Polymorphism in ML

Polymorphic functions and types, type inference, ...

Walter Cazzola

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

Polymorphism

Introduction

Polymorphism allows values of different data types to be handled 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 supports polymorphism

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

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>

Polymorphism Taxonomy

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 \( f \circ g \) without any type binding;
- its (polymorphic) type is

\[
(\alpha \to \beta) \ast (\gamma \to \alpha) \ast \gamma \to \beta
\]

\( \alpha, \beta \) 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;

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 its 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 and when the compiler deals with its polymorphic form.

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

Polymorphism @ Work

Polymorphic ADT: Stack

```ocaml
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;
```

```ocaml
[22:40] cazzola@surthur:/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

```ocaml
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>
```

```ocaml
# let il = [1;2;3;4;2;2;1;3;4;5;7;2;1] ;;
# let cl=['a';'b';'c';'a'];;
# count 'a' cl;;
- : int = 2
# count 3 il;;
- : int = 3
```

```ocaml
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>
```

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

```ocaml
let rec iter f ?(k = 0) s =
  if k < String.length s then
    f s.[k] ; iter f ~k:(k + 1) s
  else
    ();
val iter : (char -> 'a) -> ?k:int -> string -> unit = <fun>
```

Polymorphism @ Work

Reducing a List

Count the Occurrences

```ocaml
let rec count ?(tot=0) x = function
  [] -> tot | h::l1 -> if (h=x) then count -tot:(tot+1) x 11 else count -tot:tot x 11
val count : ?tot:int -> 'a -> 'a list -> int = <fun>
```

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

Reducing a List

```ocaml
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>
```

```ocaml
# remove 'a' cl;;
- : char list = ['b'; 'c']
```

Iterating on strings

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

Note
- (>:) represents a binary operator, you can use any sort of symbol.

References