Homework Assignment 2

Federico Bernacca

1 Introduzione

Per svolgere l'homework ho effettuato le seguenti scelte progettuali.
L'interprete usa l'ambiente come stack per i legami tra variabili e valori, un'altra struttura che mantiene al suo interno le varie politiche locali dei blocchi, ed infine una stringa che rappresenta la storia delle operazioni passate.
L'interprete dunque avrà un Security Manager da invocare che controlla, a partire dal blocco corrente, se la politica corrente è violata. In caso di violazione il programma termina, altrimenti, una volta esaurita la politica locale, vengono controllate le eventuali politiche più esterne (LIFO) che contengono l'appena terminata policy, così da permettere una corretta definizione di politiche innestate.

2 Estensioni

Di seguito le estensioni fatte all'interprete.

2.1 Risorse sensibili

Ho definito un nuovo costrutto sintattico per definire le risorse sensibili sulle quali le operazioni da controllare possono avere accesso.

```
(* Critical resource *)
type resource =
  | Database
  | File
  | Socket
  | Net;;
```

2.2 Espressioni

Ho deciso di modificare il costrutto sintattico delle espressioni in modo da aggiungere:

1. Possibilità di istanziare una o più policies sfruttando un nuovo costrutto per la definizione di liste.
2. Operazioni di accesso sulle risorse sensibili (α).
3. Definizione di lambda anonima ($\lambda x.e$), dove x rappresenta il parametro formale ed e il corpo della funzione.
4. Definizione di lambda ricorsiva ($\lambda_z x.e$), dove z rappresenta il nome della funzione, il significato di x ed e rimane invariato dal caso sopra.
5. Applicazione di lambda (ee').
6. Controllare uno o più blocchi di espressioni: verificare se una o più policies vengono rispettate dal corpo del blocco. ($\varphi[e]$), dove φ è la politica di sicurezza locale, ed e l'espressione da verificare che rispetti la politica all'interno del suo blocco, rappresentato da [ ].

```
type exp =
    ...
  | Elist of exp (* Eitem or Enone *)
  | Eitem of exp * exp (* Eitem(Eint 5, Enone) *)
  | Edfa of
      exp * (* Elist(Eitem(Eint 0, Eitem(Eint 1, Enone)) *)
      exp * (* Elist(Eitem(Echar 'w', Eitem(Echar 'r', Enone)) *)
      exp * (* Eint 0 *)
      exp * (* Elist(Eitem(Eint 0, Eitem(Echar 'r', Eitem(Eint 1, Enone))) *)
      exp * (* Elist(Eitem(Eint 0, Eitem(Eint 1, Enone)) *)
      exp   (* Estring "error" *)
  | Epolicy of exp (* Programmer can define policies => Elist(Eitem(Edfa...)) *)
  (* Sensible operation: access event *)
  | Read of resource
  (* Sensible operation: access event *)
  | Write of resource
   (* Sensible operation: access event *)
  | Download of resource
```

```
    (* Sensible operation: access event *)
  | Connect of resource
  (* formal parameter with function body *)
  | Lambda of string * exp
  (* function name, exp must be Lambda *)
  | LambdaRec of string * exp
  (* Lambda with acutal parameter *)
  | Apply of exp * exp;;
  (* Block where to check if policy is satisfied, second exp must be a Epolicy *)
  | Check of exp * exp
  ...
```

2.3 Valori primitivi

Come primitive del linguaggio sono stati aggiunti i seguenti valori:

```
type value =
   ...
  | List of value (* List(Item(Int 5, Item(Int 6, None))) *)
  | Item of value * value (* Item(Int 5, Item(Int 6, None)) *)
  | Dfa of
        value * (* List(Item(Int 0, Item(Int 1, None)) *)
        value * (* List(Item(Char 'w', Item(Char 'r', None)) *)
        value * (* Int 0 *)
        value * (* List(Item(Int 0, Item(Char 'r', Item(Int 1, None))) *)
        value * (* List(Item(Int 0, Item(Int 1, None)) *)
        value   (* String "error" *)
  | Policy of value (* local policies => List(Item(Dfa(...), Item(Dfa(...), None)) *)
  (* Lambda name, formal param, body, env *)
  | RecClosure of string * string * exp * value env;;
  ...
```

2.4 Interprete

Ho esteso l'interprete, quindi la funzione *eval* per permettere il passaggio delle local policies, che ad ogni valutazione di *Check()*, vengono aggiunte in testa per seguire successivamente un'estrazione con politica LIFO.
L'altro parametro di tipo *string* rappresenta la history che ad ogni chiamata di un'operazione sensibile viene aggiornata.
È inoltre possibile definire dfa e policies come primitive del linguaggio.

```
let rec rec_eval (exp: exp) (env: 'v env) (dfa_list: exp) (history : string):
  (value * exp * string) =
  match exp with
    ...
  (* Local policies as primitive language *)
  | Epolicy p -> ...
  | Edfa(states, sigma, start, transition, accepting, error) -> ...
  | Read r -> ... Epolicy p -> ieval (parse_dfa_list p) (op ^ "r")
  | Write w -> ... Epolicy p -> ieval (parse_dfa_list p) (op ^ "w")
  | Download d ->  ... Epolicy p -> ieval (parse_dfa_list p) (op ^ "d")
  | Connect c -> ... Epolicy p -> ieval (parse_dfa_list p) (op ^ "c")
  (* Head pushing the current policies *)
  | Check(block, policies) -> ... rec_eval block env (Epolicy(Elist(cons l2 l1))) op
    ...
```

Dove la funzione

```
let rec ieval (policies: dfa list) (history: string) : (value * exp * string)
```

rappresenta il Security Manager che controlla, a partire dall'ultima politica inserita, se l'operazione attualmente in esecuzione viola la politica, in caso di esito positivo, solleva un'eccezione, altrimenti si richiama ricorsivamente, seguendo un ordine LIFO, su tutte le eventuali altre politiche che contengono la corrente.

Se la funzione non solleva un'eccezione, al termine di essa viene restituito il valore risultato del calcolo dell'operazione richiesta e la storia aggiornata.

3 Test

Ho scritto alcuni test che effettuano diverse prove, runnando diversi programmi con diverse politiche innestate. Infine anche un applicazione di chiamata a funzione ricorsiva.