Ajout de la théorie logique des tableaux dans VeriFast

Pierre Nigron

29 Juin 2018

Contexte

Les programmes informatiques sont présent dans des situations critiques :

- métro (ligne 14)
- médecine (pompe à insuline)
- énergie (surêté des centrales nucléaires)
- et plein d'autres...

Correction d'un programme

Un programme est correct s'il respecte sa spécification en toutes circonstances.

Vérifier qu'un programme est correct est un problème indécidable.

Méthodes de vérification

La méthode la plus courante : Test → incomplet

Autre méthode : Prouver un programme avec la vérification déductive

Vérification déductive

La méthode déductive de VeriFast utilise la logique de Hoare.

Triplet de Hoare : précondition+programme+postcondition

```
int exemple(int a, int b)
        //@ requires a = b;
        //@ ensures result = 0;
       { return a-b;}
```

VeriFast

VeriFast est un outil de vérification déductive de programmes C ou Java. (B.Jacobs, J.Smans , F.Piessens à l'université de Leuven)

VeriFast assure entre autres :

- Pas d'accés illégaux à la mémoire
- Les contrats sont respectés. (pré/postcondition)

Cette outil est basé sur la logique de séparation et résout les énoncés mathématiques avec les SMT-solveurs Redux et Z3.

Motivations

```
void swap (int* a, int i, int j)
{int b = a[i];  a[i] = a[j];  a[j] = b;}

int partition (int* a, int lo, int hi)
{ int pivot = a[hi]; int i = lo - 1; int j;
  for (j = lo; j < hi; j++) {
    if (a[j] < pivot) {i++; if (i < j) swap(a, i, j);}
  }
  i++; if (i < hi) swap(a, i, hi); return i;}

void quicksort (int* a, int lo, int hi)
{if (lo > hi) return;
 int p = partition(a, lo, hi);
 quicksort(a, lo, p-1);
 quicksort(a, p+1, hi);}
```

Précondition : Avoir un tableau bien défini
Postcondition : Avoir un tableau bien défini, un multi-ensemble d'élements identique et un tableau trié.

Contributions

1. Automatiser la théorie des tableaux dans VeriFast.
2. Ajouter des bibliothèques pour la théorie des tableaux et des multi-ensembles.
3. Prouver le Quicksort à l'aide de VeriFast

Logique du premier ordre

Une **signature** Σ est une paire $\Sigma = (\Sigma^s, \Sigma^f)$

$$
\begin{array}{lcll}
t & ::= & x & x^\sigma \in Var \text{ et } \sigma \in \Sigma^s \\
 & | & f(t_1, \ldots, t_n) & t_1^{\sigma_1}, \ldots, t_n^{\sigma_n} \text{ et } f^{\sigma_1 \ldots \sigma_n \sigma} \in \Sigma^f
\end{array}
$$

Une **formule** du premier ordre :

$$
\begin{array}{lclll}
\phi^{FOL} & ::= & t_1 \approx t_2, & t_1^\sigma, t_2^\sigma & \text{(égalité)} \\
 & | & \neg\phi^{FOL}, & & \text{(négation)} \\
 & | & \phi_1^{FOL} \wedge \phi_2^{FOL} & & \text{(conjonction)} \\
 & | & \phi_1^{FOL} \vee \phi_2^{FOL} & & \text{(disjonction)} \\
 & | & \exists x.\phi^{FOL}, & x \in FV(\phi^{FOL}) & \text{(quantificateur existentiel)} \\
 & | & \forall x.\phi^{FOL}, & x \in FV(\phi^{FOL}) & \text{(quantificateur universel)}
\end{array}
$$

Théorie des tableaux

Soit σ_I le type des indices, σ_E le type des éléments et σ_A le type du tableau.

