Paradigmi di Programmazione - A.A. 2021-22

Esempio di Testo d'Esame n. 3

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:

$$(\lambda x.((\lambda x.\lambda y.yy)(\lambda x.(\lambda y.yy)(\lambda x.xx))))x$$

Mostrare come i passi di riduzione calcolati differiscano nei casi di strategia **call-by-name** e di strategia **call-by-value** sottolineando ad ogni passo la porzione di espressione a cui si applica la β-riduzione (redex).

SOLUZIONE:
Strategia call-by-name:

$$\underline{(\lambda x.((\lambda x.\lambda y.yy)(\lambda x.(\lambda y.yy)(\lambda x.xx))))x} \rightarrow \underline{(\lambda x.\lambda y.yy)(\lambda x.(\lambda y.yy)(\lambda x.xx))} \rightarrow \lambda y.yy$$

Strategia call-by-value:

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

Esercizio 2 [Punti 4]

Indicare il tipo delle seguenti funzioni OCaml, mostrando i passi fatti per inferirli:

1. `let f g x = (g x) = 3;;`
2. `(fun x -> fun y -> y x + 1)(3,4)`

SOLUZIONE:
(FUNZIONE 1)
Struttura del tipo:

```
G -> X -> RIS
```

Vincoli:

```
G = X -> A      (da g x)
A = int         (da (g x) = 3)
RIS = bool      (da (g x) = 3)
```

Ne consegue:

```
G = X -> int
X = X
RIS = bool
```

Tipo inferito:

```
(X -> int) -> X -> bool
```

che in sintassi OCaml diventa:

```
('a -> int) -> 'a -> bool
```

(FUNZIONE 2)
Struttura del tipo:

```
Y -> RIS   (come risultato dell'applicazione a (3,4))
```

Vincoli:

```
X = int * int   (da applicazione a (3,4))
Y = X -> int    (da y x + 1)
RIS = int       (da y x + 1)
```

Ne consegue:

```
Y = (int * int) -> int
RIS = int
```

Tipo inferito:

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

Esercizio 3 [Punti 7]

Senza utilizzare esplicitamente la ricorsione, ma utilizzando funzioni higher-order su liste (dal modulo `List`), si definisca in OCaml una funzione `split` con tipo

```
split: 'a list -> ('a -> bool) -> ('a list * 'a list)
```

in modo che `split lis p` restituisca una copia `(lis1,lis2)` tale che la lista `lis1` contenga tutti e soli gli elementi di `lis` che soddisfano il predicato `p` e la lista `lis2` contenga tutti e soli gli elementi di `lis` che non soddisfano il predicato `p`. La soluzione dovrebbe fare in modo che la lista `lis` venga scandita una sola volta.

SOLUZIONE:
Una possibile soluzione:

```
let split lis p =
  let f x (l1,l2) =
    if p x then (x::l1,l2)
    else (l1,x::l2)
  in
    List.fold_right f lis ([],[]);;
```

Esercizio 4 [Punti 15]

Si consideri il nucleo di un semplice linguaggio di programmazione funzionale, la cui sintassi è descritta da

$$\begin{array}{lll} \text{Pixel} & p ::= \langle r, g, b \rangle & \text{dove } r, g, b \in \{0, 1, \ldots, 255\} \\ \text{Identificatori} & I ::= \ldots & \\ \text{Espressioni} & e ::= I \mid p \mid \mathit{lighten}\ e \mid \mathit{darken}\ e \mid \mathit{let}\ I = e_1\ \mathit{in}\ e_2 & \end{array}$$

Intuitivamente, un *pixel* è un tipo di dato che contiene tre valori interi compresi tra 0 e 255 (il primo valore codifica *red*, il secondo *green* e il terzo *blue*). L'espressione "lighten" produce come risultato un pixel dove ogni componente del pixel passato come argomento viene incrementato di 1. L'incremento del valore 255 produce 255. L'espressione "darken" produce come risultato un pixel dove ogni componente del pixel passato come argomento viene diminuito di 1. Il decremento del valore 0 produce 0.

Si definisca l'interprete del linguaggio utilizzando OCaml come linguaggio di implementazione.

SOLUZIONE:

```
type pixel = int * int * int

type eval = Unbound | Epix of pixel

type ide = string

type exp = Ide of ide | Pix of pixel
           | Light of exp | Dark of exp
           | Let of ide * exp * exp

let check (a: int) = a > -1 && a < 256

let inc (a: int) = match a with
                   | 255 -> 255
                   | n -> n + 1

let dec (a: int) = match a with
                   | 0 -> 0
                   | n -> n - 1

let increase (v: eval) = match v with
  Epix(a, b, c) -> Epix(inc a, inc b, inc c)

let decrease (v: eval) = match v with
  Epix(a, b, c) -> Epix(dec a, dec b, dec c)

(* assumo env e bind come in MiniCaml *)
let rec sem (e: exp) (r: eval env) =
  match e with
  | Ide(i) -> r i
  | Pix (a, b, c) -> if check a && check b && check c then Epix(a, b, c) else Unbound
  | Light e -> let v = sem e r in increase v
  | Dark e -> let v = sem e r in decrease v
  | Let (i, e1, e2) -> let v = sem e1 r in let r1 = bind r i v in sem e2 r1
```