Paradigmi di Programmazione - A.A. 2021-22

Esame Scritto del 05/07/2022

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:

$$((\lambda x.\lambda y.(xy))(\lambda z.(yz)))k$$

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:

$$\begin{array}{rl} & ((\lambda x.\lambda y.(xy))(\lambda z.(yz)))k \\ \equiv_\alpha & \underline{((\lambda x.\lambda w.(xw))(\lambda z.(yz)))}k \\ \rightarrow & (\lambda w.(\underline{(\lambda z.(yz))w}))k \\ \rightarrow & \underline{(\lambda w.(yw))k} \\ \rightarrow & yk \end{array}$$

Esercizio 2 [Punti 4]

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

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

SOLUZIONE:
Struttura del tipo:

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

Uso per convenzione `X` e `Y` come variabili di tipo per le variabili `x` e `y`, `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 pattern matching)
A = B list     (da [] nel primo caso del p.m.)
RIS = Y        (da risultato nel primo caso del p.m.)
B = Z          (da z::[] nel secondo caso del p.m.)
Z = Y          (da risultato nel secondo caso del p.m.)
RIS = int      (da risultato nel terzo caso del p.m.)
```

Ne consegue:

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

Tipo inferito:

```
(int -> int list) -> int -> int
```

Esercizio 3 [Punti 7]

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

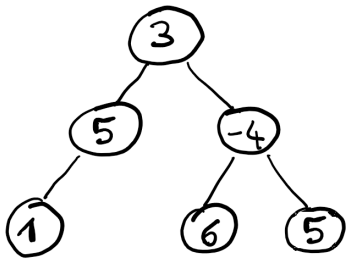
```
type btree =
| Void
| Node of int * btree * btree
```

si definisca, usando i costrutti di programmazione funzionale di OCaml, una funzione `distinct` con tipo

```
distinct : btree -> bool
```

tale che `distinct bt` restituisca `true` se i valori interi nell'albero rappresentato da `bt` sono tutti diversi tra loro, `false` se invece `bt` contiene almeno due numeri uguali.
Ad esempio dato il seguente albero binario (a destra in una rappresentazione visuale):

```
let bt =
  Node (3,
    Node (5,
      Node(1,Void,Void),
      Void
    ),
    Node (-4,
      Node(6,Void,Void),
      Node(5,Void,Void)
    )
  )
```



abbiamo che `distinct bt` restituisce `false` a causa delle due occorrenze del valore `5`.

SOLUZIONE:

Una possibile soluzione:

```
let distinct bt =
  let rec flat bt =
    match bt with
    | Void -> []
    | Node (n,bt1,bt2) -> n::((flat bt1)@(flat bt2))
  in
    let btord = List.sort (compare) (flat bt)
  in
    let f (prev,found) x =
      if x=prev then (x,false) else (x,found)
  in
    match btord with
    | [] -> true
    | x::[] -> true
    | x::lis -> let (_,res) = (List.fold_left f (x,true) lis) in res;;
```

altra soluzione:

```
let distinct bt =
  let rec count x tree =
    match tree with
    | Void -> 0
    | Node (n,tree1,tree2) -> (count x tree1)+(count x tree2)+(if x=n then 1 else 0)
  in
    let rec visit tree =
      match tree with
      | Void -> true
      | Node (n,bt1,bt2) -> (count n bt = 1) && (visit bt1) && (visit bt2)
    in
      visit bt;;
```

Esercizio 4 [Punti 15]

Si estenda il linguaggio MiniCaml visto a lezione con un nuovo tipo di dato `IntSet` che permette di dichiarare insiemi di interi. Supponiamo che il linguaggio preveda operazioni primitive per costruire e operare su insiemi di interi, come descritto di seguito:

```
Empty           /* Insieme Vuoto */
Insert(n, set)  /* Operazione di inserimento di un valore intero n
                   in un insieme di interi set */
CheckAll(predicato, set) /* Operazione che restituisce il valore true
                            se tutti gli elementi dell'insieme soddisfano il
                            predicato (funzione da interi a booleani) */
```

Esempi (in sintassi concreta):

```
let set1 = Insert(3,Insert(2,Insert(1,Empty)));;  /* risultato {1,2,3} */
let set2 = Insert(2,Insert(2,Insert(1,Empty)));;  /* risultato {1,2} (2 gia' presente) */
CheckAll((fun x -> x>2), set1);;  /* risultato true */
CheckAll((fun x -> x>2), set2);;  /* risultato false */
```

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

SOLUZIONE:
Una possibile soluzione:

```
typ exp = ...
| Empty
| Insert of exp * exp
| CheckAll of exp * exp

type evT = ... |  Set of evT list

let rec eval (e:exp) amb (evT env): evT

| Empty -> Set([])
| Insert(e1.e2) ->
     let r1 = eval e1 amb in
     let r2 = eval e2 amb in
     ( match (r1, r2) with
       | (Int(n), Set(lst)) -> if List.mem Int(n) lst
                               then Set (Int(n)::lst) else Set(lst)
       | _ -> failwith ("run-time error")
     )
| CheckAll(e1, e2) ->
  let r1 = eval e1 amb in
  let r2 = eval e2 amb in
  ( match(r1, r2) with
    | (Closure(a, b, fde), Set(lst) ->
        let rec checking(a,b,fde,lst,ret) =
        ( match lst with
          | [] -> ret
          | x::xs -> let nenv = bind fde a x in
                     let ret1 = eval b nenv in
                     if ret1 = Bool(true) then checking(a,b,fde,xs,ret1)
                     else if ret1 = Bool(false) then ret1
                     else failwith ("run-time error")
        )
    | ...
  )
```