Relazione Homework 2

Obiettivo

Lo scopo di questo homework è quello di estendere l'interprete di un semplice linguaggio funzionale di base in un interprete dotato di un meccanismo dinamico per la verifica di politiche di sicurezza locali al fine di implementare History-Dependent Access Control.
L'interprete esteso a runtime utilizza l'ambiente *env* come stack per mantenere i legami tra variabili e valori, una lista per le varie politiche locali dei security frames e una stringa *history* per tenere traccia delle operazioni sensibili svolte. Tale stringa conterrà in particolare caratteri le cui iniziali sono da ricondurre ad operazioni sensibili per la sicurezza svolte in precedenza: ad esempio la stringa RWR indica che sono state eseguite nell'ordine operazioni di Read, Write e Read.

Implementazione

Le scelte implementative che ho fatto per estendere l'interprete sono le seguenti:

1. **Valori**: ho definito due nuovi tipi di valori. *RecClosure* rappresenta il valore di chiusura per le funzioni ricorsive, mentre *Policy* rappresenta il valore di una politica locale, o una lista in caso di più politiche di sicurezza che devono valere in contemporanea.

```
type value =
    ...
    | RecClosure of string * string * expr * value env
    | Policy of dfa list
;;
```

   dfa rappresenta il tipo per un automa a stati finiti. Tramite un automa si potrà definire una politica e la sua struttura è stata fornita con il testo dell'homework e definita come segue:

```
    type dfa = {
        states : state list;
        sigma : symbol list;
        start : state;
        transitions : transition list;
        accepting : state list;
        error : string;
    }
;;
```

2. **Espressioni**: ho modificato/aggiunto i seguenti costrutti del linguaggio:

   - **Read**: operazione sensibile α_R
   - **Write**: operazione sensibile α_W
   - **Lambda**: definizione di funzione anonima λx.e con x parametro formale ed e corpo della funzione
   - **RecLambda**: definizione di funzione ricorsiva λ_zx.e con z nome della funzione, x parametro formale ed e corpo della funzione
   - **Apply**: applicazione di funzione
   - **SecurityFrame**: blocco φ[e] con φ politica locale ed e espressione che deve rispettare la politica
   - **UserPolicy**: costrutto per definire direttamente da programma una o più politiche locali di sicurezza come lista di automi a stati finiti

```
type expr =
    ...
    (* Security event, 'string' is the argument of the read (es. file, db, socket, etc...) *)
    | Read of string
    (* Security event, 'string' is the argument of the write (es. file, db, socket, etc...) *)
    | Write of string
    (* 'string' is the formal parameter, 'expr' is the body *)
    | Lambda of string * expr
    (* Recursive function, 'string' is the name of the function, 'expr' is lambda *)
    | RecLambda of string * expr
    (* Function call, 'expr' is the function, 'expr' is the actual parameter *)
    | Apply of expr * expr
    (* 'expr' represents the local policy, 'expr' is an operation or a list of operations *)
    | SecurityFrame of expr * expr
    (* 'dfa list' is a list of automata that represent security policies *)
    | UserPolicy of dfa list
;;
```

3. **Interprete**: ho esteso l'interprete classico *eval* aggiungendo come parametri la lista di security policies e la history per tenere traccia delle operazioni sensibili eseguite.

```
let rec eval (e : expr) (env : 'v env) (automa : expr) (history : string)
: (value * expr * string) =
    ...
| Read(s) -> let (a, _, _) = eval automa env automa history in
             begin match a with
               | Policy p ->  check_op "R" p history
               | _ -> failwith("ERROR: Invalid policy format in 'Read' operation!")
             end

| Write(s) -> let (a, _, _) = eval automa env automa history in
               begin match a with
                 | Policy p ->  check_op "W" p history
                 | _ -> failwith("ERROR: Invalid policy format in 'Write' operation !")
               end
```

```
| SecurityFrame(a, op) -> begin match (automa, a) with
        | (UserPolicy prev, UserPolicy curr) -> eval op env (UserPolicy(curr@prev)) history
        | (_, _) -> failwith("ERROR: Unknown types of ...")
          end

| UserPolicy(list) -> (Policy(list), automa, history)
;;
```

La valutazione dei costrutti *Read()* e *Write()* si esegue invocando la funzione *check_ops*: tale funzione ausiliaria controlla se un'operazione sensibile si trova all'interno di un blocco $\varphi[...]$ e, in caso, verifica che essa rispetti la politica di sicurezza locale φ ed eventuali altre politiche in caso di security policies innestate. Se ci trovassimo in una situazione di security policies innestate (ad esempio $\varphi[Read, \varphi'[Write]]$), la funzione controlla che l'operazione rispetti sia la politica corrente più interna (in questo caso φ'), sia quella più esterna φ, sempre tenendo conto della history eventualmente aggiornata. La funzione *check_ops* invoca al suo interno la funzione ausiliaria *check_accept*, fornita assieme al testo dell'homework, che controlla se l'automa rappresentante la politica a seguito dell'operazione termina in uno stato accettante o meno.

```
let rec check_op (c : string) (automa : dfa list) (history : string ):
(value * expr * string) =
    match automa with
        | [] -> (Int(1), UserPolicy(automa), (history ^ c))
        | x :: xs -> if (check_accepts (history ^ c) x) then
                        check_op c xs history
                     else
                        failwith(x.error)
;;
```

La valutazione di *SecurityFrame* va ad aggionare la lista di politiche di sicurezza, andando ad aggiungere in testa la politica fornita come argomento di tale costrutto. In questo modo, si potrà poi andare ad effettuare un controllo tramite la funzione *check_ops* seguendo un approccio LIFO.

Test

Nel codice è presente anche una parte dedicata ad alcuni test per controllare la giusta implementazione del meccanismo richiesto, sia in caso di politiche innestate che di funzioni ricorsive.