Ajout de la théorie logique des tableaux dans VeriFast

Pierre Nigron

22 juin 2018

Table des matières

Introduction

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 entraîne un accident dans le métro ou fait exploser une fusée, il risque d'y avoir des pertes humaines ou de lourdes pertes économiques.

Malheureusement, vérifier qu'un programme est correct est indécidable. Par conséquent, 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, ces méthodes détecteront 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.

Pour nous aider à prouver qu'un programme est correct, différents outils existent. Il y a des assistants de preuve tels 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 vérification déductive de programme en se spécialisant dans des langages précis, c'est en particulier le cas de l'outil VeriFast sur lequel nous allons travailler.

Les méthodes déductives qui vont prouver qu'un programme impératif suit sa spécification utilisent souvent une logique qui spécifie les états du programme. Cette logique est utilisé dans le triplet de Hoare muni d'axiomes et règles pour toutes les 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 : $\{B\}P\{A\}$. Le triplet est vrai si pour tout état qui rend B vraie et telle que P s'arrête, alors, après l'exécution et l'arrêt de P, A est vraie.

Par exemple, VeriFast est un outil de vérification déductive de programme, C ou Java, avec un seul ou plusieurs threads. L'outil 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 ait 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 et résout les énoncés mathématiques avec les SMT-solvers Redux et Z3.

Pour certains programmes comme les programmes de tri, il est naturel de les spécifier à l'aide des tableaux, des ensembles et des multi-ensembles. En effet, une spécification correcte d'un algorithme de tri est qu'il retourne un tableau trié contenant le même multi-ensemble d'éléments que le tableau initial. Malheureusement ces tableaux, ensembles et multi-ensembles ne sont pas prévus dans la logique de VeriFast alors qu'ils sont pourtant bien traités par Z3.

Les contributions du TRE se sont séparées en plusieurs parties dépendantes les unes des autres. Une partie était d'automatiser la théorie des tableaux dans VeriFast. En parallèle, une preuve de l'algorithme quicksort, absente dans VeriFast, a été écrite. La preuve et l'automatisation de la théorie ont conduit à la création des bibliothèques de la théorie des tableaux et des multi-ensembles.

Tout d'abord, nous nous intéresserons à la logique du premier ordre et aux procédures de décision. Ensuite, nous évoquerons la vérification déductive et plus particulièrement celle de VeriFast. Enfin, nous aborderons les contributions apportées par le TRE.

1 Logique du premier ordre

1.1 Syntaxe

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

- $\Sigma^s = \{\sigma_1, \sigma_2, \ldots\}$ est un ensemble de symboles de type (parfois aussi appellés symboles de sorte). On suppose l'existence d'un type $Bool = \{\top, \bot\}$, où nous écrivons $\top$ et $\bot$ pour les constantes, respectivement, vrai et faux.
- $\Sigma^f = \{f, g, h, \ldots\}$ est un ensemble de symboles de fonction. Pour un symbole de fonction $f^{\sigma_1\sigma_2\ldots\sigma_n\sigma}$, $n \geq 0$ est son arité et $\sigma_1\sigma_2\ldots\sigma_n\sigma$ est sa signature où $\sigma_1\sigma_2\ldots\sigma_n \in \Sigma^s$ sont les

types des arguments et σ est le type du résultat. Nous ne préciserons pas la signature du symbole de fonctions lorsque c'est inutile.

Posons $Var = x, y, z, \ldots$ un ensemble dénombrable de variables du premier ordre. Chaque variable $x^\sigma \in Var$ est associée à un type $\sigma \in \Sigma^s$.

Définition 1.1.1 *(Terme). La syntaxe des termes est définie récursivement par la grammaire suivante :*

$$\begin{array}{rcll} t & ::= & x & \textit{(variable)} \\ & | & f(t_1, \ldots, t_n) & \textit{(application)} \end{array}$$

Définition 1.1.2 *(Terme typé). Un terme t est de type $\sigma \in \Sigma^s$ sur une signature Σ (aussi appelé Σ-terme), noté t^σ, si et seulement si l'une des conditions suivantes est respectée :*

$$\begin{array}{rcll} t & = & x & \textit{et } x^\sigma \in Var \\ t & = & f(t_1, \ldots, t_n) & t_1^{\sigma_1}, \ldots, t_n^{\sigma_n} \textit{ et } f^{\sigma_1 \ldots \sigma_n \sigma} \in \Sigma^f \end{array}$$

Tout symbole de variable de type σ est un terme de type σ. Si $t_1, \ldots, t_n$ sont des termes de type $\sigma_1, \ldots, \sigma_n$ et $f^{\sigma_1 \ldots \sigma_n \sigma} \in \Sigma^f$ alors $f(t_1, \ldots, t_n)$ est un terme de type σ.

Nous noterons $\tau_\Sigma\ (\mathbf{x})$ l'ensemble de tous les termes construit utilisant les symboles de fonctions dans Σ^f et les variables dans l'ensemble $\mathbf{x}$. Nous écrivons τ_Σ pour l'ensemble $\tau_\Sigma\ (\varnothing)$ de termes ne contenant aucune variable.

Définition 1.1.3 *(Formule du premier ordre). Une formule du premier ordre sur une signature Σ (aussi appelée Σ-formule) est définie récursivement par la grammaire :*

$$\begin{array}{rcllll} \phi^{FOL} & ::= & \top & & & \textit{(vrai)} \\ & | & \bot & & & \textit{(faux)} \\ & | & t, & t^{Bool} & & \textit{(termes booléens)} \\ & | & t_1 \approx t_2, & t_1^\sigma, t_2^\sigma & & \textit{(égalité)} \\ & | & \neg\phi^{FOL}, & & & \textit{(négation)} \\ & | & \phi_1^{FOL} \wedge \phi_2^{FOL} & & & \textit{(conjonction)} \\ & | & \phi_1^{FOL} \vee \phi_2^{FOL} & & & \textit{(disjonction)} \\ & | & \exists x.\phi^{FOL}, & x \in FV(\phi^{FOL}) & & \textit{(quantificateur existentiel)} \\ & | & \forall x.\phi^{FOL}, & x \in FV(\phi^{FOL}) & & \textit{(quantificateur universel)} \end{array}$$

Les constantes $\top$ et $\bot$, des termes booléens, et l'égalité entre deux termes de même type sont des formules du premier ordre. La négation, conjonction, disjonction, les quantificateurs existentiels et universels de la logique du premier ordre sont aussi des formules de la logique du premier ordre.

Pour une formule ϕ, nous noterons $FV(\phi)$ l'ensemble des variables n'apparaissant pas dans la portée d'un quantificateur de ϕ.

Définition 1.1.4 *(Substitution). Soit un ensemble de variables $\boldsymbol{x}$ et $\boldsymbol{y}$, une substitution $\theta : \boldsymbol{x} \to \tau_\Sigma(\boldsymbol{y})$ associe chaque variable dans $\boldsymbol{x}$ à un terme dans $\tau_\Sigma(\boldsymbol{y})$.*

1.2 Sémantique

La sémantique des formules du premier ordre est définie en utilisant l'interprétation des types et des fonctions dans une signature Σ et l'évaluation des variables dans Var.

Définition 1.2.1 *(Interprétation). Une interprétation I pour Σ associe chaque symbole de type $\sigma \in \Sigma^s$ à un ensemble non-vide σ^I, chaque symbole de fonction $f^{\sigma_1,\ldots,\sigma_n\sigma} \in \Sigma^f$ avec $n > 0$ à une fonction totale $f^I : \sigma_1^I \times \ldots \times \sigma_n^I \to \sigma^I$.*

Posons I une interprétation, $f^{\sigma_1 \cdots \sigma_n \sigma}$ un symbole de fonction et $\alpha^{\sigma_1^I,\ldots,\sigma_n^I \sigma^I}$ une fonction. Nous écrivons $I[f \leftarrow \alpha]$ pour une interprétation telle que :
(i) $I[f \leftarrow \alpha](\sigma) = I(\sigma)$, (ii) $I[f \leftarrow \alpha](f) = \alpha$, et (iii) $\nu[f \leftarrow \alpha](g) = g^I$ pour tout $g \in \Sigma^f$ avec $g \neq f$.

Définition 1.2.2 *(Évaluation). Soit une interprétation I, une évaluation ν associe chaque variable $x^\sigma \in Var$ à un élément de σ^I.*

Notons V_I l'ensemble des évaluations possibles sous I. Posons I une interprétation, $\nu \in V_I$ une évaluation, $x^\sigma \in Var$ une variable et $\alpha \in \sigma^I$ une valeur. Nous noterons $\nu[x \leftarrow \alpha]$ pour une évaluation telle que : (i) $\nu[x \leftarrow \alpha](x) = \alpha$, et (ii) $\nu[x \leftarrow \alpha](y) = \nu(y)$ pour tout $y \in Var$ avec $y \neq x$.

