

Sémantique des blocs inline assembleurs x86 dans du code C

Rapport de stage L3

Jérôme Boillot

ENS de Lyon

Encadré par :

Marc Chevalier et **Jérôme Feret**

Équipe ANTIQUE, ENS de Paris, CNRS, Inria

juin-juillet 2019

Résumé

Prouver que des programmes ne peuvent pas renvoyer d'erreurs est un enjeu important dans les systèmes informatiques embarqués. L'interprétation abstraite est une méthode qui a été utilisée pour prouver des programmes C. Est-ce possible d'utiliser cette même méthode pour prouver des programmes C contenant des blocs assembleurs? J'ai prouvé pendant mon stage que cela était possible moyennant quelques restrictions.

Table des matières

1	Éléments de sémantique et d'interprétation abstraite	2
1.1	Buts et principes fondamentaux	2
1.2	Syntaxe du C sans blocs assembleurs	6
1.3	Transitions de l'état mémoire abstrait	7
1.3.1	Sémantique à petits pas sur les environnements	7
1.3.2	Sémantique à grands pas sur les environnements	7
1.3.3	Abstraction de la sémantique à grands pas	9
2	Apport personnel	11
2.1	La mise en place d'un domaine abstrait : les intervalles	11
2.2	Ajout des blocs assembleurs	13
2.3	Nouvelle syntaxe incluant une partie des blocs assembleurs	14
2.4	Évaluation des nouvelles instructions	14
2.5	Fonctionnalités supplémentaires implémentées dans mon analyseur	17
2.6	Contraintes de mon analyseur	19

Introduction

Un enjeu important de l'informatique est de prouver que des programmes ne peuvent pas renvoyer d'erreurs. C'est particulièrement nécessaire pour les systèmes informatiques embarqués, par exemple dans les avions, où les erreurs potentielles peuvent être fatales. Le théorème de RICE annihile tout espoir de prouver automatiquement l'absence d'erreurs de programmes. Plusieurs techniques de preuve d'absence d'erreur ont été mises au point, chacune d'entre elles devant faire des concessions. Dans notre cas, on s'intéresse à la technique de l'interprétation abstraite qui conserve l'automatisation des preuves au prix de l'exactitude des résultats. En effet, des erreurs peuvent apparaître, qui sont des faux positifs. En revanche, les vrais positifs seront tous détectés.

Des logiciels comme ASTRÉE ont été conçus avec cette méthode pour prouver l'absence d'erreurs de programmes C.

Cependant, ASTRÉE ne prend pas en charge, en tout cas pas pour le moment, le code inline assembleur qui est parfois nécessaire lors de la création de programmes dans l'embarqué. Or, c'est justement une de ses cibles.

Mon but pendant le stage a donc été de recréer un analyseur abstrait permettant d'analyser une partie du code C avec de l'assembleur inline dans le but de montrer qu'il est effectivement possible de tester de tels codes avec les outils de l'interprétation abstraite.

Dans un premier temps, je présenterai des éléments tirés de l'analyse bibliographique de l'interprétation abstraite sur une partie restreinte du code C ainsi qu'une syntaxe associée.

Une fois ces éléments bien appréhendés, j'étendrai dans un second temps l'analyse abstraite et la syntaxe au support des blocs assembleurs qui peuvent être intégrés dans du code C. Cette seconde partie correspond à mon apport personnel.

Un glossaire est disponible en annexe page 24 pour permettre au lecteur de retrouver facilement le sens de certaines notations.

1 Éléments de sémantique et d'interprétation abstraite

1.1 Buts et principes fondamentaux

Notre but est d'analyser statiquement un programme en un temps fini et donc forcément en faisant des approximations. Mais comment faire des approximations tout en ne se permettant pas d'oublier des erreurs? Pour cela, on va présenter des outils mathématiques qui vont nous être utiles pour être sûr que l'on ne rate pas d'erreurs. Ces outils et principes sont présentés en détail tout au long de [Cou99]. De plus, la création de la syntaxe des programmes C ainsi que la compréhension des principes introduits m'ont été possibles avec la lecture de [Fer05] et des deux premiers chapitres de [Min13].

Dans un vrai programme, toutes les variables peuvent avoir différentes valeurs, et ces valeurs peuvent dépendre des valeurs des autres variables. Ainsi, on peut représenter l'état mémoire concret comme un ensemble de couples variable-valeur. On peut penser le programme comme une suite finie d'instructions que l'on va lire de façon séquentielle, et qui vont changer l'état

mémoire. Cependant, lorsque l'on teste toutes les évaluations possibles d'un programme, on va parfois devoir lire le programme de façon non-séquentielle : l'analyseur va donc devoir explorer tous les chemins possibles à la fois.

On va alors définir un domaine abstrait qui va représenter de façon symbolique, et plus facilement manipulable, des propriétés sur les états mémoire. On a donc besoin d'une fonction qui va nous permettre de passer d'un certain état mémoire concret à un état mémoire abstrait dans lequel on pourra effectuer l'analyse. Cette étape est appelée abstraction tandis que l'étape inverse est appelée concrétisation.

