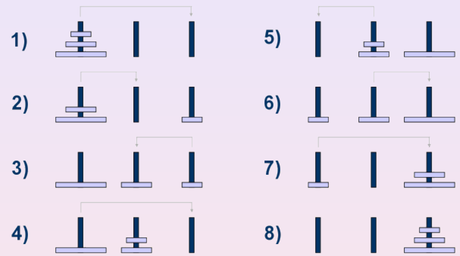
There are 3 available pegs and several holed disks that should be stacked on the pegs. The diameter of the disks differs from disk to disk each disk can be stacked only on a larger disk.



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.

Datatypes in ML

lists, tuples, arrays records, variants . . .

Walter Cazzola

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

OCaML's Primitive Datatypes

Introduction

Even if not explicitly said

- ML is a strongly and statically typed programming language;
- the type of each expression is inferred from the use.

```
[10:46]cazzola@surtur:~/lp/ml>ocaml
# 1+2*3;;
- : int = 7
# let pi = 4.0 * atan 1.0;;
Error: This expression has type float but an expression was expected of type
         int
# let pi = 4.0 *. atan 1.0;;
val pi : float = 3.14159265358979312
# let square x = x *. x;;
val square : float -> float = <fun>
# square 5;;
Error: This expression has type int but an expression was expected of type
         float
# square 5. ;;
- : float = 25.
```


OCaML's Primitive Datatypes

Booleans

OCaML provides two constants

- `true` and `false`

Operations on Booleans

logic operators
`&& || not` — logical and, or and negation respectively
relational operators
`== <>` — equal and not equal to operators
`< > <= >=` — less than, greater than, less than or equal to and greater than or equal to operators

```
[12:01]cazzola@surtur:~/lp/ml>ocaml
# true;;
- : bool = true
# true || false;;
- : bool = true
# 1<2;;
- : bool = true
# 2.5<>2.5;;
- : bool = false
```


OCaML's Primitive Datatypes

Strings

Strings

- they are native in OCaML
- several operations come from the **String** module
- since OCaML 4, strings are immutable and <- is deprecated
  - **Bytes**/**Bytes**.set must be used instead.

Operations on Strings

- ^ string concatenation
- .[] positional access to chars

```
let s1 = "walter" and s2 = "cazzola" ;;
val s1 : string = "walter"
val s2 : string = "cazzola"

# let s=s1^" "^s2;;
val s : string = "walter cazzola"
# s.[9];;
- : char = 'z'
# String.length(s);;
- : int = 14

# let b = Bytes.of_string s ;;
val b : bytes = Bytes.of_string "walter cazzola"
# Bytes.set b 0 'W'; Bytes.set b 7 'C';;
- : unit = ()
# let s = Bytes.to_string b;;
val s : string = "Walter Cazzola"
```

OCaML's Collections
Lists

Lists

- homogeneous
- cons operator ::
- concatenation operator @ (inefficient).

```
[14:54]cazzola@surtur:~/lp/ml>ocaml
# [1; 1+1; 3];;
- : int list = [1; 2; 3]
# [1;'a';3.14];;
Error: This expression has type char but an expression was expected of type
         int
# let l1 = [1;2;3] and l2 = [4;5;6];;
val l1 : int list = [1; 2; 3]
val l2 : int list = [4; 5; 6]
# l1@l2;;
- : int list = [1; 2; 3; 4; 5; 6]
# let l1 = 0::l;;
val l1 : int list = [0; 1; 2; 3]
# List.nth l1 2;;
- : int = 2
```

More operations come from the **List** module.

OCaML's Collections
Lists: Introspecting on the List

You can check if an element is in the list

```
# let a_list = [2; 7; 25; 3; 11; -1; 0; 7; 25; 25; 999; -25; 7];;
val a_list : int list = [2; 7; 25; 3; 11; -1; 0; 7; 25; 25; 999; -25; 7]
# let rec is_in l x = if l==[] then false else x==List.hd(l) || is_in (List.tl l) x;;
val is_in : 'a list -> 'a -> bool = <fun>
# is_in a_list 11;;
- : bool = true
# is_in a_list 12;;
- : bool = false
```

Count the number of occurrences

```
let count x l =
  let rec count tot x = function
    []     -> tot
  | h::tl -> if (h==x) then count (tot+1) x tl else count tot x tl
  in count 0 x l
```

```
# #use "count2.ml" ;;
val count : 'a -> 'a list -> int = <fun>
# count 7 a_list ;;
- : int = 3
```


OCaML's Collections
Lists: Introspecting on the List (Cont'd)

Look for the position of an item

```
let idx l x =
    let rec idx2 l x acc =
      if (List.hd l) == x then acc else idx2 (List.tl l) x (acc+1)
    in idx2 l x 0;;
```

```
# #use "idx.ml" ;;
# idx a_list 999;;
- : int = 10
```

Slice the list from an index to another

```
let slice i j l =          l      match l with
   let rec slice count res = function
      hd::tl when count < i  -> slice (count+1) res tl
   |  _      when count == j -> (List.rev res)
   |  hd::tl                 -> slice (count+1) (hd::res) tl
in slice 0 [] l ;;
```

```
# #use "slice.ml" ;;
val slice : int -> int -> 'a list -> 'a list = <fun>
# slice 2 5 a_list ;;
- : int list = [25; 3; 11]
```


OCaML's Collections
Tuples

Tuples are

- fixed-length heterogeneous lists.

```
# let a_tuple = (5, 'a', "a string", [1; 2; 3], 3.14);;
val a_tuple : int * char * string * int list * float =
  (5, 'a', "a string", [1; 2; 3], 3.14)
# let a_pair = (1,"w");;
val a_pair : int * string = (1, "w")
# fst a_pair ;;  (* works only on a pair *)
- : int = 1
# snd a_pair;;   (* works only on a pair *)
- : string = "w"
# let a_triplet = ( "a", 0, true);;
val a_triplet : string * int * bool = ("a", 0, true)
# fst a_triplet;;
Error: This expression has type string * int * bool
       but an expression was expected of type 'a * 'b
```

OCaML's Collections
Arrays

Arrays are
- direct-accessible, homogeneous, and mutable lists.

```
# let an_array = [|1;2;3|];;
val an_array : int array = [|1; 2; 3|]
# an_array.(2);;
- : int = 3
# an_array.(1) <- 5;;
- : unit = ()
# an_array ;;
- : int array = [|1; 5; 3|]
```

