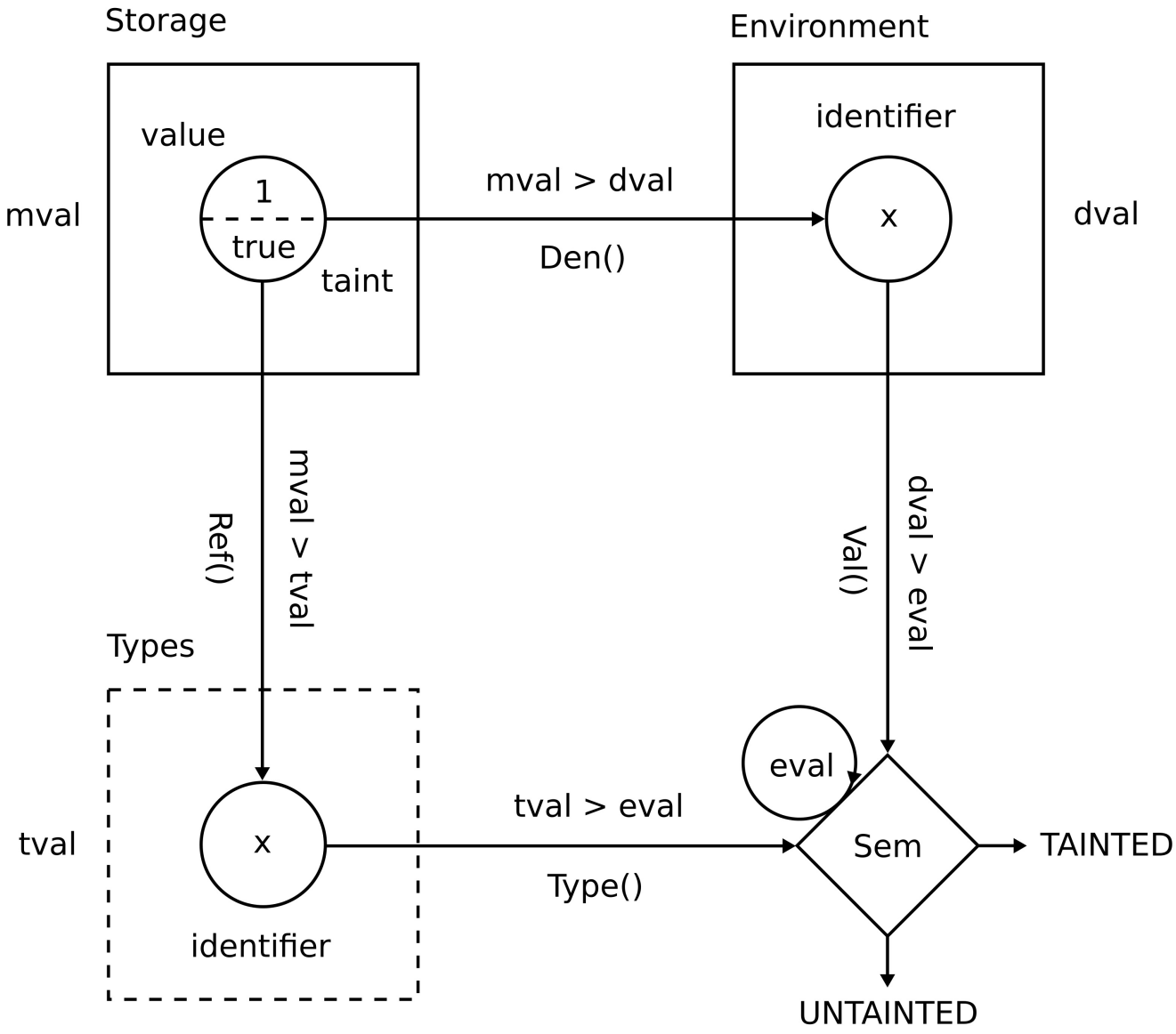
Operational recursive interpreter created in OCaml with operations on strings, parser for reflection and dynamic information control flow through taint analysis.

Index

Project



Our interpreter is written in Ocaml, characterized with an operational semantic and includes functions, block and procedures. We have a dynamic enviroment and a static environment. In our dymamic enviroment we associate identifiers with denotable values, in the static enviroment we associate a boolean value (true,false) with the type Tainted or Untainted. We use in our enviroment a static scoping, this means that in each point of the program the enviroment is identified by the lessical structures of the code. In our dynamic domain each eval, dval, mval and tval type is a tuple: the first element is the value, the second is a boolean. If the last is true, the element is tainted, otherwise it is untainted.