La théorie des tableaux est munie de la signature :
$\Sigma^s = \{\sigma_I, \sigma_E, \sigma_A\}$
et $\Sigma^f =$
$\{select^{\sigma_A\sigma_I\sigma_E}, store^{\sigma_A\sigma_I\sigma_E\sigma_A}, constant\_array^{\sigma_E\sigma_A}, array\_ext^{\sigma_A\sigma_A\sigma_I}\}$.

Exemple :

$$\exists a \in \sigma_A \exists i, j \in \sigma_I, select(a, i) \neq select(a, j)$$

Axiomes

Axiomes de select et store :

$$\forall a \in \sigma_A.\ \forall e \in \sigma_E.\ \forall i, j \in \sigma_I.\ select(store(a, i, e), i) = e$$

$$\forall a \in \sigma_A.\ \forall e \in \sigma_E.\ \forall i, j \in \sigma_I.$$
$$i \neq j \rightarrow \mathrm{select}(\mathrm{store}(a, i, e), j) = \mathrm{select}(a, j)$$

Axiome de constant_array et select :

$$\forall e \in \sigma_E.\ \forall i \in \sigma_I.\ select(constant\_array(e), i) = e$$

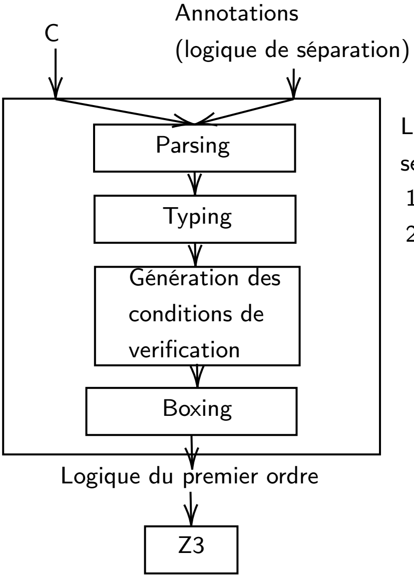
Axiome de l'extensionalité :

$$\forall a, b \in \sigma_A.\ a \neq b \rightarrow$$
$$select(a, array\_ext(a, b)) \neq select(b, array\_ext(a, b))$$

Logique de spécification de VeriFast

```
struct list { struct list *next; int value; };

inductive ints = ints_nil | ints_cons(int, ints);

predicate lists(struct list *list, ints num) =
        list == NULL ? num == ints_nil :
        num == ints_cons(?number,?num2)
        &*& list -> next |-> ?n &*& list->value |-> number
        &*& lists(n,num2);

fixpoint int length_ints(ints l){
 switch(l) {
  case ints_nil : return 0;
  case ints_cons(h,t) : return 1 + length_ints(t);
 }}

 lemma void empty_list(struct list *l)
  requires l == NULL;
  ensures lists(l,ints_nil);
 {close lists(l,ints_nil);}
```

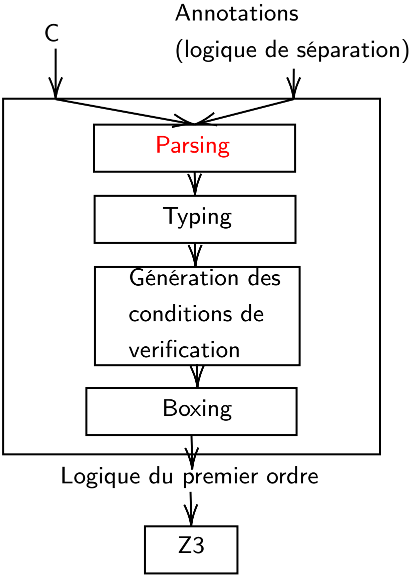
Automatisation de la théorie des tableaux



L'implémentation de l'automatisation se divise en deux parties :