Définition 1.2.3 *(Interprétation d'un terme). L'interprétation de t relative à I et ν est obtenue en replaçant chaque symbole de fonction f apparaissant dans t par son interprétation f^I et chaque variable x apparaissant dans t par son évaluation $\nu(x)$.*

Maintenant que l'on sait comment interpréter les termes avec une évaluation donnée, nous pouvons étendre la notion d'interprétation aux formules du premier ordre.

Définition 1.2.4 *(Sémantique d'une formule du premier ordre). Soit une interprétation I et une évaluation $\nu \in V_I$, nous écrivons $I, \nu \vDash \phi$ si la formule du premier ordre ϕ est interprétée par vrai sous I et ν. La relation est définie inductivement sur la structure de ϕ :*

I, ν	$\vDash$	$\top$	*toujours vrai*
I, ν	$\vDash$	$\bot$	*jamais vrai*
I, ν	$\vDash$	t	*ssi* $t_\nu^I = \top, t^{Bool}$
I, ν	$\vDash$	$t_1 \approx t_2,$	*ssi* $t_{1_\nu}^I = t_{2_\nu}^I, t_1^\sigma = t_2^\sigma$
I, ν	$\vDash$	$\neg\phi$	*ssi* $I, \nu \vDash \phi$ *est faux*
I, ν	$\vDash$	$\phi_1 \wedge \phi_2$	*ssi* $I, \nu \vDash \phi_1$ *et* $I, \nu \vDash \phi_2$
I, ν	$\vDash$	$\phi_1 \vee \phi_2$	*ssi* $I, \nu \vDash \phi_1$ *ou* $I, \nu \vDash \phi_2$
I, ν	$\vDash$	$\exists x.\phi$	*ssi* $I, \nu[x \leftarrow \alpha] \vDash \phi, x^\sigma \in FV(\phi)$, *pour certains* $\alpha \in \sigma^I$
I, ν	$\vDash$	$\forall x.\phi$	*ssi* $I, \nu[x \leftarrow \alpha] \vDash \phi, x^\sigma \in FV(\phi)$, *pour tout* $\alpha \in \sigma^I$

En utilisant leurs sémantiques, nous pouvons définir les notions de satisfaisabilité et d'implication des formules du premier ordre sous une interprétation I.

Définition 1.2.5 *(Satisfaisable et valide). Une formule du premier ordre ϕ est satisfaisable dans l'interprétation I s'il existe une évaluation ν telle que $I, \nu \vDash \phi$. Sinon, la formule est insatisfaisable. Si $I, \nu \vDash \phi$ pour tout ν, alors ϕ est valide et $\neg\phi$ est insatisfaisable.*

Définition 1.2.6 *(Implication et équivalence). Soit deux formules du premier ordre ϕ_1 et ϕ_2, nous écrivons $\phi_1 \vDash^I \phi_2$ et disons que ϕ_1 implique ϕ_2 dans l'interprétation de I ssi $I, \nu \vDash \phi_1$ implique $I, \nu \vDash \phi_2$ pour toute évaluation ν. Nous appelons ϕ_1 et ϕ_2 équivalentes si $\phi_1 \vDash^I \phi_2$ et $\phi_2 \vDash^I \phi_1$.*

Nous encapsulerons toutes les notions se rapportant à la logique du premier ordre dans les théories du premier ordre.

Définition 1.2.7 *(Théorie du premier ordre). Une théorie du premier ordre est une paire $T = (\Sigma, M)$ telle que Σ est une signature et M est un ensemble non vide de paire (I, ν), appelé un modèle de T, où I est une interprétation et $\nu \in V_I$ est une évaluation.*

Une théorie du premier ordre est cohérente s'il existe des formules non-démontrables. Nous supposons que toutes les théories, dont nous parlerons, sont cohérentes. Soit une théorie du premier ordre $T = (\Sigma, M)$, tout Σ-terme t est aussi appelé un T-terme et tout Σ-formule ϕ est aussi appelé une T-formule. Une paire $(I, \nu) \in M$ telle que $I, \nu \vDash \phi$ est un T-modèle de ϕ. Nous noterons l'ensemble des T-modèles de ϕ par $[\![\phi]\!]_T = \{(I, \nu) \in M | I, \nu \vDash \phi\}$.

Définition 1.2.8 *(T-satisfaisable et T-valide). Soit $T = (\Sigma, M)$ une théorie du premier ordre, une T-formule ϕ est T-satisfaisable si $[\![\phi]\!]_T \neq \varnothing$ sinon T-insatisfaisable. Si ϕ est T-satisfaisable si et seulement si ψ est T-satisfaisable alors ϕ et ψ sont équisatisfaisables dans T. Si $[\![\phi]\!]_T = M$ alors ϕ est T-valide et $\neg\phi$ est T-insatisfaisable.*

Définition 1.2.9 *(T-implication et T-équivalence). Soit une théorie du premier ordre $T = (\Sigma, M)$ et deux T-formules ϕ et ψ, nous écrivons $\phi \vDash^T \psi$ et posons que ϕ T-implique ψ si et seulement si $[\![\phi]\!]_T \subseteq [\![\psi]\!]_T$. Nous appelons ϕ et ψ T-équivalent si $\phi \vDash^T \psi$ et $\psi \vDash^T \phi$.*

Une théorie peut être construite à partir d'axiomes. Le modèle d'une théorie construite à partir d'axiomes est l'intersection des modèles des axiomes.

1.3 Arithmétique de Presburger

L'arithmétique de Presburger est une théorie du premier ordre. Elle est munie de la signature : $\Sigma^s = \{Nat\}$ et $\Sigma^f = \{0^{Nat}, 1^{Nat}, +^{NatNatNat}\}$ et est accompagnée des axiomes suivants :

1. $\forall x.\ \neg(0 \approx x + 1)$
2. $\forall x, y.\ x + 1 \approx y + 1 \rightarrow x \approx y$
3. $\forall x.\ x + 0 \approx x$
4. $\forall x, y.\ x + (y + 1) \approx (x + y) + 1$
5. Soit $P(x)$ une formule du premier ordre dans le langage de l'arithmétique de Presburger avec la variable libre x,
   $(P(0) \wedge \forall x.(P(x) \rightarrow P(x + 1))) \rightarrow \forall y.P(y)$

1.4 Théorie des tableaux

Un tableau logique est une expression qui associe un élément à chaque indice. Nous notons le type des indices σ_I (aussi appelé domaine) et le type des éléments σ_E (aussi appelé co-domaine). Le type du tableau est noté σ_A.

La théorie des tableaux est muni de la signature : $\Sigma^s = \{\sigma_I, \sigma_E, \sigma_A\}$ et $\Sigma^f = \{select^{\sigma_A\sigma_I\sigma_E}, store^{\sigma_A\sigma_I\sigma_E\sigma_A}, constant\_array^{\sigma_E\sigma_A}, array\_ext^{\sigma_A\sigma_A\sigma_I}\}$. Nous pouvons donc remarquer qu'il y a une signature différente pour chaque paire de types (σ_I, σ_E) et donc une théorie différente pour chaque paire de théories sur les éléments et les indices.

Posons $a \in \sigma_A$ un tableau, il y a 4 fonctions élémentaires à la théorie :

1. *Select* : Soit $i \in \sigma_I$, *select(a,i)* renvoie la valeur de l'élément à l'indice i.
2. *Store* : Soit $i \in \sigma_I$ et $e \in \sigma_E$, *store(a,i,e)* renvoie un nouveau tableau avec les mêmes valeurs que le tableau a et la valeur e à l'indice i.
3. *Constant_array* : Soit $e \in \sigma_E$, $constant\_array(e)$ renvoie un tableau tel que $\forall i \in \sigma_I$ $select(constant\_array(e), i) = e$.
4. *Extensionalité* : Soit $a, b \in \sigma_A$, $array\_ext(a, b)$ renvoie un indice i tel que $select(a, i) = select(b, i) \rightarrow a = b$

L'axiome principal utilisé pour définir les fonctions *select* et *store* est l'axiome **read-over-write** :
$\forall a \in \sigma_A.\ \forall e \in \sigma_E.\ \forall i, j \in \sigma_I.select(store(a, i, e), i) = e \wedge$
$i \neq j \rightarrow select(store(a, i, e), j) = select(a, j).$

Nous appelons les théories utilisées pour raisonner sur les indices et les éléments, *théorie des indices* et *théorie des éléments*.

