HOMEWORK ASSESSMENT

LANGUAGE BASED TECHNOLOGY FOR SECURITY
(a.a. 2023/2024)

Andrea Fabbri
Giacomo Lombardi
Giacomo Maldarella

Index

Overview

In this project it has been implemented an interpreter of an intermediate representation that is able to manage Trusted and Untrusted code, exploiting features like dynamic taint analysis and static information flow control.
The project is divided into six different files:

DIR

- → **AST.ML**
- → **UTILITIES.ML**
- → **EVAL.ML**
- → **TYPE_CHECKING.ML**
- → **TEST.ML**
- → **MAKEFILE**

1 Ast.ml

This file contains the Abstract Syntax Tree of the interpreter, consisting of all the possible types of the expressions and all the possible return values of the evaluation. Moreover, this file contains the definition of types for the trusted entities, for the information flow and for the environment.

1.1 Environment

Three different environments have been implemented:

```
(* 'env' is a list of: identifier ('ide') - value ('v') - taintness ('bool')*)
type 'v env = (ide * 'v  * bool) list
(* 'c_env' is a list of: identifier ('ide') - confidential level ('conf_level')*)
type 'v c_env = (ide * conf_level) list
(*'priv_TB' is a list of: identifier ('ide') - security level ('sec_level')*)
type 'v priv_TB = (ide * sec_level) list
```

- **Env**: The general environment, used to bind a variable with its value and with the level of taintness.
- **C_env**: Environment used to bind entities with their confidentiality level.
- **Priv_Tb**: Is a list containing the association between a trusted variable and its security level '*Secret*', '*Public*' or '*Handler*'.

1.2 Type definition

The approach used to manage Trusted code and Confidential code is based on types. In fact, two different types have been developed: '*sec_level*' and '*conf_level*'.

```
type ide = string

type sec_level=
  | Secret
  | Public
  | Handler

type conf_level =
  | High
  | Low
```

Sec_level: To distinguish between trusted and untrusted code a trust block has been introduced. Inside a trust block all the statements have a security level defined by '*sec_level*'. A trusted expression can be:

- **Secret**: Secret member of the block.
- **Public**: Public member of the block.
- **Handler:** A public member of the block that can be reached from outside the block.

Conf_level: To prevent information leaks, a confidentiality level has been assigned to each variable. Only two different levels are possible:

- **Low:** Assigned to not confidential variables, like normal variables and handlers.
- **High:** Assigned to confidential variables, like secret members of trusted blocks.

1.3 Abstract syntax tree

```
type exp =
| Eint of int
| Ebool of bool
| Estring of string
| Var of ide
| Prim of ide * exp * exp
| If of exp * exp * exp
| Let of ide * exp * exp
| LetIn of ide * exp * exp  (* to test the taint analysis *)
| Fun of ide * exp
| Apply of exp * exp
| Trustblock of exp
| LetSecret of ide * exp * exp
| LetHandle of ide * exp
| LetPublic of ide * exp * exp
| Include of exp
| Execute of exp
| CallHandler of exp * exp
| Assert of exp
| End                         (* end of a trust block*)
```

Exp consist of all the possible expressions that the interpreter can evaluate. Some of those expressions are:

- **Let**: used for the definition of a statement outside a trust block.
- **LetIn**: Construct that emulates a user input, included to better test the taint analysis.
- **Fun and Apply**: Expressions used to define and invoke a function.
- **Trustblock**: Expression that define a trust block, that is a block that only contains trusted elements, made by a trusted user.
- **LetSecret:** Same as Let but it can be used only in a trust block, and it's used to define a secret member of the block.
- **LetPublic:** Same as Let but it can be used only in a trust block, and it's used to define a public member of the block.
- **LetHandle:** This statement is used to define a Handler, that's a member of the block that can be accessed from outside of the block. It takes as parameters a public member and another statement or the End.
- **Include:** Expression used to include untrusted code (plugin) and therefore, it can't be adopted inside a trust block. The plugin can't be accessed or checked by the interpreter (black box).
- **Execute:** The statement used to run the plugin.
- **CallHandler:** The expression used to invoke a handler from outside the trust block. It takes as parameters the name of the trust block and the handler.
- **Assert**: Auxiliary function that checks if a variable is tainted or not.
- **End:** The last statement that close a trust block and returns a '*Secure_Block*'.

The ‘*evt*’ type defines all the possible return values of the evaluation, that are:

- **Int**
- **Bool**
- **String**
- **Closure:** Contains the result of the evaluation of a function.
- **ClosureInclude:** Contains the result of the evaluation of an include block.
  ```
  | ClosureInclude of exp * evt env
  ```
- **Secure_Block:** Contains the result of the evaluation of a trust block. It consists of the updated ‘*priv_TB*’ list and the new environment ‘*newEnv*’.
  ```
  | Secure_Block of evt priv_TB * evt env
  ```

2 Utilities.ml

The file “utilities.ml” contains the utilities functions used for the evaluation and for the type checking of the code.

