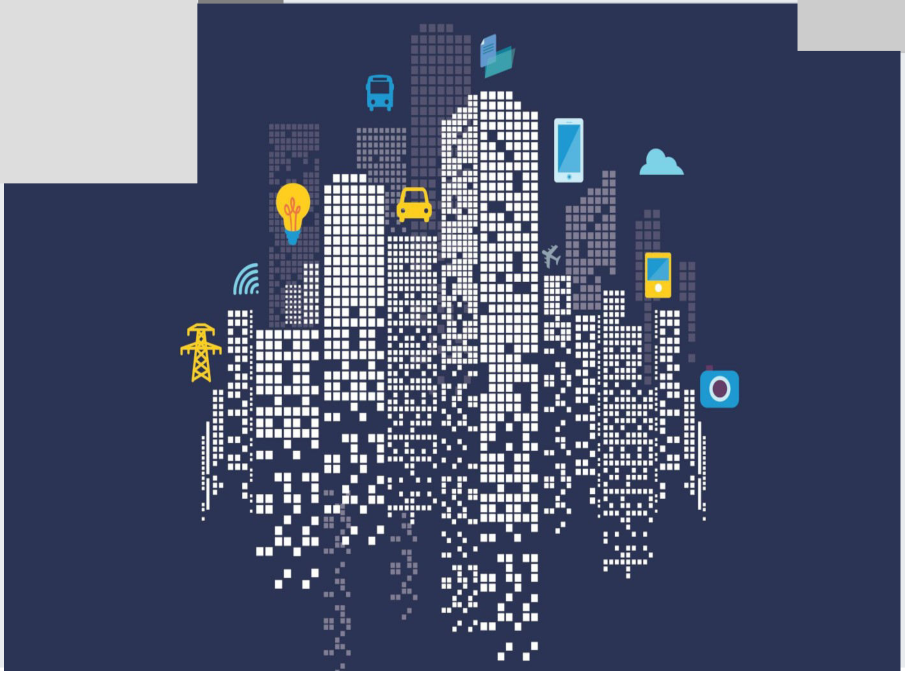


comarketing-news.fr

Les méthodes formelles dans la ville connectée

Youssef MILED

22888

MPI

Motivation

- Objets connectés
- Cyberattaques
- Language-based security
- Information flow

www.usinenouvelle.com

En quoi l'approche de Language-based security nous permet de résoudre les problèmes de confidentialité et d'intégrité dans les programmes informatiques ?

