Paradigmi di Programmazione - A.A. 2021-22

Esempio di Testo d'Esame n. 2

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]

Eseguire le seguenti sostituzioni come da definizione di capture-avoiding substitution:

- $(\lambda y.y(\lambda z.xz))\{x := \lambda z.y\}$
- $(x(\lambda y.\lambda x.x))\{x := (yz)\}$
- $(x(\lambda y.\lambda x.zx))\{x := \lambda y.y\}\{z := x\}$

Nei primi due casi mostrare direttamente l'espressione finale ottenuta. Nel terzo caso, mostrare sia l'espressione ottenuta dopo la prima sostituzione, sia l'espressione finale.

SOLUZIONE:

- $\lambda z.z(\lambda z.(\lambda z.y)z)$
- $(yz)(\lambda y.\lambda x.x)$
- $((\lambda y.y)(\lambda y.\lambda x.zx))\{z := x\} = (\lambda y.y)(\lambda y.\lambda k.xk)$

Esercizio 2 [Punti 4]

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

```
let f x y z =
  match x with
  | ('a',1) -> y
  | (x1,x2) -> if x1=y then z
               else failwith "Error";;
```

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 = char * int  (da pattern matching)
Y = char        (da x1=y)
Z = Y = RIS     (da casi del pattern matching)
```

Ne consegue:

```
X = char * int
Y = char
Z = char
RIS = char
```

Tipo inferito:

```
char * int -> char -> char -> char
```

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 `count` con tipo

```
count : btree -> (int -> bool) -> int
```

tale che `(count bt p)` restituisca il numero dei nodi in `bt` i cui **figli** soddisfano il predicato `p`. Ad esempio, dato il seguente predicato:

```
let positivo x =
  match x with
  | Void -> false
  | Node (i,_,_) -> i>0
```

e dato il seguente albero binario:

```
let bt =
  Node (3,
    Node (5,Void,Void),
    Node (-4,
      Node(6,Void,Void),
      Node(8,Void,Void)
    )
  )
```

abbiamo che `count bt positivo = 1` in quanto solo il nodo contenente `-4` ha entrambi i figli che soddisfano il predicato.

SOLUZIONE:

Una possibile soluzione:

```
let rec count bt p =
  match bt with
  | Void -> 0
  | Node (i,bt1,bt2) ->
      let c1 = count bt1 p in
      let c2 = count bt2 p in
      if (p bt1) && (p bt2) then c1+c2+1
                            else c1+c2 ;;
```

Esercizio 4 [Punti 15]

Si estenda il linguaggio MiniCaml visto a lezione con il costrutto `CodaLimitata` per la definizione di code con lunghezza massima prefissata. In aggiunta, il linguaggio è esteso con le operazioni primitive `insert` e `remove`, che rispettano la politica FIFO, e `peek`, che restituisce l'elemento in cima alla coda. Si mostri come deve essere modificato l'interprete OCaml del linguaggio.

SOLUZIONE:
Una possibile soluzione:

```
type tname = ... | TCoda

type exp = ...
           | CodaLimitata of exp
           | Insert of exp*exp
           | Remove of exp
           | Peek of exp

type evT = ...
           | Coda of evT list * int

let typecheck (x,y) = match x with
                      ...
                      | TCoda -> (match y with
                                  | Coda (lst,n) -> true
                                  | _ -> false
                                  )

let rec eval e s = match e with
                   ...
                   | CodaLimitata e1 -> let size = eval e1 s in
                                        if typecheck(TInt,size)
                                           then (match size with
                                             | Int n -> Coda ([],n)
                                             | _ -> failwith "Error"
                                            )
                                           else failwith "Type error"
                   | Insert (elem,e1) -> let coda = eval e1 s in
                                         (match coda with
                                          | Coda(lst,size) ->
                                              if List.length lst < size
                                              then let x = eval elem s in
                                                   Coda(lst@[x],size)
                                              else failwith "CODA PIENA"
                                          | _ -> failwith "Type error"
                                          )
                   | Remove e1 -> let coda = eval e1 s in
                                  (match coda with
                                   | Coda ([],size) ->
                                      failwith "CODA VUOTA"
                                   | Coda (x::lst,size) ->
                                      Coda (lst,size)
                                   | _ -> failwith "Type error"
                                  )
                   | Peek e1 -> let coda = eval e1 s in
                                (match coda with
                                 | Coda ([],size) ->
                                    failwith "CODA VUOTA"
                                 | Coda (x::lst,size) ->
                                    x
                                 | _ -> failwith "Type error"
                                )
```