Eval:

- **lookup**: Function that search for a variable in the environment ‘*env*’ and return its value.
- **t_lookup:** Same as lookup but returns the taintness.
- **secret_lookup**: Returns true if it finds a variable with a “*Secret*” security level in ‘*priv_TB*’.
- **public_lookup**: Returns true if it finds a variable with a “*Public*” security level in ‘*priv_TB*’.
- **handle_lookup**: Returns true if it finds a variable with a “*Handler*” security level in ‘*priv_TB*’. It takes an expression as input.

  ```
  (ide * sec_level) list -> exp -> bool
  let rec handle_lookup priv_TB x =
    match priv_TB, x with
    | [], _ -> false
    | _, Var y -> (
        match priv_TB with
        | [] -> false
        | (z, s) :: r -> if y = z && s = Handler then true else handle_lookup r x)
    | _, _ -> false
  ```

- **bind_env**: Environment binding function, taking as input a variable, its value and its level of taintness.
- **bind_tb**: Binding function for ‘*priv_TB*’, taking as input a variable and its security level.

Type checking:

- **c_lookup**: Used to returns the confidentiality level of a variable doing the type checking.

  ```
  (string * 'a) list -> string -> 'a
  let rec c_lookup env x =
    match env with
    | [] -> failwith (x ^ " not found in c_lookup")
    | (y, c) :: r -> if x = y then c else c_lookup r x
  ```

- **lattice_checking**: Takes two variables as input α and β, and returns true if β has a security level greater or equal than α.

  ```
  'a -> 'b -> 'c
  let lattice_checking a b = match a,b with
    | Low, _ -> true
    | _, High -> true
    | _, _ -> false;;
  ```

- **join**: Makes the join of the confidentiality level of two different variables.

  ```
  'a -> 'b -> 'b
  let join e e' = if e = Low then e' else High ;;
  ```

- **bind_ cf**: Binding function for ‘*c_env*’, taking as input a variable and its confidentiality level.

  ```
  (ide * conf_level) list -> ide -> conf_level -> (ide * conf_level) list
  let bind_cf c_env (x:ide) (cf:conf_level) = (x,cf)::c_env
  ```

3 Eval.ml

In the eval.ml file we can find the interpreter of our functional programming language.

```
exp -> evt env -> bool -> (ide * sec_level) list -> bool -> evt * bool
let rec eval (e : exp) (env:evt env) (t : bool) (priv_TB: evt priv_TB) (inTrustBlock: bool): evt * bool
```

The recursive function eval takes five input parameters and returns a pair consisting of two elements.
Input Parameters:

- **e**: Represents the expression to be evaluated.
- **env**: The evaluation environment that keeps track of the values associated with variables during the evaluation of the expression.
- **t**: A Boolean flag indicating whether the current context is tainted or untainted. This is used to manage integrity.
- **priv_tb**: This parameter is used to keep track of information that can be processed within trust blocks by associating each variable with its security level.
- **inTrustBlock**: A Boolean flag indicating whether the current expression is being evaluated within a trust block or not.

Output Parameters:

- **Param1 (evt)**: Represents the value resulting from the evaluation of the expression '*e*'.
- **Param2 (bool)**: Indicates whether the resulting value is tainted (true) or untainted (false).

Let's now evaluate the individual expressions within the function.

3.1 General expressions

```
| Eint n -> (Int n, t)
| Ebool b -> (Bool b, t)
| Estring s -> (String s, t)
| Var x ->
  if inTrustBlock  && (secret_lookup priv_TB x) || (pub_lookup priv_TB x) then
    (lookup env x, t_lookup env x)
  else
    if secret_lookup priv_TB x then failwith "Can't access a secret entity"
    else if handle_lookup priv_TB (Var x) || not (pub_lookup priv_TB x) then (lookup env x, t_lookup env x)
    else failwith "Can't access a variable not trusted"
| Prim (op, e1, e2) ->
  begin
    let v1, t1 = eval e1 env t priv_TB inTrustBlock in
    let v2, t2 = eval e2 env t priv_TB inTrustBlock in
    match (op, v1, v2) with
    | "*", Int i1, Int i2 -> (Int (i1 * i2), t1 || t2)
    | "+", Int i1, Int i2 -> (Int (i1 + i2), t1 || t2)
    | "-", Int i1, Int i2 -> (Int (i1 - i2), t1 || t2)
    | "=", Int i1, Int i2 -> (Bool (if i1 = i2 then true else false), t1 || t2)
    | "<", Int i1, Int i2 -> (Bool (if i1 < i2 then true else false), t1 || t2)
    | ">", Int i1, Int i2 -> (Bool (if i1 > i2 then true else false), t1 || t2)
    | _, _, _ -> failwith "Unexpected primitive."
  end
| If (e1, e2, e3) ->
  begin
  let v1, t1 = eval e1 env t priv_TB inTrustBlock in
  match v1 with
    | Bool true -> let v2, t2 = eval e2 env t priv_TB inTrustBlock in (v2, t1 || t2)
    | Bool false -> let v3, t3 = eval e3 env t priv_TB inTrustBlock in (v3, t1 || t3)
    | _ -> failwith "Unexpected condition."
  end
```

We won't delve into the detailed explanation of these constructs since we've already covered them in class. Nonetheless, it's crucial to highlight that they have been implemented with all the notion of taint analysis.