La logique des indices peut autoriser les quantificateurs pour modéliser des propriétés telles que "Il existe un élément du tableau qui est égal à zéro" ou "tout élément du tableau est supérieur ou égal à zéro". Un exemple de théorie adaptée est l'arithmétique de Presburger.

Définition 1.4.1 *(Tableau logique). La syntaxe d'un tableau logique est définie par extension des règles de syntaxe de la logique des indices et la logique des éléments. Nous notons $form_I/term_I$ et $form_E/term_E$ les formules/termes de, respectivement, la logique des indices et la logique des éléments :*

$form ::= form_I \mid form_E \mid \neg form \mid form \wedge form \mid \forall$ *array-identifier*$.form$
$term_A ::=$ *array-identifier* $\mid store(term_A, term_I, term_E) \mid constant\_array(term_E)$
$term_E ::= select(term_A, term_I)$
$term_I ::= array\_ext(term_A, term_A)$

Nous pouvons observer que la grammaire n'autorise pas l'égalité entre les tableaux. L'égalité se fait avec la fonction d'extensionalité, en supposant que l'égalité est autorisée entre les éléments du tableaux.

1.5 Procédures de décision

Définition 1.5.1 *(Problème de décision). Le problème de décision pour un formule ϕ est de déterminer si ϕ est valide.*

Définition 1.5.2 *(Correction d'une procédure). Une procédure pour un problème de décision est correcte si quand elle retourne Valide, alors la formule en entrée est valide.*

Définition 1.5.3 *(Complétude d'une procédure). Une procédure pour un problème de décision est complète si elle retourne valide pour toutes les formules valides.*

Définition 1.5.4 *(Procédure de décision). Soit T une théorie, une procédure est appelée procédure de décision pour T si elle est complète et correcte pour toutes les formules de T.*

Un aspect important, pour prouver des programmes avec l'aide de la logique, est d'utiliser des théories décidables.

Définition 1.5.5 *(Théorie décidable). Soit T une théorie logique du premier ordre, T est décidable si et seulement s'il existe une procédure de décision.*

Par exemple, l'arithmétique de Presburger est décidable. Mais en général, une théorie n'est pas décidable. Pour palier à ce problème, les théories doivent être syntaxiquement restreintes. On parle alors de fragment de la logique.

1.5.1 Décidabilité de la théorie des tableaux

La théorie des tableaux actuelle n'est pas décidable, même si la combinaison entre la théorie des indices et la théorie des éléments est décidable. Il faut limiter la syntaxe de la théorie pour la rendre décidable.

Tableaux comme des fonctions non-interprétées Considérons le fragment de la logique qui n'autorise pas les quantificateurs sur les tableaux. Un chemin possible pour rendre la théorie des tableaux décidable est de réduire chaque formule en une combinaison d'autres théories qui traîte les tableaux comme des fonctions non-interprétées.

Les fonctions non-interprétées sont utilisées pour abstraire, ou généraliser, des théorèmes. Contrairement aux autres symboles de fonctions, elles ne devraient pas être interprétées comme une partie de la formule. Dans la formule suivante, par exemple, F et G sont non-interprétées alors que le symbole de fonction binaire "+" est interprété comme la fonction addition :

$F(x) = F(G(y)) \vee x + 1 = y$

Remplacer une fonction avec une fonction non-interprétée est une technique courante pour simplifier le raisonnement. En contre-partie, une formule valide peut devenir invalide. Pour que ce remplacement soit correct, il faut que les fonctions non-interprétées suivent un axiome. L'axiome de la théorie des fonctions non interprétées est que si on leur donne une entrée identique, la sortie sera identique. C'est ce qu'on appelle une **cohérence fonctionnelle** ou encore **congruence fonctionnelle**.

Quand le tableau est une fonction non-interprétée, l'indice devient le seul argument de la fonction. La fonction *store* peut être manipulé en remplaçant chaque expression de la forme $store(a, i, e)$ par une nouvelle variable $a' \in \sigma_A$ et en ajoutant deux formules qui correspondent directement à l'axiome **read-over-write** :

1. $select(a', i) = e$ pour la valeur qui est écrite,
2. $\forall j \neq i. select(a', j) = select(a, j)$ pour les valeurs qui sont inchangées.

On l'appelle la régle d'écriture. Elle est une transformation équivalente dans les formules des tableaux logiques.

Maintenant, la logique des tableaux peut être réduite à la combinaison entre la logique des indices et les fonctions non-interprétées. La combinaison de l'arithmétique de Presburger et des fonctions non-interprétées est indécidable. Or, dans un programme, les indices d'un tableau sont des entiers. Il est donc intéressant de connaître la restriction nécessaire de l'ensemble des formules.

Définissons maintenant une classe réduite des formules de la logique des tableaux pour obtenir la décidabilité. Nous considérons les formules qui sont des combinaisons booléennes de **propriétés de tableau**.

Définition 1.5.6 *(Propriété de tableau). Une formule de la logique des tableaux est appelée* ***propriété de tableau*** *si et seulement si elle est de la forme :*

$\forall i_1, \ldots, i_k \in \sigma_I . \phi_I(i_1, \ldots, i_k) \Longrightarrow \phi_V(i_1, \ldots, i_k)$, *et satisfait les conditions suivantes :*

1. *La fonction booléenne* ϕ_I*, appelée* ***index guard****, doit suivre la grammaire suivante :*
   $\phi_I : \phi_I \wedge \phi_I \mid \phi_I \vee \phi_I \mid iterm \leq iterm \mid iterm = iterm$
   $iterm : i_1 \mid \ldots \mid i_k \mid term$
   $term$: *integer-constant* $\mid$ *integer-constant* $\times$ *index-identifier* $\mid$ $term{+}term$
   Le "index-identifier" utilisé dans "term" ne doit pas être l'un des $i_1 \ldots i_k$.
2. *Les indices* $i_1 \ldots i_k$ *ne peuvent être utilisé que dans une expression de la forme* $select(a, i_j)$.

La fonction booléennes ϕ_V *est appelée* ***contrainte de valeur*** *et est restreinte par la deuxième règle.*

Il existe un algorithme, que nous ne préciserons pas, acceptant le fragment des propriétés de tableau et le réduisant à une formule équisatisfaisable qui utilise la théorie des éléments et la théorie des indices.

Nous supposons que les opérations suivantes sont définies dans les théories des indices et des éléments, et que nous avons une procédure de décision pour la combinaison de théories :

— Pour la théorie des indices, nous supposons que l'arithmétique linéaire sur les indices est permise (celle de Presburger par exemple).
— Pour la théorie des éléments, nous supposons seulement que l'égalité entre deux éléments est permise.

1.6 Combinaison de théories décidables

Avant de parler de combinaison de théories décidables, nous allons définir quelques notions.

Définition 1.6.1 *(Combinaison de théories). Soit* $T_1 = (\Sigma_1, M_1)$ *et* $T_2 = (\Sigma_2, M_2)$ *deux théories, la combinaison de théories* $T_1 + T_2$ *est une* $(\Sigma_1 \cup \Sigma_2)$*-théorie définie par les ensembles* $(M_1 \cup M_2)$.

Définition 1.6.2 *(Problème de combinaison de théorie). Soit* ϕ *une* $(\Sigma_1 \cup \Sigma_2)$*-formule. Le problème de combinaison de théorie est de décider si* ϕ *est* $T_1 + T_2$*-valide.*

La combinaison des théories est, en général, indécidable même si les théories sont décidables. Ainsi, pour rendre les combinaisons décidables, il est nécessaire de les restreindre. La procédure de Nelson-Oppen est une procédure de décision pour la combinaison de théories répondant à des restrictions précises. Nous n'allons pas aborder son fonctionnement mais seulement ses restrictions.

Définition 1.6.3 *Pour que la procédure de Nelson-Oppen soit applicable, les théories* $T_1, \ldots, T_n$ *doivent suivre les restrictions suivantes :*

1. $T_1, \ldots, T_n$ *sont des théories du premier ordre, sans quantificateur et avec l'égalité.*
2. *Il y a une procédure de décision pour chaque théorie* $T_1, \ldots, T_n$.
3. *Les signatures sont disjointes, c'est-à-dire,* $\forall 1 \leq i < j \leq n$, $\Sigma_i \cap \Sigma_j = \varnothing$.
4. $T_1, \ldots, T_n$ *sont des théories qui sont interprétées sur un domaine infini.*

Il existe des extensions de la procédure de Nelson-Oppen pouvant surmonter chacune des restrictions. Les techniques de combinaison de décision sont essentielles à la construction des SMT solveurs.

