TIPI DI DATO

Abbiamo visto che OCaml offre vari tipi di dato

- int
- float
- char
- string
- bool
- ⋮

} tipi primitivi

- $\alpha, \beta, \ldots$ } variabili di tipo $\Longrightarrow$ polimorfismo

$id : \alpha \to \alpha$

- $T \to S$
- T list
- T option

} costruttori di tipo

Altri tipi di dato interessanti:

- record
- tuple
- tipi di dato algebrici
- ⋮

Record

```
type studente = {
    nome : string
    matricola : int
}
```

definizione di tipo (cf. dichiarazione)

campi

{nome : string , matricola : int}

int → bool

bool list

type <id> = T

```
let george = {
    nome = "george boole"
    matricola = 01
}
```

⇝ val george : studente

george.matricola ⇝ 01 : int

Sintassi

tipo record: $\{ \text{campo}_1 : T_1 \,;\, \dots \,;\, \text{campo}_n : T_n \}$

identificatori

espressioni: $t ::= \dots \mid \{ \text{campo}_1 = t_1 \,;\, \dots \,;\, \text{campo}_n = t_n \} \mid e.\text{campo}$

identificatori

valori: $v ::= \dots \mid \{ \text{campo}_1 = v_1 \,;\, \dots \,;\, \text{campo}_n = v_n \}$

Statica

$$\frac{t_1 : T_1 \quad \dots \quad t_n : T_n}{\{c_1 = t_1; \dots; c_n = t_n\} : \{c_1 : T_1; \dots; c_n : T_n\}}$$

Dinamica

$$\frac{t_1 \Longrightarrow v_1 \quad \dots \quad t_n \Longrightarrow v_n}{\{c_1 = t_1; \dots\ c_n = t_n\} \Longrightarrow \{c_1 = v_1; \dots; c_n = v_n\}}$$

$$\frac{t \Longrightarrow \{c_1 : v_1; \dots; c_n : v_n\}}{t.c_i \Longrightarrow v_i}$$

NB. I record sono immutabili

```
type persona = {
    nome : string
    eta : int
}

val pippo = {
    name = "pippo"
    eta = " 99"
}
```

Non posso fare $\boxed{\text{pippo.}\overset{\text{eta}}{\text{name}} = \text{pippo.}\overset{\text{eta}}{\text{name}} + 1}$

Non posso aggiungere campi $\longrightarrow$ cambiamento tipo

Possiamo pensare ad un record come ad una forma basica ed immutabile di classe

match v with
{capacita; add} → ...

```
type calcolatrice = {
    capacita : int            → "campi"
    add: int → int → int option   → "metodi"
}

val calc1 = {
    capacita = 100
    add = fun n m →
              let p = n+m
              in  if p < 100
                  then some p
                  else none
}
```

Possiamo chiamare

calc1.add 10 10

(⤳ some 20)

NB. Non possiamo aggiornare la capacità

Tuple

Le tuple sono record i cui campi sono dati dalla posizione nella tupla

$\Downarrow$

ordine è cruciale (irrilevante nel record)

```
type punto = float * float
let zero = (.0, .0)
let plus_1 (p: punto) : punto =
    match p with
      (x,y) -> (x+.1., y+.1.)
```

possiamo usare pattern matching anche per i record

float * int * bool * string

$T_1 * \dots * T_n$

$(x_1, \dots, x_n) \rightarrow \dots$

```
let zero = (.0, .0)

let plus_1 (p:punto) : punto =
    match p with
      | (x,y) -> (x+.1., y+.1.)
```

```
plus_1 zero

  -> match (.0, .0) with
         | (x,y) -> (x +. 1., y +. 1.)

  -> (.0 +. 1., .0 +. 1)

  -> (.1, .1)
```

Un tipo particolare di tupla sono le coppie

$T_1 * T_2$

Data $p : T_1 * T_2$, possiamo usare

$p.fst$

$p.snd$

per accedere agli elementi della coppia

Domanda. Possiamo pensare a fst, snd come funzioni su coppie:
qual è il loro tipo

$fst : \alpha * \beta \rightarrow \alpha$

Domanda. Possiamo pensare al tipo tupla $int * string * bool$
come zucchero sintattico per un tipo coppia?