3.2 Let

```
| Let (x, value, body) ->
    if inTrustBlock then failwith "You can't use Let in a TrustBlock"
    else
      let v, taintness = eval value env t priv_TB inTrustBlock in
      let newEnv = bind_env env x v taintness in
      eval body newEnv t priv_TB inTrustBlock
```

If the current context is within a trust block, the interpreter raises an exception because it is not permitted to use Let within a trust block.
If Let is used properly the function recursively evaluates the *'value'* expression to determine its value *'v'* and its taint status *'t'*.
Then, it updates the environment env to include a new binding between the variable *'x'* and the value *'v'*, along with its taint status, using the *'bind_env'* function.
Finally, it recursively evaluates the *'body'* of the Let expression in the new environment *'newEnv'*.

3.3 LetIn

```
| LetIn (x, value, body) ->
    if (inTrustBlock) then failwith "You can't take a var from input in a TrustBlock"
    else
    let v, taintness = eval value env true priv_TB inTrustBlock in
    let newEnv = bind_env env x v taintness in
    eval body newEnv true priv_TB inTrustBlock
```

The LetIn clause mimics scenarios where data from untrusted sources (such as external inputs) might be introduced into the program, allowing for robust handling of potential security risks through taint analysis.
The interpreter menages the LetIn operations in the same of the Let clause. The only difference is that here, the true parameter passed to *'eval'* indicates that the value is considered tainted a-priori.

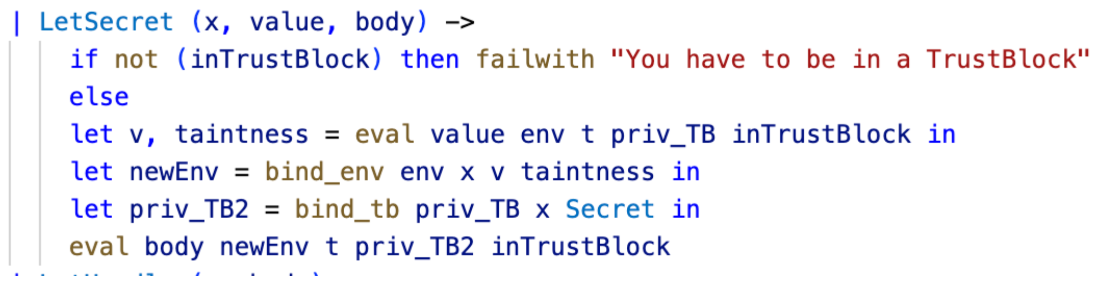
3.4 Trustblock

```
| Trustblock content ->
    if inTrustBlock then failwith "You can't create a TrustBlock inside a TrustBlock"
    else
    if t then failwith "The content of the TrustBlock is tainted"
    else
      eval content env t priv_TB true
```

The function begins by enforcing critical contextual constraints to maintain the integrity of trust blocks. It first checks whether the current evaluation context is within a trust block. This restriction ensures that trust blocks cannot be nested within each other, preventing potential complexities

and security vulnerabilities that may arise from nested security contexts.
After verifying the absence of nested trust block, the function proceeds to check the taint status *‘t’* of the current context. If the context is tainted, indicating the presence of potentially compromised data or operations, it raises an exception. This preemptive check prevents any tainted information from entering or contaminating the trust block, ensuring the sanctity of its content.
Upon passing the initial checks for nested trust block and tainted status, the function proceeds to evaluate the content enclosed within the trust block. Since the evaluation is performed within a trust block, *‘inTrustBlock’* is defined true.

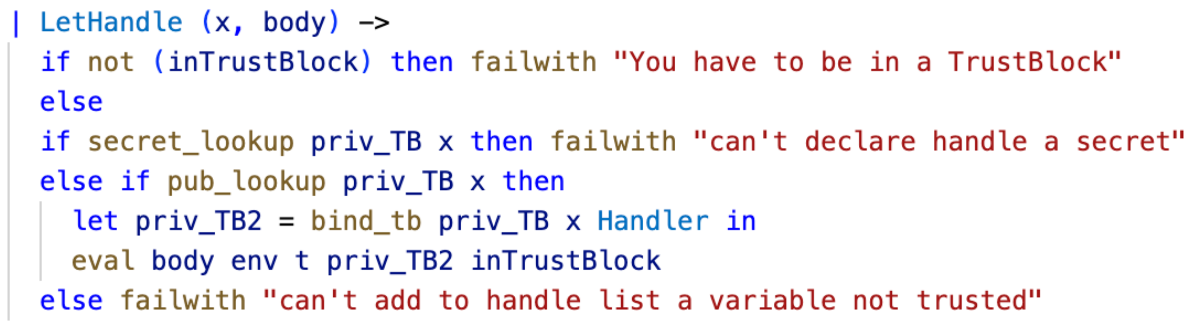
3.5 LetSecret



```
| LetSecret (x, value, body) ->
    if not (inTrustBlock) then failwith "You have to be in a TrustBlock"
    else
    let v, taintness = eval value env t priv_TB inTrustBlock in
    let newEnv = bind_env env x v taintness in
    let priv_TB2 = bind_tb priv_TB x Secret in
    eval body newEnv t priv_TB2 inTrustBlock
```

The LetSecret construct is designed for handling sensitive data within trust block, ensuring secure management of confidential information.
Most of the operations performed by the interpreter to handle the LetSecret are equal to the Let clause.
Additionally, a new entry is made in the *‘priv_TB’* to associate variable *‘x’* with a *‘Secret’* security level, ensuring sensitive handling within the trust block context. The body expression is then evaluated within the updated environment *‘newEnv’* and the list *‘priv_TB2’*, ensuring that subsequent operations maintain the integrity of sensitive data.

