Tipi Algrebrici

Definire nuovi tipi

In OCaml è possibile definire nuovi tipi di dato tramite il costrutto `type`

```
type data = int*int*int ;;
```

```
type data = int * int * int
```

```
let controlla_data (d:data) =
    match d with
     | gg,mm,aaaa -> gg>0 && gg<=31
                                && mm>0 && mm<=12
                                && aaaa>1900 && aaaa<=2030 ;;
```

```
val controlla_data : data -> bool = <fun>
```

```
let accoda_data (d:data) lis =
    if controlla_data(d) then d::lis else lis ;;
```

```
val accoda_data : data -> data list -> data list = <fun>
```

Tuple

Il prodotto cartesiano di due insiemi (es. $N \times N$) è un insieme di coppie.

$$(7,4) \in N \times N$$

Allo stesso modo il prodotto di due tipi (es. int * int) è un tipo di coppie

`(7,4) : int * int`

Record

I record sono un modo più avanzato di definire tipi prodotto, infatti possiamo dare dei nomi agli elementi, detti *campi*

```
type punto_2d = { x: float; y: float; } ;;
```

```
type punto_2d = { x : float; y : float; }
```

```
let p = { x = 3.; y = -4. } ;;
```

```
val p : punto_2d = {x = 3.; y = -4.}
```

Un valore di tipo record può essere decomposto usando il *pattern matching*
Ad esempio, calcoliamo il quadrante del piano cartesiano in cui ricade un punto

```
let quadrante {x = x_pos; y = y_pos } =
    match x_pos>=0.,y_pos>=0. with
     | true,true -> 1
     | false,true -> 2
     | false,false -> 3
     | true,false -> 4 ;;

quadrante {x=(-3.); y=2.};;
```

```
val quadrante : punto_2d -> int = <fun>
- : int = 2
```

Record *vs* Oggetti

I record assomigliano ai dizionari di JavaScript, e si potrebbe pensare che "assegnando" funzioni ai loro elementi quello che si ottiene siano oggetti (come accadrebbe in JavaScript).

```
type finto_oggetto = { n: int; update: int -> int } ;;
```

```
type finto_oggetto = { n : int; update : int -> int; }
```

La differenza con gli oggetti è che ***i record sono immutabili***

Inoltre i campi funzione (metodi) non possono accedere agli altri campi (variabili d'istanza)

```
let obj = { n = 10; update = fun x -> x } ;;
obj.n = 11 ;; (* non posso assegnare... *)
```

viene propagato un errore:

```
File "[17]", line 1, characters 0-11:
1 | obj.n <- 11 ;;
      ^^^^^^^^^^^
Error: The record field n is not mutable
```

Tipi Unione

Mentre in alcuni linguaggi (ad es. TypeScript) si possono unire direttamente due tipi, scrivendo *ad esempio:*

```
type t = string | int
```

l'operazione di unione prevista da OCaml prevede di etichettare i valori dei tipi da unire
Ad esempio, per unire i tipi
`string` e `int`, dobbiamo scegliere due etichette (ad esempio `Txt` e `Num`) e usarle in questo modo:

```
type numero_testo =
     | Txt of string
     | Num of int ;;
```

```
type numero_testo = Txt of string | Num of int
```

In sostanza, ogni riga della definizione corrisponde a un caso possibile di tipo di valore, ed è caratterizzato da una etichetta distinta (detta
costruttore)

> 📌 I costruttori devono necessariamente iniziare con la lettera maiuscola

```
let x = Txt "34" ;;
let y = Num 26 ;;
```

Dato un valore, è possibile ricostruire a quale dei tipi appartenga, tra quelli che sono stati uniti, usando i costruttori nel pattern matching

```
match x with
     | Txt s -> "testo"
     | Num n -> "numero" ;;
```

```
- : string = "testo"
```