$(int * string) * bool$
$\cong ?$
$int * (string * bool)$

Sintassi

tipi: $T ::= \dots \mid T_1 * \dots * T_n$

$T * (S * U) \cong (T * S) * U$

espressioni: $t ::= \dots \mid (t_1, \dots, t_n) \mid \text{match } t \text{ with } (p_1, \dots, p_n) \to t$

$v ::= \dots \mid (v_1, \dots, v_n)$

Semantica.

$$\frac{t_1 : T_1 \quad \dots \quad t_n : T_n}{(t_1, \dots, t_n) : T_1 * \dots * T_n}$$

$$\frac{t_1 \Rightarrow v_1 \quad \dots \quad t_n \Rightarrow v_n}{(t_1 \dots t_n) \Rightarrow (v_1 \dots v_n)}$$

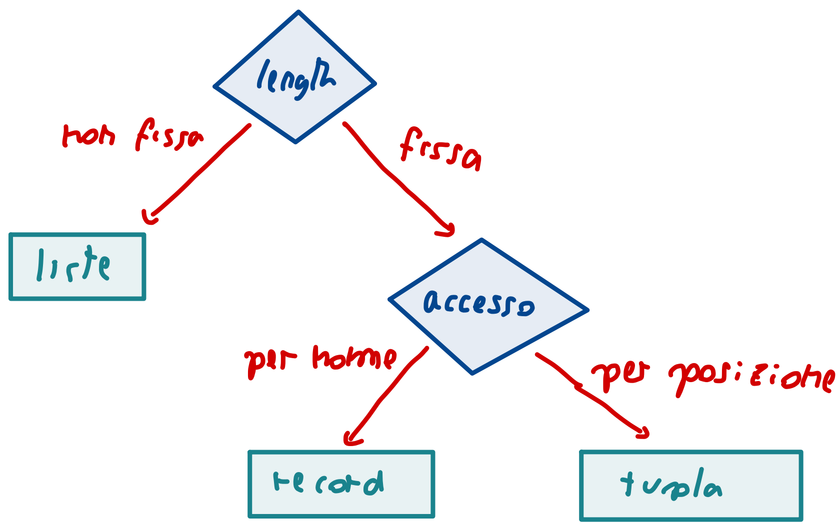
Match (v_i, p_i) ✓

$$\frac{t \Rightarrow \overbrace{(v_1 \dots v_n)} \qquad s[v_1/p_1 \dots v_n/p_n] \Rightarrow w}{\text{match } t \text{ with } \underbrace{(p_1 \dots p_n)} \to s \Rightarrow w}$$

match (2, 3) with

((x, y), z) $\rightarrow$ 5

Liste, Record, Tuple



VARIANTI (VARIANTS)

Possiamo usare i tipi varianti per enumerare le costanti di un determinato tipo

$$\text{type colore\_primario} = \underbrace{\text{Rosso} \mid \text{Blu} \mid \text{Verde}}_{\text{stiamo dicendo che tutti i valori di questo tipo sono delle costanti, e le elenchiamo}}$$

lettera maiuscola

let a = Red ⇝ val a : colore_primario

Creiamo un variante per figure geometriche

```
type point = float * float
type figure =
  | Circle of point * float
  | Rectangle of point * point
```

Costruttore

(| Circle of {center : point; radius : float})

informazione aggiuntiva

tutti e soli i valori di tipo figure sono della forma
Circle p r oppure Rectangle p q

Domanda Quanti valori contiene primary-color?

E figure?