1.7 SMT-solveur

Un SMT-solver permet de tester la validité des formules logiques à l'aide de procédures de décision. Nous allons discuter d'une méthode générale sur laquelle se basent la plupart des SMT-solvers.

Soit $T = (\Sigma, M)$ une théorie du premier ordre sans quantificateur et DP_T une procédure pour le fragment des conjonctions de T (DP_T peut décider une conjonction de T-litéraux). La méthode combine DP_T avec un SAT-solveur (solveur de contraintes de la logique propositionnelle) de diverses manières afin de construire une procédure de décision pour T. Cette approche a en pratique de gros avantages, car elle est très modulaire et très efficace. Les deux principaux outils de cette méthode travaillent en collaboration : le SAT-solver choisit la valeur des littéraux afin de satisfaire la structure booléenne de la formule et DP_T vérifie que le choix est T-satisfaisable.

Soit un T-littéral l, nous lui associons une variable booléenne unique $e(l)$, que nous appelons l'**encodeur** booléen de ce littéral. Étendons l'idée aux formules, soit une T-formule ϕ, $e(\phi)$ correspond à la formule booléenne où chaque littéral a été substitué par son encodeur booléen. $e(\phi)$ est appelée le **squelette propositionnel** de ϕ. En utilisant ces notations, nous allons étudier un aperçu de cette méthode.

Soit T la théorie de l'égalité, une formule $\phi := x = y \wedge ((y = z \wedge x \neq z) \vee x = z)$.
Nous transformons cette formule en son squelette propositionnel,

$e(\phi) := e(x = y) \wedge (e(y = z) \wedge e(x \neq z)) \vee e(x = z)$.

Soit B une formule booléenne, initialement $e(\phi)$. La seconde étape consiste à passer B au SAT-solver. Supposons que le SAT-solver retourne l'évaluation satisfaisante

$\nu := \{e(x = y) \mapsto vrai,\ e(y = z) \mapsto vrai,\ e(x \neq z) \mapsto vrai,\ e(x = z) \mapsto faux\}$.

La procédure de décision DP_T doit maintenant décider si la conjonction des littéraux correspondant à cette évaluation est satisfaisable. Nous notons cette conjonction $Th(\alpha)$, $Th(\alpha) := x = y \wedge y = z \wedge x \neq z \wedge \neg(x = z)$.

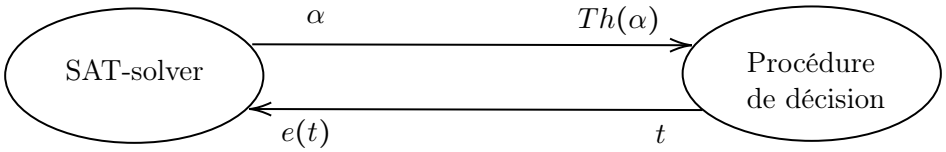
Cette formule n'est pas satisfaisable donc sa négation est valide. Ainsi, nous ajoutons $e(\neg Th(\alpha))$ à B, l'encodeur booléen de cette tautologie : $e(\neg Th(\alpha)) := (\neg e(x = y) \vee \neg e(y = z) \vee \neg e(x \neq z) \vee e(x = z))$.

Cette formule est contradictoire à l'évaluation précédente et donc empêche le SAT-solver de se répéter. Après avoir ajouté la nouvelle clause, le SAT-solver est de nouveau appelé et suggère une autre affectation : $\nu := \{e(x = y) \mapsto vrai,\ e(y = z) \mapsto vrai,\ e(x \neq z) \mapsto faux,\ e(x = z) \mapsto vrai\}$.

La T-formule suivante,

$Th(\alpha') := x = y \wedge y = z \wedge x = z \wedge \neg(x \neq z)$

est satisfaisable, ce qui prouve que ϕ, la formule originelle, est satisfaisable.



2 Vérification déductive avec VeriFast

La vérification déductive se base sur l'annotation des programmes. Chaque fonction est vérifiée séparément, en utilisant les contrats des autres fonctions et non leurs implémentations.

Le contrat d'une fonction consiste en une précondition et une postcondition. La précondition spécifie les attentes, concernant le tas et la pile, au début de la fonction. La postcondition spécifie les garanties offertes, concernant la pile et le tas, à la sortie de la fonction.

2.1 Programme

VeriFast permet la vérification des programmes C et Java. La sémantique d'un programme peut être donnée par l'ensemble de ses exécutions. Une exécution est définie par une séquence fini ou infini de configurations, chacune liée à l'autre par une instruction du programme. Une configuration est un état et une continuation. Un état est une pile et un tas. La pile spécifie la valeur des variables du programme. Le tas spécifie la valeur des cellules de mémoire. La continuation précise ce qu'il va se passer ensuite. Une continuation est soit *une continuation d'instructions*, soit une *fin de fonction*, soit une *fin de programme*.
Dans un programme, toutes les variables précédement définies doivent toujours avoir une valeur dans la pile et chaque espace mémoire doit être réprésenté dans le tas. Pour représenter le tas, VeriFast utilise la logique de séparation.

2.2 Logique de séparation

2.2.1 Syntaxe

La syntaxe de la logique de séparation (SL)[1] est construite sur la syntaxe de la logique du premier ordre, définie à la subsection 1.1.1. Nous considérons une théorie du premier ordre $T = (\Sigma, M)$ telle que Σ^s contient les types Loc et $Data$ et Σ^f contient une constante nil^{Loc}.

Définition 2.2.1 *(Formule de la logique de séparation). Une formule SL s'écrit avec les mêmes règles qu'une formule du premier ordre en ajoutant ces règles :*

ϕ^{SL}	::=	$\ldots$		
	$\mid$	emp		*(tas vide)*
	$\mid$	$t \mapsto u,$	t^{Loc}, u^{Data}	*(tas singleton)*
	$\mid$	$\phi_1^{SL} * \phi_2^{SL},$		*(conjonction de séparation)*

Les deux nouveaux atomes décrivant le tas vide et le tas singleton sont des formules $\mathrm{SL}(T)$.

Si une $\mathrm{SL}(T)$-formule contient au moins une fois emp, $\mapsto$ ou $*$ alors c'est une formule *spatiale* sinon c'est une formule *pure*.

La plupart des définitions de structure de données, telles que les listes ou les arbres, utilisent un fragment restreint sans quantificateur appelé *tas symbolique*.

Définition 2.2.2 *(Formule de tas symbolique). Une formule de tas symbolique est une conjonction $\Pi \wedge \Theta$ entre une partie pure (Π) et une partie spatiale (Θ), définie ainsi :*

Π	::=	$\top$		*(vrai)*
	$\mid$	$\bot$		*(faux)*
	$\mid$	$t,$	t^{Bool}	*(terme booléen)*
	$\mid$	$t_1 \approx t_2,$	t_1^{Loc}, t_2^{Loc}	*(egalité)*
	$\mid$	$\neg(t_1 \approx t_2),$	t_1^{Loc}, t_2^{Loc}	*(inegalité)*
	$\mid$	$\Pi_1 \wedge \Pi_2,$		*(conjonction)*
Θ	::=	emp		*(tas vide)*
	$\mid$	$t \mapsto u,$	t^{Loc}, u^{Data}	*(tas singleton)*
	$\mid$	$\Theta_1 * \Theta_2,$		*(conjonction de séparation)*

2.2.2 Sémantique

Définition 2.2.3 *(Tas).Soit une interprétation I, le tas est une partie finie associant une location avec une donnée, $h : Loc^I \rightarrow_{fin} Data^I$. Nous utiliserons Tas^I pour représenter l'ensemble de tous les tas sous l'interprétation de I.*

Deux tas h_1 et h_2 sont disjoints si $\text{dom}(h_1) \cap \text{dom}(h_2) = \varnothing$. L'union disjointe sera symbolisée par $h_1 \uplus h_2$ qui est indéfinie si h_1 et h_2 ne sont pas disjoints. Nous écrivons $\biguplus H$ pour l'union disjointe des tas dans l'ensemble $H \subseteq Tas$.