The new environment gives the possibility to make static analysis on the code of the interpreter. We have created a parser in order to include in our semantic the command Reflect. This command takes as input a string that contains a command list. Through the parser the string is evaluated in order to recreate the command list. If the reflect try to modify existing values in the store, our dynamic control

look into the taint of these values. If there are some tainted values an exception will be launched and the program will be terminated.

Files

- **/src/**: source code of interpreter
  - **/mod/**: modules of project
    - **environment.ml**: environment of interpreter to link an identifier to a location in storage
    - **storage.ml**: storage of interpreter to store a value as an element of a list
  - **domain.ml**: domains of types, type's conversion and exceptions
  - **operations.ml**: operations of semantic's interpreter and typechecking
  - **parser.ml**: parser for reflect
  - **semantic.ml**: semantic of interpreter
  - **syntax.ml**: syntax of expressions, commands, etc.
- **/test/**: folder about tests
  - **library.ml**: library of tests to be executed in **test.ml**
  - **test.ml**: execution of tests' implementation from **library.ml**

Environment

Exception of wrong bind list of `bindlist`

```
exception WrongBindList
```

New polyformic type about environment, that takes as input a string, the identificator and return the correspondant dval, and return the type of that because an environment can be of different types.

```
type 't env =
    string -> 't
```

Definition of function that creates new empty environment taking as input a type and returning a new environment of that type.

```
let emptyenv (x) =
    function y -> x
```

Function created to apply a new environment.

```
let applyenv (x,y) =
    x y
```

Bind returns a new function, which is the new environment with the association of the new variable.

```
let bind (r,l,e) =
    function lu ->
        if lu = l
            then e
        else applyenv(r,lu)
```

As for bind, `bindlist` takes three parameters and matches `il` and `el` with two empty list returning the new reference; if `il` and `el` aren't two empty list, then bind them with relative function with the new reference `r`; else raise `WrongBindList` because `il` and `el` aren't lists.

```
let rec bindlist (r,il,el) =
    match (il,el) with
        | ([],[]) -> r
        | i::il1,e::el1 -> bindlist(bind(r,i,e),il1,el1)
        | _ -> raise WrongBindList
```

Storage

Function created to apply a new storage.

```
type loc = int
```

New polyformic type that takes as input a loc and returns the correspondent mval.

```
type 't store =
    loc -> 't
```

In this snippet of code there are two definitions of function: `newloc` and `initloc`. Then I assign to name `count` the int reference of `-1` used in the following two functions: the first assigns to `count` its value added by `+1` and then, with the operator `!`, it dereferences its to get out the contents; while the

second function assigns to `count` the value of `-1`.

```
let (newloc,initloc) =
    let count =
        ref(-1) in
            (fun () -> count := !count +1; !count),
            (fun () -> count := -1)
```

Definition of function that creates new empty storage taking as input a new reference, executing `initloc()` and returning a new reference as output.

```
let emptystore (x) =
    initloc ();
    function y -> x
```

Function created to apply a new storage.

```
let applystore (x,y) =
    x y
```

In this function you allocate a new value through `newloc()` function and assign its location to name `l`. Then if `l` already exists you return `e`, that is initially the new value to allocate casted to a polyformic type, else if it not exists run `applystore(r,lu)` function with the storage reference and the new loc created as parameters.

```
let allocate ((r:'a store),(e:'a)) =
    let l =
        newloc () in
            (l,function lu ->
                if lu = l
                    then e
                else applystore(r, lu))
```

In this function if `l` casted as `loc`, the location passed as parameter, already exists then return `e`, that is the value to store; else you run `applystore(r,lu)`

```
let update ((r:'a store),(l:loc),(e:'a)) =
    function lu ->
        if lu = l
            then e
        else applystore(r,lu)
```

Parser

```
parseComList
parseList
parseCom
parseTerminal
parseParam
parseExp
    parseTerminal
    parseParam
parseInt
parseBool
```

```
<function>
    <ricursion>
        <terminal, substring>
            match <terminal>
                "terminal"
                    <expression, substring>
                        Example( , ), remain
        <application> <params>
```

