Esercizi Vari del 26 Aprile 2018

Esercizio (1 - tratto da Esercizi di Lezione8)

Si supponga di disporre di un linguaggio $\texttt{C}^{\texttt{N}}$ *ottenuto estendendo C con trasmissione per Nome. È possibile in* $\texttt{C}^{\texttt{N}}$ *sostituire ogni occorrenza di* `"while E do C;"`*, per qualsivoglia espressione* `E` *e qualsivoglia comando* `C`*, con l'invocazione di un'opportuna procedura avente comportamento equivalente.*

a. *Si mostri la definizione* $\texttt{C}^{\texttt{N}}$ *di una tale procedura;*

b. *Si confrontino gli stack di AR ottenuti nel caso:*
`while n<k do {res = res * n; n=n+1};`

c. *Si discutano i risultati di tale confronto.*

a.
```
void whileDo (bool name Exp , void name Cmd{
    if E {Cmd; whileDo(Exp,Cmd);}
}
```

b. La dichiarazione della procedura `whileDo` deve avere nel proprio scope ogni blocco contenente un comando:
(1) while E do C;
In particolare, il comando dove:
`E = n<k` mentre `C = {res = res * n; n=n+1}`.
Tale comando sarà rimpiazzato con la seguente invocazione:
(2) whileDo(E,C);
Sia $\texttt{AR}_\texttt{K}$ l'activation record in cui eseguiremo (1) nel programma originale. Sia $\texttt{AR}^{\texttt{m}}_\texttt{K}$ quello in cui eseguiremo (2) nel programma modificato. Allora $\texttt{AR}_\texttt{K}$ e $\texttt{AR}^{\texttt{m}}_\texttt{K}$ sono lo stesso activation record a meno di una ridenominazione dei valori modificabili acceduti [1]. Sia $\texttt{AR}_{\texttt{whileDo}}$ quello introdotto dall'invocazione (2). Allora $\texttt{AR}_{\texttt{whileDo}}$ ha frame contenente i parametri Exp e Cmd rispettivamente legati alle chiusure $(\texttt{E}, \texttt{AR}^{\texttt{m}}_\texttt{k})$ e $(\texttt{C}, \texttt{AR}^{\texttt{m}}_\texttt{k})$.

[1] In particolare, ogni identificatore nel frame di $\texttt{AR}_\texttt{K}$ è presente nel frame di $\texttt{AR}^{\texttt{m}}_\texttt{k}$ con stesso valore denotabile se non modificabile, altrimenti con i rispettivi modificabili aventi valore memorizzabile identico (a meno di ride...)

Esercizi Vari del 26 Aprile 2018

Esercizio (1 - tratto da Esercizi di Lezione8 : Soluzione: continua)

Si supponga di disporre di un linguaggio $\texttt{C}^{\texttt{N}}$ *ottenuto estendendo C con trasmissione per Nome. È possibile in* $\texttt{C}^{\texttt{N}}$ *sostituire ogni occorrenza di* `"while E do C;"`*, per qualsivoglia espressione* `E` *e qualsivoglia comando* `C`*, con l'invocazione di un'opportuna procedura avente comportamento equivalente.*

a. *Si mostri la definizione* $\texttt{C}^{\texttt{N}}$ *di una tale procedura;*

b. *Si confrontino gli stack di AR ottenuti nel caso:*

```
while n<k do {res = res * n; n=n+1};
```

c. *Si discutano i risultati di tale confronto.*

b.
(continua)
Quando andremo a valutare il corpo della procedura in $\texttt{AR}_{\texttt{whileDo}}$ ci troveremo a valutare prima il parametro `E`. Questo conduce a valutare la chiusura $(\texttt{E}, \texttt{AR}_{\texttt{k}}^{\texttt{m}})$.
Questa valutazione conduce a creare un nuovo activation record $\texttt{AR}_{\texttt{E}}$ con frame vuoto e accesso di catena statica CS (ai non-locali) con valore $\texttt{AR}_{\texttt{k}}^{\texttt{m}}$. Ciò conduce ad ottenere lo stesso valore e le stesse modifiche (sotto la ridenominatine fatta) della memoria che otteniamo valutando l'espressione `E` di (1) nel programma originale e nell'activation record $\texttt{AR}_{\texttt{k}}$.
La computazione del parametro `C`, se richiesta[2] ha analogo comportamento. In tutti i casi, ad ogni successiva invocazione di `whileDo(E,C);` nel corpo di `{Cmd; whileDo(Exp,Cmd);}` calcoleremo le chiusure $(\texttt{E}, \texttt{AR}_{\texttt{k}}^{\texttt{m}})$ ed (eventualmente) $(\texttt{C}, \texttt{AR}_{\texttt{k}}^{\texttt{m}})$ in un $\texttt{AR}_{\texttt{k}}^{\texttt{m}}$ identico a $\texttt{AR}_{\texttt{K}}$, a meno di una ridenominazione di modificabili, in una memoria che mantiene le stesse modifiche (sotto la ridenominatine fatta).
OSSERVAZIONE E SPUNTI: In aula abbiamo visto una descrizione grafica dell'evoluzione dello stack di controllo della computazione. Seguendo quanto descritto sopra si riproduca anche la rappresentazione grafica vista.
c.
L'invocazione di whileDo introdurre ulteriori AR a causa della definizione ricorsiva. Osserviamo però che la ricorsione è di tipo tail recursive e se trattata in tal modo richiede di utilizzare lo AR $\texttt{AR}_{\texttt{whileDo}}$ della prima invocazione anche per tutte le successive annidate.