3.6 LetHandle



```
| LetHandle (x, body) ->
  if not (inTrustBlock) then failwith "You have to be in a TrustBlock"
  else
  if secret_lookup priv_TB x then failwith "can't declare handle a secret"
  else if pub_lookup priv_TB x then
    let priv_TB2 = bind_tb priv_TB x Handler in
    eval body env t priv_TB2 inTrustBlock
  else failwith "can't add to handle list a variable not trusted"
```

The LetHandle construct is specifically designed for managing variables that act as handlers.
If we are not in a trust block an exception is raised. Otherwise, it checks if the variable *‘x’* is marked as secret in *‘priv_TB’* using *‘secret_lookup’*. If *‘x’* is secret an exception is raised with the message "can't declare handle a secret" ensuring that secret variables cannot be declared as handlers. If x is not secret but is publicly accessible, it proceeds to create a new entry in *‘priv_TB’* that associate at *‘x’* the security level *‘Handler’*.
Finally, the body expression is evaluated within the updated list ‘priv_TB2’.
If *‘x’* is neither secret nor publicly accessible, an exception is raised with the message "can't add to

handle list a variable not trusted". This construct enforces strict security controls, ensuring that only trusted and appropriately classified variables are managed as handlers within trust block.

3.7 LetPublic

```
| LetPublic (x, value, body) ->
    if not (inTrustBlock) then failwith "You have to be in a TrustBlock"
    else
    let v, taintness = eval value env t priv_TB inTrustBlock in
    let newEnv = bind_env env x v taintness in
    let priv_TB2 = bind_tb priv_TB x Public in
    eval body newEnv t priv_TB2 inTrustBlock
```

The LetPublic operations is managed by the interpreter in the same way of LetSecret. The only difference is that it updates the list ‘priv_TB2’ by associating *‘x’* with the *‘Public’* security level. This ensures that the variable x is marked as public within the trust block, distinguishing it from secret or handle variables.

3.8 Include

```
| Include content -> (
  if inTrustBlock then failwith "You can't include a file in a TrustBlock"
  else (ClosureInclude (content, env), t))
```

The Include construct allows our language to integrate external code, treating it as a black box whose internal workings cannot be inspected, with visibility limited to inputs and outputs.
It checks if the current context is within a trust block. If it is not the function returns a tuple ‘*ClosureInclude (content, env), t*’. Here, ‘*ClosureInclude*’ encapsulates the content along with the current environment ‘*env*’, preserving any taintness information ‘*t*’.

3.9 Execute

```
| Execute e ->
  if inTrustBlock then failwith "You can't execute a file in a TrustBlock"
  else
  begin
    let v1, t1 = eval e env t priv_TB inTrustBlock in
    match v1 with
    | ClosureInclude (iBody, iEnv) -> eval iBody iEnv true priv_TB inTrustBlock
    | _ -> failwith "Unexpected condition."
  end
```

The Execute construct is used to run external code integrated before with the include construct.
First, it checks if the current evaluation context is within a trust block.
Then, the function evaluates the expression ‘*e*’ with the current environment ‘*env*’, taint status ‘*t*’, and ‘*priv_TB*’. The result of this evaluation is stored in ‘*v1*’ (evt) and ‘*t1*’ (bool).
After that it checks if ‘v1’ is a ‘*ClosureInclude’* construct and proceeds to evaluate the ‘*iBody*’ in the ‘*iEnv*’, with the taint status set to true, indicating that the external code is considered tainted by

default. This decision is made to enforce a conservative security measure, ensuring that any code coming from external sources is treated as potentially unsafe.
Also, the decision to return a closure from the call of the include was made to have a referment to the external code '*iBody*' and to evaluate it in the environment of the declaration of the include construct '*iEnv*' due to the Static Scoping.

3.10 CallHandler

```
| CallHandler (trustblock, handler) -> (
    let v1, t1 = eval trustblock env t priv_TB inTrustBlock in
      match (v1, t1) with
      | Secure_Block(priv_TB1, secondEnv), t1 ->
        if not (handle_lookup priv_TB1 handler)
          then failwith "You can access only an handle var"
        else
          eval handler secondEnv t1 priv_TB1 inTrustBlock
      | _ -> failwith "the access must be applied to an trustblock")
```

The CallHandler construct is used to invoke a handler previously declared.
The trust block parameter is evaluated first to obtain its value *'v1'* and taint status *'t1'*. This ensures that the trust block is correctly identified before proceeding.
If the result of the evaluation is a *'Secure_Block'* (which includes a specific *'priv_TB1'* and environment *'secondEnv'*), it checks if the handler exists with the '*handle_lookup*' function.
If the handler is found, it proceeds to evaluate the handler with *'secondEnv'* and *'priv_TB1'*, preserving the taint status *'t1'*. This adheres to the principle of static scoping, which ensures that the variables and security policies within the handler are bound to the values and policies of the environment where it was originally declared. Evaluating the handler in its original environment avoids potential conflicts and ensures the integrity and isolation of both the main program and the trust block.

