LISTE in CAML

Titolo nota 22/10/2015

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

append : 'a list → 'a list → 'a list

$l1 @ l2$ operatore predefinito

$[1;2] @ [3;4] = [1;2;3;4]$

PROPRIETA' di @

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

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

Vogliamo dimostrare che append realizza effettivamente @

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

Relazione di precedenza indotta da append

$\forall l1, l2, l1', l2' \in \text{'a list}$

$$\langle l1, l2\rangle \sqsubset \langle l1', l2'\rangle \equiv l2 = l2' \wedge l1 = tl\ l1'$$

N.B. le coppie del tipo

$\langle [\,], l\rangle$ sono tutte e sole le coppie minimali

Caso base append [] ℓ = [] @ ℓ

append [] ℓ

= { 1° pattern }

ℓ

= { 1ª proprietà di @ }

[] @ ℓ

Caso induttivo

append xs ℓ2 = xs @ ℓ2 ← ip. induttiva

⇒ append x::xs ℓ2 = (x::xs) @ ℓ2 ←

append x::xs ℓ2

= { 2° patt. di append }

x:: (append xs ℓ2)

= { Ip. induttiva , <xs, ℓ2> ⊏ <x::xs, ℓ2> }

x:: (xs @ ℓ2)

= { 2ª proprietà di @ : x::(xs @ ℓ2) = (x::xs) @ ℓ2 }

(x::xs) @ ℓ2

TAKE

Titolo nota 22/10/2015

Vogliamo definire una funzione che "generalizza" hd

take n l

calcola la lista che contiene i primi n elementi di l

take 2 [1;2;3;4;5] = [1;2]

take 3 [1;2] = [1;2]

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

pattern condizionato: un valore "matcha" con questo pattern quando soddisfa ANCHE il vincolo dopo when.

PATTERN MATCHING su oggetti che non sono necessariamente liste.

match ($\boxed{n}$, l) with

($\boxed{0}$, - -)

non possiamo utilizzare pattern su interi del tipo

match $\boxed{n}$ with

$\boxed{n+k}$ -> - - - .

risultato del calcolo di una funzione

I pattern possono essere solo

- costanti
- variabili (o _)
- strutture: oggetti costruiti mediante costruttori di tipo (come ::)

Titolo nota 08/10/2015

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

take : $\underbrace{\text{int}}_{n} \rightarrow \underbrace{\text{'a list}}_{l} \rightarrow \underbrace{\text{'a list}}_{x's}$ = <fun>

take: int -> 'a list -> 'a list

take 2 ['a'; 'b'; 'c'; 'd']

= { 3° patt. n=2, x='a', xs=['b'; 'c'; 'd'], 2>0 }

'a' :: (take 1 ['b'; 'c'; 'd'])

= { 3° patt. n=1, x='b', xs=['c'; 'd'], 1>0 }

'a' :: ('b' :: (take 0 ['c'; 'd']))

= { 1° pattern }

'a' :: 'b' :: []

= ['a'; 'b']

take 3 ['a'; 'b']

= { 3° patt. n=3, x='a', xs=['b'], 3>0 }

'a' :: (take 2 ['b'])

= { 3° patt. n=2, x='b', xs=[], 2>0 }

'a' :: ('b' :: (take 1 []))

= { 2° patt, n=1, 1>0 }

'a' :: 'b' :: []

= ['a'; 'b']

take (-2) ['a'; 'b'; 'c']

= { 3° patt: n = -2 x = 'a' xs = ['b'; 'c']

-2 $\not>$ ∅ non si applica

errore a tempo di esecuzione

DROP : guardiamo tl

Titolo nota

08/10/2015

Funzione drop n l restituisce la lista ottenuta da l eliminando i primi n elementi

```
drop 3 [1;2;3;4;5] = [4;5]
drop 2 [1] = []

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

drop: int → 'a list → ~~'b~~ 'a list = <fun>

drop: int → 'a list → 'a list

Relazione di precedenza indotta da take e drop

