Chapitre 1

Introduction

1.1 Objectif du TRE

1.1.1 Intérêt de prouver des programmes

Aujourd'hui, les programmes informatiques sont de plus en plus présents dans notre quotidien et ont certaines fois notre vie entre leurs lignes. En effet, si un bug informatique empêche le métro de s'arrêter ou fait exploser une fusée, il risque d'y avoir des pertes humaines.

Malheureusement, vérifier qu'un programme est correct est indécidable. Du coup, la méthode la plus courante est de tester un programme dans différents environnements. Cependant, on ne peut pas tester le nombre infini de vols possibles d'une fusée. Les méthodes de test permettent de connaître le comportement réel du programme dans un environnement précis ; à l'inverse, d'autres méthodes utilisent un modèle de l'exécution du programme pour déterminer une approximation du comportement du programme dans de nombreux environnements. Même si le résultat est une approximation, cette méthode détectera si le programme se comporte correctement. Par contre, elle peut indiquer qu'un programme n'est pas correct alors qu'il l'est. Mais ceci n'apporte aucun préjudice à la sécurité.

1.1.2 Logique de Hoare

Les méthodes déductives qui vont prouver qu'un programme suit sa spécification utilise la logique de Hoare.

Cette logique se base sur le triplet de Hoare muni d'axiomes et règles pour toutes le instructions de base d'un langage impératif. Le triplet est constitué d'une précondition B, d'une postcondition A et d'un programme P représenté ainsi : $\{B\}P\{A\}$. Deux types de correction peuvent être prouvées :

— Une correction partielle : le triplet est vrai si pour tout valuation qui rend B vraie et telle que P s'arrête, alors, après l'exécution et l'arrêt de P, A est vraie.

— Une correction totale : le triplet est vrai si pour tout valuation qui rend B vraie, P s'arrête et, après sont exécution, A est vraie.

En pratique, une programme calcule la plus faible précondition nécessaire pour que la postcondition soit respecté. On la notera $wp(P, A)$ et on aura donc $\forall B'$ tel que $\{B'\}P\{A\}$, $B' \rightarrow wp(P, A)$

Axiome de skip

$$\frac{}{\{\mathrm{P}\}\textbf{skip}\{\mathrm{P}\}}$$

Un programme vide ne change pas l'état du programme.

Axiome de l'affectation

L'affectation est l'instruction $x := E$, associant à la variable x la valeur de l'expression E. $P[E/x]$ désigne l'expression P dans laquelle les occurrences de la variable x ont été remplacées par l'expression E.

$$\frac{}{\{P[E/x]\}\ x\ :=\ E\{P\}}$$

$$wp(x := E, P) = P[E/x]$$

Règle de composition

La règle de composition pour les programmes S et T s'ils sont exécutés séquentiellement, où S s'exécute avant T.

$$\frac{\{P\}S\{Q\},\ \{Q\}T\{R\}}{\{P\}\ S;T\ \{R\}}$$

$$wp(S;T, R) = wp(S, wp(T, R))$$

Règle de la conditionnelle

La règle de la conditionnelle a une postcondition R qui doit être commun à **alors** et **sinon**.

$$\frac{\{B\ \wedge\ P\}S\{Q\},\ \{\neg B\ \wedge P\}T\{Q\}}{\{P\}\ \textbf{si}\ B\ \textbf{alors}\ S\ \textbf{sinon}\ T\ \{R\}}$$

$$wp(\text{ si } B \text{ then } S \text{ else } T, R) = (B \rightarrow wp(S, R) \wedge \neg B \rightarrow wp(T, R))$$

Règle de la conséquence

La règle de la conséquence permet d'affailblir les préconditions et postconditions en les remplaçant par des conséquences logiques.

$$\frac{P\ \rightarrow P',\ \{P'\}S\{R'\},\ R'\ \rightarrow R}{\{P\}S\{R\}}$$

Règle de l'itération

Dans une boucle, on doit s'assurer qu'à chaque itération des propriétés sont respectés, ce sont les invariants de boucle. Cette règle n'assure pas la terminaison de la boucle.

$$\frac{\{I \wedge B\} S \{I\}}{\{I\}\textbf{tant que } B \textbf{ faire } S\{\neg B \wedge I\}}$$ où I est l'invariant de boucle.

