# 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

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

$\alpha, \beta$ and $\gamma$ are type variables denoted by 'a, 'b and 'c respectively;

- the type is inferred from time to time; in compose' the possible values for $\alpha$ and $\beta$ 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
## 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 `[]`: $\rightarrow \alpha$ **list** $\forall\alpha$;
2. `h::l1`, `::` is a binary operator `:::` $\alpha \times \alpha$ **list** $\rightarrow \alpha$ **list** so the type of h is $\alpha$ and the type of `l1` is $\alpha$ **list**;
3. the type of f is a function whose input has type $\alpha$ nothing can be said on the return type (denoted by $\beta$);
4. so the second occurrence of `::` should be $\beta \times \beta$ **list** $\rightarrow \beta$ **list** due to the type of f; that means
5. `map f l1` should have type $\beta$ **list**

and this is possible only if

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

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





# 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 @ 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)

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

```
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

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