1. Ajouter un type tableau
2. Ajouter les fonctions de la théorie

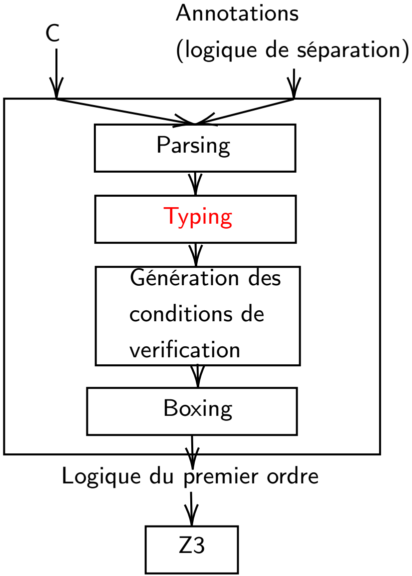
Logique de spécification



La syntaxe du type tableau est :
"$array(\sigma_I, \sigma_E)$"

Par exemple un tableau qui associe des entiers à des booléens s'écrit "array(int,bool)"

Logique de spécification 2

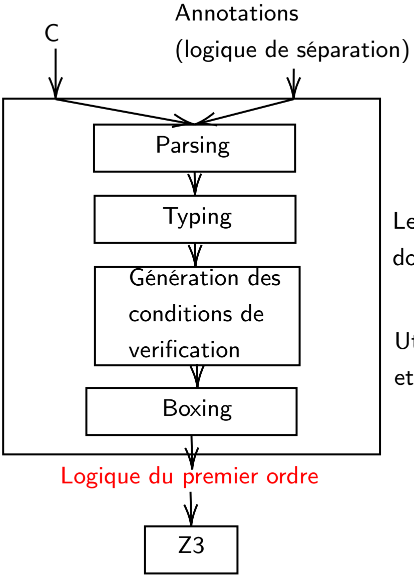


Le typing est une phase vérifiant la cohérence des types.

Les tableaux, seuls, ne posent pas de problème de cohérence.

Pour certaines utilisations, il faut vérifier la finitude ou l'habitation du type.

Logique du premier ordre de VeriFast



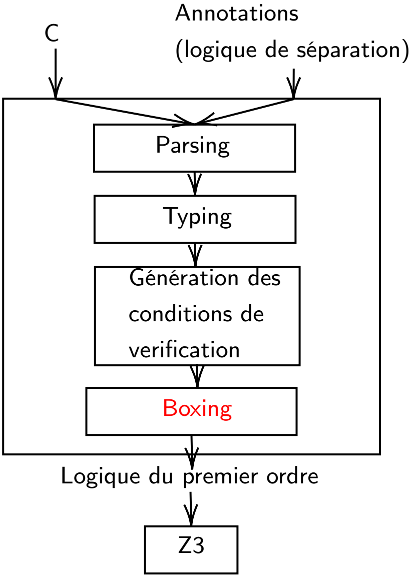
Σ^s = (inductive,int,real,boolean)

$\Sigma^f = (boxed_{int}, unboxed_{int}, \ldots, +, -, \ldots)$

Le type inductive représente plusieurs types dont les types génériques.

Utilisation de fonction de boxing et d'unboxing

Boxing



$boxed_\sigma : \sigma \rightarrow \text{inductive}$

$boxed_{array}$:
array(inductive,inductive) $\rightarrow$ inductive

$unboxed_\sigma \circ boxed_\sigma = \text{id}$

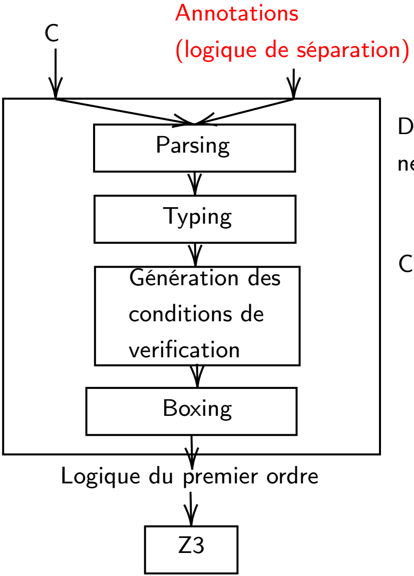
$boxed_\sigma \circ unboxed_\sigma = \text{id}$ si σ est infini

Ajout des fonctions

Pour terminer l'implémentation, il faut ajouter les fonctions de la théorie. Il y a deux étapes :

- Les intégrer à la logique de spécification.
- Les relier aux fonctions de Z3.

