Esempio moduli in OCaml

October 28, 2021

1 Tipi di dato astratti in OCaml

1.1 Esempio di uso dei moduli

OCaml consente di definire tipi di dato astratti tramite il meccanismo dei moduli.

Vediamo la definizione di un tipo di dato astratto `BOOL` che imita il funzionamento dei booleani, con valori `yes` e `no` e un'operazione `choose` simile ad un `if`.

Il tipo di dato astratto sarà definito dalla sua **segnatura** (o **firma**, o **signature**). La segnatura elenca le componenti del tipo di dato astratto (valori e operazioni) con i loro tipi, in modo parametrico rispetto al tipo della sua rappresentazione (`t` in questo esempio) che sarà istanziato successivamente, in fase di implementazione.

[27]:
```
module type BOOL = sig
    type t
    val yes: t
    val no: t
    val choose: t -> 'a -> 'a -> 'a
end;;
```

[27]:
```
module type BOOL =
  sig type t val yes : t val no : t val choose : t -> 'a -> 'a -> 'a end
```

La segnatura di per sè non è utilizzabile, ma descrive precisamente quello che il tipo di dato **è** e **può fare** (che è tutto quello che serve a un programmatore per poterlo usare). Non viene invece descritta la rappresentazione del tipo, ossia la struttura dati (di tipo `t`) usata per implementare il tipo definito. Questa informazione non è necessaria al programmatore che dovrà utilizzare questo nuovo tipo di dato.

Un'**implementazione** di un tipo di dato astratto deve andare a definire la rappresentazione del dato e le funzioni che realizzano le operazioni, come specificato dalla segnatura. Spesso, usando meccanismi di compilazione separata, l'implementazione è in un file separato e non è visibile al programmatore che la utilizza.

Ad esempio, il modulo `M1` implementa il tipo di dato astratto `BOOL` usando un valore di tipo `unit option` come rappresentazione.

[28]:
```
module M1 : BOOL = struct
    type t = unit option
    let yes = Some ()
    let no = None
    let choose v ifyes ifno =
        match v with
        | Some () -> ifyes
        | None -> ifno
end ;;
```

[28]:
```
module M1 : BOOL
```

Data un'implementazione, il programmatore la può utilizzare secondo quanto specificato dalla segnatura, usando la dot notation per riferire ai nomi definiti nel modulo. Si noti che l'interprete nasconde la rappresentazione del valore (indicando `<abstr>` al posto di, ad esempio, `Some ()`).

[30]:
```
let x = M1.yes ;;
let y = M1.no ;;
M1.choose x 3 4 ;;
```

[30]:
```
val x : M1.t = <abstr>
```

[30]:
```
val y : M1.t = <abstr>
```

[30]:
```
- : int = 3
```

E' possibile specificare più di una implementazione per lo stesso tipo di dato astratto. Implementazioni diverse tipicamente useranno rappresentazioni diverse, con vantaggi diversi. Ad esempio, se stiamo descrivendo il tipo di dato astratto di un insieme di interi, in certi casi potrà essere conveniente usare come rappresentazione interna una lista di interi (per rendere efficiente l'inserimento di nuovi elementi) mentre in altri casi potrà essere più conveniente usare un albero (per rendere più efficiente la ricerca di un elemento).

In questo esempio, implementiamo una versione di `BOOL` in cui i valori sono rappresentati come interi.

[31]:
```
module M2 : BOOL = struct
    type t = int
    let yes = 1
    let no = 0
    let choose b ifyes ifno =
        if b=1 then
            ifyes
        else
```

```
            ifno
end ;;
```

[31]:
```
module M2 : BOOL
```

Le due implemtazioni di `BOOL` sono del tutto equivalente dal punto di vista dell'uso, in quanto implementano lo stesso tipo di dato astratto. Corrispondono però a tipi concreti diversi, e i loro valori non sono confrontabili.

[33]:
```
let x = M1.yes ;;
let y = M2.yes ;;
x = y ;;
```

[33]:
```
val x : M1.t = <abstr>
```

[33]:
```
val y : M2.t = <abstr>
```

```
File "[33]", line 3, characters 4-5:
3 | x = y ;;
        ^

Error: This expression has type M2.t but an expression was expected of type
         M1.t
```

Ultima osservazione, tramite la direttiva `open` è possibile dire una volta per tutte che si utilizzerà un certo modulo, e questo consentira di usare tutti i nomi in esso definiti senza bisogno di usare la dot notation (cioè, `open` importa il *name space* del modulo).

[35]:
```
open M2 ;;
let x = yes;;
choose x 3 4 ;;
```

[35]:
```
val x : M2.t = <abstr>
```

[35]:
```
- : int = 3
```

In questo modo, cambiando sostituendo semplicemente `M1` con `M2` nella direttiva `open`, il programma continua a funzionare sebbene ovunque verrà utilizzata l'altra implementazione del tipo di dato astratto.