3.11 Assert

```
| Assert param -> (
      let v1, t1 = eval param env t priv_TB inTrustBlock in
      if t1 = true then failwith "Assertion failed"
      else (Int 1,true)
      )
```

The Assert construct serves to validate a parameter '*param*' by evaluating its taint status. The interpreter evaluates param within the current environment '*env*', taint status '*t*', and '*priv_TB*', respecting the context set by *'inTrustBlock'*.
Then it checks if the value of taintness '*t1*' is true, this means that '*param*' is tainted.
Indeed, if the value is not tainted as an example pattern, we output (Int 1) and (true).

3.12 End

```
| End ->  Secure_Block(priv_TB,env),t
```

The End construct marks the conclusion of a trust block, encapsulating its internal operations. It returns a ‘*Secure_Block*’ tuple: containing the final state of the list ‘*priv_TB’* and the environment ‘*env*’, and the tainteness ‘t’.

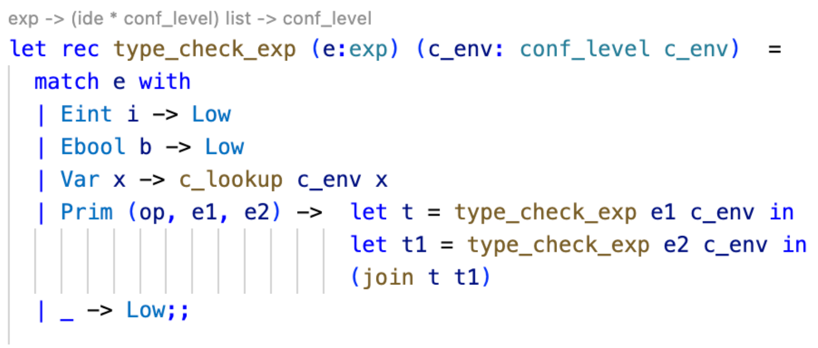
4 Type_checking.ml

This part of the interpreter ensures that there are no confidentiality leaks within the analyzed program. Specifically, it verifies that outputs with *'Low'* confidentiality level do not depend on inputs with a *'High'* security level.
The implementation of the type checker involves the following key components:

- Two levels of confidentiality, High and Low.
- A confidentiality lattice Low <= High.
- A typing environment *'c_env'* that maps entities to confidentiality levels.

The core of the type checker is implemented by the *'type_check_com'* function, which relies on several auxiliary functions: *'lattice_checking'*, *'join'*, *'bind_cf'*, and *'type_check_exp'*.

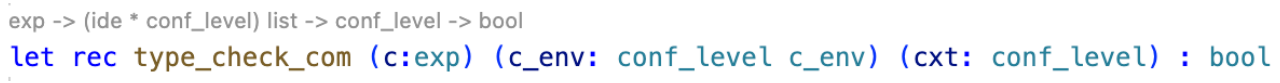
4.1 Type_check_exp



```
exp -> (ide * conf_level) list -> conf_level
let rec type_check_exp (e:exp) (c_env: conf_level c_env)  =
  match e with
  | Eint i -> Low
  | Ebool b -> Low
  | Var x -> c_lookup c_env x
  | Prim (op, e1, e2) ->  let t = type_check_exp e1 c_env in
                          let t1 = type_check_exp e2 c_env in
                          (join t t1)
  | _ -> Low;;
```

This function recursively returns the confidentiality level of an entire expression by joining the levels of its subexpressions. We assume that the default case evaluates the expressions to *'Low'*.

4.2 Type_check_com



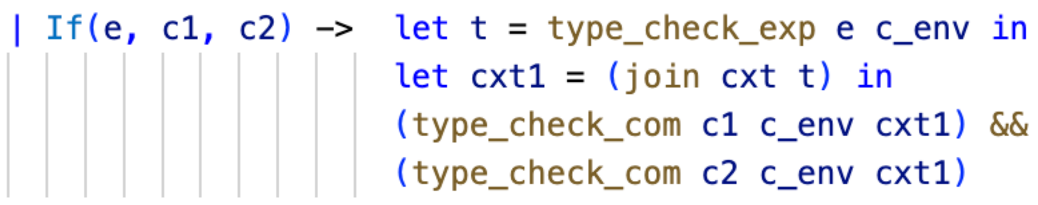
```
exp -> (ide * conf_level) list -> conf_level -> bool
let rec type_check_com (c:exp) (c_env: conf_level c_env) (cxt: conf_level) : bool
```

The *'type_check_com'* function recursively type checks a program, returning a Boolean value that is true if the program is well-typed. If the program is not well typed, it raises an error. This function takes three parameters:

- The program to be analyzed *'exp'*.
- The typing environment *'c_env'*.
- The context *'cxt'*.

Here is a detailed breakdown of how *'type_check_com'* handles different commands.