Correction totale

La logique de Hoare peut permettre d'assurer la terminaison. La règle de l'itération doit être enrichi avec le notion de variant : une fonction des états dans un ensemble bien fondé dont la valeur décroit strictement à chaque tour de boucle. L'absence de chaîne infinie décroissante dans un ensemble bien fondé garantie donc la terminaison du programme. La règle devient alors :

$$\frac{\{I \wedge B \wedge t \in V \wedge t = z\} S \{I \wedge t \in V \wedge t < z\}}{\{I \wedge t \in V\} tant\ que\ B\ faire\ S\{\neg B \wedge I \wedge t \in V\}}$$

1.1.3 Les outils

Ils existent différents outils pour prouver ou certifier des programmes. Il y a des assistants de preuve tel que Coq ou Isabelle qui permettent de démontrer des théorèmes mathématiques. En modélisant mathématiquement le comportement du programme, il est donc possible de prouver sa correction.

D'autres logiciels permettent la preuve de programme en se spécialisant dans des langages précis. Par exemple, Verifast a été conçu pour le C et le java.

1.1.4 Courte description de Verifast

Verifast est un outil de vérification déductive de programme, C ou Java, avec un seul ou plusieurs threads. L'outil vous assure qu'il n'y a pas d'accès illégaux à la mémoire, que les préconditions et postconditions sont bien respectées et qu'il n'y est pas de problème de concurrence. Il est principalement conçu par Bart Jabocs, Jan Smans et Frank Piessens à l'université de Leuven en Belgique. Verifast est basé sur la logique de séparation, la logique multi-sorted et résout les énonces mathématiques avec les SMT-solvers Redux et Z3.

1.1.5 Exemple : Quicksort avec verifast et d'autres....

1.1.6 Contributions

Il y a eu trois contributions concernant verifast. Pour des raisons pratiques, les trois contributions ont été fait simultanément.

Integrer la theorie des tableaux

Z3 a des fonctions primitives pour la théorie des tableaux mais verifast ne les utilise pas. La première contribution a été d'enrichir la logique de verifast en y ajoutant des fonctions représentant la théories des tableaux afin de les relier aux fonctions primitives de Z3.

La théorie des tableaux permet de simplifier certaines preuves comme celle du Quicksort.

Bibliothèque sur les tableaux et les multi-ensembles

Afin de prouver l'algorithme du quicksort, deux bibliothèques ont été crées. La première concerne les tableaux et la deuxième les multi-ensembles.

Preuve de quicksort

Il n'y avait pas de preuve du quicksort dans verifast.

Prouver l'algorithme permettait à la fois d'enrichir les preuves de verifast mais aussi de vérifier l'intégration de la théorie des tableaux.

Chapitre 2

Logique du premier ordre

2.1 Logique du premier ordre

2.1.1 Syntaxe

On introduit la syntaxe de la logique du premier ordre dans le contexte d'une paire $\sum = (\sum^s, \sum^f)$, que nous appelerons la signature tel que :

— $\sum^s = \{\sigma_1, \sigma_2, ...\}$ un ensemble de symboles de type. On suppose l'existence du type $Bool = \{\top, \bot\}$.

Signature : un ensemble non vide de sort, un ensemble de symboles de fonction et un ensemble de symboles de relation non logique.

Il existe une réduction vers one-sorted logic (logique du premier ordre classique).

2.2 Décidabilité des théories

Une théorie est un ensemble d'axiomes accompagnés de règles de la logique.

Sous une signature σ et une théorie τ_σ, une formule ϕ, construite dans σ, est satisfaisable in τ_σ, si ϕ est évalué à vrai sous toutes les interprétations de τ_σ. Certaines théories sont décidables. Une théorie est décidable s'il existe un algorithme qui peut répondre par oui ou par non à la question de savoir si un énoncé donné est démontrable dans cette théorie.

Une théorie est cohérente s'il existe des propositions non démontrable.

Une théorie est complète si toute proposition ou sa négation est démontrable.

2.2.1 Presburger

L'arithmétique de Presburger est la théorie décidable du premier ordre des nombres entiers naturels muni de l'addition. En 1929, Mojzesz Presburger a démontré que son arithmétique est cohérente et complète.

2.2.2 Array

Les tableaux sont une data-structure présentes dans la plupart des langages de programmation. Il est donc intéressant de savoir quelles sont les propriétés décidable.

