giovedì 23 novembre 2017 16:06

@

[3] @ [3;4) = [3;3;4]

append

Curryed

let rec append l1 l2 =

parametri

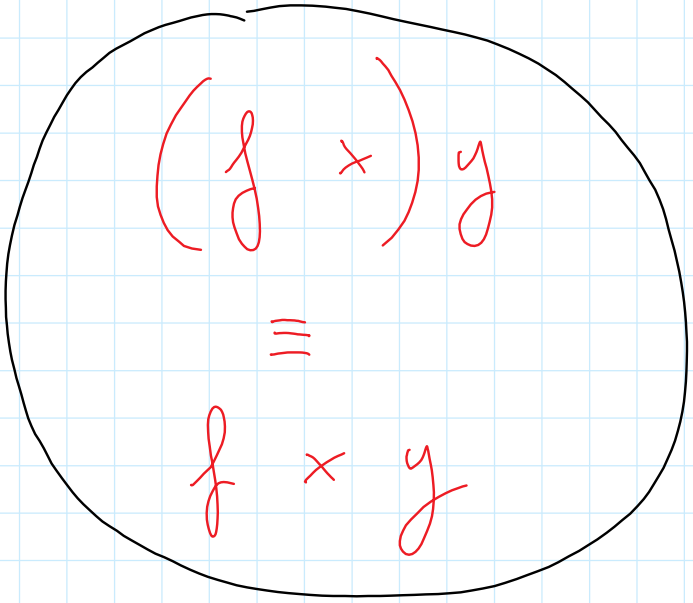
nome della funzione

parametri della definizione

argomenti della chiamata

append [3] [3;4) ;;

argomenti



applicazione di funzioni associa a sinistra

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

`[]@l ≡ l` (non servono)

uguaglia solamente liste non vuote

append : 'a list → 'a list → 'a list = <fun>

'a list: tipo di l1; 'a list: tipo di l2; 'a list: tipo out

```
append [3;4] ['c';'d'];;
```

'a = int

append : int list → int list → int list

append [3;4] [5];;

= { def. append : 2° pattern }

3 :: append [4] [5]

= { def append, 2° p }

3 :: (4 :: append [] [5])

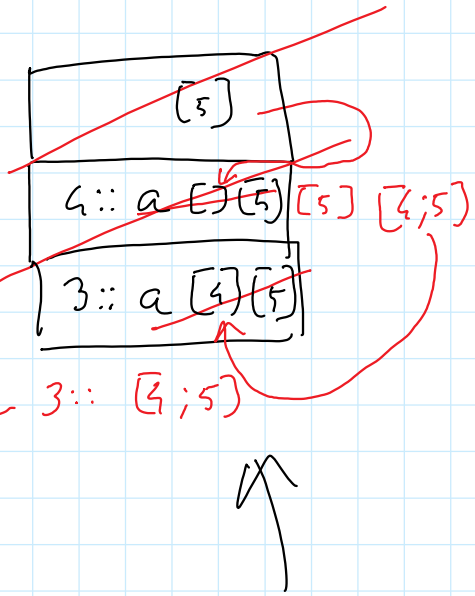
= { def append, 1° p }

3 :: (4 :: [5])

= { notazione }

[3;4;5]

STACK (PILA) di attivazione



```
let rec append l1 l2 = ..... ;;
```

giovedì 23 novembre 2017 16:26

```
let a1 = append [3;4];;
a1 : int list -> int list = <fun>

a1 [5;6];;
- : int list = [3;4;5;6]
```

```
let g(x,y) = append x y;;
g : 'a list * 'a list -> 'a list = <fun>

g [3;4];;   sbagliato

g ([3,4],[4;5]);;
- : int list = [3;4;4;5]
```

```
let somma (x,y) = x+y;;
somma : int * int -> int = <fun>

let s1 x y = somma (x,y);;
s : int -> int -> int = <fun>
```

int → tipo x; int → tipo y; int → tipo uscita

```
let s2 = s1 3;;
s2 : int -> int = <fun>

s2 5;;
- : int = 8
```

giovedì 23 novembre 2017 16:37

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

append : 'a list * 'a list → 'a list

$\forall l_1, l_2 \in$ 'a list. $(xs, l_2) < (x::xs, l_2)$

append $(l_1, l_2) = l_1 @ l_2$

proprietà di @

1) $[] @ l \equiv l @ [] \equiv l$

2) $x :: (l_1 @ l_2) \equiv (x :: l_1) @ l_2$

precedenza indotta dalla definizione ricorsiva.

