Certification en Coq

de propriétés de sécurité issues du typage

Benjamin Gregoire – Projet COQ et CRISTAL

Que peut-on garantir avec le typage?

Notion habituelle: les programmes bien typés vont s'executer correctement (plus de *segmentation fault*).

Mais aussi: Garantir la protection des informations.

Notre But

Formaliser et établir la correction des preuves de l'article de Xavier Leroy et Francois Rouaix:

Security properties of typed applets

Pour se chauffer: Une propriété de base.

Plus dur: Encapsulation procédurale et le *containment lemma*.

Plus tard: Encapsulation avec types abstraits.

Nos moyens : le système Coq

Formalisme : le calcul des constructions inductives (C.C.I).

Utilise l'isomorphisme de Curry-Howard:

- Les preuves sont des objets.
- Les propositions sont des types.
- Une preuve d'une proposition P est un objet p de type P.

Les types inductifs

- Ils correspondent à une définition par plus petit point fixe.
- D'un point de vue informatique ce sont les types concrets de ML.
- D'un point de vue logique,les prédicats par clauses à la prolog.

Quelques Rappels

Le langage : λ-calcul avec des types de base, des paires et des références.

La sémantique : Sémantique opérationelle “Big Step” + “Store Control”

exemple :

$$\frac{\varphi, e, s| - a_1 \rightarrow l/s_1 \quad \varphi, e, s_1| - a_2 \rightarrow v_2/s_2 \quad l \in \mathit{Dom}(\varphi) \wedge v_2 \notin \varphi(l)}{\varphi, e, s| - a_1 := a_2 \rightarrow err}$$

Un premier exemple de syntaxe en Coq: représentation des valeurs

$$
\begin{array}{rcl}
v & ::= & b \\
 & | & \lambda x.a[e] \\
 & | & (v_1, v_2) \\
 & | & l
\end{array}
$$

$$
\begin{array}{lll}
\texttt{Inductive} & & \mathsf{values} : \mathsf{Set} := \\
 & | & \mathsf{constv} : \mathsf{Const} \rightarrow \mathsf{values} \\
 & | & \mathsf{absv} : \mathsf{Ident} \rightarrow \mathsf{terms} \rightarrow \mathsf{env} \rightarrow \mathsf{values} \\
 & | & \mathsf{pairv} : \mathsf{values} \rightarrow \mathsf{values} \rightarrow values \\
 & | & \mathsf{st\_loc} : \mathsf{loc} \rightarrow \mathsf{values}.
\end{array}
$$

Représentation des stores et des environements

- Rester le plus abstrait possible.

exemple : `Variable` env : Set.

`Variable` app_e : env $\rightarrow$ Ident$\rightarrow$ res_app.

`Variable` subs_e : env $\rightarrow$ Ident $\rightarrow$ values $\rightarrow$ env.

Une propriété de base : l'accessibilité

Seules les adresses accessibles depuis l'environement ou créées pendant l'exécution sont modifiables.

$RL(v, s)$ = les adresses accessibles depuis la valeur v dans l'état mémoire s

$$
\begin{aligned}
RL(b, s) &= \emptyset \\
RL(l, s) &= \{l\} \bigcup RL(s(l), s) \\
RL(\lambda x.a[e], s) &= RL(e, s) \\
RL(e, s) &= \bigcup_{x \in Dom(e)} RL(e(x), s)
\end{aligned}
\tag{1}
$$

Représentation de RL :
encore une définition inductive

```
Inductive Rl : values → stores → loc → Prop:=                    1

   Rl_abs : (x,y:Ident)(a:terms)(e:env)(s:stores)(v:values)(l:loc)
          (app_e e y)=(def v) → (Rl v s l) → (Rl (absv x a e) s l)
  |Rl_pair_L : (v1,v2 : values ) (s:stores) (l:loc)
          (Rl v1 s l) → (Rl (pairv v1 v2) s l)
  |Rl_pair_R : (v1,v2 : values) (s:stores) (l:loc)
          (Rl v2 s l) → (Rl (pairv v1 v2) s l)
  |Rl_st_loc_id :(s:stores) (l:loc) (Rl (st_loc l) s l)
  |Rl_st_loc_rec : (v:values)(l:loc)(s:stores)(l':loc)
          (app_st s l)=(def v) → (Rl v s l') → (Rl (st_loc l) s l').
```

Proposition de base

`Lemma` Lemma_1: (p:sensitive_location;e:env;s:stores;a:terms;r:resul ts)
(Evaluation (Prot p) e s a r) $\rightarrow$ ((l:loc)(Rl_env e s l) $\rightarrow$ (p l)=false)
$\rightarrow$ (exist_lemme_1 p s r).

(exist_lemme_1 p s r) := $\exists$ v,s' | (Rl v,s') $\cap$ p = $\emptyset$
$\wedge$ $\forall$ w (Rl w,s) $\cap$ p = $\emptyset$ $\rightarrow$ (Rl w,s') $\cap$ p = $\emptyset$.