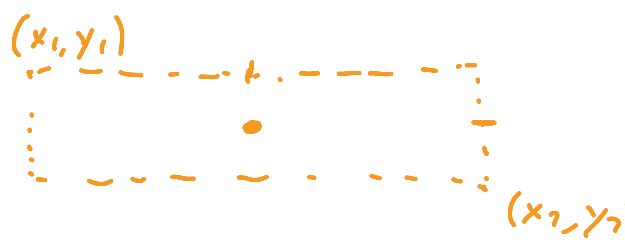
Possiamo utilizzare espressioni di tipo variante usando il

pattern matching

```
let centro (f : figura) : point =
   match f with
    | Circle p r  -> p
    | Rectangle p q ->
        let (x1,y1) = p  in          -> pattern matching
        let (x2,y2) = q  in
        let media = fun a b -> a +. b / 2.  in

        (media x1 x2 , media y1 y2)
```

```
match p with
| (x1,y1) -> match q with
              | (x2,y2) ->
                 ...
```



Qual è il significato intuitivo di un tipo variante?

```
type figure =
  | Circle of point * float
  | Rectangle of point * point
```

A∪B

Stiamo dicendo che un valore di tipo figure è, di fatto,

un valore point * float oppure un valore point * point

⟹ abbiamo una sorta di unione di tipi

I costruttori fungono da tag e fanno sì che $\underbrace{\text{l'unione sia disgiunta}}_{\text{somma di tipi } T+S}$

```
type figure =
  | Rectangle of point * point
  | Square of point * point
```

I tipi varianti sono chiamati anche **tipi algebrici**, essendo <u>somme di prodotti</u> (o record)

$$\text{figure} \approx \underbrace{(\text{point} * \text{float})}_{\text{Circle}} + \underbrace{(\text{point} * \text{point})}_{\text{Rectangle}} + \underbrace{(\text{point} * \text{point})}_{\text{Square}}$$

SINTASSI

tipi $\quad T ::= \ldots \mid c_1 \; [\text{of } T_1] \mid \ldots \mid c_n \; \underbrace{[\text{of } T_n]}$

↳ costruttori

opzionale; diciamo allora che c_n è una <u>costante</u>

<u>NB</u>. OCaml consente l'utilizzo di tipi di dato algebrici tramite type definitions

type <identifier> = c_1 of T_1 | … | c_n of T_n

↳ utile per ricorsione (vedremo dopo)

espressioni $\quad t ::= \ldots \mid c \mid c\ t \mid \text{match } t \text{ with } p_1 \to t_1 \mid \ldots \mid p_n \to t_n$

valori $\quad v ::= \ldots \mid c \mid c\ v$

STATICA

$$\frac{e : T_i \qquad [\text{type } T = C_1 \text{ of } T_1 \mid \ldots C_n \text{ of } T_n]}{C_i\, e : C_1 \text{ of } T_1 \mid \ldots C_n \text{ of } T_n \quad [T]}$$

DINAMICA

$$\frac{e \Downarrow v}{C\, e \Downarrow C\, v}$$

$$\frac{e \Downarrow C\, v \qquad \text{Match}(v, p) \qquad s[v/p] \Downarrow w}{\text{case } e \text{ of } \ldots \mid \underbrace{C\, p}_{} \to s \mid \;\; \Downarrow w}$$

estendiamo la grammatica dei pattern con

$p ::= \ldots \mid C\,p \mid \ldots$

Tipi Algebrici Ricorsivi

```
type intlist =
    | Nil
    | Cons of int * intlist
```

} Definisce i valori di tipo intlist induttivamente

Valori di tipo intlist sono

Nil, Cons (1, Nil), Cons (1, Cons (2, Nil)), ...

