Liste CAML

Titolo nota 24/09/2015

Sequenze di valori dello stesso tipo

- lista vuota []
- le liste si costruiscono con l'operatore cons (::)

:: ha come primo argomento un elemento di lista
secondo argomento una lista
il risultato è la nuova lista che ha in testa il
primo argomento di cons e come lista rimanente il
secondo argomento di cons $3 :: \underbrace{[4;5]} = [3;4;5]$

notazione per le liste

Titolo nota 24/09/2015

```
# [];;
- : 'a list = []

# [3;4];;
- : int list = [3;4]
```

liste di interi

```
# 3 :: [4];;
- : int list = [3;4]

# 'a' :: [];;
- : char list = ['a']
```

$x :: [\,] = [x]$

```
# ('a', 3) :: [];;
- : (char * int) list = [('a', 3)]

# ('a',3) :: [(5, 'b')];;
```

char * int (int * char) list

errore di tipo

```
# true :: [3;4];;
```

errore di tipo

Titolo nota 24/09/2015

```
# [2] :: [[-2;3];[]] ;;

- : int list list = [ [2]; [-2;3]; [] ]

# [[]] ;;
- : 'a list list = [[]]
```

$\underbrace{[2]}_{\text{int list}}$ :: $\underbrace{[[-2;3];[]]}_{\text{int list list}}$

$\underbrace{\text{'a list}}$ list

```
# let f n = n+1
  in [f];;
- : (int -> int) list = [<fun>]

# let g n = n+1
  and f n = n+2
  in [f;g];;
- : (int -> int) list = [<fun>;<fun>]
```

Titolo nota 24/09/2015

Altre operazioni su liste (oltre alla costante [] e al costruttore di valori ::) predefinite

hd head (testa)
tl Tail (coda)

head dà il primo elemento di una lista (hd)
tail dà la lista senza il primo elemento (Tl)

```
# hd;;
- : 'a list -> 'a = <fun>
```

```
# Tl;;
- : 'a list -> 'a list = <fun>
```

```
# hd [3;4];;
- : int = 3

# Tl [3;4];;
- : int list = [4]
```

undefinite su liste vuote

```
# hd [];;
undefined
# Tl [];;
undefined
```

funzione ricorsiva che presa una lista mi dà la sua lunghezza (numero degli elementi)

len [] = ∅

len [3;4] = 2

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

```
let rec len l = if l = [] then 0
                else 1 + len (Tl l);;
```

$\sqsubset$

len [3;4];;

= {def len, ramo else}

1 + len [4]

= {def len, ramo else}

1 + 1 + len []

= {def len, ramo then}

1 + 1 + 0

= {calcolo}

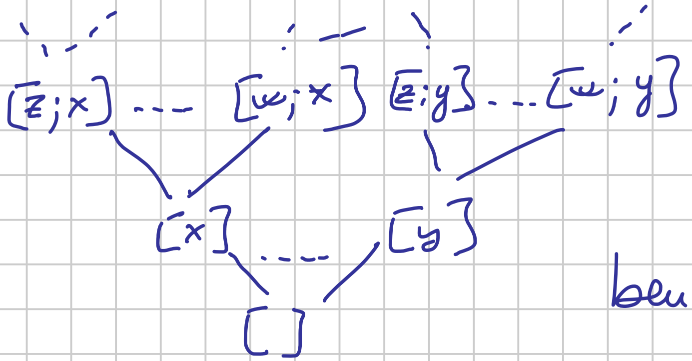
2

Precedenza indotta dalla definizione

intuitivamente: $Tl\ l \sqsubset l$

formalmente

$(\forall\, l, l' \in \text{'a list}.\ \ l \sqsubset l' \equiv l = \underline{\underline{Tl}}\ l')$



ben fondata

$$\left(\forall l, l' \in \text{'a list}.\ \ l \sqsubset l' \equiv \ l = \text{Tl}\ l'\right)$$ usando tl

$$\equiv$$ equivalenti

$$\left(\forall l, l' \in \text{'a list}.\ \ l \sqsubset l' \equiv \left(\exists\, x \in \text{'a},\ y \in \text{'a list}.\ \ l' = x :: y \ \wedge\ l = y\right)\right)$$ usando ::

Come definire len senza usare if ... then ... else ...

PATTERN (modelli)

i pattern sono espressioni che usano costruttori di valori,

costanti e variabili (nomi).

costanti → []

costruttori di valori → ::

Esempio di pattern su liste

[]	pattern
3 :: []	"
x :: []	"
x :: xs	"

x: variabile che sta per un singolo elemento

xs: variabile che indica una lista

pattern possono essere uguagliati a valori instanziando opportunamente le variabili

Es:

il pattern x :: xs può essere uguagliato alla lista

[3; 4; 5]

x → 3

xs → [4; 5]

x :: xs è 3 :: [4; 5] = [3; 4; 5]

può essere uguagliato solamente a liste non vuote

$x :: xs$

$[3]$

$x \to 3$

$xs \to [\,]$

$3 :: [\,] = [3]$

$x :: xs$

$\neq$

$[\,]$

NO!

può essere uguagliato solamente a liste più lunghe di 1

$x :: (y :: ys)$

può essere uguagliato a $[3;4]$?

Si $x \to 3$ $y \to 4$ $ys \to [\,]$

$3 :: (4 :: [\,]) = [3;4]$

$x :: y :: ys$

può essere uguagliato a $[3]$?