More operations come from the **Array** module.

```
# let a = Array.make 5 0;;
val a : int array = [|0; 0; 0; 0; 0|]
# Array.concat [a; an_array] ;;
- : int array = [|0; 0; 0; 0; 0; 1; 5; 3|]
# let a_matrix = Array.make_matrix 2 3 'a' ;;
val a_matrix : char array array = [|[|'a'; 'a'; 'a'|]; [|'a'; 'a'; 'a'|]|]
# a_matrix.(1).(2) <- 'z';; (* .(1).(2) is equivalent to (1,2) *)
- : unit = ()
# a_matrix ;;
- : char array array = [|[|'a'; 'a'; 'a'|]; [|'a'; 'a'; 'z'|]|]
```


Datatypes
Walter Cazzola

Primitive Types
Booleans
Strings
Collections
Lists
Tuples
Arrays
Records
User-Defined
Aliasing
Variants
References

OCaML's Collections
Records

Recods are
- name accessible (through field names),
- heterogeneous, and
- mutable (through the **mutable** keyword) tuples.

```
# type person = {name: string; mutable age: int};;
type person = { name : string; mutable age : int; }
# let p = {name = "Walter"; age = 35} ;;
val p : person = {name = "Walter"; age = 35}
# p.name;;
- : string = "Walter"
# p.age <- p.age+1;;
- : unit = ()
# p ;;
- : person = {name = "Walter"; age = 36}
# p.name <- "Walter Cazzola";;
Error: The record field label name is not mutable
```


Datatypes
Walter Cazzola

Primitive Types
Booleans
Strings
Collections
Lists
Tuples
Arrays
Records
User-Defined
Aliasing
Variants
References

User Defined Datatype in OCaML
Aliasing & Variants

Aliasing.
The easiest way to define a new type is to give a new name to an existing type.

```
# type int_pair = int*int;;
type int_pair = int * int
# let a : int_pair = (1,3);;
val a : int_pair = (1, 3)
# fst a;;
- : int = 1
```

Any type can be aliased.

Variants.
A variant type lists all possible shapes for values of that type.
- Each case is identified by a capitalized name, called a constructor.

```
# type int_option = Nothing | AnInteger of int ;;
type int_option = Nothing | AnInteger of int
# Nothing;;
- : int_option = Nothing
# AnInteger 7;;
- : int_option = AnInteger 7
```


Datatypes
Walter Cazzola

Primitive Types
Booleans
Strings
Collections
Lists
Tuples
Arrays
Records
User-Defined
Aliasing
Variants
References

User Defined Datatype in OCaML
Variants

Mutually recursive type must be declared via the **and** keyword.

```
type card = Card of regular | Joker
  and regular = { suit : card_suit; name : card_name; }
  and card_suit = Heart | Club | Spade | Diamond
  and card_name = Ace | King | Queen | Jack | Simple of int;;
let value = function
  Joker                   ->  0
| Card {name = Ace}       -> 11
| Card {name = King}      -> 10
| Card {name = Queen}     ->  9
| Card {name = Jack}      ->  8
| Card {name = Simple n} ->  n ;;
```

This code defines 4 types.
- the value function gives a value to each card.

```
# #use "cards.ml";;
type card = Card of regular | Joker
and regular = { suit : card_suit; name : card_name; }
and card_suit = Heart | Club | Spade | Diamond
and card_name = Ace | King | Queen | Jack | Simple of int
val value : card -> int = <fun>
# value ( Card { suit = Heart; name = Jack } ) ;;
- : int = 8
```

User Defined Datatype in OCaML
Variants.

Compared to OO programming,

– a variant type is equivalent to a class hierarchy composed of an abstract base class or interface representing the type and derived classes representing each of the variant type constructors.

Moreover, it is possible to manipulate them by pattern matching.

```
type state = On | Off;;
let turn = function
   On -> Off
 | Off -> On ;;
```

```
# #use "state.ml";;
val turn : state -> state = <fun>
# let s = Off;;
val s : state = Off
# turn s;;
- : state = On
```


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.

The OCaML Module System

Abstract and concrete data types

Walter Cazzola

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

Modules are used to realize data type (ADT and implementation) and collecting functions.

Modules are composed of two parts:
- a (optional) public interface exposing the types and operations defined in the module (**sig** ... **end**);
- the module implementation (**struct** ... **end**).

Modules can abstract data and hide implementation details

```
module A :
  sig
    ...
  end =
  struct
    ...
  end ;;
```

Modules are useful for organizing large implementations in smaller self-contained pieces of code.


```
module PrioQueue =
   struct
     type priority = int
     type char_queue = Empty | Node of priority * char * char_queue * char_queue
     exception QueueIsEmpty

     let empty = Empty
     let rec insert queue prio elt =
       match queue with
         Empty -> Node(prio, elt, Empty, Empty)
       | Node(p, e, left, right) ->
           if prio <= p
           then Node(prio, elt, insert right p e, left)
           else Node(p, e, insert right prio elt, left)

     let rec remove_top = function
         Empty -> raise QueueIsEmpty
       | Node(prio, elt, left, Empty) -> left
       | Node(prio, elt, Empty, right) -> right
       | Node(prio, elt, (Node(lprio, lelt, _, _) as left),
                         (Node(rprio, relt, _, _) as right)) ->
           if lprio <= rprio
           then Node(lprio, lelt, remove_top left, right)
           else Node(rprio, relt, left, remove_top right)

     let extract = function
         Empty -> raise QueueIsEmpty
       | Node(prio, elt, _, _) as queue -> (prio, elt, remove_top queue)
   end;;
```



```
# #use "char_pqueue.ml" ;;
module PrioQueue :
  sig
    type priority = int
    type char_queue =
        Empty
      | Node of priority * char * char_queue * char_queue
    exception QueueIsEmpty
    val empty : char_queue
    val insert : char_queue -> priority -> char -> char_queue
    val remove_top : char_queue -> char_queue
    val extract : char_queue -> priority * char * char_queue
  end
# let pq = empty ;;
val pq : PrioQueue.char_queue = Empty
# let pq = insert pq 0 'a' ;;
val pq : PrioQueue.char_queue = Node (0, 'a', Empty, Empty)
# let pq = insert (insert pq 3 'c') (-7) 'w';;
val pq : PrioQueue.char_queue =
  Node (-7, 'w', Node (0, 'a', Empty, Empty), Node (3, 'c', Empty, Empty))
# let pq = extract pq;;
val pq : PrioQueue.priority * char * PrioQueue.char_queue =
  (-7, 'w', Node (0, 'a', Empty, Node (3, 'c', Empty, Empty)))
```

The OCaML Module System
Signature (Sig ... End)

```
module type CharPQueueAbs =
  sig
    type priority = int          (* still concrete *)
    type char_queue              (* now abstract *)
    val empty : char_queue
    val insert : char_queue -> int -> char -> char_queue
    val extract : char_queue -> int * char * char_queue
    exception QueueIsEmpty
  end;;
```

