

2023-10-10 - Polimorfismo, Tipi deboli, Applicazioni parziali, Parametri di default

Polimorfismo

Il polimorfismo permette di gestire valori di **diversi tipi** sfruttando un'**interfaccia** comune.

Esistono sia *funzioni* che *tipi di dato* polimorfi:

- una **funzione polimorfa** può prendere in *input* o generare un *output* un tipo di dato polimorfo
- un **tipo di dato polimorfo** può apparire come una *generalizzazione* di esso

Esistono diversi tipi di polimorfismo:

- **Polimorfismo ad Hoc**: una funzione fornisce *diverse implementazioni* dipendenti da diverse combinazioni di *tipi in ingresso* (implementato da molti linguaggi attraverso l'*overloading*)
- **Polimorfismo parametrico** (*implementato da OCaml*): il codice è scritto *senza menzionare* nessun tipo specifico, che viene *inferito* dall'utilizzo
- **Polimorfismo per sottotipazione**: vengono supportati i *sottotipi*, che vengono accettati dove viene accettato il tipo padre (implementato nei linguaggi OOP tramite *ereditarietà*)

Polimorfismo in OCaml

Il polimorfismo *parametrico* è implementato *nativamente* in OCaml.

Un dato è *polimorfo* fino a quando non **viene utilizzato**. A quel punto il suo tipo viene **inferito dall'utilizzo** e tutta l'espressione viene *tipizzata*.

```
let identity x = x;;
val identity : 'a -> 'a = <fun> (* il parametro è di tipo generico, alpha *)
identity 5;;
- : int = 5 (* il parametro è di tipo int, quindi anche il risultato sarà int *)

let succ x = x+1;;
val succ : int -> int = <fun>
(* dato che viene effettuata un operazione tra interi su x,
allora anche x (il parametro) sarà intero *)
```

Tipi deboli (weak types)

Niente che è il **risultato dell'applicazione di una funzione** ad un argomento può essere **polimorfo**. Semplicemente dipende dal **tipo di dato in ingresso**, ma è **strettamente legato** a questo, **non** è davvero *polimorfo*.

Questo meccanismo prende il nome di **tipo debole** (*weak type*).

In pratica il compilatore non è **ancora** riuscito ad **inferire** il tipo di quel dato, ma a **runtime** lo **saprà** sicuramente.

Si notano tipi deboli soprattutto attraverso applicazioni parziali di funzioni.

Applicazioni parziali

Un'**applicazione parziale** di una funzione è l'applicazione di **non tutti i parametri** necessari a valutare il *risultato* della funzione. In questo modo verranno **chiusi** (*chiusure* - *closures*) nella funzione i parametri passati, lasciano **liberi** quelli mancanti.

```
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>
```

La funzione *compose* prende come parametri due funzioni (*f*, *g*) ed un valore (*x*).

compose' è un'applicazione parziale di *compose*, viene associato al parametro *f* di *compose* (*fun c -> int_of_char c*), lasciando liberi *g* ed *x*. Quindi *compose'* sarebbe *int_of_char (g x)*, con parametri *g* ed *x*.

Parametri di default

È possibile definire dei **parametri di default** nella dichiarazione di una funzione attraverso la *sintassi* `?(name=value)`. In caso si voglia **specificare il parametro**, allora sarà necessario usare in *modo esplicito* il nome del parametro `~name:value`.

```
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;;
```

Internamente la funzione `count` passa *esplicitamente* il parametro `tot` (`~tot:(tot+1)`), mentre esternamente è possibile *ometterlo*, passando solo `x` e la lista (anonima grazie alla sintassi `function`).