A string will be converted in an expression, command, expressions' list or commands' list through a recursive analysis, character by character, in order to identify the first terminal and its relative parameters. During each search operation, the string is divided in two parts. The first part to which it is associated the semantic meaning and the remaining substring.

parseExp

Here each terminal including its parameters is reconstructed. `parseTerminal` returns the terminal (Int, String, Char, Bool, etc.). `parseParam` returns all the characters between (). With a pattern matching for each terminal, the semantic meaning is associated in order to create each expressions. In `paramRemain` we have the remaining substring not yet analyzed.

```
let rec parseExp s =
    let terminal, terminalRemain = parseTerminal s in
        let params, paramRemain = parseParam terminalRemain in
            match terminal with

                | "Int" ->
                    let ide, restide = parseInt params in
                        let b, restbool = parseBool restide in
                            Int(int_of_string(ide), b), paramRemain

                ...
```

```
                | _ -> failwith ("unhandled exp: " ^ terminal)
```

parseTerminal

Returns the terminal from expression and command.

e.g.
INPUT: XXX(YYY(CCC(1), ZZZ),3
OUTPUT:
XXX
(YYY(CCC(1), ZZZ),3

```
let parseTerminal s =
    let rec extractTerminal stringToExtract terminalName =
        let trimmedStringToExtract = String.trim(stringToExtract) in
        (* if the length of str is 0 -> finish *)
        if String.length(trimmedStringToExtract) = 0
        then
            (* returns the terminal and the remaining string *)
            terminalName, trimmedStringToExtract
        else
            let charToElaborate = String.sub trimmedStringToExtract 0 1 in (*takes the
            let l = String.length(trimmedStringToExtract) in
            (* remaining string without the first char *)
            let stringLeftToElaborate = String.sub trimmedStringToExtract 1 (l-1)  in
            match charToElaborate with
            | " " | "\n" | "," -> extractTerminal stringLeftToElaborate terminalName (
            | "(" | "[" | ";" ->    if String.length terminalName = 0
                            then extractTerminal stringLeftToElaborate terminalName (*
                            else terminalName, trimmedStringToExtract (* finish *)
            (* continue recall recoursive extractTerminal and we save the char *)
            | _ -> extractTerminal stringLeftToElaborate  (terminalName ^ charToElabor
    in extractTerminal s "";;
```

parseParam

Returns the params included in the string.

e.g.
INPUT: (YYY(CCC(1), ZZZ),3
OUTPUT:
YYY(CCC(1), ZZZ ,3

```
let parseParam stringToParse =
    (* t = str remain, e = str extracted, o = open '('*)
```

```
    let rec extractParams t e o =
        if String.length(t) = 0
        then
            e, t (*finish*)
        else
            let c = String.sub t 0 1 in
            let l = String.length(t) in
            let r = String.sub t 1 (l-1)  in
            match c with
            (*if c=')' and o=1 means that I have found the all parameter*)
            | ")" -> if o = 1
                then e, r
                (*call recoursively extractParams ^ c*)
                else extractParams r (e ^ c) (o -1)
            | "(" -> extractParams r (e ^  c) (o +1)
            (*if no '(' opened than return r*)
            | _ -> if o = 0
                then extractParams r e o
(*if o =! 0 keep calling extractParams and save c*)
                else extractParams r ( e ^ c)  o
    in extractParams stringToParse "" 0;;
```

parseIde

Reads the characters inclueded in "" and returns the correspondant ide.

e.g. INPUT: "CIAO", XXX(YYY(3))
OUTPUT:
CIAO
,XXX(YYY(3)

parseInt

Reads the int.

e.g.
INPUT: (3), CCC(ZZZ(3))
OUTPUT:
3
, CCC(ZZZ(3))

parseChar

Reads the char.

e.g.
INPUT: 'c', CCC(ZZZ(3))