On va nommer D le domaine concret, $D^\#$ le domaine abstrait, $\alpha : D \rightarrow D^\#$ la fonction d'abstraction et $\gamma : D^\# \rightarrow D$ la fonction de concrétisation.

Dans le cas du domaine concret, on pourrait considérer que l'état mémoire est un ensemble d'associations variable valeur. On va ici, pour simplifier, négliger les relations entre les variables. On a alors un état mémoire concret qui est une association de variables à des ensembles de valeurs. Par exemple si lorsque $a = 1$, $b = 2$ et que lorsque $a = 2$, $b = 4$, on perd la relation entre a et b et l'on conserve seulement que $a \in \{1, 2\}$ et $b \in \{2, 4\}$.

On a alors D et $D^\#$ qui forment des treillis complets¹ :

- (1) $(D \triangleq \mathcal{P}(I), \sqsubseteq \triangleq \subseteq, \perp \triangleq \emptyset, \top \triangleq I, \sqcup \triangleq \cup, \sqcap \triangleq \cap)$ où $I = \{x \mid -2^{32} \leq x \leq 2^{32} - 1\}$
- (2) $(D^\#, \sqsubseteq^\#, \perp^\#, \top^\#, \sqcup^\#, \sqcap^\#)$

Passons en revue les objets introduits dans la notation d'un treillis complet représenté par un sextuplet :

1. L'ensemble des éléments du treillis.
2. L'ordre partiel du treillis.
3. L'élément minimum que l'on appelle parfois *bottom*.
4. L'élément maximum que l'on appelle parfois *top*.
5. La fonction qui a deux éléments renvoie le plus petit élément qui est plus grand que ces deux éléments. On appelle parfois cette fonction *join*. On sait que cette fonction est définie pour tout couple d'éléments du treillis car toute partie du treillis possède une borne supérieure par définition. Plus intuitivement, cela revient à considérer les cas qui sont dans un des deux éléments.
6. La fonction qui a deux éléments renvoie le plus grand élément qui est plus petit que ces deux éléments. On appelle parfois cette fonction *meet*. On sait que cette fonction est définie pour tout couple d'éléments du treillis car toute partie du treillis possède une borne inférieure par définition. Plus intuitivement, cela revient à considérer les cas qui sont dans les deux éléments à la fois.

On peut imaginer l'ordre partiel \sqsubseteq comme la comparaison de précision : $x \sqsubseteq y$ quand x comporte au plus autant de cas que y . On peut également imaginer $\sqsubseteq^\#$ comme la comparaison des contraintes : $x^\# \sqsubseteq^\# y^\#$ quand $x^\#$ contient au moins autant de contraintes que $y^\#$.

1. Un ensemble partiellement ordonné tel que tout partie possède une borne inférieure (un minimum) et supérieure (un maximum).

On voit ainsi apparaître l'ébauche d'une propriété essentielle à l'interprétation abstraite : le concept de correction qui consiste à imposer des contraintes de telle sorte qu'on n'omette pas de cas :

- On impose de ne pas gagner de précision lorsque l'on abstrait puis que l'on concrétise et de ne pas perdre des contraintes lorsque l'on concrétise puis que l'on abstrait, soit :

$$(3) \quad x \sqsubseteq \gamma \circ \alpha(x), \quad \forall x \in D$$

$$(4) \quad \alpha \circ \gamma(x^\#) \sqsubseteq^\# x^\#, \quad \forall x^\# \in D^\#$$

- De plus, un élément abstrait (resp. concret) moins précis doit aussi l'être dans sa version concrète (resp. abstraite) associée. Pour cela, on impose d'avoir α et γ monotones.

De telles propriétés correspondent en fait à la définition d'une correspondance de GALOIS antitone que l'on appellera par la suite correspondance de GALOIS. Une définition équivalente à celle donnée précédemment est :

$$(5) \quad \alpha(x) \sqsubseteq^\# x^\# \iff x \sqsubseteq \gamma(x^\#) \quad \text{où } x \in D \text{ et } x^\# \in D^\#$$

Plus intuitivement, quand un élément concret (via l'abstraction) représente plus de contraintes qu'un élément abstrait, alors il y a nécessairement plus de cas dans l'élément abstrait que dans le concret (via la concrétisation), et réciproquement.

Mais alors que peut être ce domaine abstrait ? On peut penser par exemple aux intervalles. Une variable a sa valeur qui pourra être dans un certain intervalle à un certain moment. Par exemple, si la valeur abstraite d'une certaine variable a est $\llbracket 1, 5 \rrbracket$ cela signifie que la concrétisation de cette valeur abstraite pourra être 1, 2, 3, 4 ou 5. On va, dans la suite de ce rapport, travailler sur le domaine abstrait que sont les intervalles tout en gardant à l'esprit une généralisation future à des modèles plus complexes permettant une analyse plus fine (on peut par exemple penser aux entiers multiples d'un certain nombre).