En logique, contrairement aux programmes, un tableau est de dimension infini.

Les tableaux sont équivalent à une fonction de type index vers range et nous allons leurs associées quatre opérations et un axiome proposé par McCarthy.

Select

Select(array a, index i) renvoie la valeur du tableau a à l'index i.

Store

Store(array a, index i, range e) renvoie un nouveau tableau avec les mêmes valeurs que le tableau a et la valeur e à l'index i.

Extensionality

Extensionality(array a, array b) renvoie un index i tel que
$\text{select}(a, i) = \text{select}(b, i) \rightarrow a = b$

Constant_array

Constant_array(range v) renvoie un tableau a tel que
$(\forall i : \text{index})\ \text{select}(\text{constant\_array}(v), i) = v$.

Axiome

L'axiome principal est :
$(\forall a : \text{array})\ (\forall e : \text{range})(\forall i, j : \text{index})\ i = j \rightarrow \text{select}(\text{store}(a, i, e), j) = e$
$\wedge\ i \neq j \rightarrow \text{select}(\text{store}(a, i, e), j) = \text{select}(a, j)$.

2.2.3 Multiset

Représentation des multiset en array(index, nat)

2.3 Combinaison de théories décidables

La combinaison de théories décidables est omni-présent dans la vérification de programme et il n'est pas évident que la combinaison reste décidable. Pour éviter d'avoir à produire une procédure de décision pour chaque combinaison, on peut faire appel à des techniques de combinaison de procédures de décisions.

En 1979, la méthode de combinaison de Nelson-Oppen est présenté pour la première fois. 20 ans plus tard, cette méthode est adoptée dans la plupart des

solvers SMT. Cette méthode se base sur le fait que si deux théories $\tau 1$ et $\tau 2$ sont disjointes et stablement infinies alors la satisfaisabilité de $\tau 1 \vee \tau 2$ peut être déduite de la satisfaisabilité de $\tau 1 \vee \Lambda$ et $\tau 2 \vee \Lambda$, où Λ est un ensemble d'informations partagées. De ce fait, si on a deux procédures de décision, une pour $\tau 1$, et une autre pour $\tau 2$, l'effort supplémentaire pour vérifier la satisfaisabilité de l'union est de trouver Λ. Ceci peut être étendu à plus de deux théories.

Deux théories sont disjointes si aucun symbole n'apparaît dans les deux théories à la fois, excepté des variables et le symbole d'égalité.

Une théorie est stablement infinie si toute formule sans quantificateur satisfaisable dans la þéorie a un modèle fini.

Les techniques de combinaison de décision sont essentielles à la construction des SMT solvers.

2.4 SMT-solver

Il existe de nombreux SMT-solver permettant de résoudre des énoncés mathématiques. Tous ne résolvent pas les énoncés des mêmes théories et ne sont aussi rapide. Verifast utilise deux SMT-solver, redux et Z3. Redux est plus rapide mais Z3 peut résoudre des énoncés d'un plus grand nombre de théorie.

- théorie avec fonction non interprété
- Communication entre les théories et un sat-solver
- Communication pour les axiomes "infini" entre le quantifier instantiation et le sat.

Chapitre 3

Vérification déductive avec VeriFast

3.1 Vérification déductive de programmes

La vérification déductive se base sur l'annotation des programmes. Le programme est annoté avec des formules logiques. Il y a les préconditions (exigences sur les arguments de fonctions), les postconditions (garanties sur les résultats de fonctions) et les invariants de boucle. Il restera à vérifier que pour chaque fonction, la fonction, accompagnée de ces préconditions, implique les postconditions et qu'à chaque appel de fonction, les préconditions sont respectées. Ces vérifications se feront à l'aide des SMT-solvers.

3.1.1 Logique de séparation

swap(int*x, int*y)

3.2 Logique de spécification en VeriFast

La spécification d'un programme est indiqué en commentaires à l'aide de formules logiques. Ces formules logiques peuvent être dans des prédicats (objets logiques) ou dans des lemmes (fonctions logiques).

