ESERCIZI TYPED LAMBDA CALCULUS - Alcune soluzioni -

Esercizio 1

Vedere testi.

Una proprieta' che listingue per esempio e' che, dato un termine ed un contesto, in un caso il tipo e' univocamente determinato, nell'altro possiamo avere molti tipi per lo stesso termine.

Esercizio 5

Si tratta della proprieta' di subject reduction. Infatti ci dice che in una sequenza di riduzioni che parte da un termine ben tipato ogni termine rimarra' ben tipato e del tipo del termine di partenza. Cosi', osservando che per ridurre un redex ("eseguire" una beta riduzione cioe' un passo di computazione) bisogna tener conto solo della struttura del termine e non dei tipi, ecco che, una volta controllato all'inizio che il termine e' ben tipato, i tipi non concorrono piu' alla computazione.

Esercizio 8

il principal type e' $a \rightarrow b \rightarrow a$ per come si arriva a tale risultato, consultare testi.

L'operatore di punto fisso non e' normalizzabile, se avesse anche un solo tipo sarebbe addirittura fortemente normalizzabile, quindi...

Esercizio 11

$$\begin{array}{l} b:t1 \rightarrow t2, c:t1, y:t2 \mid - c:t1 \qquad b:t1 \rightarrow t2, c:t1 \mid - b:t1 \rightarrow t2 \qquad b: t1 \rightarrow t2, c: t1 \mid - c: t1 \\ \hline b:t1 \rightarrow t2, c:t1 \mid - \lambda y.c:t2 \rightarrow t1 \qquad b: t1 \rightarrow t2, c: t1 \mid - (bc): t2 \\ \hline b:t1 \rightarrow t2, c:t1 \mid - (\lambda y.c)(bc): t1 \\ \hline b:t1 \rightarrow t2 \mid - \lambda c.((\lambda y.c)(bc)): t1 \rightarrow t1 \\ \hline \mid - \lambda bc.((\lambda y.c)(bc)): (t1 \rightarrow t2) \rightarrow t1 \end{array}$$

Esercizio 12

$$\begin{array}{l} b:t1 \rightarrow t2, c:t1, y:t2 \mid - c:t1 \qquad b:t1 \rightarrow t2, c:t1 \mid - b:t1 \rightarrow t2 \qquad b: t1 \rightarrow t2, c: t1 \mid - c: t1 \\ \hline b:t1 \rightarrow t2, c:t1 \mid - \lambda y.c:t2 \rightarrow t1 \qquad b: t1 \rightarrow t2, c: t1 \mid - (bc): t2 \\ \hline b:t1 \rightarrow t2, c:t1 \mid - (\lambda y.c)(bc): t1 \\ \hline c:t1 \mid - \lambda b.((\lambda y.c)(bc)): (t1 \rightarrow t2) \rightarrow t1 \end{array}$$

Esercizio 15

Per induzione sulla deduzione di Gamma |-Church M : alpha

Esercizio 16

```
< empty; (a-->b)-->a-->b >
```

dove a e b sono variabili di tipo e empty e' il contesto vuoto

Esercizio 20

Provate a vedere come si comporta l'operatore esplicito di punto fisso per PCF.

Esercizio 22 (By Guastella and Tumbarello)

Il termine che in PCF genera una lista di tutti i numeri naturali è il punto fisso di un funzionale.

In particolare consideriamo il seguente funzionale :

```
F = \f : (nat -> listint) -> listint . \n : nat . \ls : listint . cons n ( f  
( ( n+1 ) ls ) ).
```

Il termine cercato sarà quindi :

```
InfList = ( fix F ) 0 [ ] - >> ( F ( fix F ) ) 0 [ ] =  
= ( \f : (nat -> listint) -> listint . \n : nat . \ls : listint . cons n ( f  
( ( n+1 ) ls ) ) ) ( fix F ) ) 0 [ ] ->  
-> \n : nat . \ls : listint . cons n ( ( fix F ) ( ( n+1 ) ls ) ) - >  
-> \ls : listint . cons n ( ( fix F ) ( ( 0+1 ) ls ) ) ->  
-> cons n ( ( fix F ) ( ( 0+1 ) [ ] ) ) ,
```

dove

fix è il trovatore di punti fissi di un funzionale (F nel nostro caso)
cons , listint sono le costanti definite nell'esercizio precedente .

InfList costruisce una lista infinita < 0 , 1 , ... , n , ... > partendo dal seme 0
e da una lista vuota.

N.B. InfList ha tipo ListInt (il tipo restituito da cons) .

Costruiamo adesso una funzione PCF che restituisce il primo numero pari di una
lista.

Supponiamo di avere i seguenti predicati primitivi :

```
if : bool -> nat -> nat
```

```
divisible? : ( nat -> nat ) -> bool ; presi due naturali restituisce vero se il  
primo è divisibile per il secondo.
```

La funzione cercata sarà :

```
FirstEven = \ls : listint . if ( divisible? ( ( car ls ) 2 ) ) ( car ls )  
( FirstEven ( cdr ls ) )
```

Esempio di applicazione su una lista finita :

```
FirstEven (cons 6 (cons 5 [ ])) ->>
->>FirstEven (< 6 , < 5 , [ ] >>) ->
-> if ( divisible? ( ( car < 6 , < 5 , [ ] >> ) 2 ) ) ( car < 6 , < 5 , [ ] >> )
( FirstEven ( cdr < 6 , < 5 , [ ] >> ) ) ->
-> if ( divisible? ( 6 2 ) ) ( car < 6 , < 5 , [ ] >> ) ( FirstEven ( cdr < 6 , <
5 , [ ] >> ) ) ->
-> if tt ( car < 6 , < 5 , [ ] >> ) ( FirstEven ( cdr < 6 , < 5 , [ ] >> ) ) ->
-> ( car < 6 , < 5 , [ ] >> ) ->
-> 6 .
```

Applicazione di FirstEven a InfList con strategia di riduzione lazy :

```
FirstEven InfList = ( \ls : listint . if ( divisible? ( ( car ls ) 2 ) ) ( car ls )
( FirstEven ( cdr ls ) ) ) InfList - >
-> if ( divisible? ( ( car InfList ) 2 ) ) ( car InfList ) ( FirstEven ( cdr
InfList ) ) ->>
->> if ( divisible? ( ( car InfList' ) 2 ) ) ( car InfList ) ( FirstEven ( cdr
InfList ) ) ->
-> if ( divisible? ( ( 0 2 ) ) ( car InfList ) ( FirstEven ( cdr InfList ) ) ->
-> if tt ( car InfList ) ( FirstEven ( cdr InfList ) ) ->
-> ( car InfList ) ->>
->> ( car InfList' ) ->
-> 0
```

N.B.1 Abbiamo scritto -> e ->> intendendo ->lazy e ->>lazy

N.B.2 InfList' è cons 0 ((fix F) ((0+1) [])) , cioè la lista incompleta < 0 , ... >

N.B.3 FirstEven funziona sia con liste finite che infinite

N.B.4 La strategia lazy non valuta interamente InfList ma solo lo stretto necessario.