On peut maintenant passer du domaine abstrait au domaine concret et inversement autant de fois qu'on le veut sans oublier de cas.

Comment ensuite appliquer des fonctions concrètes aux domaines abstraits de telle sorte que l'on n'oublie pas de cas mais qu'on ait tout de même un résultat le plus précis possible une fois concrétisé ?

On veut assurer la propriété de correction : ne pas omettre de cas. Or, tous les cas possibles sont décrits par la concrétisation du domaine abstrait. Ainsi, on veut que le résultat de l'application de la fonction au domaine abstrait englobe tous les cas décrits par l'abstraction de toutes les applications de la fonction concrète aux différents éléments de la concrétisation.

Plus formellement, si on a une fonction concrète $f : D \rightarrow D$, on pourra choisir une fonction abstraite associée $f^\# : D^\# \rightarrow D^\#$ qui respecte la propriété suivante :

$$(6) \quad \alpha \circ f \circ \gamma(x) \sqsubseteq^\# f^\#(x) \quad \forall x \in D^\#$$

On sait qu'une telle fonction existe car la fonction constante égale à $\top^\#$ vérifie cette propriété. Cependant ce choix rendrait l'analyse peu efficace.

On veut maintenant stocker et faire des calculs sur les domaines abstraits des différentes variables. On va pour l'instant se focaliser seulement sur la représentation mathématique et non pas sur une possible implémentation.

On va tout d'abord définir ce que j'appellerai l'état mémoire ou le contexte : c'est une fonction partielle qui à une variable dans l'ensemble des variables possibles \mathbb{V} associe un élément du domaine concret D .

Soit $(C \triangleq \mathcal{P}(\mathbb{V} \rightarrow D), \underline{\dot{\sqsubseteq}}, \dot{\perp}, \dot{\top}, \dot{\sqcup}, \dot{\sqcap})$ le treillis complet associé qui est défini de la même façon que celui associé à D .

Plus intuitivement, on peut dire que cet ordre signifie qu'il y a au moins autant de cas contenus dans l'état mémoire \dot{y} que dans \dot{x} .

Nous avons défini la notion de contexte. Cependant, celui-ci va évoluer au fur et à mesure de l'exécution du programme. On va donc tout d'abord définir les points de programme, éléments de $\mathcal{L} \triangleq \mathcal{L}_c$, qui représentent une position dans un programme à laquelle on pourra associer un contexte. On va associer un ordre total $\leq_{\mathcal{L}}$ à l'ensemble \mathcal{L} qui va nous permettre de nous repérer dans le programme. Si une instruction est avant une autre, les points de programme associés respecteront cet ordre.

On peut maintenant introduire l'association des points de programme à leur contexte ce que j'appelle environnement.

Soit $(E \triangleq \mathcal{P}(\mathcal{L} \rightarrow C), \underline{\ddot{\sqsubseteq}}, \ddot{\perp}, \ddot{\top}, \ddot{\sqcup}, \ddot{\sqcap})$ le treillis complet associé qui est défini de la même façon que celui associé à D .

Plus intuitivement et de façon analogue aux contextes, on peut dire que cet ordre signifie qu'il y a au moins autant de cas contenus dans l'environnement \ddot{y} que dans \ddot{x} .

On veut maintenant appliquer des transformations aux environnements que l'on manipule. On définit donc $(T \triangleq \mathcal{P}(E \rightarrow E), \underline{\dot{\dot{\sqsubseteq}}}, \dot{\dot{\perp}}, \dot{\dot{\top}}, \dot{\dot{\sqcup}}, \dot{\dot{\sqcap}})$ le treillis complet associé qui est défini de la même façon que celui associé à D .

De même que pour les cas précédents, on peut penser plus intuitivement que cet ordre signifie qu'il y a au moins autant de cas contenus dans les transitions entre environnements de $\dot{\dot{x}}$ que dans celles $\dot{\dot{y}}$.

On définit les contextes, les environnements et les transitions entre environnements abstraits en rajoutant syntaxiquement des \sharp au dessus de D, C, E et T ainsi que sur les différents objets apparaissant dans les définitions des treillis. On a alors toujours des treillis car D^\sharp est un treillis et que C^\sharp, E^\sharp et T^\sharp sont définis à partir de D^\sharp .

Dans le cas des intervalles choisis comme domaine abstrait ($D^\sharp = \text{Intervalles}(\mathbb{N})$), il en découle assez simplement que :

$$(7) \quad \llbracket x_1, x_2 \rrbracket \sqcap^\sharp \llbracket y_1, y_2 \rrbracket = \begin{cases} \perp^\sharp \triangleq \emptyset & \text{si } x_2 < y_1 \text{ ou } y_2 < x_1 \\ \llbracket \max(x_1, y_1), \min(x_2, y_2) \rrbracket & \text{sinon} \end{cases}$$

$$(8) \quad \llbracket x_1, x_2 \rrbracket \sqcup^\sharp \llbracket y_1, y_2 \rrbracket = \llbracket \min(x_1, y_1), \max(x_2, y_2) \rrbracket$$

Définissons maintenant une syntaxe simplifiée du langage C pour ensuite étudier le système d'abstractions que l'on peut en tirer.