WRT the previous implementation this:

- opacifies the type char_pqueue and hides the remove_top operation.

```
# #use "CharPQueueAbs.mli" ;;
module type CharPQueueAbs =
  sig
    type priority = int
    type char_queue
    val empty : char_queue
    val insert : char_queue -> int -> char -> char_queue
    val extract : char_queue -> int * char * char_queue
    exception QueueIsEmpty
  end
# module AbstractPrioQueue = (PrioQueue: CharPQueueAbs);;
module AbstractPrioQueue : CharPQueueAbs
# AbstractPrioQueue.remove_top;;
Error: Unbound value AbstractPrioQueue.remove_top
# AbstractPrioQueue.insert AbstractPrioQueue.empty 1 'a' ;;
- : AbstractPrioQueue.char_queue = <abstr>
```


The OCaML Module System
Separate Compilation

Modules and their interface can be separately compiled.

```
[17:11]cazzola@surtur:~/lp/ml/mod-01>ls
CharPQueueAbs.mli  CharPQueue.ml      main.ml
[17:11]cazzola@surtur:~/lp/ml/mod-01>ocamlc -c CharPQueueAbs.mli
[17:12]cazzola@surtur:~/lp/ml/mod-01>ocamlc -c CharPQueue.ml
[17:16]cazzola@surtur:~/lp/ml/mod-01>ocamlc -o main CharPQueue.cmo main.ml
[17:19]cazzola@surtur:~/lp/ml/mod-01>ls
CharPQueueAbs.cmi  CharPQueueAbs.mli  CharPQueue.cmi  CharPQueue.cmo  CharPQueue.ml  main
```

```
open CharPQueue.AbstractPrioQueue;;
let x = insert empty 1 'a' ;;
```

In this case the file names for the module implementation and interface must be different (and start with a capital letter).

The OCaML Module System
Separate Compilation (Cont'd).

The implementation and interface of the module can share the same file name (apart of the suffix)

- module, sig and struct keywords are dropped

The module name comes after the module file name.

```
[17:39]cazzola@surtur:~/lp/ml/mod-02>ls
CharPQueue.mli  CharPQueue.ml      main.ml
[17:39]cazzola@surtur:~/lp/ml/mod-02>ocamlc -c CharPQueue.mli
[17:39]cazzola@surtur:~/lp/ml/mod-02>ocamlc -c CharPQueue.ml
[17:39]cazzola@surtur:~/lp/ml/mod-02>ocamlc -o main CharPQueue.cmo main.ml
[17:39]cazzola@surtur:~/lp/ml/mod-02>ls
CharPQueue.cmi  CharPQueue.cmo  CharPQueue.ml  CharPQueue.mli  main*  main.cmi  main.cmo
```

This is how the signature looks:

```
type priority = int          (* still concrete *)
type char_queue              (* now abstract *)
val empty : char_queue
val insert : char_queue -> int -> char -> char_queue
val extract : char_queue -> int * char * char_queue
exception QueueIsEmpty
```


The OCaML Module System
Functors.

Functors are «functions» from structures to structures.

This means

- fixed the signatures of the input and output structures; then
- the implementation details can change without affecting any of the modules that use it.

Functors allow to

- avoid duplication and
- increase orthogonality

in a type safe package.

The OCaML Module System
Functors: an Example.

`is_balanced()` checks that a string uses balanced parenthesis.

```
let is_balanced str =
  let s = Stack.empty in try
    String.iter
      (fun c -> match c with
          '(' -> Stack.push s c
        | ')' -> Stack.pop s
        | _ -> ()) str;
      Stack.is_empty s
  with Stack.EmptyStackException -> false
```

```
module type StackADT =
 sig
  type char_stack
  exception EmptyStackException
  val empty : char_stack
  val push : char_stack -> char ->  unit
  val top :  char_stack -> char
  val pop :  char_stack -> unit
  val is_empty :  char_stack -> bool
end
```

The idea is to iterate on the string and

- to push any open parenthesis on a stack; and
- to pop it when a close parenthesis is encountered

If the algorithm ends with an empty stack the string is balanced otherwise it is unbalanced.

The OCaML Module System
Functors: an Example.

`is_balanced()` checks that a string uses balanced parenthesis.

```
module Matcher (Stack : StackADT.StackADT) =
  struct
    let is_balanced str =
      let s = Stack.empty in try
        String.iter
          (fun c -> match c with
              '(' -> Stack.push s c
            | ')' -> Stack.pop s
            | _ -> ()) str;
          Stack.is_empty s
      with Stack.EmptyStackException -> false
end
```

```
module type StackADT =
 sig
  type char_stack
  exception EmptyStackException
  val empty : char_stack
  val push : char_stack -> char ->  unit
  val top :  char_stack -> char
  val pop :  char_stack -> unit
  val is_empty :  char_stack -> bool
end
```

Matcher is a functor that binds our algorithm to a **Stack** abstract data type.

```
#use "balanced.ml" ;;

module Matcher :
  functor (Stack : StackADT.StackADT) -> sig val is_balanced : string -> bool end
```

Instantiation make concrete the algorithm.

The OCaML Module System
Functors: an Example (Cont'd).

Functors *must* be instantiated.

```
module UnboundedStack = struct
  type char_stack = {
      mutable c : char list
  }
  exception EmptyStackException

  let empty = { c = [] }

  let push s x = s.c <- x :: s.c

  let pop s =
    match s.c with
      hd::tl -> s.c <- tl
    | []      -> raise EmptyStackException

  let top s =
    match s.c with
      hd::_ -> hd
    | []     -> raise EmptyStackException

  let is_empty s = (s.c = [])
end;;
```

```
module BoundedStack = struct
  type char_stack = {
      mutable c: char array;
      mutable top: int }
  exception EmptyStackException

  let empty = {top=0; c=Array.make 10 ' '}
  let push s x =
     s.c.(s.top) <- x; s.top <- s.top+1

  let pop s =
    match s.top with
      0 -> raise EmptyStackException
    | _ -> s.top <- s.top -1

  let top s =
    match s.top with
      0 -> raise EmptyStackException
    | _ -> s.c.(s.top)

  let is_empty s = (s.top = 0)
end;;
```

Both implementations adhere to the **StackADT** interface.

The OCaML Module System
Functors: an Example (Cont'd).

```
[18:15]cazzola@surtur:~/lp/ml>ocaml
# module M0 = Matcher(BoundedStack);;
module M0 : sig val is_balanced : string -> bool end

# module M1 = Matcher(UnboundedStack);;
module M1 : sig val is_balanced : string -> bool end

# M0.is_balanced "a(b)+c(a+d(e-f))";;
- : bool = true
# M0.is_balanced "a(b(+c(a+d(e-f))";;
- : bool = false
# M1.is_balanced "a(b)+c(a+d(e-f))";;
- : bool = true
# M1.is_balanced "a(b(+c(a+d(e-f))";;
- : bool = false
```

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.