Predicats inductifs

Il est souvent utile d'utiliser des prédicats inductifs pour représenter des propriétés. Par exemple, si on veut parler d'un tableau d'entier trié entre deux bornes b et e, on peut construire le prédicat suivant :

```
predicate sorted(array(int,int) arr,int b, int e) =
    (b >= e) ? true :
    select(arr,b) <= select(arr,b+1) &*& sorted(arr,b+1,e);
```

Contrats

Chaque fonction est accompagné d'au moins deux contrats, potentiellement vide. Les pre-conditions et les post-conditions, appelé "requires" et "ensures". Ces contrats doivent être respectés pour que le programme soit considéré comme correct. Par exemple, pour la fonction qui fait la moyenne de deux entiers :

```
int mean(int x, int y)
        //@ requires true;
        //@ ensures (x+y)/2 == result;
        {
          return x + (y - x) / 2;
        }
```

Pour cette exemple, requires est un contrat vide. En effet, on a besoin d'aucune condition supplémentaire que ce que nous donne déjà le programme C. Par contre, en post-condition, nous voulons être sur que le programme retourne bien la moyenne entre x et y.

Des conditions qui aurait pu être utile mais qui ne sont pas dans cette exemples aurait été de vérifier un possible overflow.

Appel de lemme et Ouverture/Fermeture de prédicat

En plus des pre et post conditions, il est possible de faire appel à des lemmes ou d'ouvrir/fermer des prédicats au milieu du programme.

Invariant de boucle

3.2.1 Decreases pour assurer la terminaison des boucles

Chapitre 4

Implémentation

L'objectif de l'implémentation est d'intégrer les fonctions de la théorie des tableaux dans verifast pour qu'elles soient directement retransmis à Z3. Ces fonctions manipulent un type tableau qui était absent dans la logique de verifast. La première étape fut donc d'ajouter un type primitif array dans la logique de verifast.

4.1 Création du type array

La syntaxe du type array est "array(a,b)", a étant le domaine et b le range.

4.1.1 Règle de typage

Généricité

En logique, le domaine et le range du type tableau sont deux type générique, c'est à dire, ils peuvent être de n'importe quel type. Un objectif était donc d'implémenter un type tableau tout aussi générique. Toutes les fonctions ont été implementées ainsi mais verifast a posé une limite. Il n'y a pas de fonction générique pour représenter les pointeurs définis. En effet, il y a seulement des fonctions "integer(int*,int)", "character(char*,char)", etc... qui représente des pointeurs d'entier, de char, etc. Il n'est donc pas possible actuellement de compléter la bibliothèque des tableaux avec uniquement des lemmes générique. Il est, par exemple, impossible de dire qu'un tableau est bien défini génériquement. La fonction array_model, ci-dessous, indique qu'un tableau d'entier est bien défini entre (a+b) et (a+e) si un entier est bien défini en (a+b) et qu'un tableau est bien défini entre (a+b+1) et (a+e).

```
.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));
```

Limite du domaine

S'il existe un type t de type : t → p, p étant un type quelconque, des termes tel que $(\lambda x.\neg(xx))(\lambda x.\neg(xx))$ deviennent typable alors qu'ils ne sont pas cohérent. Or, un tableau est comme fonction de type : domain → range. Je n'ai pas de preuve qu'avec les axiomes et les fonctions sur les tableaux le système devient incohérent, mais dans le doute, les tableaux de ce type ont été interdit.

Type habité

Dans verifast, tous les types doivent être habité, c'est à dire, tous les types doivent avoir au moins un élément. En effet, il existe cet axiome dans verifast :

```
fixpoint t default_value<t>();
```

Cet axiome retourne un élément du type t. Donc s'il existe un type non-habité alors le système n'est plus cohérent.

Par exemple, avec le type stream :

```
inductive stream = Cons (int, stream);
```

Il est possible de prouver faux avec ces deux fonctions :

```
lemma void no_stream (stream x)
        //@ requires true;
        //@ ensures false;
        { switch (x) : {
                case Cons(_,s) { no_stream(s); }
        }

lemma void absurd()
        //@ requires true
        //@ ensures false
        { no_stream(default_value<stream>())}
```

De plus, l'implication $\forall x.P(x) \rightarrow \exists x.P(x)$ est utilisé dans la logique de verifast et n'est vrai que si le type de x est habité.

Type infini

Exemple : utilisation de l'inégalité pour montrer l'égalité. si x peut être a, b ou c. x != a et x != b -> x == c.

4.2 ProverArray

La logique multi-sorted de verifast était composé des types int, boolean, real et inductive. Le type inductive est un type générique, c'est à dire, il existe des fonctions de boxing/unboxing pour transformer un type vers le type inductive et inversement.