1.2 Syntaxe du C sans blocs assembleurs

Définitions générales :

$$\begin{aligned} \text{type} & ::= \text{int} \mid \text{type}^* && \text{(Types C)} \\ l_c & \in \mathcal{L}_c \subseteq \mathcal{L} && \text{(Points mémoires C)} \end{aligned}$$

Programme :

$$\text{prog} ::= {}^{l_c}\text{global_stat} {}^{l_c}$$

Ensemble de fonctions :

$$\begin{aligned} {}^{l_c}\text{global_stat} {}^l & ::= {}^{l_c}\text{type } \text{name_fun}(\text{type } \text{param}_1, \dots, \text{type } \text{param}_n)\{\text{stat_c}\} {}^{l_c} \\ & \mid {}^{l_c}\text{global_stat} {}^{l_c}\text{global_stat} {}^{l_c} \end{aligned}$$

Instructions C :

$$\begin{aligned} {}^{l_c}\text{stat_c} {}^l & ::= {}^{l_c}\text{stat_c} {}^{l_c}\text{stat_c} {}^{l_c} \\ & \mid {}^{l_c}\text{type } \text{name_var}; {}^{l_c} \\ & \mid {}^{l_c}\text{name_var} := \text{arith_expr}; {}^{l_c} \\ & \mid {}^{l_c} * \text{name_var} := \text{arith_expr}; {}^{l_c} \\ & \mid {}^{l_c}\text{name_var} := \text{name_fun}(\text{arith_expr}, \dots, \text{arith_expr}); {}^{l_c} \\ & \mid {}^{l_c}\text{if } \text{bool_expr} \text{ then } {}^{l_c}\text{stat_c} {}^{l_c}\text{endif} {}^{l_c} \\ & \mid {}^{l_c}\text{while } {}^{l_c}\text{bool_expr} \text{ do } {}^{l_c}\text{stat_c} {}^{l_c}\text{done} {}^{l_c} \\ & \mid {}^{l_c}\text{return } \text{name_var}; {}^{l_c} \end{aligned}$$

Expression arithmétique :

$$\begin{aligned} \text{arith_expr} & ::= \text{name_var} \\ & \mid x \quad \text{où } x \in \llbracket -2^{32}, 2^{32} - 1 \rrbracket && \text{(Constantes)} \\ & \mid [x, y] \quad \text{où } -2^{32} \leq x \leq y \leq 2^{32} - 1 && \text{(Intervalle de valeurs possibles)} \\ & \mid * \text{name_var} \\ & \mid \&\text{name_var} \\ & \mid \circ \text{arith_expr} && \text{(Opérateurs unaires arithmétiques)} \\ & \mid \text{arith_expr} \diamond \text{arith_expr} && \text{(Opérateurs binaires arithmétiques)} \end{aligned}$$

Expression booléenne :

$$\begin{aligned} \text{bool_expr} & ::= \text{bool_expr} \star \text{bool_expr} && \text{(Opérateurs binaires booléens)} \\ & \mid \text{arith_expr} \bowtie \text{arith_expr} && \text{(Comparaisons arithmétiques)} \\ \bowtie & ::= = \mid \neq \mid < \mid > \mid \leq \mid \geq \\ \star & ::= \wedge \mid \vee \\ \circ & ::= - \mid \sim \\ \diamond & ::= + \mid - \mid \times \mid / \end{aligned}$$

1.3 Transitions de l'état mémoire abstrait

On veut maintenant « exécuter » la version abstraite de notre programme. On va d'abord décrire une « sémantique à petits pas » qui exécute simultanément un pas de chaque instruction du programme de manière glissante. Ensuite, l'exécution des ces « petits pas » jusqu'à ce que le programme termine permet d'obtenir la « sémantique à grands pas ».

1.3.1 Sémantique à petits pas sur les environnements

Pour cela on va définir la relation binaire $\tau[stat]$ avec $stat \in {}^lstat_c^{l'}$ qui représente l'ensemble des transitions d'un environnement vers celui dans lequel on a exécuté un petit pas *i.e.* que l'on a exécuté, pour tous les points de programme présents dans l'instruction, le passage au point de programme qui doit suivre dans l'exécution, et avec le contexte à ce point-là qui a été modifié par l'instruction lue. La transition peut également mener à un état d'erreur qui est un élément de Ω que l'on notera ω .

On peut ainsi définir τ par induction structurelle sur ${}^lstat_c^{l'}$ (8 cas).

