Paradigmi di Programmazione - A.A. 2021-22

Esempio di Testo d'Esame n. 4

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 alla seguente λ-espressione fino a raggiungere una espressione non ulteriormente riducibile o ad accorgersi che la derivazione è infinita:

$$(\lambda x.\lambda y.yx)(\lambda x.\lambda y.x(yy))(\lambda x.xz(\lambda y.yy))$$

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:

$$
\begin{array}{ll}
 & \underline{(\lambda x.\lambda y.yx)(\lambda x.\lambda y.x(yy))}(\lambda x.xz(\lambda y.yy)) \\
\rightarrow & \underline{(\lambda y.y(\lambda x.\lambda y.x(yy)))(\lambda x.xz(\lambda y.yy))} \\
\rightarrow & \underline{(\lambda x.xz(\lambda y.yy))(\lambda x.\lambda y.x(yy))} \\
\rightarrow & \underline{(\lambda x.\lambda y.x(yy))z}(\lambda y.yy) \\
\rightarrow & \underline{(\lambda y.z(yy))(\lambda y.yy)} \\
\rightarrow & z(\underline{(\lambda y.yy)(\lambda y.yy)}) \\
\rightarrow & z(\underline{(\lambda y.yy)(\lambda y.yy)}) \\
\rightarrow & \text{... derivazione infinita}
\end{array}
$$

Esercizio 2 [Punti 4]

Indicare il tipo della seguente funzione OCaml:

```
let f x y z =
  match (x y) with
  | [] -> (z y)
  | _::_ -> (z y) + 1;;
```

SOLUZIONE:
Struttura del tipo:

```
X -> Y -> Z -> RIS
```

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

Vincoli:

```
X = Y -> A    (da x y)
A = B list    (da pattern matching)
Z = Y -> int  (da (z y) +1)
RIS = int     (da (z y) +1)
```

Ne consegue:

```
X = Y -> B list
Y = Y
Z = Y -> int
RIS = int
```

Tipo inferito:

```
(Y -> B list) -> Y -> (Y -> int) -> int
```

che in sintassi di OCaml diventa:

```
('a -> 'b list) -> 'a -> ('a -> int) -> int
```

Esercizio 3 [Punti 7]

Definire, usando i costrutti di programmazione funzionale in OCaml, una funzione `g` con tipo

```
g : int list -> int * int
```

che, data una lista non vuota di interi, restituisce la coppia formata dal massimo elemento della lista e dal numero di volte che esso occorre nella lista stessa. Ad esempio: `g [1;-4;5;-1;5;-6;5] = (5,3)`. L'applicazione di `g` alla lista vuota causa invece un'eccezione.

SOLUZIONE:

Due possibili soluzioni:

```
(* prima soluzione *)
let g lis =
  let rec max lis m =
    match lis with
    | [] -> m
    | x::lis' -> if x>m then max lis' x
                        else max lis' m
  in let rec count lis m =
    match lis with
    | [] -> 0
    | x::lis' -> if x=m then 1+count lis' m
                 else count lis' m
  in
    match lis with
    | [] -> failwith "Errore"
    | x::lis' -> let m = max lis' x in
                 let c = count lis m in
                 (m,c);;
-----------------------------------------------------------------
(* seconda soluzione *)
let g lis =
  match lis with
  | [] -> failwith "Errore"
  | x::lis' -> let compare acc y = if y>acc then y else acc in
               let max = List.fold_left compare x lis' in
               let count acc y = if y=max then acc+1 else acc in
               let c = List.fold_left count 0 lis in
               (max,c) ;;
```

Esercizio 4 [Punti 15]

Si consideri il linguaggio didattico funzionale MiniCaml, e se ne estenda la sintassi astratta e l'interprete del linguaggio in modo gestire il costrutto iterativo for-each. La sintassi concreta del costrutto è la seguente:

`for-each (lista-interi; funzione)`

dove `lista-interi` rappresenta una lista di interi non vuota e `funzione` è una funzione non ricorsiva che viene eseguita passandole sequenzialmente tutti i valori presenti nella lista. Il costrutto `for-each` restituisce come risultato la somma dei valori calcolati dall'invocazione della funzione.

SOLUZIONE:
Una possibile soluzione:

```
type exp = ...
           | ForEach of exp list * exp

let rec eval e s = match e with
                   ...
                   | ForEach e1 e2 -> ( let lis = evalSeq e1 s in
                                        let f = eval e2 s in
                                        match (lis, f) with
                                        | (lst, Closure (i,e,s')) ->
                                                    Int (evalList lst (i,e,s') s)
                                        | (_ , _) -> failwith("ForEach error")
                                        )
                   ...
and evalSeq (lis : exp list) (s : evT env) : int list =
          match lis with
          | [] -> []
          | e::lis' -> let i = eval e s in
                       (match (typecheck(TInt,i),i) with
                       | (true,Int v) -> v :: (evalSeq lis' s)
                       | (false,_) -> failwith("IntList error")
                       )
and evalList (lst : int list) (funB : ide*exp*evT env) (s : evT env) : int =
          match (lst, funB) with
          | ([] , _) -> 0
          | (v::lst1 , (arg,fBody,fDecEnv)) ->
                 (match (eval fBody (bind fDecEnv arg (Int v))) with
                 | Int v' -> v' + (evalList lst1 funB s)) ;;
```