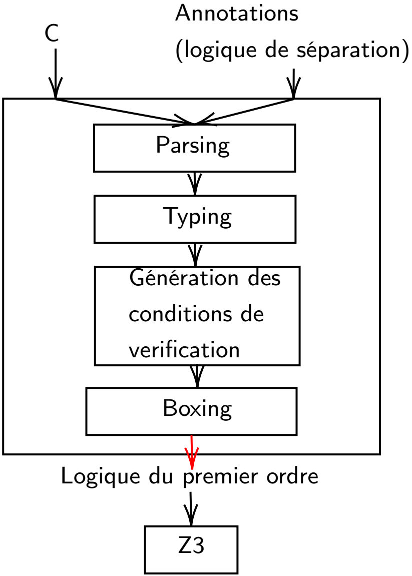
Fonction dans la logique de spécification



Deux fixpoints, prédicats ou lemme
ne peuvent pas avoir un nom identique.

Création d'une nouvelle bibliothèque

Liaison avec Z3



Bibliothèques de la théorie des tableaux

```
fixpoint u select<t,u> (array(t,u) arr, t x);
fixpoint array(t, u) store<t, u> (array(t, u) arr, t x, u y);
fixpoint array(t, u) constant_array<t,u> (u v);
fixpoint t array_ext<t, u> (array(t, u) a, array(t, u) b);

lemma void constant_select<t,u> (u v, t i)
  requires true;
  ensures select (constant_array<t,u>(v), i) == v;
  {}

lemma void select_store<t,u> (array(t,u) arr, t x, u y, t z)
  requires true;
  ensures select (store(arr, x, y), z)
              == ((x == z) ? y : select (arr, z));
  {}

lemma void array_extensionality<t, u>(array(t, u) a,
                                      array(t, u) b)
  requires select(a, array_ext<t, u>(a, b))
              == select(b, array_ext<t, u>(a, b));
  ensures a == b;
  {}
```

Bibliothèques de la théorie des tableaux 2

```
predicate array_model (int* a, int b, int e,
                            array(int,int) arr) =
  (b >= e) ? true : (integer(a+b,?v)
            &*& select(arr, b) == v
            &*& array_model(a, b+1, e, arr));

lemma void array_model_init(int* a, int length)
requires ints(a, length, _) &*& length >= 0;
ensures array_model(a, 0, length, _);
{...}
```

Bibliothèque des multi-ensembles

```
inductive multiset<t> = mk_multiset (array(t, nat));

fixpoint nat multiset_select<t>(multiset<t> m, t i)

fixpoint multiset<t> empty_multiset<t>()

fixpoint multiset<t> multiset_add<t>(multiset<t> m, t i)

fixpoint multiset<int> array_multiset(int b, nat n,
                                      array(int,int) arr)

lemma t multiset_ext<t>(multiset<t> m1, multiset<t> m2)
  requires m1 != m2;
  ensures multiset_select(m1, result) !=
          multiset_select(m2, result);
  {...}
```

Bibliothèque des multi-ensembles 2

```
predicate same_multiset(array(int,int) a1,
                        array(int,int) a2, int b, int e) =
  array_multiset(b, nat_of_int(e-b), a1) ==
  array_multiset(b, nat_of_int(e-b), a2);
```