Si possono definire casi che consistono solo del costruttore, usando i soli costruttori, si possono definire tipi enumerazione (*In molti linguaggi chiamati enum*)

```
type giorno = Lun | Mar | Mer | Gio | Ven | Sab | Dom ;;
```

```
type giorno = Lun | Mar | Mer | Gio | Ven | Sab | Dom
```

```
let is_weekend g =
    match g with
     | Sab | Dom -> true
     | _ -> false ;;
```

```
val is_weekend : giorno -> bool = <fun>
```

Costruttori che descrivono casi particolari

- `NaN` (sta per Not a Number) è il "numero" che si ottiene da un'operazione tra interi che restituisce un risultato non definito
- `Plus_inf` rappresenta $+\infty$
- `Minus_inf` rappresenta $-\infty$

```
type int_ext =
    | Num of int
    | NaN
    | Plus_inf
    | Minus_inf ;;
```

```
type int_ext = Num of int | NaN | Plus_inf | Minus_inf
```

Operazioni aritmetiche tramite questi casi particolari:

```
let sum x y =
    match x,y with
     | NaN,_ | _,NaN -> NaN
     | Plus_inf,Minus_inf | Minus_inf,Plus_inf -> NaN
     | Plus_inf,_ | _,Plus_inf -> Plus_inf
     | Minus_inf,_ | _,Minus_inf -> Minus_inf
     | Num n1,Num n2 -> Num (n1+n2) ;;
```

Tipi opzione e tipi polimorfi

Consente di specificare un valore che può essere assente e usa i costruttori `Some` e `None`

`Some` prende un valore e lo avvolge nell'opzione

```
Some 4;;
```

```
- : int option = Some 4
```

`None` che rappresenta l'assenza di un valore

```
None ;;
```

```
- : 'a option = None
```

```
let x = Some 42 (* x è un'opzione che contiene il valore 42 *)
let y = None    (* y è un'opzione che rappresenta l'assenza di un valore *)
```

I tipi opzione consentono di definire funzioni che restituiscono None in casi particolari

```
let rec massimo lis =
    match lis with
     | [] -> None
     | x::lis' -> match massimo lis' with
                                    | None -> Some x
                                    | Some max -> if x>max then Some x
                                                                    els
```

```
 val massimo : 'a list -> 'a option = <fun>
```

Tipi record e variant insieme

Si possono creare tipi strutturati combinando a piacere `record`, `variant`, `tuple`, *ecc...*
Si mostra in seguito una combinazione possibile:

```
type punto_multidimensionale =
    | DueDim of {x: float; y: float; }
    | TreDim of {x: float; y: float; z: float; }
```

```
type punto_multidimensionale =
    DueDim of { x : float; y : float; }
     | TreDim of { x : float; y : float; z : float; }
```

```
let p1 = DueDim {x=10.; y=10.} ;;
let p2 = TreDim {x=4.; y=6.; z=8.} ;;
```

```
val p1 : punto_multidimensionale = DueDim {x = 10.; y = 10.}
val p2 : punto_multidimensionale = TreDim {x = 4.; y = 6.; z = 8.}
```

Tipi ricorsivi

Sono tipi variant definiti in termini di se stessi

```
type lista_di_int =
    | Nil
    | Elem of int * lista_di_int ;;

let lst = Elem (3 ,Elem (4, Elem (6,Nil))) ;;
```

```
type lista_di_int = Nil | Elem of int * lista_di_int
val lst : lista_di_int = Elem (3, Elem (4, Elem (6, Nil)))
```

Una lista di interi è fatta di elementi di tipo int concatenati l'uno all'altro. Ogni elemento, quindi, deve prevedere anche un "riferimento" all'elemento successivo.

L'ultimo elemento della lista avrà riferimento nullo (o meglio, a un elemento nullo), in quanto non esiste un successivo.

Nel codice precedente abbiamo due casi:

- ***La lista è vuota***: possiamo rappresentarla con un singolo elemento nullo che rappresenteremo con il costruttore `Nil`
- La lista non è vuota: allora esiste un primo elemento che rappresenteremo con il costruttore Elem. Tale elemento prevederà un intero e un riferimento all'elemento successivo.