Définition 2.2.4 *(Sémantique des formules de la logique de séparation). Soit une interprétation I, une évaluation $\nu \in V_I$ et un tas $h \in Tas$, nous écrivons $I, \nu, h \vDash^{SL} \phi$ si une formule SL ϕ est interprétée à vrai sous I, ν et h. Cette relation est définie par induction dans la structure de ϕ :*

$I, \nu, h \vDash^{SL} emp$ *ssi* $dom(h) = \varnothing$

$I, \nu, h \vDash^{SL} t \mapsto u$ *ssi* $t^I_\nu \neq nil^I$ *et* $h = \{(t^I_\nu, u^I_\nu)\}, t^{Loc}, u^{Data}$

$I, \nu, h \vDash^{SL} \phi_1 * \phi_2$ *ssi* $\exists h_1 \exists h_2 . h = h_1 \uplus h_2$ *et* $I, \nu, h_1 \vDash^{SL} \phi_1$ *et* $I, \nu, h_2 \vDash^{SL} \phi_2$

Un triplet (I, ν, h) tel que $(I, \nu) \in M$ et $I, \nu, h \vDash^{SL} \phi$ est un SL(T)-modèle pour la SL(T)-formule ϕ. Nous noterons l'ensemble des SL(T)-modèles de ϕ par $[\![\phi]\!]_{SL(T)} = \{(I, \nu, h) | (I, \nu) \in M, h \in Tas^I \text{ et } I, \nu, h \vDash^{SL} \phi\}$.

La logique de séparation ajoute la règle du cadre à la logique de Hoare :

$$\frac{\{\text{P}\}\text{C}\{\text{Q}\}}{\{\text{P} * \text{R}\}\text{C}\{\text{Q} * \text{R}\}}$$

, où R est une formule qui ne concerne pas les variables utilisées par C.

Informellement, la règle dit que la preuve de C peut ignorer la partie du tas ne conçernant pas C.

2.3 Annotations et assertions

En VeriFast, les annotations sont exprimées sous forme d'*assertions*. Les assertions sont des formules de la logique de séparation.

Exemple 1 *Exemple : Soit une fonction r,*

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

La précondition de l'*exemple* assure que l'égalité entre a et b est respectée. La postcondition de *exemple* assure que la valeur retournée par le programme est 0, en supposant que la précondition est respectée.

L'une des notions de base de la vérification déductive est les prédicats. En effet, le contrat des fonctions doit pouvoir être exprimé pour les tas de taille quelconque, de manière symbolique. Un prédicat est simplement une formule. La définition d'un prédicat est de la forme

$$\textbf{predicate } p(x_1, \ldots, x_n) \coloneqq a$$

où p est le nom du prédicat, $x_1, \ldots, x_n$ sont les noms des variables et a une formule. $x_1, \ldots, x_n$ sont appelées les paramètres de p et a le corps de p.

En utilisant les prédicats, un tas symbolique peut être dérivé depuis un tas concret. Un tas symbolique ne contient pas de cellules de mémoire mais des instances de prédicats. Un tas symbolique est obtenu depuis un tas concret en fermant des instances de prédicats. La fermeture d'une instance de prédicat, *close*, remplace la partie du tas décrite par le corps de l'instance du prédicat par l'instance du prédicat elle-même. Inversement, l'ouverture d'une instance de prédicat, *open*, remplace l'occurence du prédicat avec un fragment du tas correspondant au corps du prédicat. Nous disons qu'un tas symbolique fait abstraction d'un tas concret si le tas concret peut être obtenu à travers un nombre fini d'opérations *open*.

Les instructions *open* et *close* sont les *instructions fantômes*. Nous appelons l'effacement d'un programme annoté, le fait, de supprimer toutes les annotations d'un programme.

Exemple 2

```
predicate equal(x,y) := x=y

int exemple'(int a, int b)
//@ requires equal(a,b);
//@ ensures equal(result,0);
{
//@ open equal(a,b);
return a-b;
//@ close equal(result,0);
}
```

Nous allons aborder les notions de consommation et de production d'assertion.

2.3.1 Sémantique des assertions

Les assertions sont interprétées par rapport à un tas symbolique. Appelons points-to assertion, l'assertion représentant la formule *tas singleton* (voir SL). Nous définissons l'ensemble des éléments du tas symbolique comme l'union des éléments points-to et des instances de prédicat :

$AbsHeapElems = \{l \mapsto v | l \in Addresses, v \in \mathbb{Z}\} \cup \{p(v_1, \ldots, v_n) | v_1, \ldots, v_n \in \mathbb{Z}\}$

Nous définissons un état symbolique comme un triplet d'une pile, d'un tas symbolique et d'un chemin de conditions.

2.4 Logique de spécification en VeriFast

Une notion, que nous n'avons pas abordée, et qui est primordiale dans la logique de spécification de VeriFast, est l'invariant de boucle. L'invariant de boucle est une formule qui est vraie avant, pendant et après la boucle. Il permet donc d'assurer que le code est correct pendant et après la boucle. Nous allons maintenant voir les annotations plus complexes, dont nous n'avons pas encore parlées, dans VeriFast.

Types et predicats inductifs Pour permettre des spécifications riches, VeriFast supporte les types inductifs. Les types inductifs permettent à VeriFast de représenter des structures inductives. Souvent, chaque élément de la structure doit respecter certaines propriétés. Par exemple, dans une pile, chaque élément doit avoir un espace alloué. Pour représenter ces conditions, on utilise des prédicats inductifs.
Dans l'exemple suivant, le type *ints* représente la liste des éléments dans la pile. Le prédicat *nodes* indique que de l'espace alloué pour chaque élément et que les éléments de *ints* correspondent bien aux éléments de la pile.