OUTPUT:
c
, CCC(ZZZ(3))

```
let parseChar stringToParse =
    let fullLenght = String.length stringToParse in
    let findSingleQuote = String.index stringToParse '\'' in
    let remain = String.sub stringToParse (findSingleQuote+3) (fullLenght-findSingleQu
    (*returns the char in the array string at the index of the first ' found *)
    stringToParse.[findSingleQuote+1], remain
```

parseList

Takes a string and returns a substring created with the first list found and all the remaining substring.

```
let parseList stringToParse =
    let rec extractList t e o =
        if String.length t = 0
        then
            e, t
        else
            let c = String.sub t 0 1 in
            let l = String.length(t) in
            let r = String.sub t 1 (l-1)  in
            match c with
            | "]" -> if o = 1
                then e, r
                else extractList r (e ^ c) (o -1)
            | "[" ->
                if o = 0
                then extractList r e (o +1)
                else extractList r (e ^  c) (o +1)
            | ";" -> extractList r e o
            | _ -> if o = 0
             (*if there's no more [ returns r ed e*)
                then extractList r e o
                else extractList r ( e ^ c)  o
    in extractList stringToParse "" 0;;
```

parseIdeList

In the function we need to be able to identify a list of ide. This function takes a string and create that list of ide.

```
let parseIdeList stringToParse =
    let idelist, remain = parseList stringToParse in
```

```
    let rec parseIdes (toExtract : string) (extracted : string list) =
        let parsed, remain = parseIde toExtract  in
        if parsed = ""
        then extracted
        else parseIdes remain (parsed :: extracted)
    in parseIdes idelist [], remain
```

parseBool

Identify the taint in the primitive value (Int, String, Char, Bool, etc).

```
let rec parseBool stringToParse =
        let trimmedString = String.trim(stringToParse) in
        if (String.length trimmedString) < 4
            then  failwith ("cannot parse bool from: " ^ stringToParse)
        else if trimmedString.[0] = ','
            then parseBool (String.sub trimmedString 1 (String.length(trimmedString)-1
        else if trimmedString.[0] = '\n'
            then parseBool (String.sub trimmedString 1 (String.length(trimmedString)-1
        else
            let tt = String.sub trimmedString 0 4 in
                if (tt="true") then let rr = String.sub trimmedString 4 (String.length
            else
                let ff = String.sub trimmedString 0 5 in
                    if (ff="false") then let rr = String.sub trimmedString 4 (String.l
            else failwith ("cannot find true or false")
```

splitListElement

From a string that should represent a list, extracts the elements. Needed for the Block.

```
let splitListElement stringToParse =
    let rec extractParamList t e o rlist =

            if String.length t = 0
(* finish but I need to check if I need to return the list with the element appended o
             then if String.length e = 0 then rlist else e :: rlist
             else
                let c = String.sub t 0 1 in
                let l = String.length(t) in
                let r = String.sub t 1 (l-1)  in
            match c with
            | ";" ->
            if o = 0
(* if I have found ; and the o = 0 means that I'm inside the list between to parameter
If the '()' are opened, means that I'm inside one parameter inside one list of lists.
case 1:[(.....); (.....)], .......
```

```
case 2:[([..;..;..;]), .......... *)
                then if String.length e = 0
(*if there is no story to save, skip the char*)
                    then extractParamList r "" o (rlist) (* skip *)
                    else extractParamList r "" o (e :: rlist)
                else extractParamList r (e^c) o rlist (* element not complete, save th
            | "(" -> extractParamList r (e^c) (o+1) rlist
            | ")" -> extractParamList r (e^c) (o-1) rlist
            | _ -> extractParamList r (e^c) o rlist
in extractParamList stringToParse "" 0 [];;
```

parseExpList

Returns a list of string and the remaining string to be evaluated.

```
and parseExpList stringToParse =
    let explist, remainExplist = parseList stringToParse in
    let rec parseExps (toExtract : string) (extracted : exp list) =
        if String.trim toExtract = ""
        then
            extracted
        else
            let parsed, remain = parseExp toExtract  in
                parseExps remain ( parsed :: extracted)
    in parseExps explist [], remainExplist
```

parseStringIdexExp

From a string extract ide*exp. Needed in the function below parseIdexExpList.

```
let parseStringIdexExp stringToParse =
    let ide, remain0 = parseIde stringToParse in
     let exp0, remain1 = parseExp remain0 in
        ide, exp0;;
```

parseIdexExpList

From a list of string extracts (ide*exp) list.

```
let parseIdexExpList (listideexp:string list)  =
    List.rev_map (fun x -> parseStringIdexExp x) listideexp;;
```

parseCom

As parseExp above, the string that contains a command or a list of commands is evaluated and the commands reconstructed.

```
let rec parseCom s =
    let terminal, terminalRemain = parseTerminal s in
        let params, paramRemain = parseParam terminalRemain in
            match terminal with

                | "Assign" ->
                    let exp0, restexp0 = parseExp params in
                        let exp1, restexp1 = parseExp restexp0 in
                            Assign(exp0, exp1), paramRemain

                ...

                | _ -> failwith ("Error in parsing: " ^ s)
```

parseComList

Returns a list of commands and the remaining string to be evaluated.

```
and parseComList stringToParse =
    let comlist,remain = parseList stringToParse in
    let rec parseCommands (toExtract : string) (extracted : com list) =
        if String.trim toExtract = ""
        then
            extracted
        else
            let parsed, remain = parseCom toExtract  in
                parseCommands remain (parsed :: extracted)
    in parseCommands comlist [], remain
```