1. Modélisation du problème
2. Définition d'un système de type adapté
3. Implémentation en OCaml
4. Test du modèle

Modélisation du problème

Attaquant / pirate

Agent extérieur ayant accès aux résultats du programme

Non-interférence

Entrées différentes => aucune information sur les variables protégées

Confidentialité + Intégrité

Données privées /

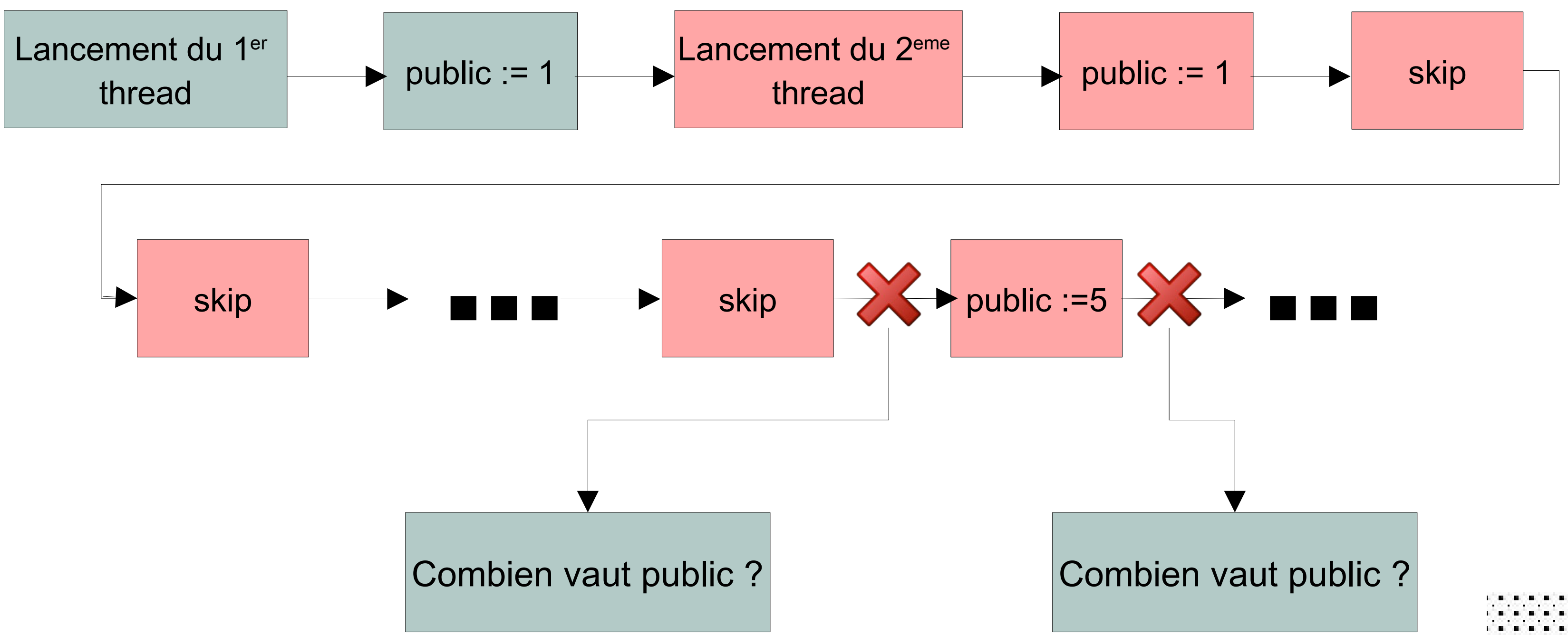
données publiques

www.pngegg.com

$public := 42$
while $private \geq 0$ **do**
    $skip$

;
$public := 51$

Problème



Grammaire du langage :

- Expression $e ::= n \mid x \mid e1 \odot e2 \mid$
- Commandes $c ::= x := e \mid skip \mid$
  $if\ e\ then\ c1\ else\ c2 \mid$
  $while\ e\ do\ c \mid$
  $c1; c2$
- Phrases $p ::= e \mid c$

Exemple de phrase : a := 5; b := 2; while a do a := a - b

Types définis

- Types des données $\tau ::= L \mid H$
- Types des phrases

$$\rho ::= \tau$$
$$\mid \tau\ var$$
$$\mid \tau\ cmd\ n$$

Affecte à des variables de type $\geq \tau$ et se termine en n étapes

$$\mid \tau_1\ cmd\ \tau_2$$

Affecte à des variables de type $\geq \tau_1$ et dont la durée d'exécution dépend de variables de type $\leq \tau_2$

Algorithm 1 Exemple τ *cmd n*

if $x < 0$ **then**
$x := x + x;$
$x := x + x$
else
$x := x \times x;$
$x := x \times x$

Algorithm 2 Exemple τ_1 *cmd* τ_2

$a := 42;$
while a **do**
$a := a - 1$

Règles de typage : [6]

Règles d'inférence :

(R-VAL) $$\frac{\gamma(x) = \tau \ var}{\gamma \vdash x : \tau}$$

(INT) $$\gamma \vdash n : L$$

(SUM) $$\frac{\gamma \vdash e_1 : \tau, \quad \gamma \vdash e_2 : \tau}{\gamma \vdash e_1 + e_2 : \tau}$$

(ASSIGN) $$\frac{\gamma(x) = \tau \ var, \quad \gamma \vdash e : \tau}{\gamma \vdash x := e : \tau \ cmd \ 1}$$

(SKIP) $$\gamma \vdash \textbf{skip} : H \ cmd \ 1$$

(IF) $$\frac{\begin{array}{l}\gamma \vdash e : \tau \\ \gamma \vdash c_1 : \tau \ cmd \ n \\ \gamma \vdash c_2 : \tau \ cmd \ n\end{array}}{\gamma \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 : \tau \ cmd \ n+1}$$

$$\frac{\begin{array}{l}\gamma \vdash e : L \\ \gamma \vdash c_1 : \tau_1 \ cmd \ \tau_2 \\ \gamma \vdash c_2 : \tau_1 \ cmd \ \tau_2\end{array}}{\gamma \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 : \tau_1 \ cmd \ \tau_2}$$

$$\frac{\begin{array}{l}\gamma \vdash e : H \\ \gamma \vdash c_1 : H \ cmd \ H \\ \gamma \vdash c_2 : H \ cmd \ H\end{array}}{\gamma \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 : H \ cmd \ H}$$

$$\frac{\gamma \vdash A \quad \gamma \vdash B}{\gamma \vdash A \wedge B}$$

Règles de typage : [6]

(WHILE)

$$\frac{\gamma \vdash e : L \qquad \tau_2 \subseteq \tau_1 \qquad \gamma \vdash c : \tau_1 \ cmd \ \tau_2}{\gamma \vdash \textbf{while } e \textbf{ do } c : \tau_1 \ cmd \ \tau_2}$$

$$\frac{\gamma \vdash e : H \qquad \gamma \vdash c : H \ cmd \ H}{\gamma \vdash \textbf{while } e \textbf{ do } c : H \ cmd \ H}$$

(COMPOSE)

$$\frac{\gamma \vdash c_1 : \tau_1 \ cmd \ L \qquad \gamma \vdash c_2 : \tau_1 \ cmd \ \tau_2}{\gamma \vdash c_1 ; c_2 \ : \ \tau_1 \ cmd \ \tau_2}$$

$$\frac{\gamma \vdash c_1 : \tau \ cmd \ H \qquad \gamma \vdash c_2 : H \ cmd \ H}{\gamma \vdash c_1 ; c_2 \ : \ \tau \ cmd \ H}$$

$public := 42$
while $private \geq 0$ **do**
  $skip$

;
$public := 51$

⚠ Illégale

Règles de typage : [6]

Relations :

(BASE) $L \subseteq H$

(CMD) $$\frac{\tau_1' \subseteq \tau_1, \quad \tau_2 \subseteq \tau_2'}{\tau_1 \ cmd \ \tau_2 \subseteq \tau_1' \ cmd \ \tau_2'}$$

$$\frac{\tau' \subseteq \tau}{\tau \ cmd \ n \subseteq \tau' \ cmd \ n}$$

$$\tau \ cmd \ n \subseteq \tau \ cmd \ L$$

(REFLEX) $\rho \subseteq \rho$

(TRANS) $$\frac{\rho_1 \subseteq \rho_2, \quad \rho_2 \subseteq \rho_3}{\rho_1 \subseteq \rho_3}$$