Pour bien comprendre comment τ est définie on va étudier le cas ${}^lcstat_1{}^{l'_c}stat_2{}^{l'_c}$.

$$\begin{aligned} \tau[{}^lcstat_1{}^{l'_c}stat_2{}^{l'_c}] = & \left\{ \left(\langle l_c, \rho \rangle, \langle l'_c, \rho' \rangle \right) \mid \rho \in C, \left(\langle l_c, \rho \rangle, \langle l'_c, \rho' \rangle \right) \in \tau[{}^lcstat_1{}^{l'_c}] \right\} \\ & \cup \left\{ \left(\langle l'_c, \rho \rangle, \langle l'_c, \rho' \rangle \right) \mid \rho \in C, \left(\langle l'_c, \rho \rangle, \langle l'_c, \rho' \rangle \right) \in \tau[{}^lcstat_2{}^{l'_c}] \right\} \end{aligned}$$

Plus intuitivement, on a fait passer le contexte qui se trouvait en l_c par l'instruction $stat_1$ tandis que l'on a fait passer le contexte qui se trouvait en l'_c par l'instruction $stat_2$.

On peut également regarder le cas ${}^lcX := v{}^{l'_c}$:

$$\begin{aligned} \tau[{}^lcX := v{}^{l'_c}] = & \left\{ \left(\langle l_c, \rho \rangle, \langle l'_c, \rho' \rangle \right) \mid \rho \in C, X \in \text{Dom}(\rho), \rho' = \rho[X := \{\text{Évaluation de } v \text{ dans } \rho\}] \right\} \\ & \cup \left\{ \left(\langle l_c, \rho \rangle, \omega \right) \mid \rho \in C, X \notin \text{Dom}(\rho) \right\} \end{aligned}$$

Plus intuitivement, si dans le contexte au point mémoire l_c la variable X est déjà définie alors on remplace juste l'ancienne valeur par la nouvelle au point l'_c et si elle n'était pas définie on renvoie un état d'erreur.

Les 6 autres cas sont décrits dans [Cou99].

1.3.2 Sémantique à grands pas sur les environnements

On peut donc maintenant calculer l'évolution lors de la lecture d'une instruction des points mémoires et des contextes associés. Cependant, pour le moment, si l'on a 3 instructions qui se suivent on aura 3 nouveaux contextes qui seront passés seulement par une seule instruction chacun. On veut donc définir une fonction décrivant les transitions entre environnements lors de la lecture d'une instruction complète. Pour cela, on va considérer la clôture réflexive transitive

de la relation binaire $\tau[stat]$ que l'on écrit $\tau^*[{}^lstat{}^r] \in \mathcal{P}((\mathcal{L} \times C)^2)$ pour ${}^lstat{}^r \in \text{stat_c}$. On peut se représenter cette clôture comme l'ensemble de tous les contextes qui peuvent être associés à un moment à un point de programme.

Plus formellement, on définit la composition de relations binaires de la façon suivante :

$$(9) \quad t \circ r = \{ \langle s, s'' \rangle \mid \exists s' \in S, \langle s, s' \rangle \in t \wedge \langle s', s'' \rangle \in r \} \quad \text{où } (t, r) \in (\mathcal{P}(S^2))^2$$

On peut ainsi définir les puissances d'une relation binaire de la façon suivante avec $t \in \mathcal{P}(S^2)$:

$$(10) \quad \begin{cases} t^n \triangleq \emptyset & \text{pour } n < 0 \\ t^0 \triangleq 1_S \triangleq \{ \langle s, s \rangle \mid s \in S \} \\ t^{n+1} \triangleq t \circ t^n \triangleq t^n \circ t & \text{pour } n \geq 0 \end{cases}$$

Il faut bien noter dans la suite que la composition pour les relations binaires et pour les fonctions sont différentes. En particulier $t^n \neq (\lambda r. t \circ r)^n$ pour $t \in \mathcal{P}(S^2), n \in \mathbb{N}$.

Pour avoir une image de la composition des relations binaires on peut prendre l'exemple de la relation binaire \mathcal{R} telle que :

$$\mathcal{R}_{i,j} \Leftrightarrow \{ \text{il existe un chemin direct entre } i \text{ et } j \}$$

On a alors

$$\mathcal{R}_{i,j}^n \Leftrightarrow \{ \text{il existe un chemin de longueur au maximum } n \text{ de } i \text{ à } j \}$$

On peut alors calculer t^* pour ensuite pouvoir calculer $\tau^*[stat]$:

$$(11) \quad t^* \triangleq \bigcup_{n \geq 0} t^n = \bigcup_{n \geq 0} \left(\bigcup_{i \leq n} t^i \right) = \bigcup_{n \geq 0} (\lambda r. 1_S \cup t \circ r)^n(\emptyset) = \text{lfp}(\lambda r. 1_S \cup t \circ r) \quad (\text{c.f. } ^2)$$

On peut maintenant calculer τ^* pour l'instruction $X := v$ que l'on a précédemment décrites :

$$\tau^*[{}^lc X := v{}^rc] = 1_{\mathcal{L} \times C} \cup \tau[X := v] \quad \text{car } \tau^2[X := v] = \tau[X := v]$$