`Lemma` Proposition_1 : (p:sensitive_location;e:env;s:stores;a:terms)
((l:loc)(Rl_env e s l)$\rightarrow$(p l)=false) $\rightarrow$ $\neg$ (Evaluation (Prot p) e s a err).

$$\text{(Prot p)} = \begin{cases} \text{undef} & \text{si l} \notin p \\ \emptyset & \text{sinon} \end{cases}$$

L'accessibilité une condition trop forte ?

Exemple : 2 fonctions qui ne présentent pas de risque

- $\lambda x.!r$

- $\lambda x.$ *if je suis appelé par l'applet"*
  then error()
  else r:=x

Que veut-on ?

- S'assurer que les fonctions de l'environement font les bons contrôles d'accès.
- S'assurer que les références "sensibles" ne sont manipulables que par du code qui provient de l'environnement :
  pas de fuite de références "sensibles" de l'environnement vers l'applet (*containment*).

Exemple de fuite possible.

Références sensibles = de type τ *ref* $\in I$

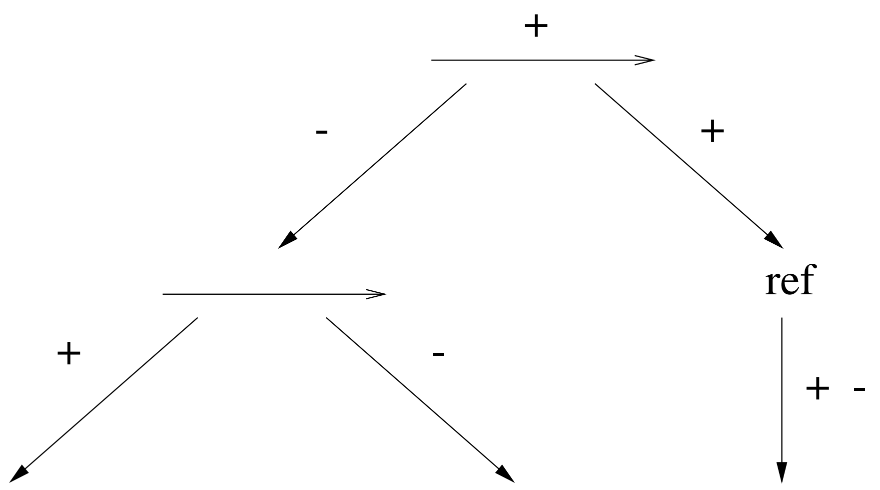
x :τ *ref*	direct
f : *int*$\rightarrow$ τ *ref*	le résultat est une référence
g : (τ *ref*$\rightarrow$ int) $\rightarrow$ *int*	fonction d'ordre supérieur
h : (τ *ref*) *ref*$\rightarrow$ *int*	tordu

