LISTE in CAML

Titolo nota 26/11/2015

Rappresentazione di liste

$[x_1; x_2; x_3; \ldots; x_n]$ è una lista di n elementi (tutti dello stesso tipo)

ed è una abbreviazione per questa costruzione

$(x_1 :: (x_2 :: (x_3 :: \ldots (x_{n-1} :: (x_n :: [])) \ldots)))$ [] è la costante polimorfa che rappresenta la lista vuota

:: 'a * 'a list → 'a list

Scriviamo una funzione

null : 'a list → bool calcola true se la lista è vuota, false altrimenti

```
let null l = if l = [] then true else false ;;
null : 'a list -> bool = <fun>
```

Scriviamo una funzione che data una lista (non vuota) restituisce il primo elemento della lista

```
let head l = if l = x::xs then x else ?
```

È una funzione PREDEFINITA in CAML

```
hd : 'a list → 'a
```

è indefinita se il suo argomento è la lista vuota

È predefinita una funzione

```
tl : 'a list → 'a list
```

restituisce la lista senza il primo elemento

PATTERN MATCHING

```
let foo l = if (hd l) mod 2 = 0 then true else false
foo : int list → bool = <fun>
```

dice true se il 1° elemento della lista è pari, false altrimenti

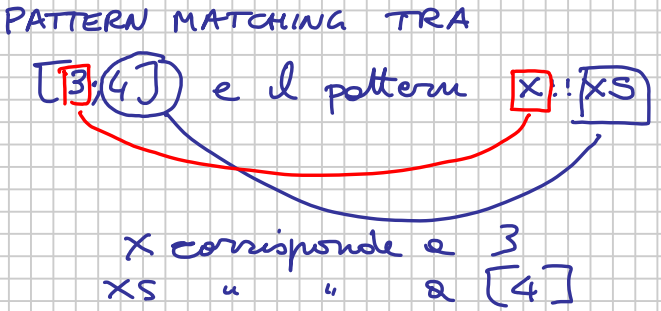
```
let foo l = match l with
    x::xs -> x mod 2 = 0 ;;
```

x sta per il 1° elemento di l
xs " " " la coda di l

pattern (modello) che corrisponde ad una lista NON vuota
il modello attribuisce il nome x al primo elemento della lista
e il nome xs alla coda della lista

Come avviene la valutazione di foo ?

```
  foo [3;4]
= {p.m.  x=3   xs=[4]}
  3 mod 2 = 0
=
  false
```



PATTERN MATCHING

```
match espressione with
  pattern1 -> espressione 1 |
  pattern2 -> espressione 2 |
  ...
  patternn -> espressionen
```

definizione per casi

un nome utilizzato qui ha senso solo nell'espressione 2 (corrispondente al pattern)

Ogni caso corrisponde ad un pattern

```
match l with
  []      -> ∅ |
  x :: xs -> 1 |
  x :: y :: xs -> 2
```

INUTILE

```
match l with
  []          -> ∅ |
  x :: y :: xs -> 2 |
  x :: []      -> 1
```

[x]

Funzione che seleziona il 2° elemento di una lista

```
let second l = match l with
    x :: y :: xs -> y ;;
```

second : 'a list → 'a = <fun>

('a list = l, 'a = ris.)

```
let secondo l = hd (tl l) ;;
```

secondo : 'a list → 'a = <fun>

second [4;1;8;25]

= { p.m. x=4, y=1, xs=[8;25] }

1

second [2]

= { p.m. x=2, fallimento, non riusciamo a matchare y con un elemento }

errore

second [3;8] (sta per 3 :: 8 :: [])

= { p.m. x=3, y=8, xs=[] }

8

second []

= { p.m. fallimento }

errore

DOMANDA

```
let f l = match l with
    []          -> ∅   |   <- int
    x::[]       -> ∅   |   <- int
    x::y::xs    -> y/1 ;;  <- int quindi y deve avere tipo int
                           poiché y è un elemento di l, 'a è int
```

f : ~~'a~~ int list → int = <fun>

(tipo l: int list; tipo ris.: int)

f : 'a list → int = <fun>

Osservazioni sui pattern

`x::y::xs -> y`

`_::y::_ -> y`

sono oggetti di tipo 'a

è un oggetto di tipo 'a list

perché?

`::` : 'a -> 'a list

quando una parte del pattern non viene utilizzata nella parte destra (espressione) può essere rimpiazzata dal pattern speciale "_"

Il pattern speciale "_" matcha con qualunque valore.

```
let null l = match l with
    []  -> true |
    _   -> false ;;
```

```
let null_sbagliata l = match l with
    _   -> false |
    []  -> true ;;
```

DOMANDA

Come faccio a scrivere una funzione come "second" ma che abbia tipo

int list → int

Più in generale, come faccio a scrivere una funzione che "riconosce" il tipo di una lista? Cioè ad esempio che calcola true se l'argomento è una lista di interi, false altrimenti.

```
let secondI l = match l with
    _ :: y :: _ -> y + 0 ;;
```

```
secondI : 'a list -> int = <fun>
```

'a → int

(prefix +) : int -> int -> int
(prefix +.) : float -> float -> float

```
let liste_interi l = match l with
    x :: _ -> if (x > 0 or x <= 0) then true else ??
```

??? non si può !!