(SUBSUMP) $$\frac{\gamma \vdash p : \rho_1, \quad \rho_1 \subseteq \rho_2}{\gamma \vdash p : \rho_2}$$

| τ_1 cmd τ_2

Affecte à des variables de type ≥ τ_1 et dont la durée d'exécution dépend de variables de type ≤ τ_2

| τ cmd n

Affecte à des variables de type ≥ τ et termine en n étapes

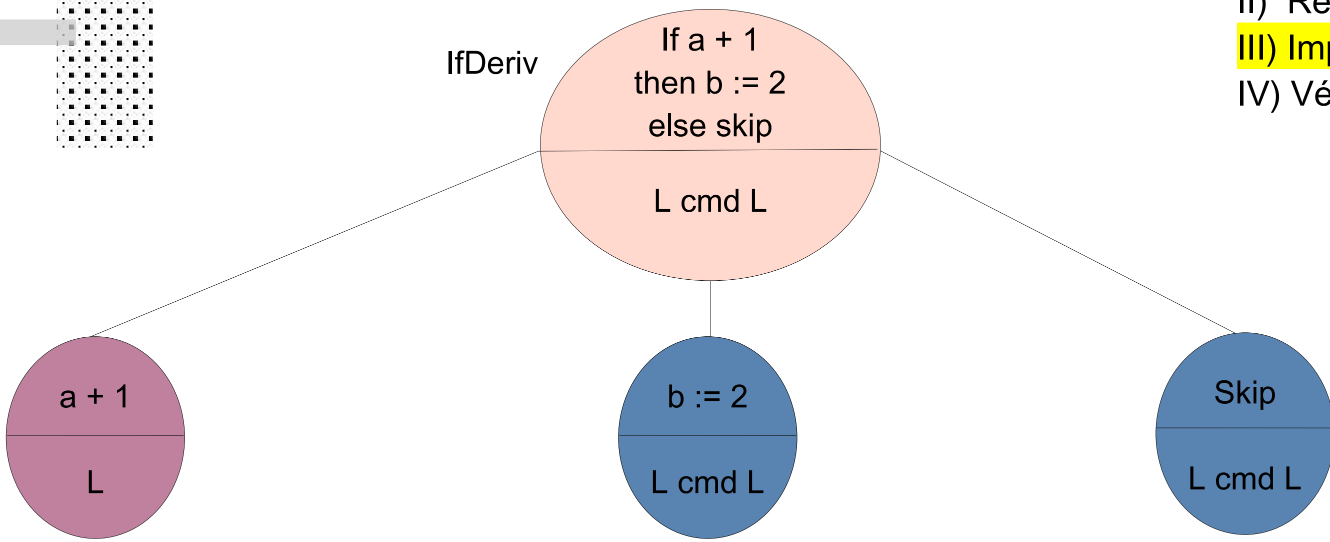
Implémentation en OCaml

On définit :

- Types des données
- Phrases avec expressions et commandes
- Types des phrases
- Arbre de dérivation
- Fonction type checker

```
type tree =
  | Empty                                                   (* Red *)
  | VarDeriv of string * phrase_type                        (* Orange *)
  | ConstDeriv of int                                       (* Red *)
  | BinopDeriv of phrase * phrase_type * tree * tree        (* Purple *)
  | SkipDeriv of phrase_type                                (* Red *)
  | AssignDeriv of string * phrase * phrase_type * tree * tree  (* Light green *)
  | IfDeriv of phrase * phrase_type * tree * tree * tree    (* Pink *)
  | WhileDeriv of phrase * phrase_type * tree * tree        (* Yellow *)
  | SeqDeriv of phrase * phrase_type * tree * tree          (* Dark green *)
  (* For the subtyping rules : *)
  | SubDeriv of phrase * phrase_type * phrase_type * tree * tree  (* Dark blue *)
  | SubRules of phrase_type * phrase_type * tree * tree     (* Light blue *)
```