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

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

predicate nodes(struct node *node, ints num) =
        node == 0 ?
        num == ints_nil :
        num == ints_cons(?number,?num2)
        &*& node -> next |-> ?n &*& node->value |-> number
        &*& malloc_block_node(node) &*& nodes(n,num2);
```

La notation &*& permet de simplifier * qui joue le rôle de la conjonction ($\wedge$).

Fixpoints En plus des types inductifs, VeriFast permet aussi les fixpoints. Un fixpoint est une fonction avec au moins un argument de type inductif. Le corps du fixpoint doit être un **switch** sur l'un de ses arguments inductifs. Pour s'assurer que les fixpoints sont bien définis, le corps d'un fixpoint f peut utiliser un autre fixpoint g seulement si g apparaît avant f dans le programme. Si f est récursif,l'une des composantes d'un argument inductif de f doit être utilisée à la place de cet argument. Par exemple, le fixpoint ci-dessous indique la longueur d'un élément de type *ints*.

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

Lemmes VeriFast a en réalité une instruction supplémentaire pour le code fantôme, en plus des *open* et *close*. En effet, il est possible d'appeler des lemmes. Comme un fonction C, un lemme prend des arguments, à une précondition et une postcondition. Le corps d'un lemme représente une preuve. Cette preuve du lemme démontre que la précondition implique la postcondition pour toutes les valeurs possibles des arguments. L'appel d'un lemme correspond donc à l'application d'un théorème.

Pour que la preuve soit valide, le corps doit satisfaire certaines restrictions. Dans un premier temps, le corps ne doit pas affecter l'état concret, plus précisement, il ne doit pas modifier de

champs ou appeler des fonctions C. Deuxièmement, l'exécution du corps doit terminer. Pour assurer la terminaison, les boucles et les appels de lemmes sont limités.
Les boucles doivent avoir un variant. Un variant de boucle doit strictement décroitre à chaque tour de boucle et doit être défini dans un ensemble bien fondé.
Un lemme x peut appeler un lemme y s'il est défini avant dans le programme ou si l'appel est récursif. Si l'appel est récursif, alors l'une des restrictions suivantes doit être respectée :

1. L'appel récursif réduit la taille du tas. Plus précisement, après la consommation des préconditions d'un appel récursif, il doit rester un champ dans le tas.
2. L'appel récursif réduit le type d'un argument inductif. Le corps du lemme est alors un **switch** sur un argument inductif et l'un des paramètres de l'argument est utilisé à la place de l'argument inductif dans l'appel récursif.
3. L'appel récursif réduit la profondeur de la première conjonction de la précondition. Plus précisement, le corps du lemme n'est pas un switch et la consommation du premier prédicat dans la précondition de l'appel récursif est obtenue en ouvrant le premier prédicat produit par la précondition du lemme.

2.5 Preuve de programme

Consommation d'une assertion Dans un état symbolique, consommer une assertion, revient à vérifier qu'il existe un fragment de tas qui correspond à l'assertion et de supprimer le fragment du tas. La consommation effectue un filtrage.

Une points-to assertion et un prédicat peuvent contenir des motifs de variables de la forme $?x$. Un motif de variable correspond à n'importe quelle valeur et associe la valeur à la variable.

Définissons la fonction *consume*, $consume(s, H, \Pi, a, Q)$ indique que la consommation de l'assertion a dans un état symbolique (s, H, Π) réussit, et que le prochain état satisfait la consommation de la postcondition Q.

Pour la consommation des points-to et des prédicats, la fonction *consume* a deux fonctions auxiliaires, *match_pattern* et *match_patterns*.
$match\_pattern(s, v, \pi)$ tente de faire correspondre la valeur v au motif π dans la pile s. Si le filtrage réussit, la fonction retourne un singleton contenant la pile résultante, c'est-à-dire, la pile s après avoir associée la valeur v à π. Sinon, elle retourne l'ensemble vide.
De la même façon, la fonction $match\_patterns(s, \bar{v}, \bar{\pi})$ tente de faire correspondre une liste de valeur $\bar{v}$ et une liste de motifs $\bar{\pi}$. Si le filtrage réussit, la fonction retourne un singleton contenant la pile résultante, c'est-à-dire, la pile s après avoir associée la liste de valeurs $\bar{v}$ à la liste de motifs $\bar{\pi}$. Sinon, elle retourne l'ensemble vide.

Nous avons les propriétés suivantes :

Lemme 1 *(Affaiblissement de la postcondition de consommation). Si la consommation d'une assertion par rapport à une certaine postcondition réussit, alors la consommation par rapport à une postcondition plus faible réussit aussi.*

$\forall s, H, a, Q, Q'\ consume(s, H, a, Q) \Rightarrow$
$(\forall s', H'\ Q(s', H') \Rightarrow Q'(s', H')) \Rightarrow consume(s, H, a, Q')$

Lemme 2 *(Encadrement de la consommation). Si la consommation d'une assertion réussit, alors elle réussit lorsque des éléments sont ajoutés au tas symbolique, que ces éléments sont toujours présents dans chaque post-état et la postcondition d'origine est conservée après avoir supprimés les éléments ajoutés.*

Nous disons qu'un état symbolique satisfait une assertion si la consommation de l'assertion dans l'état symbolique réussit et le tas symbolique résultant est vide.

Lemme 3 *(Consommation correcte). Si la consommation d'une assertion réussit dans un état symbolique, alors le tas symbolique peut être séparé en un fragment qui satisfait l'assertion et un fragment qui satisfait la postcondition.*

Production d'une assertion L'opérateur de production d'assertion est l'inverse de celui de consommation : produire une assertion dans un état symbolique donné étend le tas avec un fragment arbitraire de tas satisfaisant l'assertion.
Plus précisement, la fonction $produce(s, H, \Pi, a, Q)$ signifie que la postcondition Q est valable dans tous les états obtenus en étendant l'état symbolique (s, H, Π) avec un fragment de tas qui satisfait l'assertion a.

La fonction auxiliaire $produce\_pattern(s, \pi, Q)$ signifie que la postcondition Q est valable pour chaque pair (s', v) telle que la valeur v filtre avec le motif π dans la pile s et s' est s mise à jour avec le motif approprié.

Lemme 4 *(Affaiblissement de la postcondition de production). Si la production d'une assertion réussit, alors la production réussit aussi avec une postcondition plus faible.*

$\forall s, H, a, Q, Q'\ produce(s, H, a, Q) \Rightarrow$
$(\forall s', H'\ Q(s', H') \Rightarrow Q'(s', H')) \Rightarrow produce(s, H, a, Q')$

Lemme 5 *(Production Tas Inutile). Produire une assertion dans un état symbolique $E = (s, H, \Pi)$ est équivalent à produire la même assertion dans un tas vide et d'ajouter le tas H au tas post-état.*

Lemme 6 *(Production correcte). Si un état symbolique $E = (s, H, \Pi)$ satisfait une assertion et que la production de cette assertion réussit dans un tas vide, alors la postcondition de la production est valable dans le tas H.*

VeriFast vérifie si un programme satisfait sa spécification avec des exécutions symboliques et en utilisant un SMT-solver. Nous allons définir une exécution symbolique.

Les règles d'une **exécution symbolique** sont définies par *continuation passing style*. La continuation Q représente le travail qu'il reste à faire sur le chemin actuel. Par soucis de brièveté, seulement une partie des règles est présentée.

Produire une assertion points-to revient à ajouter un fragment avec un nouveau symbole au tas et à lier le symbole à la variable.

$$produce(s, H, \Pi, f \mapsto ?x, Q) \equiv$$
$$\text{let } \sigma = fresh(s, H, \Pi) \text{ in } Q(s[x := \sigma], h \uplus \{(f, \sigma)\}, \Pi)$$

Produire une assertion pure, telle que $e_1 = e_2$, revient à ajouter une formule équivalente au chemin de conditions.

$$produce(s, H, \Pi, e_1 = e_2, Q) \equiv \Pi \not\vdash_{SMT} \neg[\![e_1 = e_2]\!] \rightarrow Q(s, H, \Pi \cup \{[\![e_1 = e_2]\!]_s\})$$

Produire une assertion conditionnelle revient à créer deux branches dans l'exécution symbolique. L'une où l'assertion est ajoutée et une autre où sa négation est ajoutée. Nous noterons qu'une branche n'est pas atteignable si le chemin de conditions est incohérent.

$$produce(s, H, \Pi, e_1 = e_2?A_1 : A_2, Q) \equiv$$
$$(\Pi \not\vdash_{SMT} \neg[\![e_1 = e_2]\!] \rightarrow produce(s, H, \Pi \cup \{[\![e_1 = e_2]\!]_s\}, A_1, Q)) \wedge$$
$$(\Pi \not\vdash_{SMT} [\![e_1 = e_2]\!] \rightarrow produce(s, H, \Pi \cup \{\neg[\![e_1 = e_2]\!]_s\}, A_2, Q))$$

Produire deux ensembles d'assertions séparées par la conjonction de séparation revient à ajouter l'un puis l'autre.

$$produce(s, H, \Pi, A_1 * A_2, Q) \equiv$$
$$produce(s, H, \Pi, A_1, (\lambda h, s, \Pi.\ produce(s, H, \Pi, A_2, Q)))$$

Les règles de consommation sont inverses.

$consume(s, H, \Pi, f \mapsto ?x, Q) \equiv$
$\exists t_1, h'.h = \{(f, t_1)\} \uplus h' \wedge \Pi \vdash_{SMT} Q(h', s[x := t_1], \Pi)$

$consume(s, H, \Pi, e_1 = e_2, Q) \equiv \Pi \vdash_{SMT} [\![e_1 = e_2]\!]_s \wedge Q(s, H, \Pi)$

$consume(s, H, \Pi, e_1 = e_2?A_1 : A_2, Q) \equiv$
$(\Pi \not\vdash_{SMT} \neg[\![e_1 = e_2]\!] \rightarrow consume(s, H, \Pi \cup \{[\![e_1 = e_2]\!]_s\}, A_1, Q)) \wedge$
$(\Pi \not\vdash_{SMT} [\![e_1 = e_2]\!] \rightarrow consume(s, H, \Pi \cup \{\neg[\![e_1 = e_2]\!]_s\}, A_2, Q))$

$consume(s, H, \Pi, A_1 * A_2, Q) \equiv$
$consume(s, H, \Pi, A_1, (\lambda h, s, \Pi.\ consume(s, H, \Pi, A_2, Q)))$

À chaque appel de fonction, les préconditions doivent pouvoir être consommées et les postconditions produites. Une fonction est valide si après avoir produit les préconditions avec des valeurs arbitraires pour les paramètres des fonctions et exécuter symboliquement le corps de la fonction, la postcondition peut être consommée. La continuation finale assure l'absence de fuite de mémoire en vérifiant que la pile est vide après avoir consommée la postcondition. Un programme est valide si toutes les fonctions sont valides.

3 Contributions

3.1 Automatisation de la théorie des tableaux

L'objectif de l'implémentation est d'intégrer les fonctions de la théorie des tableaux dans VeriFast pour qu'elles soient directement retransmises à Z3.

L'intérêt d'automatiser la théorie des tableaux est de faciliter la manipulation des tableaux mais aussi de faciliter le raisonnement sur d'autres types algébriques tels que les ensembles et les multi-ensembles.

Les fonctions de la théorie des tableaux manipulent un type tableau qui était absent dans la logique de VeriFast. En effet, la logique de spécification de VeriFast ne possède pas de type tableau et la logique du premier ordre de VeriFast permettant la communication avec Z3 est construite sur 4 types, les types int, booléen, réel et *inductive*.
Le type inductive représente plusieurs types dont les types génériques. Par conséquent, il existe des fonctions de *boxing* et d'*unboxing*. Les fonctions de boxing permettent aux éléments d'un type différent d'*inductive* d'être utilisés dans des fonctions génériques en étant tranformés en type *inductive*. Par exemple, *mk_ boxed_ int* transforme un élément de type int en un élément de type *inductive*. Le type de *mk_ boxed_ int* est alors $int \rightarrow inductive$. Une fonction d'*unboxing* permet de transformer un élément du type *inductive* vers un élément d'un autre type σ. Le

type de la fonction d'unboxing est *inductive* $\rightarrow \sigma$. Une propriété sur ces fonctions est qu'unbox un élément boxé redonne le même élément si les deux fonctions manipulent le même type :

$$mk\_unboxed\_int(mk\_boxed\_int(2)) = 2$$

Attention, si i est de type *inductive*, $box(unbox(i))$ ne redonne pas forcément i, même si les fonctions de boxing/unboxing manipulent le même type. La raison est que le type doit être infini.

La première étape du travail du TRE a donc été d'ajouter un type primitif array dans la logique de spécification de VeriFast. Il a ensuite fallu ajouter un type array dans les communications avec Z3 et pour finir, ajouter les fonctions de la théorie des tableaux dans la logique de spécification mais aussi dans la logique de premier ordre afin de pouvoir transmettre le type array à Z3.

3.1.1 Ajout du type array

Soit σ_A le type d'un tableau, la syntaxe de σ_A dans VeriFast est "array(σ_I,σ_E)". Par exemple, pour définir un tableau a des entiers vers les booléens, nous écrivons $array(int, bool)$ a. Nous allons maintenant parler de la généricité des tableaux dans VeriFast, des contraintes pour préserver la cohérence de la théorie et nous finirons par l'ajout du type tableau à la logique de premier ordre.

Généricité En logique de spécification, le type des indices et le type des éléments du type tableau sont deux types génériques. Cela signifie qu'ils peuvent être de n'importe quel type. Un des objectifs a donc été d'implémenter un type tableau tout aussi générique. Le type et les fonctions ont été implementées ainsi mais VeriFast a posé une limite. Il n'y a pas de fonction générique dans VeriFast, pour représenter les pointeurs. En effet, il y a seulement des fonctions "integer(int*,int)", "character(char*,char)", etc... qui représentent des pointeurs d'un type précis, respectivement int et char. Il n'est donc pas possible actuellement de compléter la bibliothèque des tableaux avec uniquement des lemmes et prédicats génériques. Il est, par exemple, impossible de dire qu'un tableau est bien défini génériquement. La fonction array_model, ci-dessous, indique que le tableau d'entiers a 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 Un axiome assez naturel que des utilisateurs de VeriFast pourraient vouloir ajouter est que $\forall f \in \sigma_I \rightarrow \sigma_E$, $\exists a \in \sigma_A$ tel que $\forall i \in I, f(i) = select(a, i)$. Or, l'axiome n'est pas compatible avec un type t équivalent à array(t,p), p étant un type quelconque. Il est par conséquent interdit de définir un type *inductive* t avec un paramètre de type tableau ayant t comme type du domaine. Par exemple, la définition suivante est interdite :