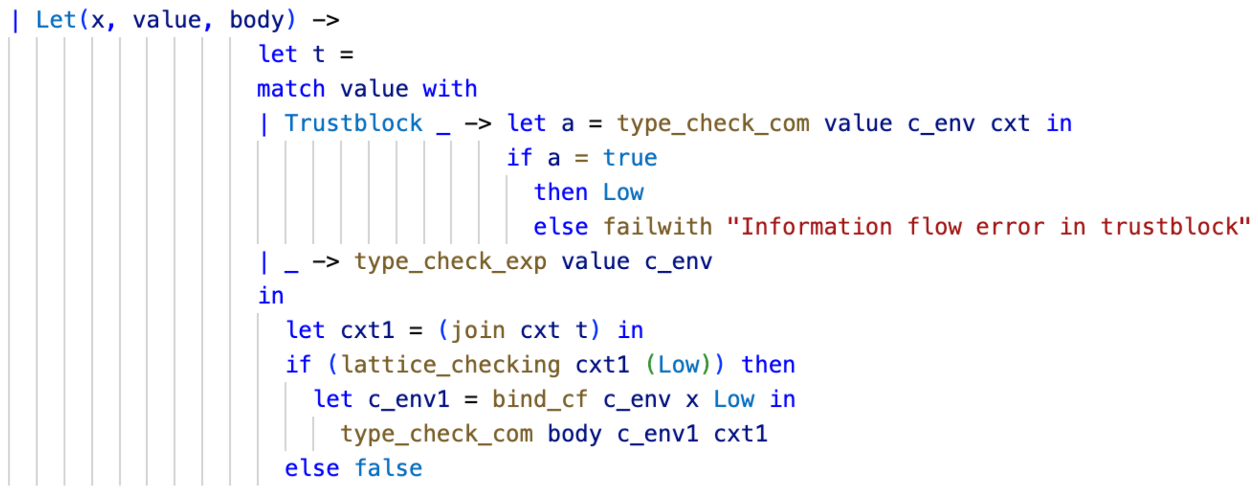
4.3 If



```
| If(e, c1, c2) ->  let t = type_check_exp e c_env in
                    let cxt1 = (join cxt t) in
                    (type_check_com c1 c_env cxt1) &&
                    (type_check_com c2 c_env cxt1)
```

First, the type checker evaluates the type of the condition *'e'* using the environment *'c_env'*. Then, it combines the current context *'cxt'* with the confidentiality level of *'e'* using *'join'*. After that, it checks the commands in both branches of the If (*'c1'* and *'c2'*) with the new context *'cxt1'*.

4.4 Let

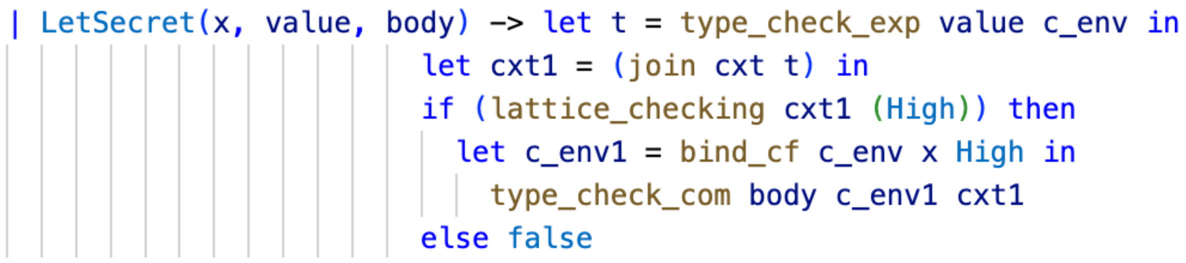


```
| Let(x, value, body) ->
                        let t =
                        match value with
                        | Trustblock _ -> let a = type_check_com value c_env cxt in
                                         if a = true
                                           then Low
                                           else failwith "Information flow error in trustblock"
                        | _ -> type_check_exp value c_env
                        in
                          let cxt1 = (join cxt t) in
                          if (lattice_checking cxt1 (Low)) then
                            let c_env1 = bind_cf c_env x Low in
                              type_check_com body c_env1 cxt1
                          else false
```

To check if the operation is well-typed, the function checks the type of the value assigned to *‘x’*. If *‘value’* is *‘Trustblock’*, it calls *‘type_check_com’* on *‘value’* with the environment *‘c_env’* and the current context. If the trust block is typed correctly, the function assigns *‘Low’* to it; otherwise, it raises an error.

The function combines the current context *‘cxt’* with *‘t’* using *‘join’* and checks if the new context *‘cxt1’* is compatible with *‘Low’* using *‘lattice_checking’* (since variables defined with Let are considered to have low confidentiality). If compatible, indicating that the assignment doesn’t leak confidential information, the function binds *‘x’* with *‘Low’* in the environment and checks the body with the new context. It returns false if the assignment is not well-typed.

4.5 LetSecret

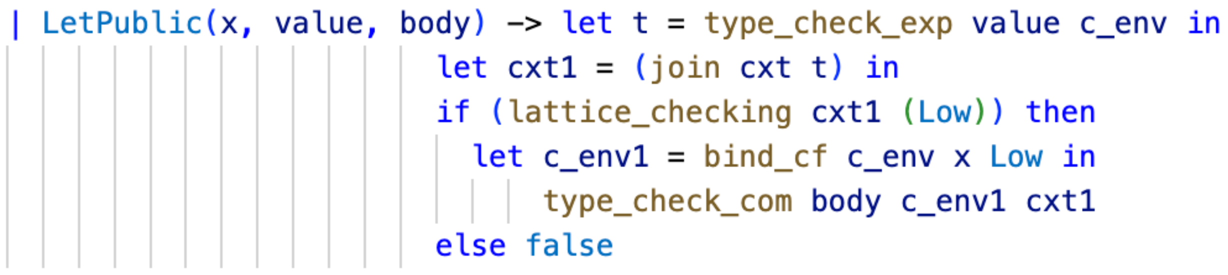


```
| LetSecret(x, value, body) -> let t = type_check_exp value c_env in
                               let cxt1 = (join cxt t) in
                               if (lattice_checking cxt1 (High)) then
                                 let c_env1 = bind_cf c_env x High in
                                   type_check_com body c_env1 cxt1
                               else false
```

The LetSecret command declares secret variables within a trust block, assigning ‘*High*’ confidentiality to *‘x’*.