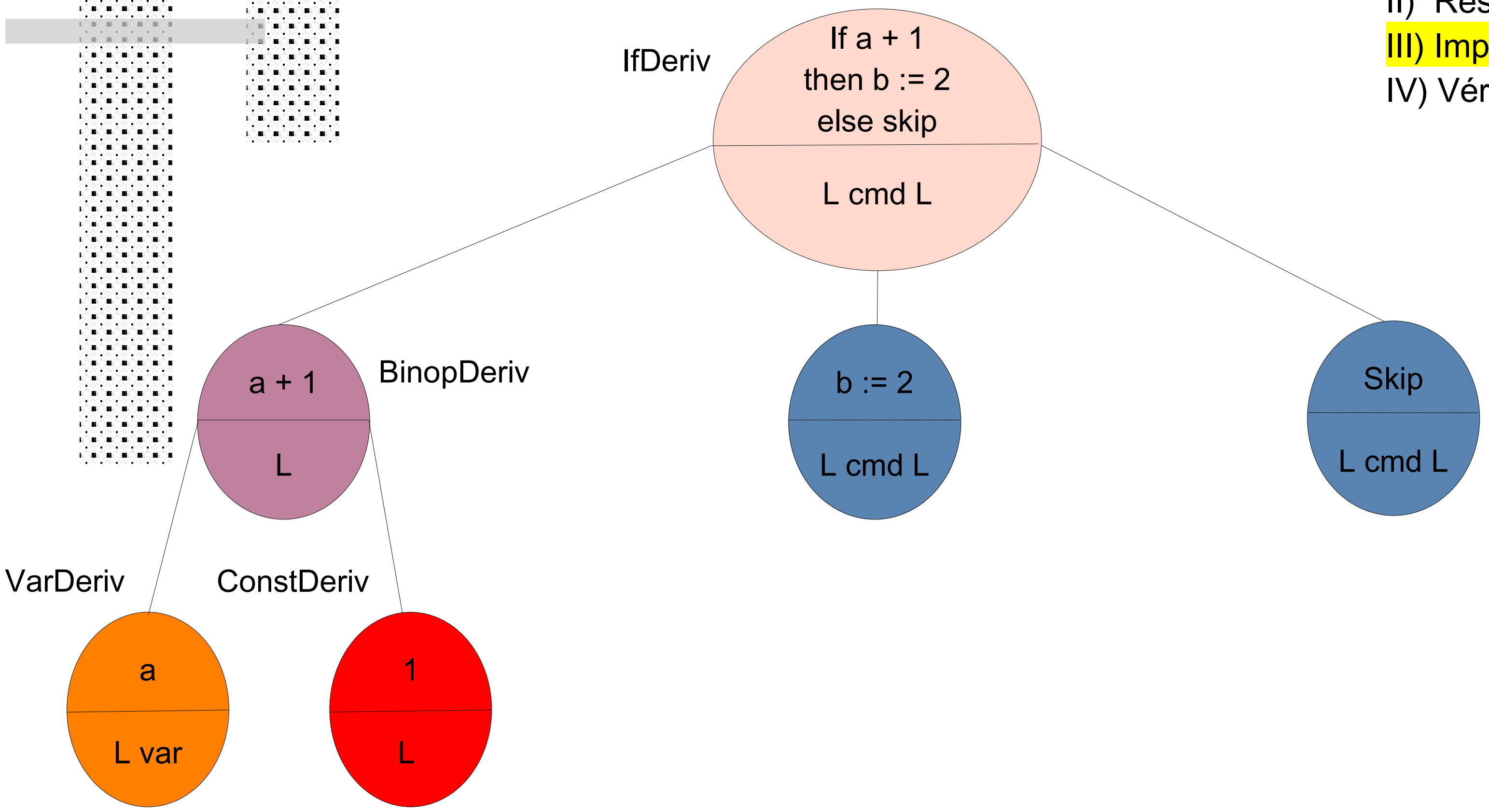
Exemple :



$$\frac{\gamma \vdash e : L \qquad \gamma \vdash c_1 : \tau_1\ cmd\ \tau_2 \qquad \gamma \vdash c_2 : \tau_1\ cmd\ \tau_2}{\gamma \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \tau_1\ cmd\ \tau_2}$$

γ = [(a, L var); (b, L var)]

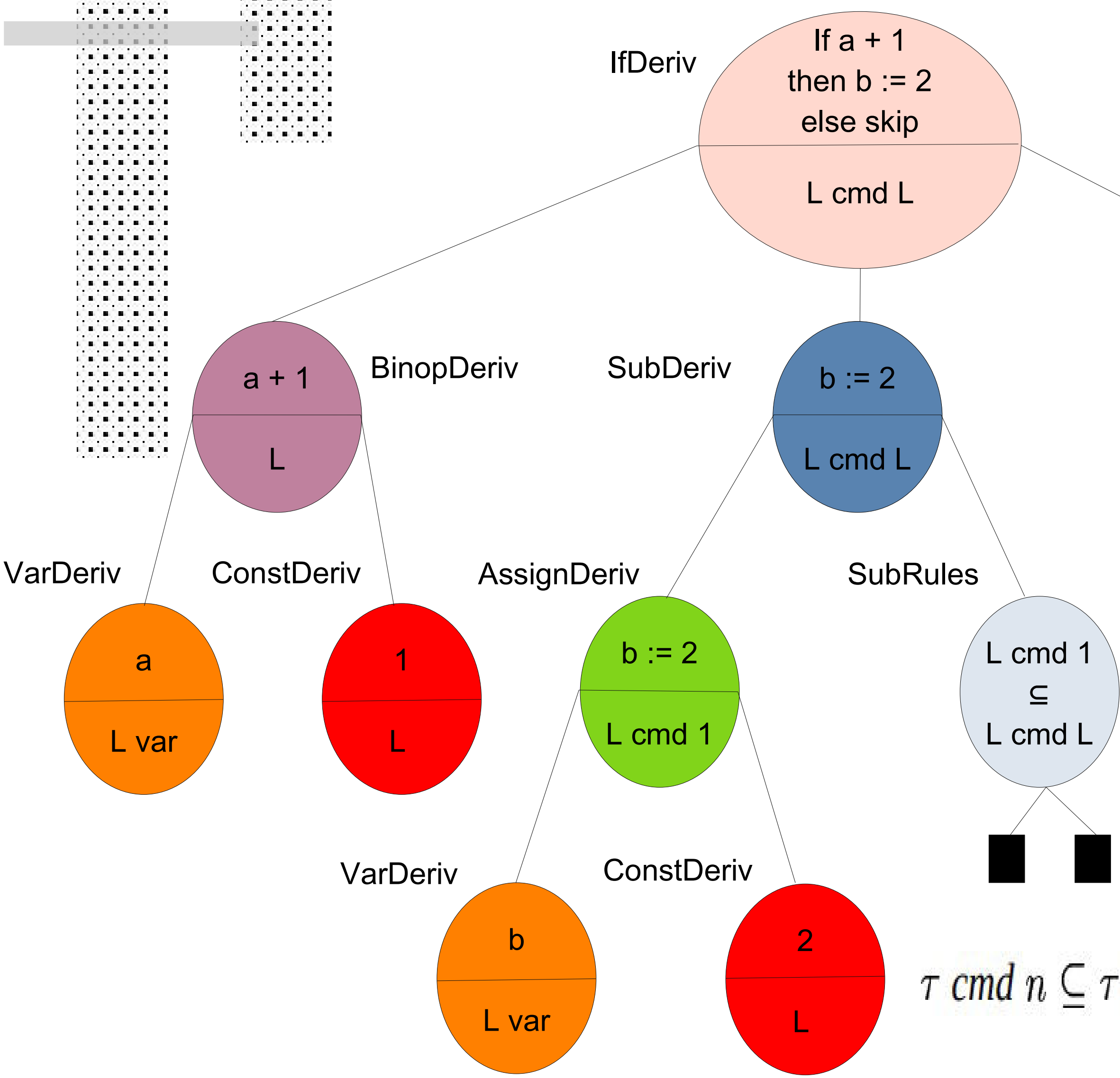
Exemple :