Polymorphism in ML

Polymorphic functions and types, type inference, . . .

Walter Cazzola

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

Polymorphism

It permits to handle values of different data types by using a uniform interface.

- A function that can evaluate to or be applied to values of different types is known as a polymorphic function.
- A data type that can appear to be of a generalized type is designated as a polymorphic data type.

OCaML/ML natively supports polymorphism

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

```
[15:34]cazzola@surtur:~/lp/ml>ocaml
# #use "compose.ml" ;;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# compose char_of_int int_of_char ;;
- : char -> char = <fun>
# compose (not) (not) ;;
- : bool -> bool = <fun>
# compose (fun x -> x+1) int_of_char ;;
- : char -> int = <fun>
```


Ad Hoc Polymorphism

- the function/method denotes different implementations depending on a range of types and their combination;
- it is supported in many languages by overloading.

Parametric Polymorphism

- all the code is written without mention of any specific type and thus can be used transparently with any number of new types;
- it is widely supported in statically typed functional programming languages or in object-orientation by generics or templates.

Subtype Polymorphism

- the code employs the idea of subtypes to restrict the range of types that can be used in a particular case of parametric polymorphism;
- in OO languages is realized by inheritance and sub-classing.

OCaML supports parametric polymorphism.

- compose implements f∘g without any type binding;
- its (polymorphic) type is

$$(\alpha \to \beta) * (\gamma \to \alpha) * \gamma \quad \to \quad \beta$$

α, β and γ are type variables denoted by 'a, 'b and 'c respectively;

- the type is inferred from time to time; in compose' the possible values for α and β are restricted to **char** and **int**

```
[17:13]cazzola@surtur:~/lp/ml>ocaml
# let compose f g x = f (g x);;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# let compose' = compose (fun c -> int_of_char c) ;;
val compose' : ('_a -> char) -> '_a -> int = <fun>
```

compose' is weak-typed ('_a).

Polymorphism
Weak Typed

Nothing that is the result of the application of a function to an argument can be polymorphic

– if we don't know yet exactly what is its type, then it's a weak type.

The type `'a -> 'a` means:

– for all type `'a`, this is the type `'a -> 'a`.

Whereas, the type `'_a -> '_a` means:

– there exist one and only one type `'_a` such that this is the type `'_a -> '_a`.

Shall we say that what is potentially polymorphic turns to monomorphic in practice when the compiler deals with its polymorphic form.

```
# let a = ref [];;
val a : '_a list ref = {contents = []}
# let b = 1::!a ;;
val b : int list = [1]
# a;;
- : int list ref = {contents = []}
```


Polymorphism
Walter Cazzola
Polymorphism
introduction
taxonomy
Polymorphism in ML
parametric
weak typed
Type Inference
@ Work
datatype
References

Polymorphism
Type Inference

```
let rec map f l = match l with
  h::l1 -> f h::map f l1
| _ -> [];;
```

Let us calculate the type of map

1. `[]`, `[]` is a zerary function `[]`: $\to \alpha$ **list** $\forall \alpha$;
2. `h::l1`, `::` is a binary operator `:::` $\alpha \times \alpha$ **list** $\to \alpha$ **list** so the type of h is α and the type of l1 is α **list**;
3. the type of f is a function whose input has type α nothing can be said on the return type (denoted by β);
4. so the second occurrence of `::` should be $\beta \times \beta$ **list** $\to \beta$ **list** due to the type of f, that means
5. `map f l1` should have type β **list**

and this is possible only if

6. the type of map is $(\alpha \to \beta) \times \alpha$ **list** $\to \beta$ **list**

```
# #use "map.ml" ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```


Polymorphism
Walter Cazzola
Polymorphism
introduction
taxonomy
Polymorphism in ML
parametric
weak typed
Type Inference
@ Work
datatype
References

Polymorphism @ Work
Polymorphic ADT: Stack

```
module Stack = struct
  type 'a stack = { mutable c : 'a list }
  exception EmptyStackException
  let empty () = { c = [] }
  let push s x = s.c <- x :: s.c
  let pop s =
    match s.c with
      hd::tl -> s.c <- tl
    | []     -> raise EmptyStackException
end;;
```

```
[22:40]cazzola@surtur:~/lp/ml>ocaml
# #use "adtstack.ml";;
# let s = Stack.empty();;
val s : '_a Stack.stack = {c = []}
# Stack.push s 7;;
- : unit = ()
# Stack.push s 25;;
- : unit = ()
# s ;;
- : int Stack.stack = {c = [25; 7]}
# let s1 = Stack.empty();;
val s1 : '_a Stack.stack = {c = []}
# Stack.push s1 "Hello";;
- : unit = ()
# Stack.push s1 "World";;
- : unit = ()
# s1;;
- : string Stack.stack = {c = ["World"; "Hello"]}
```


Polymorphism
Walter Cazzola
Polymorphism
introduction
taxonomy
Polymorphism in ML
parametric
weak typed
Type Inference
@ Work
datatype
References

Polymorphism @ Work
Iterating on Collections

Count the occurrences

```
let rec count ?(tot=0) x = function
  [] -> tot | h::l1 -> if (h==x) then count ~tot:(tot+1) x l1 else count ~tot:tot x l1
```

```
val count : ?tot:int -> 'a -> 'a list -> int = <fun>
# let il = [1;2;3;4;2;2;1;3;4;5;7;3;2;1] ;;
# let cl=['a';'b';'c';'a'];;
# count 'a' cl;;
- : int = 2
# count 3 il;;
- : int = 3
```

Reducing a List

```
let rec remove x = function
  [] -> [] |  h::l1 -> if (h = x) then (remove x l1) else (h::(remove x l1))
```

```
val remove : 'a -> 'a list -> 'a list = <fun>
# remove 3 il;;
- : int list = [1; 2; 4; 2; 2; 1; 4; 5; 7; 2; 1]
# remove 'a' cl;;
- : char list = ['b'; 'c']
```

Iterating on strings

```
let rec iter f ?(k = 0) s =
  if k < String.length s then ( f s.[k] ; iter f ~k:(k + 1) s ) ;;
```

```
val iter : (char -> 'a) -> ?k:int -> string -> unit = <fun>
```

Polymorphism @ Work
Sorting (Quicksort)