Gli argomenti della chiamata ricorsiva precedono gli argomenti della funzione

$(xs, l_2) < (l_1, l_2)$

=

$(xs, l_2) < (x::xs, l_2)$ ben fondata?

giovedì 23 novembre 2017 16:41

$\forall l_1, l_2 \in$ 'a list.

append $(l_1, l_2) = l_1 @ l_2$

Induzione $(xs, l_2) < (x :: xs, l_2)$

Caso base

append $([], l_2) = [] @ l_2$

Caso induttivo

append $(xs, l_2) = xs @ l_2$

$\Rightarrow$

append $(x :: xs, l_2) = (x :: xs) @ l_2$

Caso base

append $([], l_2)$

$=$ { def. append, 1° p }

l_2

$=$ { data la prop. $[] @ l_2 = l_2$ }

$[] @ l_2$

Caso induttivo

giovedì 23 novembre 2017 16:50

$$append(xs, l_2) = xs \mathbin{@} l_2 \Rightarrow append(x :: xs, l_2) = (x :: xs) \mathbin{@} l_2$$

ip. induttiva

$$\begin{aligned}
&append(x :: xs, l_2) \\
&= \{ \text{def. append, } 2^\circ \text{ p.} \} \\
&\quad x :: append(xs, l_2) \\
&= \{ \text{ip. induttiva} \} \\
&\quad x :: (xs \mathbin{@} l_2) \\
&= \{ \text{prop } @ : \; x :: (l_1 @ l_2) = (x :: l_1) @ l_2 \} \\
&\quad (x :: xs) @ l_2
\end{aligned}$$

take n l generalizzazione di hd

take 3 [3;4;5;6] = [3;4;5]
take 7 [3;4;5;6] = [3;4;5;6]
take 0 [3;4;5;6] = []

```
let rec take n l = match (n, l) with
    (0, l) -> []
  | (n, []) -> []
```

Sono diverse parametro variabile del pattern

Sono mutuamente esclusivi ??

clausole when

x::xs when condizione →

Questo pattern si applica se è possibile unificare il pattern al valore che consideriamo e la condizione è vera !!

```
let rec take n l = match (n,l) with
```

Sono mutuamente esclusivi

```
  (0,l) -> []
| (n,[]) when n>0 -> []
| (n,x::xs) when n>0 -> x:: take (n-1) xs;;
```

take : int → 'a list → 'a list = <fun>

tipo n — tipo l — tipo ns

take 2 [3;4;5];;

= { def take, 3° p }

3:: take 1 [4;5]

= { def take, 3° p }

3 :: (4:: take 0 [5])

= { def take, 1° p }

3 :: (4 :: [])

= { notazione }

[3;4]

x = 3

xs = [4;5]

x :: xs = [3;4;5]

drop generalizzazione di te

giovedì 23 novembre 2017 17:13

drop n l restituisce la lista l senza i primi n elementi.

drop 2 [3;4;5;6] = [5;6]
drop 7 ["] = []
drop 0 ["] = [3;4;5;6]
drop 5 [] = []

```
let rec drop n l = match (n, l) with
    (0, l) -> l
  | (n, []) when n>0 -> l = []      equivalente
  | (n, x::xs) when n>0 -> drop (n-1) xs ;;
```

drop : int -> 'a list -> 'a list = <fun>

int: tipo n; 'a list: tipo l; 'a list: tipo xs

nth l'ennesimo elemento di una lista

nth n l

nth 0 l indefinito
nth 3 [] indefinito
nth 2 [3;4;5] = 4
nth 4 [3;2] indefinito

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

Non ci serve

nth: int → 'a list → 'a = <fun>

int: n
'a list: tipo l
'a: tipo xs

Somme di tutti gli elementi di una lista di interi

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

sum : int list → int = <fun>

tipo l — tipo ris