$$\frac{\gamma \vdash e_1 : \tau, \quad \gamma \vdash e_2 : \tau}{\gamma \vdash e_1 + e_2 : \tau}$$

γ = [(a, L var); (b, L var)]

Exemple :

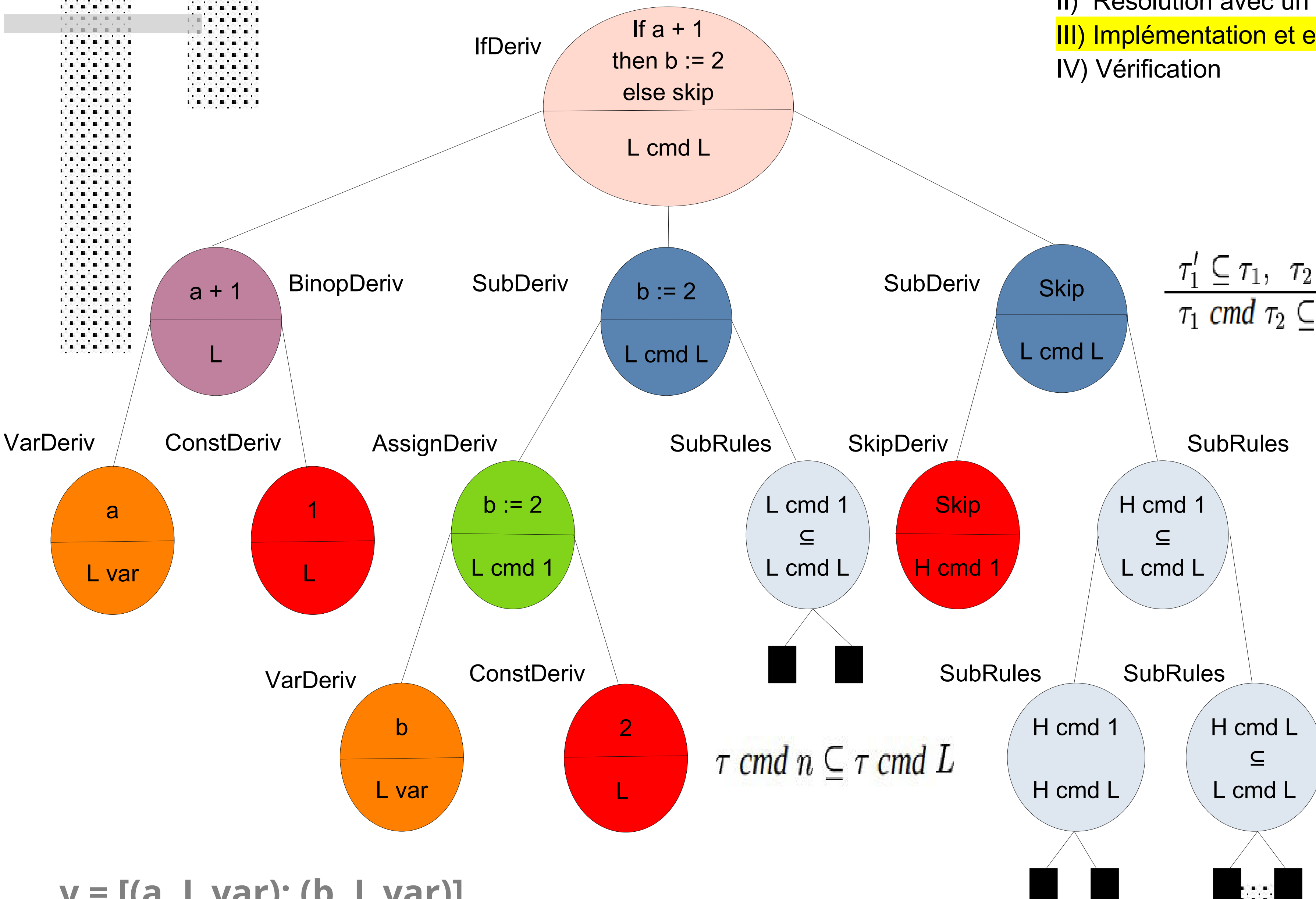


$$\frac{\gamma(x) = \tau\ var, \quad \gamma \vdash e : \tau}{\gamma \vdash x := e : \tau\ cmd\ 1}$$

$$\tau\ cmd\ n \subseteq \tau\ cmd\ L$$

γ = [(a, L var); (b, L var)]

