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.,

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

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.,

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

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

- forall that return true when all the 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 function 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 a

- an astract data type (OptVarADT)

```
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.