```
let rec take n l = ...
   .....
   (n, x::xs) when n>0 -> x::(take (n-1) xs)
```

$(n-1, xs) \sqsubset_t (n, \_::xs)$

$(n-1, xs) \sqsubset_d (n, \_::xs)$

Indichiamo questa relazione con

$\sqsubset$

Dimostriamo per induzione ben fondata rispetto a $\sqsubset$ su $\mathbb{N} \times$ 'a list

$$\boxed{\forall n \in \mathbb{N}.\ \forall l \in \text{'a list}.\ (\text{take } n\ l) \,@\, (\text{drop } n\ l) = l}$$

1° caso base $(0, l)$

$(\text{take } 0\ l) \,@\, (\text{drop } 0\ l)$

$= \{$ 1° patt. di take, 1° patt. di drop $\}$

$[\,] \,@\, l$

$= \{$ prop. di @ $\}$

l

2° caso base $(k, [\,])$

$(\text{take } k\ [\,]) \,@\, (\text{drop } k\ [\,])$

$= \{$ 2° patt. di take, drop $\}$

$[\,] \,@\, [\,]$

$=$

$[\,]$

$\langle n, [x_1; \dots; x_k] \rangle$

$\langle n-1, [x_2; \dots; x_k] \rangle$

$\vdots$

1° caso: $\langle 0, [\dots] \rangle$ 2° caso: $\langle n, [\,] \rangle$

$\sqsubset$ è ben fondata

Titolo nota 08/10/2015

Caso induttivo $(\text{take } \underline{n}\ \text{xs}) @ (\text{drop } \underline{n}\ \text{xs}) = \text{xs}$ ← ip. induttiva

$$\Rightarrow (\text{take } \underbrace{(n+1)\ x::\text{xs}}) @ (\text{drop } \underbrace{(n+1)\ x::\text{xs}}) = x::\text{xs}$$

$(n, \text{xs}) \sqsubset (n+1, x::\text{xs})$ coppie NON MINIMALE rispetto a $\sqsubset$

$$(\text{take } (n+1)\ x::\text{xs}) @ (\text{drop } (n+1)\ x::\text{xs})$$

$= \{ 3^\circ \text{ patt. di take e drop} \}$

$$(x::(\text{take } n\ \text{xs})) @ (\text{drop } n\ \text{xs})$$

$= \{ 2^a \text{ prop. di } @ \quad (x::\ell) @ \ell' = x::(\ell @ \ell') \}$

$$x::(\underbrace{(\text{take } n\ \text{xs}) @ (\text{drop } n\ \text{xs})}_{=\ \text{xs}})$$

$= \{ \text{Ip. induttiva} \}$

$x::\text{xs}$ c.v.d.

nth : selezionare l'n-esimo elemento di una lista

nth $\underline{n}$ l restituisce l'n-esimo elemento di l (indefinito se tale elemento non esiste).

nth 3 ['a'; 'b'; 'c'; 'd'; 'e'] = 'c'

- nth 3 ['a'; 'b'] = ?? indefinito

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

$$\text{nth} : \underbrace{\text{int}}_{n} \to \underbrace{\text{'a list}}_{l} \to \underbrace{\text{'a}}_{n\text{'s}} \quad = \text{<fun>}$$

nth $\boxed{3}$ ['a'; $\boxed{\text{'b'; 'c'; 'd'; 'e'}}$]

= { 2° patt. n=3, xs=['b'; 'c'; 'd'; 'e'], 3>1 }

nth 2 ['b'; 'c'; 'd'; 'e']

= { 2° patt, n=2, xs=['c'; 'd'; 'e'] }

nth 1 ['c'; 'd'; 'e']

= { 1° patt., x='c' }

'c'

Relazione di precedenza indotta?

$\langle n-1, xs \rangle \sqsubset \langle n, x::xs \rangle$

$\forall n \in \mathbb{N} \setminus \{0\} \ \forall l \in \text{'a list} \setminus \{[\,]\}$

$\langle 1, l \rangle$ sono tutte e sole le coppie minimali

Titolo nota 08/10/2015

CALCOLO della SOMMA degli elementi di una lista di interi

sum : int list → int

```
let rec sum l = if l = [] then 0 else (hd l) + sum (tl l);;
```

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

sum [3; -1; 5]

= { 2° patt. $\boxed{x}$ = 3, xs = [-1; 5] }

3 + sum [-1; 5]

= { 2° patt. $\boxed{x}$ = -1, xs = [5] }

3 + (-1) + sum [5]

= { 2° patt. $\boxed{x}$ = 5, xs = [] }

3 + (-1) + 5 + sum []

= { 1° patt. }

3 + (-1) + 5 + 0

= 7

```
let rec sum l =
   if l = [] then 0
     else match l with
          x::xs -> x + (sum xs);;
```

```
match exp with
p1 -> e1 |
p2 -> e2 |
 :
pn -> en
```

è una espressione come tutte le altre, può stare ovunque può stare una espressione