Exemple :



$$\tau \; cmd \; n \subseteq \tau \; cmd \; L$$

$$\frac{\tau_1' \subseteq \tau_1, \quad \tau_2 \subseteq \tau_2'}{\tau_1 \; cmd \; \tau_2 \subseteq \tau_1' \; cmd \; \tau_2'}$$

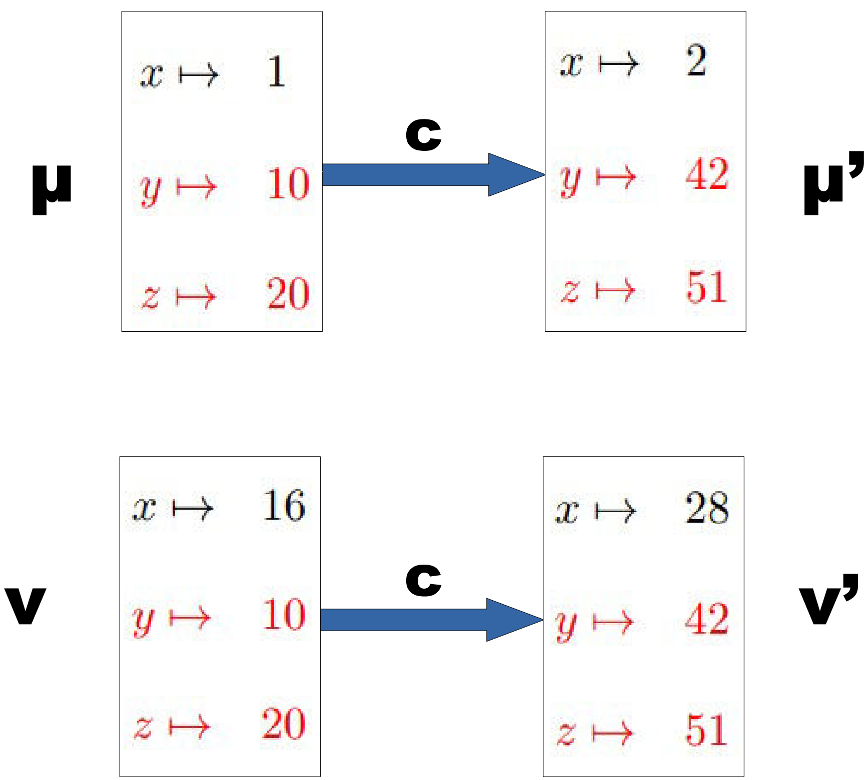
γ = [(a, L var); (b, L var)]

```
utop # check_type ex gamma;;
- : bool = true
```

Résultat :

Non Interférence :

$$\forall\, \gamma \in F(I, \{L\ var, H\ var\}), \forall c \in commands,\ \forall\, \mu, v \in F(I, N)$$
$$(\exists\, \rho,\ \gamma \vdash c : \rho\ \wedge\ \mu \sim_\gamma v\ \wedge\ (c, \mu) \rightarrow^* \mu') \Rightarrow (\exists\, v',\ (c, v) \rightarrow^* v'\ \wedge\ \mu' \sim_\gamma v')$$



Test du modèle

y = [(y, L var); (x, H var); (tmp, H var) ; (rest, H var)]

Algorithm Objet connecté

Input : x (private), y (public)

$tmp := x;$ — H cmd 1 ⊆ L cmd L

while $y > 0$ **do**

$\quad y := y - 1;$ — L cmd 1 ⊆ L cmd L

$\quad x := x - 1$ — H cmd 1 ⊆ L cmd L

L cmd L

;

if $x \leq 0$ **then**

$\quad rest := -x;$

$\quad tmp := tmp + rest;$

$\quad x := 2 \times tmp$

H cmd 3 ⊆ L cmd H

else

$\quad skip;$

$\quad skip;$

$\quad skip$

H cmd 3 ⊆ L cmd H

L cmd H

L cmd H

Bilan

Assurance

Modèle prouvé correct

Difficultés

Difficile à automatiser

Taille des formules très grande pour des phrases simples et courtes

Améliorations possibles

Assistant de preuve

Annexe

[1] Flemming Nielson, Hanne Riis Nielson : Formal methods, an appetizer :

[2] David Basin : Formal methods for security

[3] Xavier Leroy : Le logiciel, entre l'esprit et la matière : Leçon inaugurale prononcée au Collège de France le jeudi 15 novembre 2018

[4] Vincent Simonet : Flow Caml in a Nutshell

[5] Nevin Heintze, Jon G.Riecke : The Slam Calculus : Programming with Secrecy and Integrity