(une fois que l'on est en lc on ne peut plus avancer)

Une fois que l'on a calculé la clôture, on a l'ensemble de toutes les transitions impliquées par la lecture de ${}^lstat{}^r$. On n'a donc plus besoin des transitions mais juste des états finaux calculés à partir d'un certain état initial. On définit donc $\text{post} : (\mathcal{P}(S^2) \rightarrow S \rightarrow \mathcal{P}(S))$ telle que $\text{post}[t]P = \{ s' \mid \exists \langle s, s' \rangle \in t, s \in P \}$. On a ainsi sélectionné le nouvel état du système à partir d'un certain état P après avoir effectué les transitions de t .

En particulier, quand $t = \tau^*[{}^lstat{}^r]$, l'environnement initial P donne l'environnement final après lecture de l'instruction ${}^lstat{}^r$. On va noter cette fonction particulière $\text{Post}[{}^lstat{}^r]P \triangleq \text{post}[\tau^*[{}^lstat{}^r]]P$.

2. Cette égalité est prouvée en annexe à la page 22.

1.3.3 Abstraction de la sémantique à grands pas

On veut maintenant décrire les abstractions et les concrétisations de fonctions.

Commençons par le cas des fonctions agissant sur les contextes (abstrait ou non). Soit $f : C \rightarrow C$ et $g : C^\# \rightarrow C^\#$. On veut décrire l'abstraction de f que l'on notera $\dot{\alpha}(f) : C^\# \rightarrow C^\#$ ainsi que la concrétisation de g que l'on notera $\dot{\gamma}(g) : C \rightarrow C$.

De façon assez intuitive on peut définir $\dot{\alpha}$ et $\dot{\gamma}$ telles que :

$$(12) \quad \begin{cases} \dot{\alpha}(f) = \alpha \circ f \circ \gamma \\ \dot{\gamma}(g) = \gamma \circ g \circ \alpha \end{cases}$$

En effet, l'abstraction d'une fonction consiste à appliquer la fonction au domaine concret en entrée puis à ré-abstraire le résultat. Avec les propriétés de α et de γ , on garde la propriété de correction.

On définit de la même façon les abstractions et les concrétisations pour les fonctions agissant sur les environnements et sur les transitions entre environnements :

$$(13) \quad \begin{cases} \ddot{\alpha}(f) = \dot{\alpha} \circ f \circ \dot{\gamma} \in E^\# \rightarrow E^\# & \text{où } f : E \rightarrow E \\ \ddot{\gamma}(g) = \dot{\gamma} \circ g \circ \dot{\alpha} \in E \rightarrow E & \text{où } g : E^\# \rightarrow E^\# \end{cases}$$

$$(14) \quad \begin{cases} \dot{\dot{\alpha}}(f) = \ddot{\alpha} \circ f \circ \ddot{\gamma} \in T^\# \rightarrow T^\# & \text{où } f : T \rightarrow T \\ \dot{\dot{\gamma}}(g) = \ddot{\gamma} \circ g \circ \ddot{\alpha} \in T \rightarrow T & \text{où } g : T^\# \rightarrow T^\# \end{cases}$$

On pourrait à partir de ce moment déjà utiliser les fonctions que l'on a définies pour faire l'évaluation abstraite du programme. Cependant, passer constamment du domaine concret au domaine abstrait fait que l'on exécuterait toutes les fonctions dans leur version concrète et donc qu'on aurait exactement les mêmes problèmes que ceux posés par l'exécution de tout le programme dans le domaine concret. On ne tirerait aucun avantage de l'utilisation du domaine abstrait comme le fait que l'évaluation se termine en temps fini. On va donc, pour chaque fonction, définir une sur-approximation abstraite.

Ainsi, pour $(A, \leq_A, \alpha_A) \in \{(C, \dot{\subseteq}, \dot{\alpha}), (E, \ddot{\subseteq}, \ddot{\alpha}), (T, \dot{\dot{\subseteq}}, \dot{\dot{\alpha}})\}$ et pour $f : A \rightarrow A$ on va définir $f^\# : A^\# \rightarrow A^\#$ telle que $\alpha_A(f) \leq_A f^\#$.

Plus intuitivement, on peut dire que cela signifie que tous les cas que faisaient apparaître l'application de f apparaissent aussi quand on applique $f^\#$.

On va donc maintenant définir la fonction $\text{APost}[\llbracket \text{stat}^{\prime\prime} \rrbracket]P$ qui va être la fonction abstraite associée à la fonction $\text{Post}[\llbracket \text{stat}^{\prime\prime} \rrbracket]$ que l'on a définie plus haut.

Cette fonction va donc représenter l'évolution de l'environnement abstrait à partir de l'environnement P quand on lit l'instruction $\text{stat}^{\prime\prime}$.

