Esercizi Vari del 27 Aprile 2018

Esercizio (0)

Completamento della soluzione dell'esercizio 3 di ieri, 26 aprile.

.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 = (match els with [] -> false | x::xs -> (x = e) || (sin e xs)) in
   let rec inner old news = (match newl with
      | [] -> []
      | x::xs -> if (sin x old) then x::(inner oldl xs) else inner (x::oldl) xs )in
   inner [] (ideList d);;
```

.c

```
(* test di uso:
let d1 = Var("x",12);;
let d2 = Var("y",10);;
let d3 = Const("y",7);;
let d4 = Array("z",11);;
let d5 = Var("y",0);;
let d6 = VarN "z";;
let s = SeqDcl(d1,SeqDcl(d2,SeqDcl(d3,SeqDcl(d4,SeqDcl(d5,d6)))));;
# collisions s;;
- :  string list = ["y"; "y"; "z"]
*)
```

Esercizi Vari del 27 Aprile 2018

Esercizio (1)

Si consideri le seguenti due definizioni in OCaml:

```
# let g x = function
     |[] -> []
     |e::els -> x
     ;;
val g : 'a list -> 'b list -> 'a list = <fun>
# let rec g x = function
     |[] -> []
     |e::els -> g els
     ;;
Error: This expression has type 'a list -> 'b list
       but an expression was expected of type 'b list
```

Si dica:

a. Quale funzione è definita dalla prima definizione di g*;*

b. Perchè la seconda definizione genera errore?

.a

La prima definizione dichiara che `g x` definisce una funzione che applica ad una lista vuota calcola la lista vuota ed applicata ad una lista non vuota calcola calcola la lista `x`. In altre parole, g è una funzione che applicata ad una prima lista `x` e ad una seconda lista, se quest'ultima ha valore `[]` calcola `[]`, se quest'ultima è diversa da `[]` allora calcola la prima lista `x`. Ad esempio `g [2] []` calcola `[]`, mentre `g [2] [3;4]` calcola [2].

.b

Anche la seconda definizione introduce una funzione binaria con primo parametro `x` e secondo parametro introdotto per casi. E anche in questa definizione il primo caso `[]->[]` afferma che il valore calcolato è di tipo lista generica. Ne consegue che l'invocazione ricorsiva `g els` calcola una funzione da una lista generica in un lista generica ma il valore atteso deve essere invece una lista generica (dello stesso tipo restituito dal primo caso).

Esercizi Vari del 27 Aprile 2018

Esercizio (2)

Si utilizzi (nel top level evaluator di OCaml) il comando `#use "List.ml"` *per visionare le signature di tutte le operazioni della libreria List. Alcune di queste operazioni sono implementate tail-recursive altre no: Si veda il link* `http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html`

a. *Si individui l'operazione "exists" e si mostri come può essere utilizzata nella definizione dell'operazione "isin", introdotta ieri per l'appartenenza di un valore ad una lista;*

b. *Si fornisca una definizione della funzione reverse che inverte liste polimorfe*

c. *Si fornisca una definizione tail-recursive di reverse*

d. *Si confrontino le 2 definizioni tra loro e con le funzioni List.rev e List.rev_append (facendo anche uso della funzione "elapsedTime" nei listing allegati)*

a.
```
let isin x = List.exists (fun y -> x = y);;
```
Compiliamo ed applichiamola nel top level evaluator:
```
# isin 3 [7;0;15;3;14];;
- :  bool = true
# isin 3 [7;0;15;14];;
- :  bool = false
```

b.
```
let rec reverse = function
     |[] -> []
     |e::els -> (reverse els)@[e]
     ;;
```
Compiliamo ed applichiamola nel top level evaluator:
```
# elapsedTime reverse [1;2;3;4;5;6;7;8;9];;
elapsed time:  0.000007s
- :  int list = [9; 8; 7; 6; 5; 4; 3; 2; 1]
```

Esercizi Vari del 27 Aprile 2018

Esercizio (2 - continua)

....

c. *Si fornisca una definizione tail-recursive di reverse*

d. *Si confrontino le 2 definizioni tra loro e con le funzioni List.rev e List.rev_append (facendo anche uso della funzione "elapsedTime" nei listing allegati)*

c.