Reflect

Reflect is a command that takes as input a string that contains a command list. Through the parser, the string is evaluated in order to recreate the command list.

`parser.ml`:

```
let rec parseCom s =
    let terminal, terminalRemain = parseTerminal s in
        let params, paramRemain = parseParam terminalRemain in
            match terminal with
```

`semantic.ml`:

```
| Reflect(x) ->
    let comList, str = parseComList x in
        semcl(comList,r,s,t)
```

If the reflect try to modify existing values in the store, as written before, our dynamic control look into the taint of these values. The original fuction called update in the store has been modified in order to prevent possible code injection. A new function preupdate has been created. If the value is already in the store and the taint of it is true an exception will be launched.

```
and preupdate ((r: 'a store),(l: loc),(e: mval)): ('a store) =
    if (mvaltobool e = false)
        then update (r,l,e)
    else if (mvaltobool (applystore (r,l)) = false)
        then update (r,l,e)
    else raise Untrusted
```

e.g. of reflect tainted:

```
let r = emptyenv(Unbound);;
let s = emptystore(Undefined);;
let t = emptyenv(Untyped);;

let d = [("x",Newloc(Int(4,false)));("y",Newloc(Int (1,false)))] ;;

let r1, s1 = semdv(d,r,s,t);;

let str = "[Block([],[],
[While(Not(Equ(Val(Den(\"x\")),Int(0,false))),[Assign(Den(\"y\"),Prod(Val(Den(\"y\")),
let s1 = semc(Reflect(str),r1,s1,t);;
```

```
val str : string =
"[Block([],[],[While(Not(Equ(Val(Den(\"x\")),Int(0,false))),[Assign(Den(\"y\"),Prod(Va
let s1 = semc(Reflect(str),r1,s1,t)

Exception: Domain.Untrusted.
```

Taint Analysis

An expression, command or a declaration between a tainted and an untainted input, returns an untainted output if the result is indipendent by the tainted input. Otherwise, if output depends by a tainted input, it returns a tainted output.

Each eval, dval, mval and tval type of interpreter is a tuple: the first element is the value, instead the second is a boolean. If the last is true, it is tainted, otherwise it is untainted.

```
type eval ==
    | Eint of int * bool
    ...
```

Then we have created a mirror of `enviroment.ml` for enviroment of types, to isolate taint by values in method to extract them. Infact, to get taint you could use `Ref()`, to get location in storage and converting it from mval to tval, and then `Type()` extracting taint as an eval through a type conversion. Lastly, to get taint as a tag, you could use `evaltotag` conversion out of semantic.

Strings

Len()

It get as input a stringand returns its length as an integer. If it's untainted, remains untainted, otherwise tainted.

```
let len (x) =
    if (typecheck("string",x))
        then
            match x with
                | Estring(x,t)     -> Eint(String.length(x),t)
                | _                -> failwith("Error: match")
    else failwith ("Error: type")
```

Upper()

It get as input a string and returns its updated with all characters in uppercase. If it's untainted, remains untainted, otherwise tainted.

```
let upper (x) =
    if (typecheck("string",x))
        then
            match x with
                | Estring(y,t)     -> Estring(String.uppercase(y),t)
                | _                -> failwith("Error: match")
    else failwith ("Error: type")
```

Lower()

It get as input a string `s` and returns its updated with all characters in lowercase. If it's untainted,

remains untainted, otherwise tainted.

```
let lower (x) =
    if (typecheck("string",x))
        then
            match x with
                | Estring(y,t)     -> Estring(String.lowercase(y),t)
                | _                -> failwith("Error: match")
    else failwith ("Error: type")
```

Get()