On va maintenant étudier cette fonction en définissant $\text{APost}[\llbracket X := v^{\prime\prime} \rrbracket]$. Pour cela, calculons une sur-approximation de $\dot{\dot{\alpha}} \circ \text{Post}[\llbracket X := v^{\prime\prime} \rrbracket]$:

$$\begin{aligned} \dot{\alpha} \circ \text{Post} \llbracket X := v' \rrbracket &= \ddot{\alpha} \circ \text{Post} \llbracket X := v' \rrbracket \circ \dot{\gamma} \\ \underline{\underline{\ddot{\alpha}}}^\# \lambda J. v \in \Omega \text{ ? } J \text{ ; } J[l' \leftarrow J[l]] \dot{\downarrow}^\# J[l] [X \leftarrow v' \cap^\# D^\#] &\quad (\text{c.f. }^3) \\ &\quad (\text{où } v' \text{ est l'abstraction de l'évaluation de l'expression arithmétique de } v) \end{aligned}$$

Comme pour ce qu'on a fait dans le cas des transitions entre environnements avec la définition de la fonction APost , on peut également définir d'autres fonctions abstraites qui vont représenter :

- l'évaluation abstraite d'expressions arithmétiques : $\text{Faexp}^\triangleright \llbracket \text{arith_expr} \rrbracket : C^\# \rightarrow D^\#$ (pour forward abstraction of arithmetic expressions),
- l'évaluation abstraite d'expressions booléennes : $\text{Abexp}^\triangleright \llbracket \text{bool_arith} \rrbracket : C^\# \rightarrow \{tt, ff\}$ (pour forward abstract of boolean expressions) (tt représentant le fait que l'expression booléenne peut être vraie et ff qu'elle peut être fausse). Dans le domaine abstrait, une expression booléenne peut être à la fois vraie et fausse ; par exemple si $a = \llbracket 1, 2 \rrbracket$ et que l'on teste si $a == 1$,
- $\text{Baexp}^\triangleleft \llbracket \text{arith_expr} \rrbracket(p) : C^\# \rightarrow C^\#$ où $p \in D^\#$ (pour backward abstraction of arithmetic expressions) qui retourne le contexte abstrait dans lequel l'évaluation abstraite de l'expression arithmétique est dans p . Cela est particulièrement utile lors de l'utilisation d'expressions booléennes.

Par exemple, si on exécute l'instruction suivante $l_c \text{if } (a == 1) \text{ then } l_c^1 \dots l_c^2 \text{ else } l_c^3 \dots l_c^4 l_c^5$ et qu'au point de programme l_c , $a = \llbracket 1, 2 \rrbracket$, il faut qu'au point de programme l_c^1 , $a = \llbracket 1 \rrbracket$. On voudra que le contexte en l_c^1 soit le contexte en l_c modifié de telle sorte que l'évaluation abstraite de l'expression arithmétique a soit $\llbracket 1 \rrbracket$, *i.e.* $\text{Baexp}^\triangleleft \llbracket a \rrbracket (\llbracket 1 \rrbracket) \{ \text{contexte abstrait en } l_c \}$.

On remarquera que le symbole \triangleright est associé aux fonctions qui permettent d'évaluer de façon abstraite une expression, on va vers l'avant : forward, alors que le symbole \triangleleft est associé aux fonctions qui permettent de retrouver un contexte abstrait à partir de domaines abstraits, on va vers l'arrière : backward.

3. Cette inégalité est prouvée dans [Cou99] page 65.

2 Apport personnel

2.1 La mise en place d'un domaine abstrait : les intervalles

Nous avons défini dans la partie précédente (au moins en partie) APost, mais nous n'avons pas défini les fonctions qui dépendaient du domaine abstrait que l'on a choisi : les intervalles. En effet, étant donné que l'on a choisi un domaine concret non relationnel (*i.e.* que les valeurs concrètes des variables ne sont pas considérées comme inter-dépendantes), on peut construire le domaine abstrait à partir d'une abstraction de l'ensemble des valeurs. Il est possible de montrer pour toutes ces fonctions que leur propriété de correction est bien vérifiée. On va commencer par définir les fonctions d'abstraction et de concrétisation :

$$\alpha(R) := \begin{cases} \perp^\sharp & \text{si } R = \emptyset \\ \llbracket \min(R), \max(R) \rrbracket & \text{si } R \neq \emptyset \end{cases}$$

$$\gamma(R^\sharp) := \begin{cases} \emptyset & \text{si } R^\sharp = \perp^\sharp \\ \{x \mid a \leq x \leq b\} & \text{si } R^\sharp = \llbracket a, b \rrbracket \end{cases}$$