First, the type checker retrieves the type of the value assigned to *‘x’*. Then, combines the current context with *‘t’* and checks if the new context *‘cxt1’* is compatible with *‘High’*. If compatible, it binds *‘x’* with *‘High’* in the environment and recursively checks the *‘body’* with the new context calling *‘type_check_com’*. It returns false if the assignment is not-well typed.

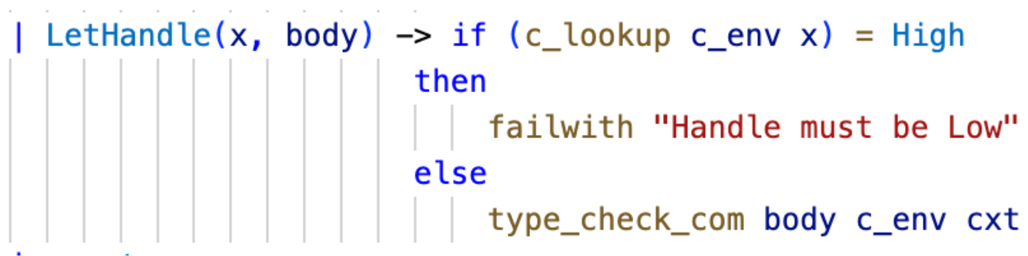
4.6 LetPublic



```
| LetPublic(x, value, body) -> let t = type_check_exp value c_env in
                               let cxt1 = (join cxt t) in
                               if (lattice_checking cxt1 (Low)) then
                                 let c_env1 = bind_cf c_env x Low in
                                   type_check_com body c_env1 cxt1
                               else false
```

The type checking on LetPublic command is implemented similarly to the one on LetSecret, with the only difference that, since it is the Let operation for declaring public variables within a trust block, by hypothesis *‘x’* is assigned *‘Low’* level of confidentiality.

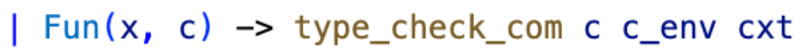
4.7 LetHandle



```
| LetHandle(x, body) -> if (c_lookup c_env x) = High
                        then
                          failwith "Handle must be Low"
                        else
                          type_check_com body c_env cxt
```

The type checker verifies that *‘x’* is not of high level. If *‘x’* is *‘High’*, it raises an error; otherwise, it checks the *‘body’* in the current context.

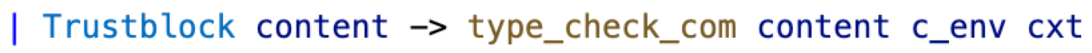
4.8 Fun



```
| Fun(x, c) -> type_check_com c c_env cxt
```

The Fun command handles function declarations. The type checker recursively checks the body *‘c’* of the function in the type environment and current context.

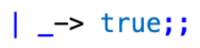
4.9 Trustblock



```
| Trustblock content -> type_check_com content c_env cxt
```

The Trustblock command allows the creation of blocks of code that are trusted. The type checker recursively calls *‘type_check_com’* on the content with the environment and the current context.

4.10 Default case



```
| _-> true;;
```

The default case handles any other unspecified command, always returning true.

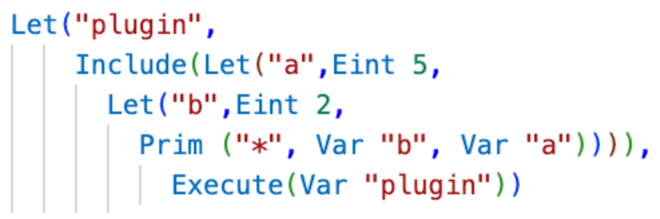
Test.ml

This section provides an overview of the suite of tests implemented to verify the correctness and functionality of the OCaml interpreter with taint analysis and type checking. Each test is designed to evaluate a specific aspect of the interpreter, covering a variety of scenarios.
Below is a list of the tests implemented:

- **Test 1**: Nesting a *'Let'* inside another *'Let'*.
- **Test 2**: Use of *'LetHandle'* functionality.
- **Test 3**: Use of *'CallHandler'* functionality.
- **Test 4**: Verification of *'CallHandler'* functionality when the handler is a function.
- **Test 5**: Use of *'Include'* and *'Execute'*.
- **Test 6**: Use of *'LetIn'*.
- **Test 7**: Attempt of a trust block accessing a secret variable declared in another trust block.
- **Test 8**: Attempt to assign the value of a secret variable to a public one, verifying information flow between a high confidentiality variable and a low confidentiality variable.
- **Test 9**: Plugin access to a secret variable.
- **Test 10**: Plugin access to a handle.
- **Test 11**: *'Include'* inside a trust block.
- **Test 12**: Attempt of using *'LetPublic'* outside a trust block.
- **Test 13:** Declaration of a trust block within a trust block.
- **Test 14**: Operation between tainted and untainted values.
- **Test 15**: *'CallHandler'* on a non-handler variable.
- **Test 16**: Verification of *'Assert'* functionality.