L'elemento successivo, però, altri non è che il primo elemento di un'altra lista: la lista di tutti gli elementi che seguono l'elemento corrente.

Qundi possiamo vedere il riferimento al prossimo elemento come un riferimento ad un'altra lista dello stesso tipo di quella che stiamo descrivendo.

Per questo motivo al costruttore `Elem` è associato il tipo `int * lista_di_int`

☝🏻 La concatenazione degli elementi sarà realizzata tramite annidamento di coppie

Ora in modo ricorsivo possiamo anche definire una funzione analoga all'operatore `::`

```
let cons x lis =
    Elem (x,lis) ;;

cons 5 lst;;
```

```
val cons : int -> lista_di_int -> lista_di_int = <fun>
- : lista_di_int = Elem (5, Elem (3, Elem (4, Elem (6, Nil))))
```

e possiamo definire anche altre operazioni utilizzando lo schema di `lista_di_int`, ad esempio:

```
let rec somma_lista lis =
    match lis with
     | Nil -> 0
     | Elem (x,lis') -> x + somma_lista lis' ;;
```

Alberi in OCaml

Grazie ai tipi ricorsivi è possibile dunque definire gli alberi

```
type albero_bin =
    | Nodo of int*albero_bin*albero_bin
    | Foglia of int ;;
```

```
type albero_bin = Nodo of int * albero_bin * albero_bin | Foglia of int
```

Operazioni su Alberi

Visita

```
let rec visita_ant a =
    match a with
     | Foglia v -> [v]
     | Nodo (v,sx,dx) -> v::((visita_ant sx)@(visita_ant dx)) ;;
```

```
val somma_albero : albero_bin -> int = <fun>
```

Somma

```
let rec somma_albero a =
    match a with
     | Foglia v -> v
     | Nodo (v,sx,dx) -> v + somma_albero sx + somma_albero dx ;;
```

```
val somma_albero : albero_bin -> int = <fun>
```

Abstract Syntax Tree

Rappresentano la struttura sintattica di un programma/espressione/termine appartenente a un certo linguaggio

Esempio: *Le espressioni aritmetiche su interi*

```
Exp ::= n | Exp op Exp | -Exp | (Exp)
op ::= + | - | * | / | %
```

La rappresentazione come albero di sintassi astratta si basa sulla definizione di un tipo variant con un costruttore per ogni produzione (caso) della grammatica in formato BNF.

```
type op = Add | Sub | Mul | Div | Mod ;;

type exp =
    | Val of int
    | Op of op*exp*exp
    | UMin of exp ;;
```

```
type op = Add | Sub | Mul | Div | Mod
type exp = Val of int | Op of op * exp * exp | UMin of exp
```

Ogni costruttore è associato a un tipo che rappresenta le informazioni espresse dalla corrispondente produzione nella grammatica. Ad esempio, il costruttore `Val`, che rappresenta la produzione `Exp ::= n` è associato al tipo `int`

Con gli
AST si può ad esempio creare una funzione che trasforma l'AST di una espressione in una stringa

```
let symbol o =
    match o with
     | Add -> "+" | Sub -> "-" | Mul -> "*" | Div -> "/" | Mod -> "%" ;;

let rec to_string e =
    match e with
     | Val n -> string_of_int n
     | Op (o,e1,e2) -> "("^ (to_string e1) ^ (symbol o) ^ (to_string e2) ^"
     | UMin e' -> "(-"^(to_string e')^")" ;;
```

```
val symbol : op -> string = <fun>
val to_string : exp -> string = <fun>
```

oppure una funzione che valuta un espressione:

```
let rec eval e =
    match e with
     | Val n -> n
     | Op (o,e1,e2) ->
        (match o with
          | Add -> (eval e1) + (eval e2)
          | Sub -> (eval e1) - (eval e2)
          | Mul -> (eval e1) * (eval e2)
            | Div -> (eval e1) / (eval e2)
            | Mod -> (eval e1) mod (eval e2)
        )
     | UMin e' -> - (eval e') ;;
```

```
val eval : exp -> int = <fun>
```