```
let qsort (>:) l =
  let rec qsort = function
    [] -> []
  | h::tl -> (qsort (List.filter (fun x -> (x >: h)) tl) )
             @ [h] @
             (qsort (List.filter (fun x -> (h >: x)) tl) )
  in qsort l
```

```
[14:58]cazzola@surtur:~/lp/ml>ocaml
# #use "qsort.ml" ;;
val qsort : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>
# let l=[11; 4; 123; 7; -8; 0; 15; 11; -7; 77; 99; 100; 1; 2; 4; -77] ;;
val l : int list = [11; 4; 123; 7; -8; 0; 15; 11; -7; 77; 99; 100; 1; 2; 4; -77]
# let l'=['a'; 'z'; 'w'; 'b'; 'f'; 'a'; 'x'] ;;
val l' : char list = ['a'; 'z'; 'w'; 'b'; 'f'; 'a'; 'x']
# qsort (>) l ;;
- : int list = [123; 100; 99; 77; 15; 11; 7; 4; 2; 1; 0; -7; -8; -77]
# qsort (<) l ;;
- : int list = [-77; -8; -7; 0; 1; 2; 4; 7; 11; 15; 77; 99; 100; 123]
# qsort (<) l' ;;
- : char list = ['a'; 'b'; 'f'; 'w'; 'x'; 'z']
```

Note

- (>:) represents a binary operator, you can use any sort of symbol
- to avoid to scan the list twice **List**.partition can be used instead of **List**.filter.

Polymorphism @ Work
Sorting (Selection Sort)

```
let lmin (<:) l =
  let rec lmin m = function
    [] -> m
  | h::tl -> lmin (if (m <: h) then m else h) tl
  in lmin (List.hd l) (List.tl l)

let filter_out x l =
  let rec filter_out acc x = function
    [] -> List.rev acc
  | h::tl when h=x -> List.rev_append tl acc
  | h::tl -> filter_out (h::acc) x tl
  in filter_out [] x l

let selection (<:) l =
  let rec selection acc = function
    [] -> List.rev acc
  | l' -> let m = (lmin (<:) l') in selection (m::acc) (filter_out m l')
  in selection [] l
```

```
[10:56]cazzola@surtur:~/lp/ml> ocaml
# let l1 = [-7;1;25;-3;0;15;77;-7] ;;
val l1 : int list = [-7; 1; 25; -3; 0; 15; 77; -7]
# #use "selection.ml";;
val lmin : ('a -> 'a -> bool) -> 'a list -> 'a = <fun>
val filter_out : 'a -> 'a list -> 'a list = <fun>
val selection : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>
# selection (<) l1 ;;
- : int list = [-7; -7; -3; 0; 1; 15; 25; 77]
# selection (>) l1 ;;
- : int list = [77; 25; 15; 1; 0; -3; -7; -7]
```


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.

Playing with Fun

Currying, Map-Filter & Reduce, Folding, . . .

Walter Cazzola

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

Currying & Partial Evaluation

Currying

Currying is a technique to transform a function with multiple arguments into a chain of functions each with a single argument (partial application). E.g.,

$$f(x,y) = \frac{y}{x} \xrightarrow{(2)} f(2) = \frac{y}{2} \xrightarrow{(3)} f(2)(3) = \frac{3}{2}$$

Currying is a predefined techniques in ML.

```
# let f x y z = x+.y*.z;;
val f : float -> float -> float -> float = <fun>
# f 5.;;
- : float -> float -> float = <fun>
# f 5. 3. ;;
- : float -> float = <fun>
# f 5. 3. 7.;;
- : float = 26.
```


Currying & Partial Evaluation

Partial Evaluation

It refers to the process of fixing a number of arguments to a function, producing another function of smaller arity. E.g.,

$$f(x,y) = \frac{y}{x} \xrightarrow{x=2} g(y) = f(2,y) = \frac{y}{2} \xrightarrow{(3)} g(3) = \frac{3}{2}$$

```
let f x y = y/.x ;;
let g = f 2. ;;
```

```
# #use "partial-eval.ml";;
val f : float -> float -> float = <fun>
val g : float -> float = <fun>
# f 2. 3. ;;
- : float = 1.5
# g 3. ;;
- : float = 1.5
```

By using named parameters

```
let compose ~f ~g x = f (g x)
let compose' = compose ~g: (fun x -> x**3.)
```

```
# #use "partial-eval2.ml" ;;
val compose : f:('a -> 'b) -> g:('c -> 'a) -> 'c -> 'b = <fun>
val compose' : f:(float -> 'a) -> float -> 'a = <fun>
# compose ~f:(fun x -> x -. 1.) ~g:(fun x -> x**3.) 2. ;;
- : float = 7.
# compose' ~f:(fun x -> x -. 1.) 2. ;;
- : float = 7.
```


Map, Filter and Reduce

Overview

Map, filter and reduce

- to apply a function to all the elements in the list (map);
- to filter out some elements from the list according to a predicate (filter) and
- to reduce the whole list to a single value according to a cumulative function (reduce).

represent the most recurring programming pattern in functional programming.

Recall, a possible map implementation

```
let rec map f = function
  h::l1 -> f h::map f l1
| _ -> [];;
```

```
# #use "map2.ml";;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# let l = [1; 2; 3; 7; 25; 4] ;;
val l : int list = [1; 2; 3; 7; 25; 4]
# map (fun x-> (x mod 2) == 0) l;;
- : bool list = [false; true; false; false; false; true]
```

Map, Filter and Reduce
Filter


```
let rec filter p = function
  [] -> []
| h::l -> if p h then h :: filter p l else filter p l
```

E.g., to skim odd elements from a list

```
# #use "filter.ml";;
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
# l ;;
- : int list = [1; 2; 3; 7; 25; 4]
# filter (fun x-> (x mod 2) == 0) l;;
- : int list = [2; 4]
```

E.g., to trim the elements greater than or equal to 7.

```
# filter (fun x -> x < 7) l ;;
- : int list = [1; 2; 3; 4]
```


Map, Filter and Reduce
Reduce


```
let rec reduce acc op = function
  []     -> acc
| h::tl -> reduce (op acc h) op tl ;;
```

```
# #use "reduce.ml";;
val reduce : 'a -> ('a -> 'b -> 'a) -> 'b list -> 'a = <fun>
# l ;;
- : int list = [1; 2; 3; 7; 25; 4]
# reduce 0 (+) l;;
- : int = 42
# reduce 1 ( * ) l ;;
- : int = 4200
```

map and reduce can be used to define two predicates on lists:

- exists that returns true if at least one element matches the predicate and

```
# let exists p l = reduce false (||) (map p l);;
val exists : ('a -> bool) -> 'a list -> bool = <fun>
# exists (fun x-> (x mod 2) == 0) l;;
- : bool = true
```

- forall that returns true when all elements match the predicate.