Esercizi Vari del 26 Aprile 2018

Esercizio (2 - tratto da Esercizio 8 di Lezione9)

Il termine λx.x x è un termine sintatticamente corretto del Lambda Calcolo e

a. *può essere "trascritto" in OCaml ottenendo il termine ?*
b. *che per quanto sintatticamente corretto non ha un tipo associabile: Perchè?*

.a

```
# fun x -> x x;;
```

.b
Quando Compiliamo:

```
# fun x -> x x;;
Error: This expression has type 'a -> 'b
       but an expression was expected of type 'a
       The type variable 'a occurs inside 'a -> 'b
```

Norate che l'errore è rilevato sulla seconda occorrenza di x nel corpo x x
Rispondiamo al "Perchè?
L'identificatore `x` si trova ad avere contemporaneamente:
* un tipo `a' -> 'b` in quando generico tipo per la sua occorrenza come temine applicando, in γ `= x x`
* un tipo `'a` per la sua occorrenza come termine applicato di un applicando di tipo `a' -> 'b`
dove la generica `'a` è la stessa variabile qualificata universalmente su tutti i tipi esprimibili (in OCaml).
Allora vediamo che non esiste termine che sostituito ad `'a` renda `'a` e `'a->'b` uno stesso tipo. E questa è la ragione dell'errore e del rifiuto di accettare `# fun x -> x x;;` come un termine OCaml.

Esercizi Vari del 26 Aprile 2018

Esercizio (3 - tratto da Esercizio 5 di Laboratorio3)

Si consideri il seguente tipo algebrico di OCaml:

`typedcl = Const of string * int | Var of string * int | VarN of string | Array of string * int | SeqDcl of dcl * dcl`

per gli alberi strati di dichiarazioni di costanti, variabili inizializzate e non, array e sequenze per SmallC.
Si fornisca una funzione OCaml:

a. *idelList che restituisce la lista degli identificatori dichiarati;*
b. *collisiionList che restituisce la lista di tutti gli identificatori che sono dichiarati più volte.*
c. *Applicare ad una sequenza di almeno 5 diverse dichiarazioni.*

.a

```
let rec ideList = function
| Const (ide,_) -> [ide]
| Var (ide,_) -> [ide]
| VarN ide -> [ide]
| Array (ide,_) -> [ide]
| SeqDcl (d1,d2) -> (ideList d1) @ (ideList d2)
;;
```

.b

Una definizione che calcola i duplicati presenti nell lista ottenuta da ideList, definita come in (a). `let collisions (d:dcl) =`

```
   let rec isin e els = (* completare:  calcola true se e occorre in els, false alt.  *) in
   let rec inner old news = (match newl with
      | [] -> []
      | x::xs -> if (sin x old) then...else...  )in
   inner [] (ideList d);;
```

Completare e discutere il 27/4/2018

Esercizi Vari del 26 Aprile 2018

Esercizio (4 - tratto da Esercizio 6 di Lezione10)

Si consideri l'ADT polimorfo per valori 'a stack, Immutable, in OCaml, introdotto a Lezione10 e riportato nella pagina successiva. Si indichino:

a. *I valori astratti introdotti dal nuovo tipo (e tipo di di dato) 'a stack.*
b. *gli stati concreti con cui tali valori sono rappresentati nell'ADT Stack.*
c. *la funzione di astrazione, AF.*
d. *l'invariante di rappresentazione, I*

a. Sono già stati indicati nel commento all'intestazione del programma

b. -d. **ancora da discutere**

ADT Polimorfo in OCaml: Stack Immutable

```
exception Error ;;

module type STACK =
(* Uno 'a stack e' un valore [v1,...,vk] per k>0 ([] per k=0), contenente k valori
   di tipo generico 'a. Il valore vk e' l'ultimo aggiunto (mediante un'operazione
   push) ed il primo ad essere estratto (mediante un'operazione pop)
*)
 sig type 'a stack
  val create_stack: unit -> 'a stack
  val push: 'a stack -> 'a -> 'a stack
  val top: 'a stack -> 'a
  val pop: 'a stack -> 'a stack
end;;
```

```
module Stack =
 (struct
  type 'a sk = E | SK of 'a sk * 'a
  type 'a stack = M of int * ('a sk)
  let create_stack = fun () -> M(0,E)
  let push (M(n,sk)) x =
        if n=100 then raise(Error)
        else M(n+1,SK(sk,x))
  let top (M(n,sk)) = match sk with
                E -> raise Error
                | SK(sk,v) -> v
  let pop (M(n,sk)) = match sk with
                E -> raise Error
                | SK(sk,v) -> M(n-1,sk)
end:STACK);;
```