FUNZIONI RICORSIVE su LISTE

Calcoliamo la lunghezza di una lista

len [3;4;1;5] = 4 len [] = ∅ len ['a'] = 1

```
let rec len l = match l with
    []    ->  ∅  |
    x::xs ->  1 + (len xs) ;;

len : 'a list -> int = <fun>
```

```
let rec len l = if l=[] then ∅
                else 1 + len (tl l) ;;
```

```
len [3;4;1]
= { p.m. xs = [4;1] }
  1 + (len [4;1])
= { p.m. xs = [1] }
  1 + 1 + len [1]
= { p.m. xs = [] }
  1 + 1 + 1 + (len [])
= { p.m. 1° pattern }
  1 + 1 + 1 + ∅
= 3
```

Funzione che selesiona il PENULTIMO elemento di una lista

DOMANDA

```
let rec penultimo l = match l with
      [x;y]          ->  x      |
    _::y::z::xs      ->  penultimo y::z::xs
```

(penultimo (tl l))

tl l

penultimo [4;5;8;-1]

= { p.m. 2° patt. y=5, z=8, xs=[-1] }

penultimo [5;8;-1]

= { p.m. 2° patt. y=8, z=-1, xs=[] }

penultimo [8;-1]

= { p.m. 1° pattern x=8 y=-1 }

8

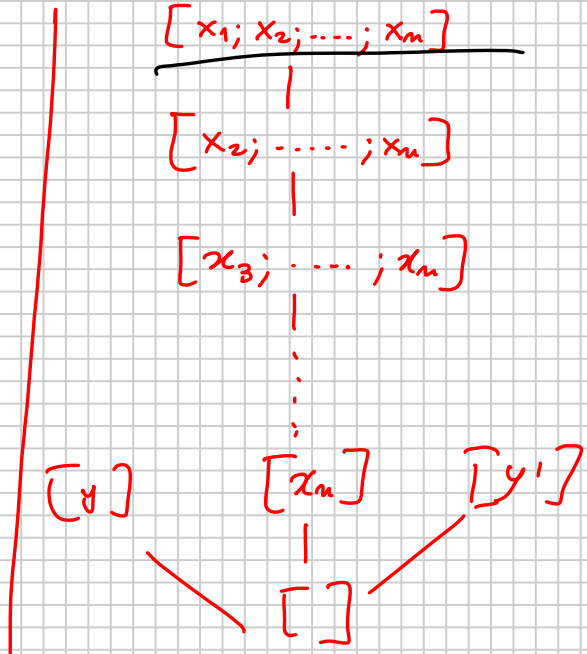
INDUZIONE BEN FONDATA

```
let rec len l = if l = [] then 0 else 1 + len (tl l) ;;
len : 'a list -> int = <fun>
```

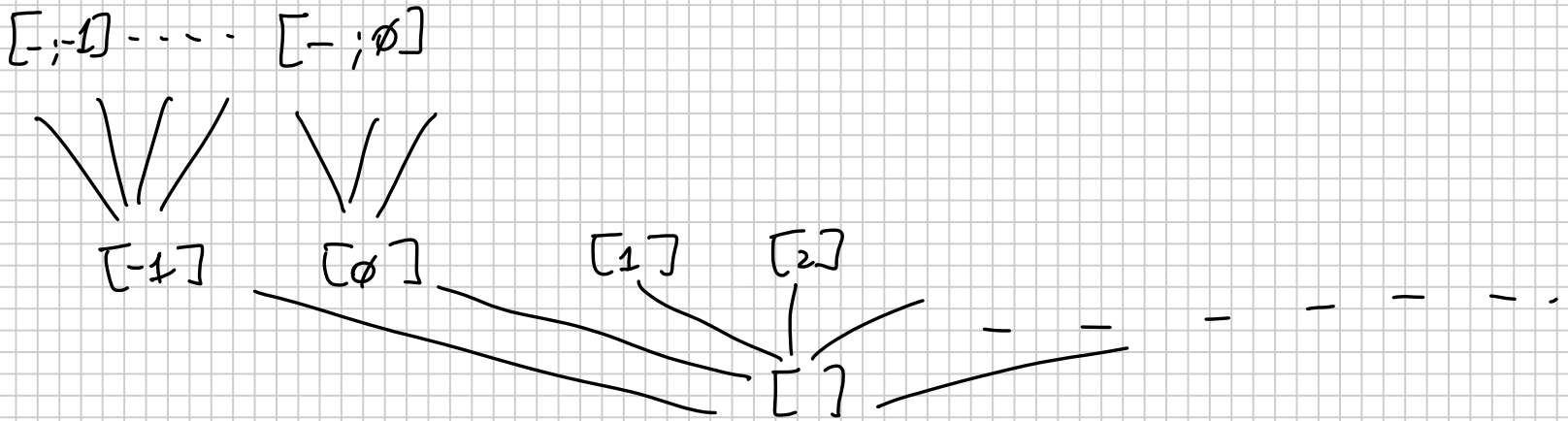
$\forall l, l' \in$ 'a list . $l \prec_{len} l' \equiv l = tl\ l'$

$\forall l, l' \in$ 'a list . $l <_{len} l' \equiv l' = x :: xs \wedge l = xs$

```
let rec len l = match l with
    []      -> 0
  | x::xs   -> (len xs) + 1 ;;
```



Relazione di precedenza "istruzione" su liste di interi



CONCATENAZIONE di LISTE

$[x_1; \ldots; x_n] \cdot [y_1; \ldots; y_k] = [x_1; \ldots; x_n; y_2; \ldots; y_k]$

Esiste un operatore PREDEFINITO @ (append) infisso

```
[1;2;3] @ [4;5;6] = [1;2;3;4;5;6]
```

```
@ : 'a list -> 'a list -> 'a list
```

```
let rec append l l' = match l with
    []    ->  l'   |
  x::xs   ->  x::(append xs l') ;;
```

```
append : 'a list -> 'a list -> 'a list = <fun>
```

ESERCIZIO: definire $\prec_{append}$ sul dominio 'a list * 'a list