```
# let forall p l = reduce true (&&) (map p l);;
val forall : ('a -> bool) -> 'a list -> bool = <fun>
# forall (fun x-> (x mod 2) == 0) l;;
- : bool = false
```


Map, Filter and Reduce
Folding

Reduce is an example of folding

- i.e., iterating an arbitrary binary function over a data set and build up a return value.
- e.g., in the previous case, we have $((((((0+1)+2)+3)+7)+25)+4)$ (due to tail recursion).

Functions can be associative in two ways (left and right) so folding can be realized

- by combining the first element with the results of recursively combining the rest (right fold), e.g., $0+(1+(2+(3+(7+(25+4)))))$ or
- by combining the results of recursively combining all but the last element, with the last one (left fold).

List provides the functions fold_left and fold_right.

```
# let l = [1.;2.;3.;4.;5.] ;;
val l : float list = [1.; 2.; 3.; 4.; 5.]
# List.fold_right (/.) l 1. ;;
- : float = 1.875
# List.fold_left (/.) 1. l ;;
- : float = 0.00833333333333333322
```


Iterating on Lists
Zip (the longest)

To couple two lists element by element

- all the exceeding elements are dropped.

```
let rec zip_longest l1 l2 =
  match (l1, l2) with
    ([],[]) | (_, []) | ([], _) -> []
  | (h1::l1', h2::l2')          -> (h1,h2)::(zip_longest l1' l2') ;;
```

```
[18:17]cazzola@surtur:~/lp/ml>ocaml
# #use "zip.ml";;
val zip_longest : 'a list -> 'b list -> ('a * 'b) list = <fun>
# let l0 = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10];;
val l0 : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
# let l1 = ['a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'];;
val l1 : char list = ['a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g']
# zip_longest l0 l1 ;;
- : (int * char) list =
[(1, 'a'); (2, 'b'); (3, 'c'); (4, 'd'); (5, 'e'); (6, 'f'); (7, 'g')]
# zip_longest l1 l0;;
- : (char * int) list =
[('a', 1); ('b', 2); ('c', 3); ('d', 4); ('e', 5); ('f', 6); ('g', 7)]
```

It is equivalent to List.assoc.

Iterating on Lists
Group By

To reorganize a list according to a numeric property.

```
type 'a group = { mutable g: 'a list } ;;
let empty_group = function x -> { g = [] } ;;
let rec group_by l ?(ris:'a group array = (Array.init 10 empty_group)) f =
  match l with
    [] -> ris
  | h::l1 ->
      ( ris.((f h)).g <- ris.((f h)).g@[h] ;
        group_by l1 ~ris:ris f );;
```

```
[17:42]cazzola@surtur:~/lp/ml>ocaml
# #use "groupby.ml" ;;
type 'a group = { mutable g : 'a list; }
val empty_group : 'a -> 'b group = <fun>
val group_by : 'a list -> ?ris:'a group array -> ('a -> int) -> 'a group array = <fun>
# let l0 = [10; 11; 22; 23; 45; 25; 33; 72; 77; 16; 30; 88; 85; 99; 9; 1];;
val l0 : int list = [10; 11; 22; 23; 45; 25; 33; 72; 77; 16; 30; 88; 85; 99; 9; 1]
# let l1 = [ "hello"; "world"; "this"; "is"; "a"; "told"; "tale" ];;
val l1 : string list = ["hello"; "world"; "this"; "is"; "a"; "told"; "tale"]
# group_by l0 (fun x -> x/10) ;;
- : int group array =
[|{g = [9; 1]}; {g = [10; 11; 16]}; {g = [22; 23; 25]}; {g = [33; 30]};
  {g = [45]}; {g = []}; {g = []}; {g = [72; 77]}; {g = [88; 85]}; {g = [99]}|]
# group_by l1 String.length ;;
- : string group array =
[|{g = []}; {g = ["a"]}; {g = ["is"]}; {g = []}; {g = ["this"; "told"; "tale"]};
  {g = ["hello"; "world"]}; {g = []}; {g = []}; {g = []}; {g = []}|]
```


Iterating on Lists
Miscellaneous

To pairwise couple the elements of a list.

```
(* l -> (l0,l1), (l1,l2), (l2, l3), ...*)
let rec pairwise = function
    h'::h''::l' -> (h',h'')::pairwise (h''::l')
  | _ -> []
```

```
#  #use "pairwise.ml";;
val pairwise : 'a list -> ('a * 'a) list = <fun>
# let l1= ['a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'; 'h'; 'i'];;
val l1 : char list = ['a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'; 'h'; 'i']
# pairwise l1;;
- : (char * char) list =
[('a','b'); ('b','c'); ('c','d'); ('d','e'); ('e','f'); ('f','g'); ('g','h'); ('h','i')]
```

To enumerate the elements of a list.

```
let enumerate l =
  let rec enumerate acc n = function
     h :: ls -> enumerate ((n,h)::acc) (n+1) ls
  |  []      -> List.rev acc
in enumerate [] 0 l
```

```
# #use "enumerate.ml";;
val enumerate : 'a list -> (int * 'a) list = <fun>
# enumerate ['a'; 'b'; 'c'] ;;
- : (int * char) list = [(0, 'a'); (1, 'b'); (2, 'c')]
```


Advance on Functions
Functions with a Variable Number of Arguments


```
let arg x = fun y rest -> rest (op x y) ;;
let stop x = x;;
let f g = g init;;
```

```
[12:12]cazzola@surtur:~/lp/ml>ocaml
# let op = fun x y -> x+y;;
val op : int -> int -> int = <fun>
# let init = 0;;
val init : int = 0
# #use "varargs.ml";;
val arg : int -> int -> (int -> 'a) -> 'a = <fun>
val stop : 'a -> 'a = <fun>
val f : (int -> 'a) -> 'a = <fun>
# f (arg 1) stop;;
- : int = 1
# f (arg 1) (arg 2) stop;;
- : int = 3
# f (arg 1) (arg 2) (arg 7) (arg 25) (arg (-1)) stop;;
- : int = 34
# let op = fun x y -> y @ [x] ;;
val op : 'a -> 'a list -> 'a list = <fun>
# let init = [] ;;
val init : 'a list = []
# #use "varargs.ml";;
val arg : 'a -> 'a list -> ('a list -> 'b) -> 'b = <fun>
val stop : 'a -> 'a = <fun>
val f : ('a list -> 'b) -> 'b = <fun>
# f (arg 1) (arg 2) (arg 7) (arg 25) (arg (-1)) stop;;
- : int list = [1; 2; 7; 25; -1]
# f (arg "Hello") (arg "World") (arg "!!!") stop ;;
- : string list = ["Hello"; "World"; "!!!"]
```


Advance on Functions
Functor for Functions with a Variable Number of Arguments

Previous approach need to be reloaded every time you need a different kind for f

- removing the previous instantiation.

To implement a functor will solve the issue, we need

- an abstract data type (**OpVarADT**)