It get as input a string `x` and an int `y` to return a character `c` of `x` at index `y`. If string `x` is untainted, the returning char `c` will be untainted, otherwise tainted.

```
let get (x,y) =
    if (typecheck("string",x) && typecheck("int",y))
        then
            match (x,y) with
                | Estring(x,a),Eint(y,b)     -> Echar(String.get x y, a)
                | _                            -> failwith("Error: match")
    else failwith ("Error: type")
```

Set()

It get as input a string `x`, an int `y` and a char `z` to return the same string `x` updated replacing its char at position `y` with the new char `z`. If string `x` after `Set()` will be the same of that before, then its taint remains the same. Otherwise, if `x` is changed, then will be made a new string through substring operation: you merge two string, the first and the last part, with in the middle the new char `z`. In this last case, will be done an `OR` between taints of string `x` and char `z` to be setted.

```
let set (x,y,z) =
    if (typecheck("string",x) && typecheck("int",y) && typecheck("char",z))
        then
            match (x,y,z) with
                | Estring(x,a),Eint(y,b),Echar(z,c) ->
                    let c1 = String.get x y in
                        if (c1 == z)
                            then Estring (x,a)
                        else
                            let start = String.sub x 0 y in
                                let finish = String.sub x (y+1) (String.length x-(y+1)
                                    Estring(start ^ (String.make 1 z) ^ finish, a || c
                | _    -> failwith("Error: type error")
    else failwith ("Error: type error")
```

Contains()

It get as input a char `x` and a string `y` to return a boolean: `true` if `x` is in `y`, otherwise `false`. If string `y` or char `x` is tainted, then the returning bool value will be tainted, otherwise untainted.

```
let contains (x,y) =
    if (typecheck("char",x) && typecheck("string",y))
        then
            match (x,y) with
                | Echar(x,a),Estring(y,b)     -> Ebool(String.contains y x, a || b)
                | _                            -> failwith("Error: match")
    else failwith ("Error: type")
```

Sub()

It get as input a string `x` and two int, `y` and `z`, to return a new substring of `x` that starts at position `y` and ends in `z`. If the substring of string `x` after `Sub()` is empty "", then it's untainted, otherwise will be return the same taint of `x`.

```
let sub (x,y,z) =
    if (typecheck("string",x) && typecheck("int",y) && typecheck("int",z))
        then
            match (x,y,z) with
                | Estring(x,a),Eint(y,b),Eint(z,c) ->
                    let w = (z - y) in
                        if w == 0
                            then Estring("",false)
                        else
                            Estring(String.sub x y w, a)
                | _    -> failwith("Error: match")
    else failwith ("Error: type")
```

Concat()

It get as input two string, `s1` and `s2`, to return a new string `s3` composed by `s1` concatenate with `s2`. If both string are empty "", then will be return an empty string "" untainted, independently of taints of two string `s1` and `s2`; if one of two string is empty, you will get the other string with its taint; otherwise will be done an OR between taints of `s1` and `s2`.

```
let concat (x,y) =
    if (typecheck("string",x) && typecheck("string",y))
        then
            match (x,y) with
                | (Estring(x,a),Estring(y,b)) ->
                    if ((x = "") && (y = ""))
```

```
                        then Estring("",false)
                    else if (x = "")
                        then Estring(y,b)
                    else if (y = "")
                        then Estring(x,a)
                    else
                        Estring(x ^ y, a || b)
                | _    -> failwith("Error: match")
    else failwith ("Error: type")
```

Operations

Minus()

It get as input an integer and returns it negative, with the same taint of before.

```
let minus (x) =
    if (typecheck("int",x))
        then
            match x with
                | Eint(x,t)     -> Eint(-x,t)
                | _                -> failwith("Error: match")
    else failwith ("Error: type")
```

Iszero()

It get as input an integer and returns a boolean: true if it is equal to 0, otherwise false with the same taint of before.

```
let iszero (x) =
    if (typecheck("int",x))
        then
            match x with
                | Eint(y,t)     -> Ebool(y = 0, t)
                | _                -> failwith("Error: match")
    else failwith ("Error: type")
```

Equ()

It get as inputs two integer and returns a boolean: true if the first is equal to second, otherwise false. If one of two is tainted, it returns tainted, otherwise untainted.

```
let equ (x,y) =
    if (typecheck("int",x) && typecheck("int",y))
        then
```

```
            match (x,y) with
                | (Eint(x,a),Eint(y,b))     -> Ebool(x = y, a || b)
                | _                            -> failwith("Error: match")
    else failwith ("Error: type")
```

