Paradigmi di Programmazione - A.A. 2023-24

Esame Scritto del 12/01/2024

CRITERI DI VALUTAZIONE:
La prova è superata se si ottengono almeno 12 punti negli esercizi 1,2,3 e almeno 18 punti complessivamente.

Esercizio 1 [Punti 4]

Applicare la β-riduzione **con strategia call-by-name** alla seguente λ-espressione fino a raggiungere una espressione non ulteriormente riducibile o ad accorgersi che la derivazione è infinita:

$$(\lambda x.x)(((\lambda x.\lambda y.\lambda z.xyz)z)x)$$

Nella soluzione, mostrare tutti i passi di riduzione calcolati, sottolineando ad ogni passo la porzione di espressione a cui si applica la β-riduzione (redex) ed evidenziando le eventuali α-conversioni.

SOLUZIONE:

Una possibile soluzione:

$$\begin{array}{rl} & \underline{(\lambda x.x)(((\lambda x.\lambda y.\lambda z.xyz)z)x)} \\ \rightarrow & \underline{(((\lambda x.\lambda y.\lambda z.xyz)z)x)} \\ \equiv_\alpha & \underline{(((\lambda x.\lambda y.\lambda k.xyk)z)x)} \\ \rightarrow & \underline{((\lambda y.\lambda k.zyk)x)} \\ \rightarrow & (\lambda k.zxk) \end{array}$$

Esercizio 2 [Punti 4]

Indicare il tipo della seguente funzione OCaml, mostrando i passi fatti per inferirlo:

```
fun x ->
   match x with
   | (1,f) -> true
   | (n,f) -> (f n) > 0
```

SOLUZIONE:
Struttura del tipo:

```
X -> RIS
```

Uso per convenzione `X` come variabile di tipo per la variabile `x`, `RIS` come variabile di tipo del risultato, e `A,B,C,...` come variabili di tipo "fresche" per la definizione dei vincoli.

Vincoli:

```
X = int * F     (da pattern)
RIS = bool      (da primo caso del p.m.)
F = int -> int  (da secondo caso del p.m.)
```

Ne consegue:

```
X = int * (int -> int)
RIS = bool
```

Tipo inferito:

```
(int * (int -> int)) -> bool
```

Esercizio 3 [Punti 7]

Assumendo il seguente tipo di dato che descrive alberi binari di interi:

```
type btree =
| Node of int * btree * btree
| Leaf of int
```

si definisca, usando i costrutti di programmazione funzionale di OCaml, tre funzioni `profondita`, `max_profondita` e `conta_max` con tipi

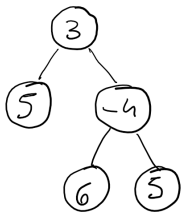
```
profondita : btree -> int
max_profondita : btree list -> int
conta_max : btree_list -> int*int
```

tali che:

- `profondita bt` restituisca la profondità dell'albero `bt`. (NOTA: un albero costituito solo da una foglia ha profondità pari a 0)
- `max_profondita btlis` restituisca il massimo tra le profondità degli alberi contenuti nella lista `btlis` (NOTA: se la lista è vuota, restituisce `0`)
- `conta_max btlis` restituisca una coppia di interi `(max_prof,n)` dove `max_prof` è la profondità massima degli alberi in `btlis` e `n` è il numero di alberi in `btlis` che hanno tale profondità. (NOTA: se la lista è vuota, restituisce `(0,0)`)

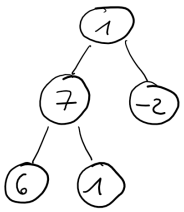
Ad esempio dati i seguenti alberi binari (a destra in una rappresentazione visuale – RIFARE FIGURE):

```
let bt1 =
  Node (3,
    Leaf 5,
    Node (-4,
      Leaf 6,
      Leaf 5
    )
  ) ;;
```




```
let bt2 =
  Leaf 12 ;;
```


```
let bt3 =
  Node (1,
    Node (7,
      Leaf 6,
      Leaf 1
    ),
    Leaf -2
  );;
```



abbiamo che:

```
profondita bt1 ;;                  (* restituisce 2 *)
profondita bt2 ;;                  (* restituisce 0 *)
profondita bt3 ;;                  (* restituisce 2 *)
max_profondita [bt1;bt2;bt3] ;;    (* restituisce 2 *)
conta_max [bt1;bt2;bt3] ;;         (* restituisce (2,2) *)
conta_max [bt3;bt3;bt2;bt3;bt3] ;; (* restituisce (2,4) *)
```

