Playing with Fun
Currying, Map-Filter & Reduce, Folding,...

Walter Cazzola
Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano
e-mail: cazzola@dico.unimi.it

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.,

\[ f(x, y) = \frac{y}{x} \]
\[ f(2) = \frac{2}{2} \]
\[ f(2)(3) = \frac{3}{2} \]

Currying is a predefined technique in ML.

```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.;
```

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

\[ f(x, y) = \frac{y}{x} \]
\[ g(y) = \frac{y}{2} \]
\[ g(3) = \frac{3}{2} \]

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

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).

```
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]
```

By using named parameters

```
let compose ~f ~g x = f (g x) ;
let compose' = compose ~g: (fun x -> x -. 1.) ;
```

```
# use "partial-eval2.ml" ;
val compose : f:('a -> 'b) -> g:('c -> 'a) -> 'c -> 'b = <fun>
val compose' : f:(float -> 'a) -> float -> 'a = <fun>
```

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>
# f = [1; 2; 3; 7; 25; 4];
val f : int list = [1; 2; 3; 7; 25; 4]
# map f ;
- : int list = [false; true; false; false; false; true]
Playing with Fun
Walter Cazzola

References

Slide 5 of 16

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

# let filter = fun p l -> reduce false (or) (map p l);
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
# 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]

Reduce

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

# l
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
# reduce 0 (+) l
- : int = 55
# reduce 1 ( * ) l
- : int = 3628800

map and reduce can be used to define two predicates on lists:
- exists that returns true if at least one element matches the predicate

# 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

Slide 6 of 16

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

# l = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
val l : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
# 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 = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10];
val l1 : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]

zip_longest l0 l1
- : (int * int) list = [(1, 1); (2, 2); (3, 3); (4, 4); (5, 5); (6, 6); (7, 7); (8, 8); (9, 9); (10, 10)]

zip_longest l1 l0
- : (int * int) list = [(1, 1); (2, 2); (3, 3); (4, 4); (5, 5); (6, 6); (7, 7); (8, 8); (9, 9); (10, 10)]

Slide 7 of 16

Iterating on Lists
Zip (the longest)

To couple two lists element by element
- all the exceeding elements are dropped

let rec zip_longest l l' = match (l, l') with
| ([], []) | (_, []) | ([], _) | (_, []) -> []
| (h::l', h2::l2') -> (h,h2)::(zip _longest l' l2')

val zip_longest : 'a list -> 'b list -> ('a * 'b) list = <fun>

it is equivalent to List.assoc.
# Group by

To reorganize a list according to a numeric property.

```ocaml
type 'a group = { mutable g : 'a list; }

let empty_group = function
  | [] -> ris
  | h::hl ->
    (* rix.(h)(h0) = rix.(h0)(h1) = g(h1); group_by hl ~ris:ris (f g) (h h0) ;
    group_by hl ~ris:ris f ;;

let pairwise_l = function
  | (h::l1) -> (h, h1) :: pairwise_l (h1 :: l1)
  | [] -> []

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

```ocaml
# pairwise_l [1; 2; 7; 25; -1];;
- : int list = [1; 2; 7; 25; -1]
# enumerate [1; 2; 7; 25; -1];;
- : int group array = [{g = [1]}; {g = [25]}; {g = [-1]}; {g = [2]}; {g = [7]}]
```

---

# Advance on Functions

## Functions with a Variable Number of Arguments

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

### Example

```ocaml
# f (arg "Hello") (arg "World") (arg "!!!") stop ;;
- : string list = ["Hello"; "World"; "!!!"]
```

---

# Advance on Functions

## Function for Functions with a Variable Number of Arguments

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

To implement a functor will solve the issue, we need a

- an abstract data type (OptVarADT)
- a functor (VarArgs)
- few concrete implementations for the ADT

## Module OptVarADT

```ocaml
module type OptVarADT = struct
  type a and b and c
  let init = c
  let op: a -> b -> c
end
```

## Module VarArgs

```ocaml
module VarArgs (OP : OptVarADT.OpVarADT) = struct
  let op = fun a b c -> op a b c
end
```

## Module StringConcat

```ocaml
module StringConcat = struct
  type a=string and b=string list and c=string list
  let init = [] ;
  let op = fun x y -> x @ [y] ;
end
```
Functor for Functions with a Variable Number of Arguments

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.

```ocaml
module M0 = VarArgs(ListConcat) ;;
# module M0 = VarArgs(ListConcat) ;;
```

If you cannot use parametrized type, you can use module language to add parametrization by making the (ListConcat) module a functor over a type.

```ocaml
module ListConcatFunctor (T : sig type t end) ->
  sig type a and b and c val op : a * b -> c val init : c end
```

```ocaml
module ListConcatFunctor (T : sig type t end) ->
  sig type a and b and c val op : a * b -> c val init : c end
```
References

- Davide Ancona, Giovanni Lagorio, and Elena Zucca.
  Linguaggi di Programmazione.
  Città Studi Edizioni, 2007.

- Greg Michaelson.
  An Introduction to Functional Programming through \(\lambda\) Calculus.
  Addison-Wesley, 1989.

- Larry C. Paulson.
  ML for the Working Programmer.