Paradigmi di Programmazione - A.A. 2024-25

Esame Scritto del 26/5/2025

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, oppure ad accorgersi che la derivazione è infinita:

$$(\lambda x.\lambda z.xz)(\lambda q.z)(\lambda a.aa)$$

Nella soluzione, (i) mostrare **tutti i passi** di riduzione uno per uno, (ii) **sottolineare** a ogni passo quale parte di espressione viene ridotta (il **redex**); ed (iii) evidenziare le eventuali α**-conversioni**.

SOLUZIONE:

Una possibile soluzione: per prima cosa rendiamo esplicite alcune delle parentesi dell'espressione:

$$((\lambda x.\lambda z.xz)(\lambda q.z))(\lambda a.aa)$$

$\underline{((\lambda x.\lambda z.xz)(\lambda q.z))}(\lambda a.aa)$
dato che la sostituzione $\{x = \lambda q.f\}$ andrebbe applicata a $(\lambda z.xz)$, ma $z \in FV(\lambda q.z) = \{z\}$: α-conversione

$\equiv_\alpha$ $\underline{((\lambda x.\lambda k.xk)(\lambda q.z))}(\lambda a.aa)$
applico la sostituzione $\{x := \lambda q.z\}$ a $(\lambda k.xk)$(unica possibile riduzione per l'assoc. a sinistra delle espressioni)

$\rightarrow$ $\underline{(\lambda k.(\lambda q.z)k)(\lambda a.aa)}$
applico la sostituzione $\{k := \lambda a.aa\}$ a $((\lambda q.z)k)$

$\rightarrow$ $\underline{((\lambda q.z)(\lambda a.aa))}$
dovrei applicare la sostituzione $\{q := \lambda a.aa\}$ a z, ma $\lambda q.z$ è una funzione costante

$\rightarrow$ z

Esercizio 2 [Punti 4]

Trovare il **tipo** della seguente funzione OCaml, mostrando tutti i **passi** fatti per inferirlo:

```
let g a b c =
 match b with
 | [] -> (a,c)
 | x :: xs -> if x then (a + 1,c) else (a - 1,c)
```

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 = int (dalle espressioni a+1 e a-1)
Y = A list  (dal pattern [])
A = bool  (dal pattern x :: xs, dove x è usato come condizione in un if)
RIS = X*Z  (dai casi del p.m.)
```

- Ne consegue:

```
X = int
Y = bool list
RIS = int * Z
```

- Tipo inferito: `int -> bool list -> Z -> int*Z`
- In sintassi OCaml:

```
int -> bool list -> 'a -> int * 'a
```

Esercizio 3 [Punti 7]

Definire in OCaml le **tre funzioni** descritte di seguito:

- `split_at : int -> 'a list -> 'a list * 'a list` che divide una lista in due nel punto `n`, ovvero prende un intero `n` e una lista e restituisce una lista di coppie di liste: la prima contiene i primi `n` elementi, mentre la seconda contiene il resto degli elementi. Solleva un'eccezione se `n` è negativo o maggiore della lunghezza della lista in ingresso.

```
split_at 2 [1; 2; 3; 4] ;;
(* Output: [[1; 2], [3; 4]] *)
```

- `group_pairs : 'a list -> ('a * 'a) list` che raggruppa gli elementi di una lista due a due. Se la lista ha lunghezza dispari, solleva un'eccezione.

```
group_pairs [1; 2; 3; 4];;
(* Output: [(1,2); (3,4)] *)
```

- `interleave : 'a list -> 'a list -> 'a list` mescola gli elementi di due liste, alternandone gli elementi. Se le liste hanno lunghezze diverse, solleva un'eccezione.

```
interleave [1;3;5] [2;4;6];;
(* Output: [1;2;3;4;5;6] *)
```

SOLUZIONE:
Una possibile soluzione:

```
SPLIT-AT
let rec split_at n lst =
 match (n, lst) with
 | (0, _) -> ([], lst)
 | (_, []) -> failwith "Index out of bounds"
 | (n, x :: xs) ->
     let (left, right) = split_at (n - 1) xs in
     (x :: left, right)

Oppure

let split_at n lst =
   if n < 0 || n > List.length lst then failwith "Index out of bounds"
   else
   let rec aux i acc rest =
     match (i, rest) with
     | (0, _) -> (List.rev acc, rest)
     | (_, []) -> failwith "Unexpected end of list"
     | (i, x :: xs) -> aux (i - 1) (x :: acc) xs
   in
   aux n [] lst ;;

GROUP
let rec group_pairs lst =
 match lst with
 | [] -> []
 | x :: y :: rest -> (x, y) :: group_pairs rest
 | [_] -> failwith "List has an odd number of elements"