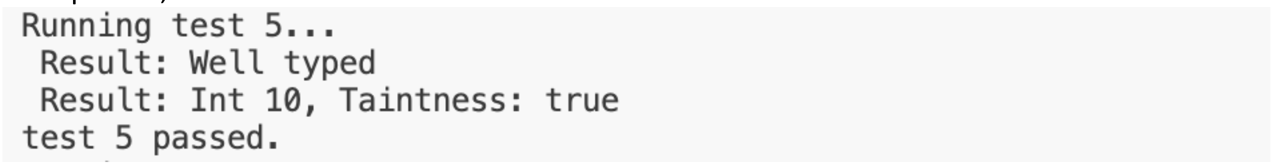
Let's see an in-depth explanation of tests 5, 8, and 14.

Test 5



```
Let("plugin",
    Include(Let("a",Eint 5,
      Let("b",Eint 2,
        Prim ("*", Var "b", Var "a")))),
          Execute(Var "plugin"))
```

This test uses a *'Let'* expression that binds the identifier "plugin" to the result of the *'Include'* construct. Inside the *'Include'* there are operations that the interpreter cannot inspect. The *'Execute'* construct then executes the "plugin".
As expected, the result of the test is:



```
Running test 5...
 Result: Well typed
 Result: Int 10, Taintness: true
test 5 passed.
```

Test 8

```
Let("mytrustB",
    Trustblock(
      LetSecret("y", Eint 22,
        LetPublic("x",Var "y",
          LetHandle("x",End)))),
            Let("a",Eint 5,
              Let("b", Eint 5,
                Prim ("*", Var "b", Var "a"))))
```

This test creates a *'Let'* expression that binds "mytrustB" to a *'Trustblock'*. Inside the trust block: *'LetSecret'* binds "y" to the integer value 22, *'LetPublic'* binds "x" to the value of "y", and *'LetHandle'* ensures the value of "x" is accessible outside "*mytrustB*". Outside the trust block, additional *'Let'* and operations are performed to complete the test.
Since, by assumption, public variables have low confidentiality level while secret variables have high confidentiality level, the result is:

```
Running test 8...
test 8 failed: Information flow error in trustblock
```

Test 14

```
Let("plugin",
  Include(Let("a", Eint 2, Let("b", Eint 4, Prim("+", Var "a", Var "b")))),
    Let("Executed", Execute(Var "plugin"),
      Let("g", Eint 5,
        Prim("*", Var "g", Var "Executed"))))
```

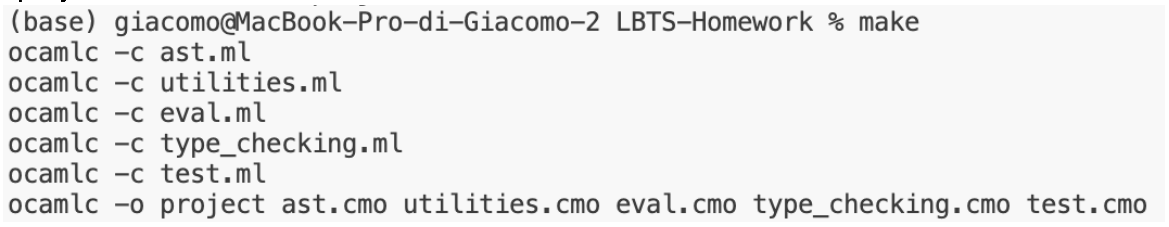
This test uses the *'Let'* expression that binds "*plugin*" to the result of the *'Include'* construct, which performs an addition operation between "a" and "b". Then, the *'Execute'* construct executes the "plugin", binding "*Executed*" to the result of "a"+"b" (which is 6). Finally, the *'Let'* expression multiplies "g" (which is 5) by the value of "*Executed*".
Since the final multiplication is between "g" (untainted) and "*Executed*" (tainted), the result of the test is:

```
Running test 16...
 Result: Well typed
 Result: Int 30, Taintness: true
test 16 passed.
```

How to Run

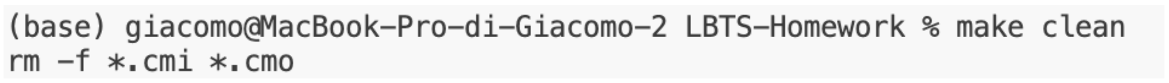
Inside the directory of the project, there's the '*Makefile*'.
To compile and run the project is sufficient to:

- Call "make" on the terminal to Compile all the files and to link them inside the executable "project".



```
(base) giacomo@MacBook-Pro-di-Giacomo-2 LBTS-Homework % make
ocamlc -c ast.ml
ocamlc -c utilities.ml
ocamlc -c eval.ml
ocamlc -c type_checking.ml
ocamlc -c test.ml
ocamlc -o project ast.cmo utilities.cmo eval.cmo type_checking.cmo test.cmo
```

- Run the program by calling the executable "./project" on the terminal.
- After the compilation, it is possible to delete all the compiled file by the command "make clean"



```
(base) giacomo@MacBook-Pro-di-Giacomo-2 LBTS-Homework % make clean
rm -f *.cmi *.cmo
```