```
module type OpVarADT =
 sig
  type a and b and c
  val op: a -> b -> c
  val init : c
 end
```

- the functor (**VarArgs**)

```
module VarArgs (OP : OpVarADT.OpVarADT) =
  struct
    let arg x = fun y rest -> rest (OP.op x y) ;;
    let stop x = x;;
    let f g = g OP.init;;
  end
```

- and few concrete implementations for the ADT

```
module Sum = struct
  type a=int and b=int and c=int
  let op = fun x y -> x+y ;;
  let init = 0 ;;
end
```

```
module StringConcat = struct
  type a=string and b=string list and c=string list
  let op = fun (x: string) y -> y @ [x] ;;
  let init = [] ;;
end
```

```
[16:00]cazzola@surtur:~/lp/ml>ocaml
# #use "OpVarADT.mli";;
module type OpVarADT =
  sig type a and b and c val op : a -> b -> c val init : c end
# #use "sum.ml";;
module Sum :
  sig
    type a = int
    and b = int
    and c = int
    val op : int -> int -> int
    val init : int
  end
# #use "concat.ml" ;;
module StringConcat :
  sig
    type a = string
    and b = string list
    and c = string list
    val op : string -> string list -> string list
    val init : 'a list
  end
# #use "varargs.ml" ;;
module VarArgs :
  functor (OP : OpVarADT.OpVarADT) ->
    sig
      val arg : OP.a -> OP.b -> (OP.c -> 'a) -> 'a
      val stop : 'a -> 'a
      val f : (OP.c -> 'a) -> 'a
    end
```


Advance on Functions
Functor for Functions with a Variable Number of Arguments
Playing with Fun
Walter Cazzola
Playing with Fun
currying
partial evaluation
map+reduce
iteration
var args
References

```
# module M0 = VarArgs(StringConcat) ;;
module M0 :
  sig
    val arg :
      StringConcat.a -> StringConcat.b -> (StringConcat.c -> 'a) -> 'a
    val stop : 'a -> 'a
    val f : (StringConcat.c -> 'a) -> 'a
  end
# module M1 = VarArgs(Sum) ;;
module M1 :
  sig
    val arg : Sum.a -> Sum.b -> (Sum.c -> 'a) -> 'a
    val stop : 'a -> 'a
    val f : (Sum.c -> 'a) -> 'a
  end
# M1.f (M1.arg 1) (M1.arg 2) (M1.arg 7) (M1.arg 25) (M1.arg (-1)) M1.stop;;
- : Sum.c = 34
# M1.f (M1.arg 1) (M1.arg 2) (M1.arg 7)  M1.stop;;
- : Sum.c = 10
# M0.f (M0.arg "Hello") (M0.arg "World") (M0.arg "!!!") M0.stop ;;
- : StringConcat.c = ["Hello"; "World"; "!!!"]
```


Advance on Functions
Functor for Functions with a Variable Number of Arguments
Playing with Fun
Walter Cazzola
Playing with Fun
currying
partial evaluation
map+reduce
iteration
var args
References

How to instantiate **OpVarADT** with a generic list?

- a generic type as `'a list` cannot match the signature **OpVarADT** since none of the types are defined as parametric; and
- an abstract type in an implementation, even if it matches the signature, has no definition at all

```
module ListConcat = struct
  type a and b = a list and c = a list
  let op = fun (x: a) y -> y @ [x] ;;
  let init = [] ;;
end
```

```
# #use "listc.ml" ;;
module ListConcat :
  sig
    type a
    and b = a list
    and c = a list
    val op : a -> a list -> a list
    val init : 'a list
  end
#  module M2 = VarArgs(ListConcat) ;;
module M2 :
  sig
    val arg : ListConcat.a -> ListConcat.b -> (ListConcat.c -> 'a) -> 'a
    val stop : 'a -> 'a
    val f : (ListConcat.c -> 'a) -> 'a
  end
# M2.f (M2.arg "Hello") (M2.arg " ") (M2.arg "World") (M2.arg "!!!") M2.stop ;;
Error: This expression has type string but an expression was expected of type ListConcat.a
```


Advance on Functions
Functor for Functions with a Variable Number of Arguments
Playing with Fun
Walter Cazzola
Playing with Fun
currying
partial evaluation
map+reduce
iteration
var args
References

If you cannot use parametrized type

- you can use module language to add parametrization, by making the (**ListConcat**) module a functor over a type

```
module ListConcatFunctor (T : sig type t end) = struct
  type a = T.t and b = a list and c = a list
  let op = fun (x: a) y -> y @ [x] ;;
  let init = [] ;;
end
```

```
# #use "ListConcatFunctor.ml";;
module ListConcatFunctor :
  functor (T : sig type t end) ->
    sig
      type a = T.t and b = a list and c = a list
      val op : a -> a list -> a list
      val init : 'a list
    end
# module M3 = VarArgs(ListConcatFunctor(struct type t = int end));;
module M3 : sig
    val arg : int -> int list -> (int list -> 'a) -> 'a
    val stop : 'a -> 'a
    val f : (int list -> 'a) -> 'a
  end
# module M4 = VarArgs(ListConcatFunctor(struct type t = string end)) ;;
module M4 : sig
    val arg : string -> string list -> (string list -> 'a) -> 'a
    val stop : 'a -> 'a
    val f : (string list -> 'a) -> 'a
  end
# M3.f (M3.arg 2) (M3.arg 3) (M3.arg 4) M3.stop;;
- : int list = [2; 3; 4]
# M4.f (M4.arg "Hello") (M4.arg "World") M4.stop;;
- : string list = ["Hello"; "World"]
```

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.

ML in Action

Graph Coverage

Walter Cazzola

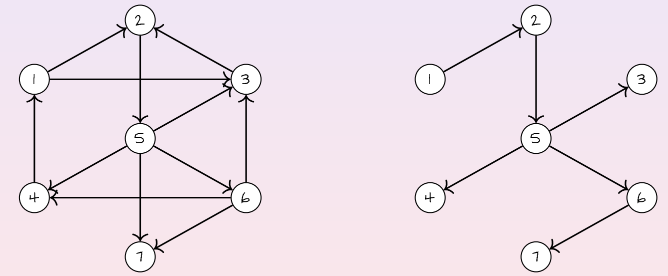
Dipartimento di Informatica
Università degli Studi di Milano
e-mail: cazzola@di.unimi.it
twitter: @w_cazzola

Depth First Search (DFS)

Problem Definition

Depth First Search

- is an algorithm for traversing graph starting from a given node and exploring as far as possible along each branch before backtracking.



Note,

- DFS depends on how out edges are ordered (in the case above they are sorted by value).
- we focus on acyclic direct graphs

Depth First Search (DFS)

Abstract Datatypes

To solve the problem we need:

- a tree datatype to represent the result of the visit