[ ] [1] [1,2]

```
let rec length (x: intlist) : int =
    match x with
    | Nil -> 0
    | Cons (n, ns) -> 1 + length ns
```

Possiamo astrarre ancora

```
type α mylist =
    | Nil
    | Cons α * α mylist
```

è esattamente il tipo α list

type α list = [ ] | (::) of α * α list

```
let rec length (α mylist) : int =
    match x with
      | Nil -> 0
      | Cons(y,ys) -> 1 + length ys
```

Esempi

Numeri Naturali

```
type nat = zero | succ of nat
```

Corrisponde alla definizione induttiva di $\mathbb{N}$

- $0 \in \mathbb{N}$
- $n \in \mathbb{N} \longrightarrow n+1 \in \mathbb{N}$
- Nient'altro è elemento di $\mathbb{N}$

$$\frac{}{\text{zero} : \text{nat}} \qquad \frac{n : \text{nat}}{\text{succ } n : \text{nat}}$$

```
let rec int_of_nat (n: nat) : int =
    match n with
      | zero -> 0
      | succ n -> 1 + (int_of_nat n)
```

Esercizio   Scrivere mat_of_int

Opzioni

```
type α option = None | Some of α
```

Un valore di tipo α option è:

- None ⇝ computazione fallita
- Some x ⇝ computazione riuscita con risultato x

```
let rec nat_of_int : int → nat option = fun n →
        if n ≥ 0
           then match n with
                    | 0 → Some zero
                    | n → Succ (nat_of_int (n-1)) ?
                              Some (Succ nat_of_int (n-1) ) ?
           else None
```

```
 ⟶  n ⟶ let res = nat_of_int (n-1)          : nat option
         in match res with
              | None -> None
              | Some p -> Some (Succ p)
```

Ripuliamo il codice

```
let apply (x : α option) (f : α -> β option) =
    match x with
      | None -> None
      | Some y -> f y

let rec nat-of-int n =
    if n < 0
      then None
      else match n with
          | 0 -> Some Zero
          | n -> apply (nat-of-int n-1) (fun x -> Some (succ x))
```

MODULARITÁ

In programmazione funzionale i programmi sono, di fatto, funzioni

Questo consente di modularizzare il codice

||

scomporre programma complesso in (tanti) programmi **semplici**

$$\underbrace{F}_{\text{prog. complesso}} = \underbrace{F_n \circ F_{n-1} \circ \dots \circ F_0}_{\text{composizione programmi semplici}}$$

let $F_1\ x_1 \dots x_n = \boxed{}$;;

$\vdots$

let $F_m\ y_1 \dots y_m = \boxed{}$;;

let $F\ z_1 \dots z_k = \boxed{F_1(\dots) \ \cdots \ F_m(\dots)}$ composizione di funzioni

Questo funziona se i tipi corrispondono

$$\frac{f : T \to S \qquad g : S \to U}{g \circ f : T \to U}$$

Domanda. Se ho un programma

$$f : T \to S \text{ option}$$

"un programma di tipo $f : T \to S$ che può fallire"

e uno

$$g : S \to U \text{ option}$$

Posso comporli ?

```
let bind : (α -> β option) -> (α option -> β option) =
    fun f -> fun x -> match x with
                      | None -> None
                      | Some y -> f y
```

```
let comp (f : α → β option) (g : β → γ option) : α → γ option =
    (bind g) ∘ f
```

Altri utili combinatori per option

```
let unit : α → α option = fun x → Some x
let option_map : (α → β) → (α option → β option) =
        fun f → bind (f comp unit)
```

```
f comp g = fun x → g (f x)
           (or fun x → x |> f |> g
            or fun x → g @@ f @@ x
```

MONADI

Costruttori di tipo M tali che abbiamo programmi

$unit : \alpha \to \alpha\, M$

$bind : (\alpha \to \beta\, M) \to (\alpha\, M \to \beta\, M)$ (e quindi anche map)

sono dette monadi, e sono una utile astrazione per modellare effetti computazionali

Esercizi

① Dimostrare che list è una monade

② Monade di stato. Dimostrare che

$$T\,S\varepsilon = S \to S * T$$

con S tipo fissato è una monade