Un contre exemple: k : τ *ref*$\rightarrow$ *int*

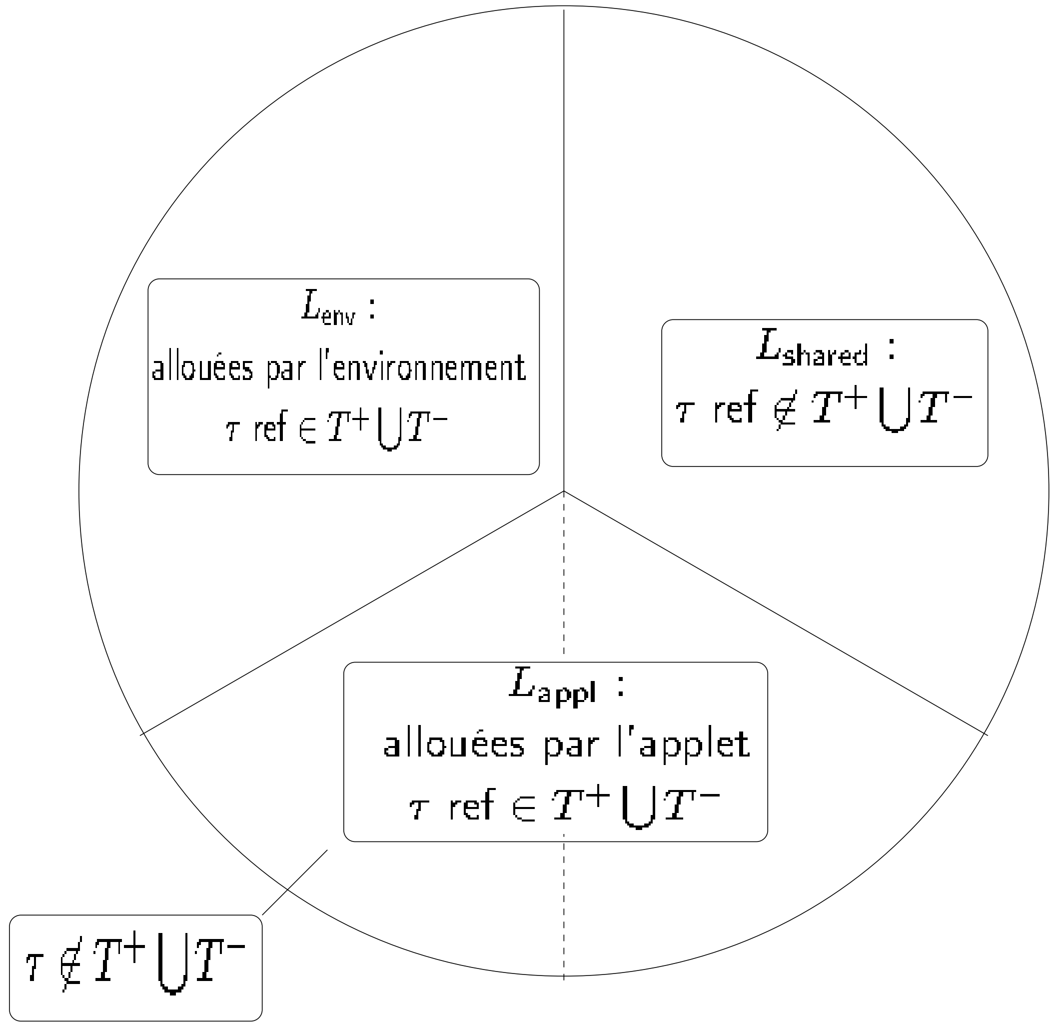
$T^+ \quad T^-$

$T^+ =$ l'ensemble des types qui contiennent un sous-terme $\tau \in I$ en occurrence positive.

$T^- =$ l'ensemble des types qui contiennent un sous-terme $\tau \in I$ en occurrence négative .



partition de la mémoire



Comment le prouver?

$C(s, S, m, v, \tau)$: Dans le stores s de type S,
la valeur v de type τ ne donne accès qu'à des références permisent au monde m.

CoInductive C : worlds $\rightarrow$ values $\rightarrow$ types $\rightarrow$ Prop:=

Le predicat C: cas de base.

```
Cconst : (m:worlds;b:Const) (C m (constv b) (const_T (Typeof b)) )

| Cpair: (m:worlds;v1,v2:values;t1,t2:types)
        (C m v1 t1) → (C m v2 t2)
        → (C m (pairv v1 v2) (prod_T t1 t2))

| Cnamed : (m:worlds;v:values;tn:T_Named;t:types)
        (TD tn)=(def_td t) → (C m v t)
        → (C m v (named_T tn t))
```

Le predicat C: les références.

```
|Cloc_app:(l:loc;v:values;t:types)
          (app_st s l)=(def v)
          → (app_S S l)=(def_T t)
          → ( (L l (world envi)) →False)
          → (C appl v t)
          → (C appl (st_loc l) (ref_T t))

|Cloc_env:(l:loc;v:values;t:types)
          (app_st s l)=(def v)
          → (app_S S l)=(def_T t)
          → ( (L l (world appl))→(T init appl (ref_T t)) →False)
          → (C envi v t)
          → (C envi (st_loc l) (ref_T t)).
```

Le predicat C:les fermetures .

```
|Cfun_intra : (m:worlds;x:Ident;a:terms;e:env;t1,t2:types;E:T_env)
            (flag a m)
            → (well_typed E (abs x a) (fun_T t1 t2))
            → (types_env S e E)
            → (C_env m e E)
            → (C m (absv x a e) (fun_T t1 t2))

|Cfun_cross : (x:Ident;a:terms;e:env;t1,t2:types;E:T_env;m:worlds)
            (flag a (compl m))
            → (well_typed E (abs x a) (fun_T t1 t2))
            → (types_env S e E)
            → ((T init (compl m) (fun_T t1 t2))→False)
            → (C_env (compl m) e E))
            → (C m (absv x a e) (fun_T t1 t2))
```

Le “containment lemma”

L’évaluation d’un terme a qui provient du monde m dans un environement contenu dans le monde m produit un résultat contenu dans le monde m.

```
Lemma containment_lemma :(e:env;s:stores;phi:st_control;a:terms;r:results)
  (Evaluation phi e s a r)
  → (E:T_env;S:T_stores;L:loc→flags→Prop; t:types;m:worlds;s':stores;v:values)
    r=(val v s')
    → (HSLT init S L)
    → (flag a m)
    → (well_typed E a t)
    → (types_env S e E)
    → (types_stores s S)
    → (C_env m e E)
    → (exist_cont_lemma s s' S m v t).
```

Autres résultats :

- Une condition suffisante pour que l'environement initial $e0 : E0$ soit contenu dans le monde "applet" est que tous les types $E0(x) \notin T^+$.
- Preuve que l'encapsulation procédurale fait les bons contrôles d'accès

Conclusion

intérêt d'une preuve formelle :

- sûreté de la preuve.
- force à expliciter les non-dits de la preuve papier
  $\Rightarrow$ permet de montrer des résultats fins.