```
type 'a tree = Leaf of 'a | Tree of ('a * 'a tree list);;
```

- a graph datatype to support the obvious needing

```
module type GraphADT =
  sig
   type 'a graph

   val empty : unit -> 'a graph

   val add_node : 'a -> 'a graph -> 'a graph
   val add_arc : 'a -> 'a -> 'a graph -> 'a graph
   val adjacents : 'a -> 'a graph -> 'a list
   val node_is_in_graph : 'a -> 'a graph -> bool
   val is_empty : 'a graph -> bool

   exception TheGraphIsEmpty
   exception TheNodeIsNotInGraph
  end;;
```


Depth First Search (DFS)

Graph Implementation

```
module Graph : GraphADT =
 struct
    type 'a graph = Graph of ( 'a list ) * ( ( 'a * 'a ) list )

    let empty() = Graph([], [])
    let is_empty = function
      Graph(nodes, _) -> (nodes = [])

    exception TheGraphIsEmpty
    exception TheNodeIsNotInGraph

    (* checks if an element belongs to the list *)
    let rec is_in_list ?(res=false) x = function
      []     -> res
    | h::tl -> is_in_list ~res: (res || (x=h)) x tl

    (* checks if a node is in the graph *)
    let node_is_in_graph n = function
      Graph(nodes, _) -> is_in_list n nodes
      ...
  end
```

Depth First Search (DFS)

Graph Implementation (Follows)

```
(* adds an element to a list if not present *)
let rec add_in_list ?(res=[]) x = function
  []    -> List.rev x::res
| h::tl when (h=x) -> List.rev_append tl (h::res)
| h::tl            -> add_in_list ~res: (h::res) x tl

(* operations to add new nodes and arcs (with their nodes) to the graph, respectively *)
let add_node n = function
  Graph( [], [] )       -> Graph( [n], [] )
| Graph( nodes, arcs ) -> Graph( (add_in_list n nodes), arcs )

let add_arc s d = function
  Graph(nodes, arcs) ->
     Graph( (add_in_list d (add_in_list s nodes)), (add_in_list (s,d) arcs) )

(* returns the nodes adjacent to the given node *)
let adjacents n =
  let adjacents n l = List.map snd (List.filter (fun x -> ((fst x) = n)) l)
in function
  Graph(_, arcs) -> adjacents n arcs
```


Depth First Search (DFS)

Ancillary Operations on Graphs

```
open Graph
(* transforms a list of arcs in a graph *)
let arcs_to_graph arcs =
  let rec arcs_to_graph g = function
    []         -> g
  | (s,d)::tl -> arcs_to_graph (add_arc s d g) tl
  in arcs_to_graph (empty()) arcs

(* extract a tree out of acyclic graph with the given node as the root *)
let graph_to_tree g root =
  let rec make_tree n = function
    [] -> Leaf(n)
  | adj_to_n -> Tree(n, (make_forest adj_to_n))
  and make_forest = function
    [] -> []
  | hd::tl -> (make_tree hd (adjacents hd g))::(make_forest tl)
  in make_tree root (adjacents root g)
```


Depth First Search (DFS)

DFS Implementation

```
open Graph

let dfs g v =
  let rec dfs g v g' = function
    []      -> g'
  | hd::tl when (node_is_in_graph hd g') -> dfs g v g' tl
  | hd::tl -> dfs g v (add_arc v hd (dfs g hd (add_node hd g') (adjacents hd g))) tl
  in
    if (is_empty g) then raise TheGraphIsEmpty
    else if not (node_is_in_graph v g) then raise TheNodeIsNotInGraph
         else graph_to_tree (dfs g v (add_node v (empty())) (adjacents v g)) v
```


Depth First Search (DFS)

DFS in Action

```
[18:08]cazzola@surtur:~/lp/ml>ocaml
# #use "tree.ml";;
type 'a tree = Leaf of 'a | Tree of ('a * 'a tree list)
# #use "GraphADT.mli";;
module type GraphADT =
  sig
    type 'a graph
    val empty : unit -> 'a graph
    val add_node : 'a -> 'a graph -> 'a graph
    val add_arc : 'a -> 'a -> 'a graph -> 'a graph
    val adjacents : 'a -> 'a graph -> 'a list
    val node_is_in_graph : 'a -> 'a graph -> bool
    val is_empty : 'a graph -> bool
    exception TheGraphIsEmpty
    exception TheNodeIsNotInGraph
  end
# #use "Graph.ml" ;;
module Graph : GraphADT
# #use "aux.ml" ;;
val arcs_to_graph : ('a * 'a) list -> 'a Graph.graph = <fun>
val graph_to_tree : 'a Graph.graph -> 'a -> 'a tree = <fun>
# #use "dfs.ml" ;;
val dfs : 'a Graph.graph -> 'a -> 'a tree = <fun>
# let g1 = arcs_to_graph [(1,2);(1,3);(4,1);(5,4);(3,2);(2,5);(5,3);(5,6);(5,7);(6,7);(6,3);(6,4)] ;;
val g1 : int Graph.graph = <abstr>
# let g7 = arcs_to_graph([("Algol", "Pascal"); ("Algol", "C"); ("Algol", "Java"); ("C", "Java");
  ("Algol", "Python"); ("Pascal", "Modula 2"); ("C", "C++"); ("Java", "Scala"); ("Lisp", "ML");
  ("Lisp", "Scala"); ("Lisp", "Python"); ("Lisp", "Erlang"); ("ML", "OCaML")]);;
val g7 : string Graph.graph = <abstr>
# dfs g1 1 ;;
- : int tree = Tree (1, [Tree (2, [Tree (5, [Leaf 4; Leaf 3; Tree (6, [Leaf 7])])])])
# dfs g7 "Algol";;
- : string tree = Tree ("Algol", [Tree ("Pascal", [Leaf "Modula 2"]);
   Tree ("C", [Tree ("Java", [Leaf "Scala"]); Leaf "C++"]); Leaf "Python"])
# dfs g7 "Lisp" ;;
- : string tree = Tree ("Lisp", [Tree ("ML", [Leaf "OCaML"]); Leaf "Scala"; Leaf "Python"; Leaf "Erlang"])
```

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.