Sum()

It get as inputs two integer and returns the sum between them as result. If one of two integer is equal to 0, the interpreter returns the other addend with its taint as result. Otherwise it returns the sum between them, with OR operation between their taints.

```
let sum (x,y) =
    if (typecheck("int",x) && typecheck("int",y))
        then
            match (x,y) with
                | (Eint(x,a),Eint(y,b)) ->
                    if (x == 0 && y != 0)
                        then Eint(y,b)
                    else if (x != 0 && y == 0)
                        then Eint(x,a)
                    else
                        Eint(x + y, a || b)
                | _    -> failwith("Error: match")
    else failwith ("Error: type")
```

Diff()

It get as inputs two integer and returns the difference between them as result. If the second integer is 0, result is indipendent by its taint so interpreter return the first integer with its taint; otherwise it returns difference between them with OR operation between their taints.

```
let diff (x,y) =
    if (typecheck("int",x) && typecheck("int",y))
        then
            match (x,y) with
                | (Eint(x,a),Eint(y,b)) ->
                    if (x != 0 && y == 0)
                        then Eint(x,a)
                    else
                        Eint(x - y, a || b)
                | _ -> failwith("Error: match")
    else failwith ("Error: type")
```

Prod()

It get as inputs two integer and returns the product between them as result. If one of two is 0, interpreter

returns a 0 untainted because the result is indipendent by inputs. Instead, if one of two is a 1, the result will be the same of the other input with its taint because 1 doesn't change inputs. Otherwise, it returns their product as result and OR operation between their taints.

```
let prod (x,y) =
    if (typecheck("int",x) && typecheck("int",y))
        then
            match (x,y) with
                | (Eint(x,a), Eint(y,b)) ->
                    if (x == 0 || y == 0)
                        then Eint(0,false)
                    else if (x == 1 && y != 1)
                        then Eint(y,b)
                    else if (x != 1 && y == 1)
                        then Eint(x,a)
                    else
                        Eint(x * y, a || b)
                | _    -> failwith("Error: match")
    else failwith ("Error: type")
```

Div()

It get as inputs two integer and returns the division between them as result. If division will be made between an integer (!=0) and a 0, interpreter raises an exception: `Exception: DivisionByZero`. Instead, if 0 will be divided by an integer, it returns always a 0 untainted, because the result is indipendent by inputs. Another case could be that when dividend is a 1, the interpreter returns the same divider (!=1) with its taint. Otherwise, this operation return division between the two values and the OR operation between their taints; so if one of two is tainted, returns tainted, otherwise untainted.

```
let div (x,y) =
    if (typecheck("int",x) && typecheck("int",y))
        then
            match (x,y) with
                | (Eint(x,a), Eint(y,b)) ->
                    if (x != 0 && y == 0)
                        then raise DivisionByZero
                    else if (x == 0 && y != 0)
                        then Eint(0,false)
                     else if (x != 1 && y == 1)
                        then Eint(x,a)
                    else
                        Eint(x / y, a || b)
                | _    -> failwith("Error: match")
    else failwith ("Error: type")
```

Logic

Not()

It get as input a boolean and returns its negated. If it's untainted, remains untainted, otherwise tainted.

```
let non (x) =
    if (typecheck("bool",x))
        then
            match x with
                | Ebool(y,t)     -> Ebool(not y, t)
                | _                -> failwith("Error: match")
    else failwith ("Error: type")
```

Or()

It get as inputs two booleans and returns `||` operation between them as result. If one of two is tainted, returns tainted, otherwise untainted.

```
let vel (x,y) =
    if (typecheck("bool",x) && typecheck("bool",y))
        then
            match (x,y) with
                | (Ebool(x,a), Ebool(y,b))     -> Ebool(x || y, a || b)
                | _                            -> failwith("Error: match")
    else failwith ("Error: type")
```

And()

It get as inputs two booleans and returns `&&` operation between them as result. If both are tainted, returns tainted, otherwise untainted.

```
let et (x,y) =
    if (typecheck("bool",x) && typecheck("bool",y))
        then
            match (x,y) with
                | (Ebool(x,a),Ebool(y,b))     -> Ebool(x && y, a || b)
                | _                            -> failwith("Error: match")
    else failwith ("Error: type")
```

Credits

Andrea Bazerla - VR377556
Valentina Mantelli - VR072986
Lorenzo Bellani - VR360742

License