NO! $x :: (y :: ys) \neq 3 :: [\,]$

Titolo nota 24/09/2015

I pattern vengono usati nelle espressioni MATCH

match (espressione) with
pattern 1 → ris 1
| pattern 2 → ris 2
⋮
| pattern n → ris_n

```
# match [2;3] with
    [] → 0
  | x :: xs → 1 ;;
```

x :: xs ↓ 2 :: [3]

- : int = 1

Titolo nota 24/09/2015

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

len [3;4]

= { def len, 2° pattern, x=3, xs=[4] }

1+ len [4]

= { def len, 2° pattern, x=4, xs=[] }

1 + 1 + len []

= { def len, 1° pattern }

1+1+0

= { calcolo }

2

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

len [3;4]

= { def len, 2° pattern, x=3, xs=[4] }

1+ len [4]

= { def len, 2° pattern, x=4, xs=[] }

1 + 1 + len []

= { def len, 1° pattern }

1+1+0

= { calcolo }

2

Inferenze dei tipi con espressioni match

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

$$\text{len} : \underbrace{\text{'a list}}_{\text{tipo di } l} \rightarrow \underbrace{\text{int}}_{\text{tipo del risultato}} = \text{<fun>}$$

```
# let rec g n = g n + 1 ;;
  g : int -> int = <fun>

# g 2 ;;
```

non termina

```
let f = ... f ...
```

```
let g n = n+1 ;;
g : int -> int = <fun>
let g n = g n + 1 ;;
g : int -> int = <fun>
```

lecita

```
# g 2 ;;
- : int = 4
```

Concatenazione tra liste

Titolo nota 24/09/2015

Operatore predefinito @ si applica a due liste

[1;2] @ [3;4;5];;

- : int list = [1;2;3;4;5]

Due proprietà

1) $y @ [\,] = [\,] @ y = y$

2) $(x :: xs) @ l = x :: (xs @ l)$

proprietà

$3 :: ([1;2] @ [4]) = (3 :: [1;2]) @ [4] = [3;1;2;4]$

@ append

```
let rec append l1 l2 = match l1 with
    [] -> l2
  | x :: xs -> x :: (append xs l2) ;;
```

Curryed

a [3;4] [5;6]

= {def a, 2° pattern, x = 3, xs = [4]}

3 :: (a [4] [5;6])

= {def a, 2° p., x = 4, xs = []}

3 :: (4 :: (a [] [5;6]))

= {def a, 1° p.}

3 :: (4 :: ([5;6]))

= {calcolo}

[3;4;5;6]

@ append

Titolo nota 24/09/2015

let rec append l1 l2 = match l1 with Curryed

[] → l2

| x :: xs → x :: (append xs l2) ;;

$$\text{append} : \underbrace{\text{'a list}}_{\text{Tipo di } l1} \rightarrow \underbrace{\text{'a list}}_{\text{Tipo di } l2} \rightarrow \underbrace{\text{'a list}}_{\text{tipo risultato}} = \text{<fun>}$$

$$\left(\forall\, l1, l2 \in \text{'a list}.\ \text{append } l1\ l2 = l1 \mathbin{@} l2\right)$$

1) $l @ [\,] = [\,] @ l = l$

2) $(x :: xs) @ l = x :: (xs @ l)$!!

Titolo nota 24/09/2015

$(\forall\ l1\ l2 \in \text{'a list}.\ \text{append}\ l1\ l2 = l1 @ l2)$

```
# let rec append l1 l2 = match l1 with
    [] -> l2
  | x :: xs -> x :: (append xs l2);;
```

Curryed

Procedura induttiva della def. (informalmente $(xs, l2) \sqsubset (x::xs, l2)$)

$$\left(\forall\ l_1, l_2, l_1', l_2' \in \text{'a list}.\quad (l_1, l_2) \sqsubset (l_1', l_2') \equiv (l_1 =_{Tl} l_1' \wedge l_2 = l_2')\right)$$

BEN FONDATA

$(x::xs, l2)$

$|$

(xs, l_2)

$\vdots$

$([\,], l2)$

infinite catene

$(\forall\ l1\ l2 \in 'a\ list.\ append\ l1\ l2 = l1 @ l2)$

Titolo nota 24/09/2015

```
# let rec append l1 l2 = match l1 with
      [] -> l2
    | x :: xs -> x :: (append xs l2) ;;
```

Curryed

Caso base $([\,], l2)$

append [ ] l_2 = [ ] @ l_2

append [ ] l2

= {def. append, 1° patt.}

l2

= {1) prop. @, [ ]@l = l}

[ ]@ l2

Caso induttivo

ip. induttiva

append xs l2 = xs @ l_2 $\Rightarrow$ append (x::xs) l2 = (x::xs)@ l_2

append (x :: xs) l_2

= {def. append, 2° p}

x :: (append xs l2)

= {ip. induttiva}

x :: (xs @ l2)

= {2) prop. @}

(x :: xs) @ l_2

take n l dà come risultato la lista che contiene i primi n elementi di l.

Titolo nota 24/09/2015

```
# take 2 [3;4;5;6];;
- : int list = [3;4]

# take 4 [3;4];;
- : int list = [3;4]

# take 0 [3;4];;
- : int list = []
```

```
let rec take n l = match (n, l) with
    (0, ys) -> []
  | (m, []) -> []
  | (m, x::xs) -> x :: (take (n-1) xs);;
```

quando la lista è vuota e n qualsiasi

quando n = 0 e la lista è qualsiasi