[6] Geoffrey Smith : A New Type System for Secure Information Flow

[7] Geoffrey Smith : A Type-Based Approach to Program Security

Code :

```
type data_types = H | L
type binop =
        Plus | Minus | Mult | Div | Eq | Neq | Lt | Gt | Leq | Geq
        | And | Or
type exp =    Int of int
                | Var of string
                | Binop of binop * exp * exp
type command =   Assign of string * exp
                   | Skip
                   | If of exp * command * command
                   | While of exp * command
                   | Seq of command * command
```

```
type phrase =   Exp of exp
              | Command of command

type phrase_type =   T of data_types
                   | Var of data_types
                   | Cmd of data_types * data_types
                   | Ncmd of data_types * int


type var_type = string * phrase_type

let less_or_eq t1 t2 =(t2 = H) || (t1 = L)

exception Type_error

exception Phrase_error
```

```
type tree =
    | Empty (* Red *)
    | VarDeriv of string * phrase_type (* Orange *)
    | ConstDeriv of int (* Red *)
    | BinopDeriv of phrase * phrase_type * tree * tree (* Purple *)
    | SkipDeriv of phrase_type (* Red *)
    | AssignDeriv of string * phrase * phrase_type * tree * tree (* Light green *)
    | IfDeriv of phrase * phrase_type * tree * tree * tree (* Pink *)
    | WhileDeriv of phrase * phrase_type * tree * tree (* Yellow *)
    | SeqDeriv of phrase * phrase_type * tree * tree (* Dark green *)
    (* For the subtyping rules : *)
    | SubDeriv of phrase * phrase_type * phrase_type * tree * tree (* Dark blue *)
    | SubRules of phrase_type * phrase_type * tree * tree (* Light blue *)
```

```
let extract_phrase_and_type tree =
    match tree with
    | VarDeriv(x, t) -> Exp(Var x), t

    | ConstDeriv(n) -> Exp(Int n), T L

    | BinopDeriv(p, t, _, _) -> p, t

    | SkipDeriv(t) -> Command Skip, t

    | AssignDeriv(x, Exp e, t, _, _) -> Command(Assign(x, e)), t

    | IfDeriv(p, t, _, _, _) -> p, t

    | WhileDeriv(p, t, _, _) -> p, t

    | SeqDeriv(p, t, _, _) -> p, t

    | SubDeriv(p, t1, t2, _, _) -> p, t1

    | _ -> raise Phrase_error
```

```
let rec check_sub_rules tree t1 t2 =
    match tree with
    | SubRules(rho1, rho2, Empty, Empty) -> (t1 = rho1 && t2 = rho2) &&
    (* here we are in the case where we don't introduce another rho3 *)
    (* we only use base and cmd subtyping rules *)
    begin
        match rho1, rho2 with
        | T L, T H -> true
        | Cmd(tau1, tau2), Cmd(tau1', tau2') →
            (less_or_eq tau1' tau1) && (less_or_eq tau2 tau2')
        | Ncmd(tau, n), Ncmd(tau', n') when n = n' -> less_or_eq tau' tau
        | Ncmd(tau, n), Cmd(tau', L) when tau = tau' -> true
        | _, _ when rho1 = rho2 -> true
        | _ -> false
    end
```

```
| SubRules(rho1, rho2, son1, son2) ->
    (t1 = rho1 && t2 = rho2) &&
    ( match son1, son2 with
        | SubRules(t1', t1'', _, _), SubRules(t2', t2'', _, _) ->
        (t1' = rho1) && (t1'' = t2') && (t2'' = rho2) &&
    (check_sub_rules son1 rho1 t1'') &&
    (check_sub_rules son2 t2' rho2)
        |_ -> false
    )

|_ -> failwith"problem in the structure of the tree (Subrules)"
```

```
let check_type tree gamma =
    let rec aux tree expected_phrase expected_type =
        match tree with
        | VarDeriv(x, T L) -> (expected_phrase = Exp(Var x))
                              && (expected_type = T L)
                              && (List.mem_assoc x gamma)
                              && (List.assoc x gamma = Var L)
        | VarDeriv(x, T H) -> (expected_phrase = Exp(Var x))
                              && (expected_type = T H)
                              && (List.mem_assoc x gamma)
                              && (List.assoc x gamma = Var H)
        | ConstDeriv(n) -> (expected_phrase = (Exp(Int n)))
                              && (expected_type = T L )
```

```
| BinopDeriv(p, t, tree1, tree2) ->
    begin match p with
        | Exp(Binop(_, e1, e2)) ->
            let verify_phrase = p = expected_phrase in
            let verify_type = t = expected_type in
            let type_tree1 = aux tree1 (Exp e1) t in
            let type_tree2 = aux tree2 (Exp e2) t in
            verify_phrase && verify_type && type_tree1 && type_tree2

        |_ -> failwith"problem in the structure of the tree (binop)"
    end

| SkipDeriv(t) -> t = Ncmd(H, 1)

| AssignDeriv(x, Exp exp, Ncmd(tau, 1), son_x, son_exp) ->
    (expected_phrase = Command (Assign(x, exp)))
    && (expected_type = Ncmd(tau, 1))
    && (List.mem_assoc x gamma)
    && (List.assoc x gamma = Var tau)
    && (aux son_x (Exp(Var x)) (T tau))
    && (aux son_exp (Exp exp) (T tau))
```

```
| IfDeriv(p, t, tree_exp, tree1, tree2) ->
begin
match p with                                                        |
Command(If(e, c1, c2)) →
    ( match t with