```
inductive t = mk_array (array(t,int));
```

Type habité Dans VeriFast, tous les types doivent être habités, 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 ci-dessous :

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

Il est possible de prouver le 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 utilisée dans la logique de VeriFast et n'est vraie que si le type de x est habité.

Pour éviter la création d'un type non-habité à l'aide des tableaux, il a été interdit de créer un type inductif t ayant uniquement comme paramètre des tableaux utilisant t pour le type des indices ou des éléments. Par exemple, ci-dessous, la première version du type t a été interdite car le type est non-habité. La deuxième est autorisée car empty est un élément de type t. :

```
1. inductive t = mk_array (array(int, t));
2. inductive t = mk_array (array(int, t))| empty;
```

Type infini VeriFast a besoin de savoir si un type est de taille infinie. Si un type est infini, il est au moins aussi grand que le type *inductive*. Une fonction f de boxing et une fonction g d'unboxing manipulant un même type infini sont inverses dans les deux sens, c'est-à-dire, $g \circ f = id$ et $f \circ g = id$, où id est la fonction identité. Si le type est fini, alors $f \circ g \neq id$. En effet, si f et g manipulent un type fini et que $f \circ g = id$ alors VeriFast deviendrait incohérent.

Posons b_b et u_b les fonctions de boxing et d'unboxing du type booléen, b_i et u_i les fonctions de boxing et d'unboxing du type int. Nous avons alors $u_i \circ b_i = id$, $b_i \circ u_i = id$ et $u_b \circ i_b = id$ car int est un type infini. Supposons que $b_b \circ u_b = id$ alors que le type booléen est fini,

$$\begin{array}{lllllll} u_b(b_i(0)) & = & vrai & \vee & u_b(b_i(0)) & = & faux \text{ par definition du type booléen} \\ b_b(u_b(b_i(0))) & = & b_b(vrai) & \vee & b_b(u_b(b_i(0))) & = & b_b(faux) \\ b_i(0) & = & b_b(vrai) & \vee & b_i(0) & = & b_b(faux) \text{ par supposition} \end{array}$$

Si l'on répète les mêmes opérations avec 1 et 2. On obtient le système suivant :

$$\begin{cases} b_i(0) = b_b(vrai) \vee b_i(0) = b_b(faux) \\ b_i(1) = b_b(vrai) \vee b_i(1) = b_b(faux) \\ b_i(2) = b_b(vrai) \vee b_i(2) = b_b(faux) \end{cases}$$

ce qui est équivalent à

$$\begin{array}{lccccc} & b_i(0) = b_i(1) & \vee & b_i(1) = b_i(2) & \vee & b_i(0) = b_i(2) \\ \equiv & u_i(b_i(0)) = u_i(b_i(1)) & \vee & u_i(b_i(1)) = u_i(b_i(2)) & \vee & u_i(b_i(0)) = u_i(b_i(2)) \\ \equiv & 0 = 1 & \vee & 1 = 2 & \vee & 0 = 2 \end{array}$$

Il est donc possible avec cette propriété de prouver le faux en partant du vrai. Donc elle rendrait VeriFast incohérent.

Le type tableau n'est pas présent dans la logique du premier ordre de VeriFast. Cette logique permet à VeriFast de communiquer avec les SMT-solvers. Après, avoir rajouté le type tableau à la logique de spécification, il fallait ajouter le type tableau à la logique du premier ordre. En effet, sans un type array, les fonctions de la théorie des tableaux implémentées dans Z3 ne sont pas utilisables.

Un type array a été ajouté. Pour représenter un type dans une fonction générique, il lui faut des fonctions de boxing et d'unboxing. Ces fonctions ont donc été implémentées.

La première idée a été d'implémenter des fonctions de boxing et d'unboxing qui prennent en compte le type des indices et des éléments. Cette implémentation était incorrecte car VeriFast déclare ces fonctions de boxing et d'unboxing à Z3 dès le début. Or, avant de lire le code, VeriFast ne peut pas connaître les types des tableaux qui vont apparaître dans la preuve.

L'alternative à ce problème a été de faire des fonctions de boxing et d'unboxing pour des tableaux de type *inductive* pour le domaine et le co-domaine. Les tableaux utilisant d'autres types utiliseront les fonctions de boxing et d'unboxing des types de leurs domaine et co-domaine pour transformer leurs indices et éléments en type *inductive*. Comme ces fonctions existent pour tous les types de Z3, cette alternative permet à VeriFast de déclarer, dès le début, la fonction et son type sans perdre l'aspect générique.

3.1.2 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 choisir comment représenter et récupérer ces fonctions dans le code de VeriFast. La deuxième est de définir comment ces fonctions vont être communiquées à Z3.

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 définies dans une nouvelle bibliothèque sous la forme de fonctions abstraites. Une fonction abstraite est une fonction qui n'a pas de corps. Ainsi, tous les programmes antérieurs restent corrects et les noms restent disponibles si la bibliothèque n'est pas incluse.

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

Une possibilité aurait été de modifier le parseur 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 modifications dans plusieurs fichiers. Par conséquent, une autre solution a été choisie.

L'autre solution consiste à trouver un point précis du code où toutes les fonctions vont être

appelées et où un maximum de vérification aura déjà été faite. Ainsi, le nombre de vérifications à modifier sera minime et donc limitera la quantité de modifications apportées sans en changer la qualité de ce qui a été implémenté.

Lors de la dernière vérification intéressante pour les fonctions abstraites, un filtrage sur le nom des fonctions permet aux primitives de la théorie d'être repérées et ainsi d'être traitées comme les fonctions de la théorie des tableaux.

Dans VeriFast, les deux solveurs ont une API commune. Ainsi, si on veut ajouter de nouvelles fonctions pour communiquer avec Z3, on doit s'assurer que Redux saura quoi en faire.

Redux ne connaît pas la théorie des tableaux. Il faut donc que lorsqu'un utilisateur de VeriFast utilise une fonction de la théorie, Redux sache comment réagir. La réaction de Redux est de considérer ces fonctions comme non-interprétées. L'utilisateur pourra prouver les mêmes choses que s'il utilisait Z3 mais il devra faire les preuves alors qu'avec les Z3, les preuves dans le bon fragment seront automatisées.

Les fonctions sont des primitives dans Z3. L'implémentation consiste donc à un appel de fonctions.

3.2 Bibliothèques

La bibliothèque de la théorie des tableaux et, par extension, la bibliothèque des multi-ensembles ont été ajoutées dans VeriFast. Je ne vais pas vous présenter l'intégralité des bibliothéques mais seulement les fixpoints et les lemmes importants.

3.2.1 Bibliothèques de la théorie des tableaux

La bibliothèque de la théorie des tableaux commencent par les fonctions de la théorie et ces axiomes.