SOLUZIONE:

Una possibile soluzione:

```
let rec profondita bt =
  match bt with
  | Leaf n -> 0
  | Node of (n,bt1,bt2) -> let (prof1,prof2) = (profondita bt1,profondita bt2)
                           in if (prof1>prof2) then prof1 else prof2 ;;

let rec max_profondita btlis =
  match btlis with
  | [] -> 0
  | bt::btlis' -> let (prof1,prof2) = (profondita bt,max_profondita btlis')
                  in if (prof1>prof2) then prof1 else prof2 ;;

let conta_max btlis =
  let max_prof = max_profondita btlis in
  let proflis = List.map profondita btlis in
  let maxlis = List.filter (fun n -> n=max_prof) proflis in
  (max_prof,List.length maxlis)
```

altre soluzioni (solo `conta_max`):

```
let conta_max btlis =
  let max_prof = max_profondita btlis in
  let rec conta lis =
    match lis with
    | [] -> 0
    | bt::lis' -> if (profondita bt) = max_prof
                  then (1+conta lis')
                  else conta lis'
  in (max_prof,conta btlis);;


let conta_max btlis =
  let f (mp,n) bt =
    if (profondita bt) = mp then (mp,n+1) else (mp,n)
  in
    List.fold_left f (max_profondita btlis,0) btlis;;
```

Esercizio 4 [Punti 15]

Si estenda il linguaggio MiniCaml visto a lezione con un nuovo tipo di dato `IntCollection` che permette di dichiarare collezioni di interi (strutture dati che contengono interi, anche ripetuti, senza un ordine particolare), con le seguenti operazioni:

```
Empty               /* Costruisce una collezione vuota */
Add(elem, coll)     /* Costruisce una collezione ottenuta aggiungendo un
                       intero descritto da elem alla collezione descritta da coll */
Remove(elem, coll)  /* Costruisce una collezione ottenuta rimuovendo tutte
                       le occorrenze dell'intero descritto da elem dalla
                       collezione descritta da coll */
```

```
Exists(pred, coll)  /* Calcola true se la collezione descritta da coll contiene
                       almeno un elemento che soddisfa il predicato (funzione da
                       interi a booleani) descritto da pred. Calcola false altrimenti */
```

Esempi (in sintassi concreta):

```
let coll1 = Add(2+2,Add(1,Empty));;      /* coll1 contiene [1;4] */
let coll2 = Add(3,Add(4,Add(3,coll1)));; /* coll2 contiene [1;4;3;4;3] */
let coll3 = Remove(3,coll2)              /* coll4 contiene [1;4;4] */
Exists((fun x -> x>2), coll3);;          /* risultato true (coll3 contiene due valori >2) */
```

Si modifichi l'implementazione dell'interprete del linguaggio con quanto serve per gestire i costrutti per operare su collezioni di interi descritte in precedenza.

SOLUZIONE:
Una possibile soluzione:

```
type exp =
...
| Empty
| Add of exp * exp
| Remove of exp * exp
| Exists of exp * exp

type evT = ... |  Coll of evT list

let rec eval e s =
  match e with
  ...
  | Empty -> Coll []
  | Add (e1,e2) ->
       let elem = eval e1 s in
       let coll = eval e2 s in
       ( match (elem, coll) with
         | (Int n, Coll lst) -> Coll (elem::lst)
         | _ -> failwith "type error"
       )
  | Remove (e1,e2) ->
       let elem = eval e1 s in
       let coll = eval e2 s in
       ( match (elem, coll) with
         | (Int n, Coll lst) -> Coll (List.filter (fun x -> (x<>elem)) lst)
         | _ -> failwith "type error"
       )

  | Exists (e1,e2) ->
       let pred = eval e1 amb in
       let coll = eval e2 amb in
       ( match(pred, coll) with
         | (Closure(arg, body, fdeclenv), Coll(lst)) ->
              let check x =
                let newenv = bind fdeclenv arg x in
                let result = eval body newenv in
                match result with
                (
                | Bool b  -> b
                | _ -> failwith "type error"
                )
              in Bool (List.exists check lst)
         | _ -> failwith "type error"
       )
```