     | Cmd(H, H) ->
        let verify_phrase = p = expected_phrase in
        let verify_type = t = expected_type in
        let verify_exp = aux tree_exp (Exp e) (T H) in
        let verify_tree1 = aux tree1 (Command c1) t in
        let verify_tree2 = aux tree2 (Command c2) t in
        verify_phrase && verify_type && verify_exp
        && verify_tree1 && verify_tree2

             | Cmd(tau1, tau2) ->
                let verify_phrase = p = expected_phrase in
                let verify_type = t = expected_type in
                let verify_exp = aux tree_exp (Exp e) (T L) in
            let verify_tree1 = aux tree1 (Command c1) t in
            let verify_tree2 = aux tree2 (Command c2) t in
            verify_phrase && verify_type && verify_exp
            && verify_tree1 && verify_tree2
```

```
| Ncmd(tau, n) when n >= 1 ->
let verify_phrase = p = expected_phrase in
let verify_type = t = expected_type in
let verify_exp = aux tree_exp (Exp e) (T tau) in
let verify_tree1 =
        aux tree1 (Command c1) (Ncmd(tau, n - 1)) in
let verify_tree2 =
        aux tree2 (Command c2) (Ncmd(tau, n - 1)) in
verify_phrase && verify_type && verify_exp
&& verify_tree1 && verify_tree2
|_ -> false

)

|_ -> failwith"problem in the structure of the tree (if-else)"

end
```

```
| WhileDeriv(p, t, tree_exp, tree_cmd) ->
    begin
        match p with
        | Command (While(e, c)) ->
            ( match t with

                | Cmd(H, H) ->
                    let verify_phrase = p = expected_phrase in
                    let verify_type = t = expected_type in
                    let verify_exp = aux tree_exp (Exp e) (T H) in
                    let verify_command = aux tree_cmd (Command c) t in
                    verify_phrase && verify_type && verify_exp
                    && verify_command

                | Cmd(tau1, tau2) ->
                    let verify_phrase = p = expected_phrase in
                    let verify_type = t = expected_type in
                    let verify_exp = aux tree_exp (Exp e) (T L) in
                let verify_command = aux tree_cmd (Command c) t in
                let comp = less_or_eq tau1 tau2 in
            verify_phrase && verify_type && verify_exp
            && verify_command && comp

                |_ -> false

            )

| _ -> failwith"problem in the structure of the tree (while)"
    en
```

```
| SeqDeriv(p, t, tree1, tree2) ->
    begin match p with
        | Command (Seq(c1, c2)) ->
            ( match t with
            | Cmd(tau, H) ->
                let verify_phrase = p = expected_phrase in
                let verify_type = t = expected_type in
                let verify_c1 = aux tree1 (Command c1) t in
                let verfiy_c2 = aux tree2 (Command c2) (Cmd(H, H)) in
                verify_phrase && verify_type && verify_c1
                && verfiy_c2
            | Cmd(tau1, tau2) ->
                let verify_phrase = p = expected_phrase in
                let verify_type = t = expected_type in
                let verify_c1 = aux tree1 (Command c1) (Cmd(tau1, L))
                in let verfiy_c2 = aux tree2 (Command c2) t in
                verify_phrase && verify_type && verify_c1 && verfiy_c2
            | _ -> false )
        |_ -> failwith"problem in the structure of the tree (Seq)"
    end
```

```
| SubDeriv(p, t1, t2, son1, subtype_tree) ->
        let verify_phrase = p = expected_phrase in
        let verify_type = t1 = expected_type in
        let verify_son = aux son1 p t2 in
        let verify_subrules = check_sub_rules subtype_tree t2 t1 in
verify_phrase && verify_type && verify_subrules && verify_son

|_ -> false

in let p, t = extract_phrase_and_type tree in aux tree p t
```

```
let tree1 = BinopDeriv(
            Exp(Binop(Plus, Var "a", Int 1)),
            T L,
            VarDeriv("a", T L),
            ConstDeriv(1)
        )


let tree2 = SubDeriv(
            Command (Assign("b", Int 2)),
            Cmd(L, L),
            Ncmd(L, 1),
            AssignDeriv(
                "b",
                Exp(Int 2),
                Ncmd(L, 1),
                VarDeriv("b", T L),
                ConstDeriv(2)
            ),
            SubRules(Ncmd(L, 1), Cmd(L, L), Empty,Empty
            )                                 )
```

```
let tree3 = SubDeriv(
        Command Skip,
        Cmd(L, L),
        Ncmd(H, 1),
        SkipDeriv(Ncmd(H, 1)),
        SubRules(
            Ncmd(H, 1),
            Cmd(L, L),
            SubRules(Ncmd(H, 1), Cmd(H, L), Empty, Empty),
                SubRules(Cmd(H, L), Cmd(L, L), Empty, Empty)
            )
        )

let ex = IfDeriv(
        Command(
            If( Binop(Plus, Var "a", Int 1), Assign("b", Int 2), Skip)
        ),
        Cmd(L, L),
        tree1,
        tree2,
        tree3
        )

let gamma = [("a", Var L); ("b", Var L)]
```