Les sorts de la logique de verifast sont les types qui pourront être transmis au solver Z3. Or, pour utiliser les fonctions sur les tableaux, le type primitif array est nécessaire. Une étape fut donc de créer un nouveau type primitif array qui pourra être transmis à Z3.

4.2.1 Intêret du type Inductive

Le type inductive a un rôle très particulier dans verifast. En plus de représenter les types génériques, le type inductive est un moyen de communiquer à la fois à des solvers multi-sorted et à des solvers one-sorted. En effet, Verifast utilise deux solvers qui sont Redux et Z3. Redux n'a pas de type alors que Z3 en a. Comme indiqué en (2.1), il existe une réduction de la logique multi-sorted vers la one-sorted. Verifast peut donc aisément communiquer à Redux même s'il n'utilise pas directement la même logique. Pour transformer tous les sorts vers un seul sort, verifast utilise des fonctions de boxing.

Fonctions de boxing

Une fonction de boxing est une fonction qui transforme une valeur d'un certain type vers le type inductive. Il existe, par dualité, des fonctions d'unboxing afin de retrouver la valeur initiale. Donc si f est la fonction de boxing des entiers et g la fonction d'unboxing alors $f(g(2)) = 2$ et $g(f(2)) = 2$.

4.2.2 Représentation du type array dans le solver Z3

Dans le Z3, le type array de Z3 est un type avec deux arguments. Ces arguments représentent le domaine et le range du tableau. Une première implémentation consista donc à récupérer les deux types et de les transmettre à Z3 au moment de la création d'un tableau.

Pourquoi array inductive inductive dans z3

Communication avec Z3

Représentation dans verifast

4.3 Ajout des fonctions

Pour terminer l'implémentation, il faut maintenant ajouter les fonctions select, store, constant_array et array_ext (extensionalité). Il y a deux étapes, la première est de définir comment ces fonctions vont être communiqué à Z3. La deuxième est de choisir comment représenter et récupérer ces fonctions dans le code de verifast.

4.3.1 Communication avec les SMT-solvers

Dans verifast, tous les solvers ont une api commune. Ainsi, si on veut ajouter de nouvelles fonctions pour communiquer avec Z3, on doit s'assurer que Redux sera aussi s'en occuper.

Redux

Redux considère les fonctions de la théorie des tableaux commme des fonctions non-primitives et donc comme des fonctions défini par l'utilisateur. Lors d'un appel d'une de ces fonctions, Redux fait donc un appel de fonction classique. Peu de choses sont possibles avec Redux et il faut donc éviter d'utiliser Redux si on veut utiliser la théorie des tableaux.

Z3

Les fonctions sont des primitives de Z3. L'implémentation consiste donc à appeler les fonctions avec les informations que nous fournira l'utilisateur.

4.3.2 Représentation dans verifast

Verifast interdit à deux fonctions ou prédicats d'avoir un nom identique. Cette règle est primordiale car elle permet à verifast de définir toutes les primitives dans des bibliothèques annexes. Les fonctions de la théorie des tableaux ont été défini dans une nouvelle bibliothèque sous la forme de fonctions pures. Ainsi, tous les programmes antérieurs sont restés correct et les noms restent disponibles si la bibliothèque n'est pas incluse.

Les fonctions de la théorie sont donc, dans le code de verifast, des fonctions comme des autres. Il faut maintenant savoir comment les repérer pour utiliser les fonctions primitives de Z3.

Une possibilité aurait été de modifier le parser et ainsi savoir directement quand est-ce qu'elles sont appelées. Avec l'implémentation actuelle, l'ajout des fonctions aurait nécessité un nombre injustifié de modification dans plusieurs fichiers pour recopier, une seconde fois, ce qui est déjà écrit. Du coup, une autre solution a été choisi.

L'autre solution consiste à trouver un point précis du code où toutes les fonctions vont être appelées et où un maximum de vérification auront déjà été fait. Ainsi, le moins possible d'étapes seront implémentées et donc produira un code propre et concis.

Lors de la dernière vérification intéressante pour les fonctions pures, un matching sur le nom des fonctions permet aux primitives de la théorie d'être repéré et ainsi d'être traité comme les fonctions de la théorie des tableaux.

Chapitre 5

Quicksort

Chapitre 6

Conclusion

Table des matières