An exercise on the implementation of a programming language with security abstractions

Playing with permissions and stack

Playing with permissions and stack with OCAML

OCAML CODE AND REPL EXAMPLES ARE AVAILABLE ON TEAMS

Permissions

A permission is a triple **Set, Entity, List of Allowed Actions**

Set is the set of the elements providing the domain of interest (e.g. File, Network...)

Entity is the name of an element of the domain (if '*' is used then the permission is for all the elements in the domain)

Allowed Actions is the list of allowed actions for a given element of the domain

```
type permission = Permission of string * string * string list

(* A permission domain is a list of permissions*)

type pdomain = permission list
```

Security Actions and Security Stack

A Security Action (secAction) is a operation acting on permissions (e.g grant of a pdomain or enable/disable a permission)

```
type secAction =
  | Grant of pdomain
  | Disable of permission
  | Enable of permission

type pstack = secAction list
```

RUN TIME STRUCTURES

Policy Manager

```
let allows (p : permission) (request : permission) =
  let (Permission (s1, r1, a1)) = p in
  let (Permission (s2, r2, a2)) = request in
  s1 = s2 && (r1 = "*" || r1 = r2) && sublist a2 a1
```

```
let rec sublist l1 l2 =
  match l1 with
  | [] -> true
  | e :: l -> if List.mem e l2 then sublist l l2 else false
```

Policy Manager (cont)

```
let rec domainInspection (set : pdomain) (request : permission) =
  match set with
  | [] -> 0
  | p :: s -> if allows p request then 1
                      else domainInspection s request
```

Policy manager: inspection policy

```
let stackInspection inspectFunction (stack : pstack) (request : permission) =
  match stack with
   | [] -> 0
   | e :: l -> inspectFunction (e :: l) request
```

Inspect the stack: if empty fails
otherwise calls an inspectFunction that applies a given inspection policy

Policy manager: inspection

```
type pdomain = permission list
type pstack = secAction list
```

```
let rec inspect (stack : pstack) (request : permission) =
  match stack with
  | [] -> 1
  | sa :: sl -> (
      match sa with
      | Grant domain ->
          if domainInspection domain request = 1 then inspect sl request else 0
      | Enable p ->
          if allows p request then 1 else stackInspection inspect sl request
      | Disable p ->
          if allows p request then 0 else stackInspection inspect sl request)
```

THE LANGUAGE

```
type expr =
  | CstI of int
  | CstB of bool
  | Var of ide
  | Let of ide * expr * expr
  (* SecLet evaluates the expressions pushing the given pdomain on top of the stack *)
  | SecLet of ide * expr * pdomain * expr
  | Prim of ide * expr * expr
  | If of expr * expr * expr
  (* Lambda: parameters, body and permission domain *)
  | Fun of ide * expr * pdomain
  | Call of expr * expr
```

```
(* Return true iff that permission is allowed *)
 | DemandPermission of permission
 (* returns if DemandPermission is true otherwise return false (Int 0) *)
 | OnPermission of permission * expr
 (* Aborts if permission is not enabled *)
 | CheckPermission of permission
 (* Evaluates the expression with the permission enabled *)
 | Enable of permission * expr
 (* Evaluates the expression with the permission disabled *)
 | Disable of permission * expr
 (* Evaluates the expression pushing the secAction on top of the stack *)
 | SecBlock of secAction * expr
 (* Reads a file iff is allowed otherwise aborts *)
 | ReadFile of string
   (* Send and evaluates expr to a file iff is allowed otherwise aborts *)
 | SendFile of expr * string
 | Abort of string
```

THE INTERPRETER

```
let rec eval (e : expr) (env : value env) (stack : pstack) : value =
 match e with
 | CstI i -> Int i
 | CstB b -> Int (if b then 1 else 0)
 | Var x -> lookup env x
 | Let (x, eRhs, letBody) ->
     let xVal = eval eRhs env stack in
     let letEnv = (x, xVal) :: env in
     eval letBody letEnv stack
 | SecLet (x, eRhs, secSet, letBody) ->
     (* xVal is evaluated in the current stack *)
     let xVal = eval eRhs env stack in
     let letEnv = (x, xVal) :: env in
     let letStack = Grant secSet :: stack in
     (* letBody is evaluated in the updated stack *)
     eval letBody letEnv letStack
]
```

```
| Prim (ope, e1, e2) -> (
    let v1 = eval e1 env stack in
    let v2 = eval e2 env stack in
    match (ope, v1, v2) with
    | "*", Int i1, Int i2 -> Int (i1 * i2)
    | "+", Int i1, Int i2 -> Int (i1 + i2)
    | "-", Int i1, Int i2 -> Int (i1 - i2)
    | "=", Int i1, Int i2 -> Int (if i1 = i2 then 1 else 0)
    | "<", Int i1, Int i2 -> Int (if i1 < i2 then 1 else 0)
    | _ -> failwith "unknown primitive or wrong type")
 | If (e1, e2, e3) -> (
    match eval e1 env stack with
    | Int 0 -> eval e3 env stack
    | Int _ -> eval e2 env stack
    | _ -> failwith "eval if")
| Fun (x, fBody, sec) -> Closure (x, fBody, sec, env)
| Call (eFun, eArg) -> (
    let fClosure = eval eFun env stack in
    match fClosure with
    | Closure (x, fBody, sec, fDeclEnv) ->
       (* xVal is evaluated in the current stack *)
       let xVal = eval eArg env stack in
       let fBodyEnv = (x, xVal) :: fDeclEnv in
       let fBodyStack = Grant sec :: stack in
       (* fBody is evaluated in the updated stack *)
       eval fBody fBodyEnv fBodyStack
    | _ -> failwith "eval Call: not a function")
| DemandPermission p -> Int (stackInspection inspect stack p)
  | OnPermission (p, e) ->
      if eval (DemandPermission p) env stack = Int 1 then eval e env stack
      else Int 0
  | CheckPermission p ->
      if eval (DemandPermission p) env stack = Int 1 then Int 1
      else eval (Abort "CheckPermission failed") env stack
  | Enable (p, e) -> eval e env (Enable p :: stack)
  | Disable (p, e) -> eval e env (Disable p :: stack)
  | SecBlock (sec, e) -> eval e env (sec :: stack)
| ReadFile f ->
     if
       eval (DemandPermission (Permission ("File", f, [ "r" ]))) env stack  = Int 1
     then Int 1 (* do read *)
     else eval (Abort ("No Read Permission for " ^ f)) env stack
  | SendFile (e, f) ->
     if
       eval (DemandPermission (Permission ("File", f, [ "w" ]))) env stack  = Int 1
     then eval e env stack (* do write *)
     else eval (Abort ("No Write Permission for " ^ f)) env stack
  | Abort msg -> failwith msg
```

More things to do ....

- From C-like boolean to booleam values
- Ocaml Option typoe to manage the inspection of permissions
- A full fledged language for security actions
- ......