```
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;
{}
```

t,u sont les types génériques.

Aujourd'hui, VeriFast indique que les axiomes sont corrects alors que la preuve est vide, ce n'était logiquement pas le cas avant l'automatisation de la théorie.

Les derniers prédicats et lemmes majeurs de la bibliothèque sont le prédicat *array_model*(int* a, int b, int e, array(int,int) arr) représentant un tableau a entre (a+b) et (a+e) par le tableau arr et le lemme *array_model_init*($int*, int$) transformant une liste d'entier en un tableau d'entier. Les prédicats *integer*($int*, int$) et *ints*($int*, int, list$) utilisés ci-dessous représentent, respectivement, un pointeur d'entier et une liste d'entiers.

```
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, _);
{...}
```

Par défaut, le résultat renvoyé malloc ou un tableau en C sont représentés comme une liste dans VeriFast, d'où l'importance du lemme *array_model_init*.

3.2.2 Bibliothéque des multi-ensembles

Les multi-ensembles sont représentés comme des tableaux avec des éléments de type *nat*.

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

Les fonctions principales sont *multiset_select* permettant de récupérer le nombre d'occurrences d'une valeur dans un multi-ensemble, *multiset_empty* permettant d'initialiser un multi-ensemble vide, *multiset_add* permettant d'ajouter un élément à un multi-ensemble, *multi_ext* étant l'égalité entre deux multi-ensembles et *array_multiset* permettant de récupérer le multi-ensemble des éléments d'un tableau.

Le prédicat le plus important de cette bibliothèque est *same_multiset*(array(int,int) a1, array(int,int) a2, int b, int e) indiquant que les tableaux $a1$ et $a2$ ont le même multi-ensemble d'éléments entre b et e.

3.3 Quicksort

L'algorithme de tri Quicksort, développé par Hoare, (voir page 23/24) doit être l'un des algorithmes le plus connu et le plus utilisé. Même si sa correction a été démontrée de nombreuses fois, certifier un programme VeriFast revient à montrer que l'ensemble du code est correct avec VeriFast. Il faut donc que les algorithmes les plus classiques soient de nouveau prouvés. L'algorithme du Quicksort n'était pas prouvé en VeriFast et était un bon exemple pour illustrer l'intérêt de la théorie des tableaux.

J'ai défini la spécification du Quicksort ainsi :
Précondition : le tableau est bien défini entre (a+lo) et (a+hi).
Postcondition : le tableau est bien défini entre (a+lo) et (a+hi), le multi-ensemble des éléments est inchangé et le tableau est trié.

L'algorithme du Quicksort utilise les fonctions *swap* et *partition*.
La spécification de *swap* a été définie ainsi :
Précondition : le tableau est bien défini entre (a+lo) et (a+hi).

Postcondition : le tableau est bien défini entre (a+lo) et (a+hi) et les valeurs du tableau à la position i et j ont été échangées.
La spécification de *partition* a été définie ainsi :
Précondition : le tableau est bien défini entre (a+lo) et (a+hi).
Postcondition : le tableau est bien défini entre (a+lo) et (a+hi), le multi-ensemble des éléments du tableau est inchangé, le pivot est à la position que retourne la fonction, tous les éléments avant le pivot dans le tableau sont plus petits que le pivot et tous les éléments après le pivot dans le tableau sont plus grands que le pivot.

La preuve de la fonction *partition* se passe principalement dans la boucle. Les invariants de boucle importants sont que les élements entre low et i sont plus petits que le pivot, que les éléments entre $i+1$ et j sont plus grands que le pivot et que le multi-ensemble reste le même. Ainsi, à la fin de la boucle, il ne reste plus qu'à incrémenter i placer le pivot à la position i et la postcondition est respectée.

La preuve de la fonction quicksort se repose sur deux cas, si le tableau est vide, c'est-à-dire, $lo > hi$, alors le tableau vide est déjà trié et le multi-ensemble des éléments reste vide. Si le tableau n'est pas vide, on appelle la fonction *partition* puis on rappelle *quicksort* récursivement sur les deux morceaux du tableau. Le morceau où tous les éléments sont plus petits que $a[p]$ et le morceau où les éléments sont plus grands que $a[p]$.

Après l'appel à la fonction *partition*, les garanties de la postcondition de *partition* sont que les éléments entre lo et $p-1$ sont plus petits que $a[p]$, que les éléments entre $p+1$ et hi sont plus grands que $a[p]$ et que le multi-ensemble des éléments n'a pas changé. Il reste donc à rappeler l'algorithme du quicksort sur les deux parties du tableaux. Après les deux appels récursifs, les postconditions des deux appels à quicksort et à partition nous garantissent ceci :

1. Avant l'appel à quicksort, les éléments entre lo et p-1 sont plus petits que $a[p]$.
2. Avant l'appel à quicksort, les éléments entre p+1 et hi sont plus grands que $a[p]$.
3. Avant l'appel à quicksort, le multi-ensemble des éléments du tableau est le même qu'au début.
4. Après l'appel à quicksort, le multi-ensemble des éléments entre lo et p-1 est le même qu'avant l'appel à quicksort.
5. Après l'appel à quicksort, le tableau est trié entre lo et p-1.
6. Après l'appel à quicksort, le multi-ensemble des éléments entre p+1 et hi est le même qu'avant l'appel à quicksort.
7. Après l'appel à quicksort, le tableau est trié entre p+1 et hi.

   Il faut maintenant combiner les propriétés pour avoir les post-conditions voulues.
8. En combinant (1) et (4), il est possible de montrer qu'après l'appel de quicksort, les éléments entre lo et p-1 sont plus petits que $a[p]$.
9. En combinant (2) et (6), il est possible de montrer qu'après l'appel de quicksort, les éléments entre p+1 et hi sont plus grands que $a[p]$.
10. En combinant (4) (6) et la valeur du pivot, il est possible de montrer que le multi-ensemble des éléments du tableau avant les appels à quicksort est le même qu'après.
11. En appliquant la transitivité de l'égalité avec (3) et (10), il est possible de montrer que le multi-ensemble des éléments du tableau est le même au début et après les appels récursifs.

12. En combinant (5) (7) (8) et (9), il est possible de montrer qu'après les appels récursifs, le tableau est bien trié.

Pour autoriser les appels récursifs, un lemme est appelé . Ce lemme dit que si un tableau est bien défini entre lo et hi et $lo <= p <= hi$, alors il existe un tableau bien défini entre lo et p-1, un autre entre p+1 et hi et la case p est bien définie. Ce découpage permet aux appels récursifs de respecter la précondition. Après les appels récursifs, un lemme inverse permet de reformer un tableau entre lo et hi. En considérant les prédicats (11) et (12), nous pouvons conclure que la postcondiction de l'algorithme du Quicksort est bien respectée.

55 lignes de code m'ont été nécesssaire pour ajouter le type tableau, ces restrictions et ces fonctions de boxing. L'ajout des fonctions de la théorie des tableaux représente 140 lignes. Les bibliothèques font 858 lignes et la preuve du Quicksort fait 554 lignes.

4 Conclusion

L'automatisation de la théorie des tableaux est aujourd'hui fonctionnelle dans VeriFast. Les preuves dans le bon fragment de la théorie des tableaux sont totalement automatisées avec Z3. Les bibliothèques de la théorie des tableaux et des multi-ensembles ont été construites et l'algorithme du Quicksort a été prouvé en VeriFast à l'aide des bibliothèques et de l'automatisation.

Une suite possible à ce travail est d'automatiser de nouvelles théories ou d'ajouter d'autres fonctions à la théorie des tableaux.

Références

[1] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.

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);
}
```