Rappel Quicksort

```
void swap (int* a, int i, int j)
{int b = a[i];  a[i] = a[j];  a[j] = b;}

int partition (int* arr, int lo, int hi)
{ int pivot = a[hi]; int i = lo - 1; int j;
  for (j = lo; j < hi; j++) {
    if (a[j] < pivot) {i++; if (i < j) swap(a, i, j);}
  }
  i++; if (i < hi) swap(a, i, hi); return i;}

void quicksort (int* a, int lo, int hi)
{if (lo > hi) return;
 int p = partition(a, lo, hi);
 quicksort(a, lo, p-1);
 quicksort(a, p+1, hi);}
```

Spécification swap

```
fixpoint array(int, int) array_swap(array(int, int) start,
                                    int i, int j) {
  return store(store(start, j, select(start, i)),
               i, select(start, j));
}

void swap (int* a, int i, int j)
  //@ requires array_model(a, ?b, ?e, ?start) &*&
               b <= i &*& i < j &*& j < e;
  //@ ensures array_model(a, b, e, array_swap(start, i, j));
```

Spécification partition

```
int partition (int* a, int lo, int hi)
  /*@ requires array_model(a, lo, hi+1, ?start) &*&
               lo <= hi &*&
               pivot == select(start, hi);
      ensures array_model(a, lo, hi+1, ?end) &*&
              same_multiset(start, end, lo, hi+1) &*&
              lo <= result &*& result <= hi &*&
              select(end, result) == pivot &*&
              upper_bound(end, lo, result, pivot) &*&
              lower_bound(end, result+1, hi+1, pivot); @*/
```

Spécification partition 2

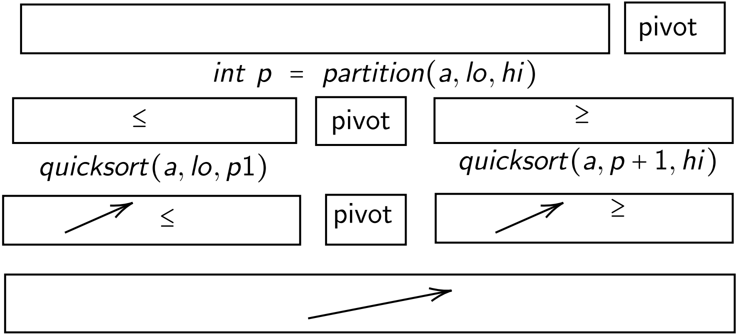
```
int pivot = a[hi];
int i = lo - 1;
for (j = lo; j < hi; j++)
/*@ invariant array_model(a,lo,hi,?arr) &*&
              lo <= j &*& j < hi+1 &*&
              i < j &*& lo -1 <= i &*&
              same_multiset(start, arr, lo, hi) &*&
              select(arr, hi) == p &*&
              upper_bound(arr,lo,i+1,p) &*&
              lower_bound(arr,i+1,j,p); @*/
{
    if (a[j] < pivot) {
      i++;
      if (i < j) swap(a, i, j);
    }
}
i++;
if (i < hi) swap(a, i, hi);
return i;
```

Spécification quicksort

```
void quicksort (int* a, int lo, int hi)
  //@ requires array_model(a, lo, hi+1, ?start);
  //@ ensures array_model(a, lo, hi+1, ?end) &*&
               same_multiset(start, end, lo, hi+1) &*&
               sorted(end, lo, hi+1);

{
  if (lo > hi) return;
  int p = partition(a, lo, hi);
  quicksort(a, lo, p-1);
  quicksort(a, p+1, hi);
}
```

Spécification quicksort 2

```
int p = partition(a, lo, hi);
quicksort(a, lo, p-1);
quicksort(a, p+1, hi);
```



Conclusion

Travail accompli :

- J'ai amélioré l'automatisation de VeriFast avec la théorie des tableaux
- Les bibliothèques de la théorie des tableaux et des multi-ensembles ont été ajoutées.
- Le Quicksort a été prouvé avec VeriFast.

Suite possible :

- Prouver d'autre programme afin d'enrichir les bibliothèques.
- Ajouter des fonctions à la théorie des tableaux.
- Ajouter d'autres théories