PROGRAMMAZIONE II - a.a. 2019-20
Soluzione– Esercitazione del 4 dicembre 2020

Esercizio 1

Si estenda il linguaggio didattico funzionale in modo da includere espressioni e valori di tipo **record**. Un *valore (di tipo) record* è un dato strutturato composto da un numero finito di coppie *nome-valore*, detti **campi**. Analogamente, una *espressione (di tipo) record* è composta da un numero finito di coppie *nome-valore*. La valutazione di una espressione record produce un valore record.

Un identificatore può esser legato a un valore record tramite il costrutto `let`: nel seguente esempio (in sintassi OCaML-like) si evidenzia che l'espressione record che compare nel `let` è valutata in un valore record

```
# let rect = record{base = 5 * 5, altezza = 10 - 6}
val rect = record{base = 25, altezza = 4}
```

Sui record è definita l'operazione di selezione di una componente. Continuando l'esempio precedente

```
# let b = rect.base
val b = 25
```

1. Si estenda la sintassi astratta del linguaggio didattico funzionale in modo da includere valori ed espressioni record e l'operatore di selezione.

Prima soluzione con lista di coppie

```
type exp = ...
   | Rec of (ide * exp) list
   | Select of exp * ide

type evT = ...
   | RecVal of (ide * evT) list
   ...
```

```
let rec eval (e : exp) (r : evT env) : evT = match e with
     ...
     Rec(lst) -> RecVal(evalList lst r) |
     ...
     Select(rd, id) ->
         (match (eval rd r) in
              RecVal(rdPairs) -> lookup id rdPairs |
              _ ->  failwith("wrong select value")) |
     ...
and evalList (lst : (ide * exp) list) (r : evT env) : (ide * evT) list = match lst with
     [ ] -> [ ] |
     (id, arg) :: rest -> (id, eval arg r) :: evalList rest r |
and lookup (id : ide) (lst : (ide * evT) list) : evT = match lst with
     [ ] -> Unbound |
     (id1, val):: rest -> if (id = id1) then val else (lookup id rest);;
```

Seconda soluzione con funzione

```
type exp = ...
   | Rec of (ide * exp) list
   | Select of exp * ide

type evT = ...
   | RecVal of (evT env)
   ...
```

```
let rec eval (e : exp) (r : evT env) : evT = match e with
     ...
     Rec(lst) -> RecVal(evalEnv lst r) |
     ...
     Select(rd, id) ->
         (match (eval rd r) in
              RecVal(r1) -> (applyenv r1 id) |
              _ ->  failwith("wrong select value")) |
     ...
and evalEnv (lst : (ide * exp) list) (r : evT env) : (evT env) = match lst with
     [ ] -> (emptyEnv Unbound) |
     (id, arg) :: rest -> (bind id (eval arg r) (evalEnv rest r));;
```

Esercizio 2

Una funzione a dominio finito è una funzione che è definita solo per un numero finito di elementi. Ad esempio si consideri la seguente funzione con una sintassi nello stile di OCaML

`let sum = fun y -> 50 + y for y in [0; 1; 2; 3; 4];;`

La funzione `sum` è definita solamente per valori del parametro attuale che appartengono all'insieme $\{0, 1, 2, 3, 4\}$, insieme che è calcolato al momento della definizione della funzione stessa.

1. Si estenda la sintassi astratta del linguaggio didattico funzionale senza funzioni ricorsive in modo da includere tali funzioni.

2. Si definiscano le regole OCaML dell'interprete per trattare la valutazione di dichiarazione e la chiamata di funzioni a dominio finito.

Assumendo di modellare solo funzioni unarie, come specificato anche durante lo svolgimento della verifica, e ricordandosi di non considerare la ricorsione, si ottiene la seguente soluzione

1.
```
type exp = ...
   | DFun of ide * exp * exp list

type evT = ...
   | DFunVal of ide * exp * evT list * evT env
```

2. Si definiscano le regole OCaML dell'interprete per trattare la valutazione di dichiarazione e la chiamata di funzioni a dominio finito.

```
let rec eval (e : exp) (r : evT env) : evT = match e with
   ...
   | DFun(i, a, exL) -> let vL = (evalList exL r) in DFunVal(i, a, vL, r)
   ...
   | FunCall(f, eArg) ->
         let fClosure = (eval f r) in
            (match fClosure with
                ...
                DFunVal(arg, fBody, fDom, fDecEnv) ->
                      let v = (eval eArg r) in
                          if (check v fDom) then eval fBody (bind fDecEnv arg v)
                                            else Unbound |
                _ -> failwith("non functional value")) |
   ...
and evalList (lst : exp list) (r : evT env) : evT list = match lst with
     [ ] -> [ ] |
     e :: rest ->
         let v = (eval e r) in v :: evalList rest r |
and check (v : evT) (lst : evT list) : bool = match lst with
     [ ] -> false |
     val :: rest -> if (evTeq v val) then true else (check v rest)
and evTeq (v1 : evT) (v2 : evT) : bool = match lst with
     ... (* adeguata nozione di eguaglianza su evT *);;
```

Esercizio 3

Si estenda il linguaggio didattico funzionale introducendo il tipo di dato `IntSet` che permette di dichiarare insieme di interi di cardinalità finita. In aggiunta, il linguaggio è esteso con le operazioni primitive `insert myset elem` e `remove myset elem` che permettono di operare su insiemi finiti di interi.

1. Si mostri come deve essere modificato l'interprete del linguaggio didattico funzionale.

Si presenta una soluzione minimale: ***`insert`*** *aggiunge sempre un intero alla lista, mentre* ***`remove`*** *ne rimuove tutte le occorrenze.*

```
type exp = ...
   | IntSet of seq | insert of exp * exp | remove of exp * exp
   ...
and seq = Empty | Item of exp * seq

type evT = ...
   | IntSetVal of int list

let rec eval (e : exp) (r : evT env) : evT = match e with
   ...
   | IntSet e1 -> IntSetVal (evalSeq e1 r)
   | insert e1 e2 = (match (eval e2 r, eval e1 r) with
        IntSetVal s, Int i -> IntSetVal (i :: s)
      | _, _ -> failwith("IntSet error") )
   | remove e1 e2 = (match (eval e2 r, eval e1 r) with
        IntSetVal s, Int i -> IntSetVal (removeSeq s i)
      | _, _ -> failwith("IntSet error") )
         ...
and let rec evalSeq (s : seq) (r : evT env) : int list = match s with
     Empty -> []
   | Item (e1, s1) -> (eval e1 r)::(evalSeq s1 r)
   | _ -> failwith("IntSet error")
and let rec removeSeq (s : int list) (i: int) : int list = match s with
     [] -> []
   | i1 :: s1 -> if (i = i1) then (removeSeq s1 i) else i1 :: (removeSeq s1 i)
```

Esercizio 4

Si consideri il seguente programma OCaML

```
let z =1;;

let f1 = fun x y  -> x + y *z;;

let rec apply_n_times f n x =
  if n<=0 then x
  else apply_n_times f (n-1) (f x);;

let rec map_n_times g n = function
  | [] -> []
  | h1::ls -> (apply_n_times g n h1) :: map_n_times g n ls;;

let z = 2;;

let ff = f1 1;;

map_n_times ff z [10;20];;
```

1. Si simuli la valutazione del programma mostrando la struttura della pila dei record di attivazione.
2. Si determini il valore calcolato dal programma. *Il valore calcolato è* $[12; 22]$