```
let rec reverse res = function
     |[] -> res
     |e::els -> (reverse (e::res) els
     ;;
```

Compiliamo ed applichiamola nel top level evaluator:

```
# elapsedTime reverse [1;2;3;4;5;6;7;8;9];;
elapsed time:  0.000003s
- :  int list = [9; 8; 7; 6; 5; 4; 3; 2; 1]
```

d.

Nei Linguaggi Funzionali (e pi in generale nei Linguaggi di Programmazione Applicativi, inclusi quelli Algebrici e quelli Logici) la misura data dal tempo intercorso tra clock di inizio e clock di fine di un'esecuzione non è una stima affidabile e talora nemmeno significativa. La ragione risiede nel fatto che un esecutore presiede non solo la gestione della computazione relativa alla valutazione dell'espressione racchiusa tra il prompt, # e ;; bensì l'intera gestione del top level evaluator. Questo include il monitoraggio dello stato dello stack di controllo (per recuperare memoria di AR deallocati) ed ancor più insondabile l'esecuzione del *garbage collector* per recuperare memoria heap, quando la disponibilità è scesa sotto una soglia critica. Queste attività e i loro costi non sono necessariamente imputabili all'ultima espressione valutata ma all'intera sessione di lavoro.
Per una misura significativa occorrerebbe misurare il tempo intercorso nelle stesse condizioni di sessione: Ad esempio, all'inizio della sessione che deve essere richiusa immediatamente dopo e riaperta per il confronto con una differente valutazione.
Una misura alternativa ed assai più significativa utilizza il numero di riduzioni eseguite sull'espressione iniziale per ottenere il valore calcolato. Ma questa funzione non è attualmente disponibile nel top level evaluator di OCaml.

Esercizi Vari del 27 Aprile 2018

Esercizio (3 - 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*

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);;
```

Esercizi Vari del 27 Aprile 2018

Esercizio (4 - variante Esempio di Lezione12)

Si consideri il tipo "Dcl" in Java, riportato nel file DclINF.java incluso nei listing allegati. Tutti i valori di tipo "Dcl" sono stati equipaggiati con metodi per le operazioni "toString" e "print()". Vogliamo aggiungere anche per questi valori le operazioni viste ieri per il tipo "dcl" definito in OCaml.

a. *Compilare il file "DclINF.java" insieme al file "Main1.java", entrambi allegati (allo scopo caricare in una directory nella quale eseguire i comandi indicati in Main1.java)*
b. *Definire idelList che restituisce la lista degli identificatori dichiarati;*
c. *Ci è richiesto di modificare le dichiarazioni aggiungendo il caso di dichiarazione di puntatore. In sintassi concreta: "var * ide;". Come possiamo procedere per fare ciò?*
d. *Si confronti quanto fatto in [c.] nel caso di Java con quanto dovremmo fare nel caso di OCaml (o di Linguaggio Procedurale tipo C).*

ancora da discutere

Esercizi Vari del 27 Aprile 2018

Esercizio (5 - Tipi Astratti in OCaml)

I vector Java sono un tipo di dato strutturato, omogeneo e polimorfo che permette un doppio accesso ai componenti: sequenziale come le liste e diretto come gli array. Ha operazioni per l'aggiunta, addE, e la rimozione, removeE, in coda, ed operazioni per l'aggiunta, add n, e la rimozione, remove n, del componente n-esimo. Un' operazione size fornisce il numero di elementi contenuti. Infine, ha operazioni getE e get n, e setE e set n, per l'accesso e la modifica rispettivamente dell'ultimo o dello n-esimo componente.
Vogliamo definire in OCaml un tipo di dato Vector con tali caratteristiche. Il nuovo tipo deve essere introdotto come tipo astratto.

a. *Definire in OCaml un modulo type per Vector polimorfi*
b. *Fornire un primo ADT utilizzando il seguente stato concreto:*
   `type 'a vector = int * (int -> 'a)`
   a questo scopo, rispondere alle seguenti richieste:
   b1. *Fornire una presentazione per i valori astratti Vector*
   b2. *Fornire funzione AF*
   b3. *Fornire invariante I*
   b4. *definire i costruttori dei valori*
   b5. *definire nell'ordine le operazioni richieste.*

Ancora da discutere