Oppure

let rec group_pairs_opt lst =
 match lst with
 | [] -> Some []
 | x :: y :: rest ->
     (match group_pairs_opt rest with
      | Some pairs -> Some ((x, y) :: pairs)
      | None -> None)
 | [_] -> None

Oppure

let group_pairs lst =
  let rec aux acc rest =
   match rest with
   | x :: y :: tl -> aux ((x, y) :: acc) tl
   | [] -> List.rev acc
   | [_] -> failwith "List has an odd number of elements"
 in
 aux [] lst

INTERLEAVE
let rec interleave l1 l2 =
 match (l1, l2) with
 | ([], []) -> []
 | (x::xs, y::ys) -> x :: y :: interleave xs ys
 | _ -> failwith "Lists must be of equal length"

let interleave l1 l2 =
 let rec aux l1 l2 acc =
   match (l1, l2) with
   | ([], []) -> List.rev acc
   | (x :: xs, y :: ys) -> aux xs ys (y :: x :: acc)
   | _ -> failwith "Lists have different lengths"
 in
 aux l1 l2 []
```

Esercizio 4 [Punti 15]

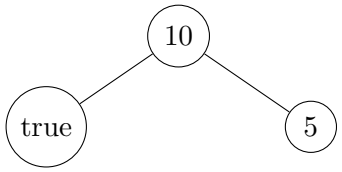
Si estenda il linguaggio **MiniCaml** visto a lezione introducendo un nuovo tipo di dato astratto `ValTree` per la definizione di **alberi binari** i cui nodi possono contenere qualsiasi valore (incluse funzioni).

La sintassi del linguaggio deve essere estesa con i seguenti costrutti:

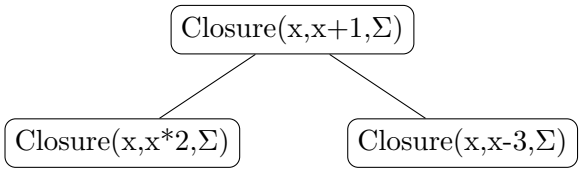
- `ValTree(v, l, r)` che costruisce un albero binario con valore `v` nel nodo radice e sottoalberi `l` (sinistro) e `r` (destro);
- `EmptyTree` che rappresenta l'albero vuoto;
- `ApplyTree(tree, v)` che, dato un albero `tree` che contiene funzioni, visita l'albero applicando via via le funzioni che incontra nei nodi. Tutte le funzioni sono applicate allo stesso valore `v`. Il risultato è un nuovo albero con la stessa struttura, ma con i nodi che contengono i risultati delle applicazioni.

Esempi (con sintassi simile a OCaml): valori e alberi corrispondenti

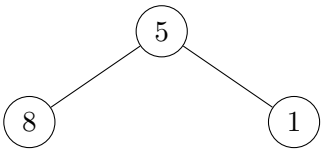
```
let t1 =
 ValTree(10,
   ValTree(True, EmptyTree, EmptyTree),
   ValTree(3+2, EmptyTree, EmptyTree)
  )
```




```
let t2 =
 ValTree(
   Fun(x -> x+1),
   ValTree(Fun(x -> x*2), EmptyTree, EmptyTree),
   ValTree(Fun(x -> x-3), EmptyTree, EmptyTree)
  )
```




```
ApplyTree(t2, 4)
```



Mostrare come modificare la sintassi e l'interprete del linguaggio MiniCaml in OCaml per gestire i nuovi costrutti `ValTree`, `EmptyTree` e `ApplyTree`.

SOLUZIONE:
Una possibile soluzione:

```
type exp =
    ...
  | EmptyTree
  | ValTree of expr * expr * expr
  | ApplyTree of expr * expr

type EvT =
    ...
  | Tree of EvT * EvT * EvT
  | Empty


let rec eval e s = match e with
    ...
  | EmptyTree -> Empty
  | ValTree (v, l, r) ->
      let v' = eval v s in
      let l' = eval l s in
      let r' = eval r s in
      Tree (v', l', r')
  | ApplyTree (t, e) ->
      let tv = eval t s in
      let arg = eval e s in
      let rec apply_tree tree =
        match tree with
        | Empty -> Empty
        | Tree (Closure (x, body, fDeclEnv), l, r) ->
            let new_val = eval body (bind x arg fDeclEnv) in
            let left_applied = apply_tree l in
            let right_applied = apply_tree r in
            Tree (new_val, left_applied, right_applied)
        | Tree (_, _, _) -> failwith "Error"
      in
      apply_tree tv
```