On peut maintenant définir Faexp[▷] qui, on le rappelle, renvoie l'élément abstrait associé à l'évaluation d'une expression arithmétique à partir d'un contexte abstrait r :

$$\begin{aligned} \text{Faexp}^\triangleright \llbracket n \rrbracket r &:= \llbracket n, n \rrbracket \\ \text{Faexp}^\triangleright \llbracket X \rrbracket r &:= r(X) \\ \text{Faexp}^\triangleright \llbracket \circ A \rrbracket r &:= \alpha\left(\left\{ \circ c \mid c \in \text{Faexp}^\triangleright \llbracket A \rrbracket r \right\}\right) \\ \text{Faexp}^\triangleright \llbracket A_1 \diamond A_2 \rrbracket r &:= \alpha\left(\left\{ a_1 \diamond a_2 \mid (a_1, a_2) \in \text{Faexp}^\triangleright \llbracket A_1 \rrbracket r \times \text{Faexp}^\triangleright \llbracket A_2 \rrbracket r \right\}\right) \end{aligned}$$

Puis Abexp[▷] qui permet de renvoyer une abstraction de tous les contextes concrets compatibles avec un contexte abstrait donné en paramètre, qui satisfont également une condition booléenne donnée :

$$\begin{aligned} \text{Abexp}^\triangleright \llbracket \text{true} \rrbracket r &:= r \\ \text{Abexp}^\triangleright \llbracket \text{false} \rrbracket r &:= \perp^\sharp \\ \text{Abexp}^\triangleright \llbracket b_1 \wedge b_2 \rrbracket r &:= \text{Abexp}^\triangleright \llbracket b_1 \rrbracket r \dot{\cap}^\sharp \text{Abexp}^\triangleright \llbracket b_2 \rrbracket r \\ \text{Abexp}^\triangleright \llbracket b_1 \vee b_2 \rrbracket r &:= \text{Abexp}^\triangleright \llbracket b_1 \rrbracket r \dot{\cup}^\sharp \text{Abexp}^\triangleright \llbracket b_2 \rrbracket r \\ \text{Abexp}^\triangleright \llbracket !b_1 \rrbracket r &:= \text{Abexp}^\triangleright \llbracket \text{applyNotOnBoolExpr}(b_1) \rrbracket r \end{aligned}$$

(où applyNotOnBoolExpr est la fonction qui, à une expression booléenne, associe sa négation : on pousse les négations sur les feuilles pour les éliminer)

$$\text{Abexp}^\triangleright \llbracket a_1 \bowtie a_2 \rrbracket r := \text{Baexp}^\triangleleft \llbracket a_1 \rrbracket (r) d_1^\sharp \dot{\cap}^\sharp \text{Baexp}^\triangleleft \llbracket a_2 \rrbracket (r) d_2^\sharp$$

où $(d_1^\#, d_2^\#) = \bowtie^\# (\text{Faexp}^\triangleright \llbracket a_1 \rrbracket r, \text{Faexp}^\triangleright \llbracket a_2 \rrbracket r)$

avec $\bowtie^\# (d_1^\#, d_2^\#) = \left(\alpha \left(\{x \in \gamma(d_1^\#) \mid \forall y \in \gamma(d_2^\#), x \bowtie y\} \right), \right. \\ \left. \alpha \left(\{y \in \gamma(d_2^\#) \mid \forall x \in \gamma(d_1^\#), x \bowtie y\} \right) \right)$

On peut itérer l'appel à la fonction $\text{Abexp}^\triangleright$ jusqu'à trouver un point fixe pour trouver les contextes les plus précis mais qui englobe tout de même tous les cas. Dans ce cas, on utilisera une certaine fonction que l'on appelle *narrowing* qui permettra d'assurer que le point fixe sera trouvé en temps fini au prix d'un résultat qui pourra être une sur-approximation du contexte optimal (plus de cas que ce qui aurait été nécessaire).

On peut finalement expliciter $\text{Baexp}^\triangleleft$ qui permet de renvoyer l'abstraction de tous les contextes compatibles avec une contexte abstrait donné en argument, qui satisfait également la condition booléenne telle que l'évaluation abstraite d'une expression arithmétique soit dans un élément abstrait p :

$\text{Baexp}^\triangleleft \llbracket n \rrbracket (r)p := n \in p \text{ ? } r \text{ ; } \perp^\#$

$\text{Baexp}^\triangleleft \llbracket X \rrbracket (r)p := r[X \leftarrow (r(X) \sqcap^\# p)]$

$\text{Baexp}^\triangleleft \llbracket \circ A \rrbracket (r)p := \text{Baexp}^\triangleleft \llbracket A \rrbracket (r)(\circ^\triangleleft (\text{Faexp}^\triangleright \llbracket A \rrbracket r, p))$

avec $\circ^\triangleleft (q, p) := \alpha \left(\{c' \in p \mid \exists c \in q, c' = \circ(c)\} \right)$

$\text{Baexp}^\triangleleft \llbracket A_1 \diamond A_2 \rrbracket (r)p := \text{Baexp}^\triangleleft \llbracket A_1 \rrbracket (r)p_1 \sqcap^\# \text{Baexp}^\triangleleft \llbracket A_2 \rrbracket (r)p_2$

où $(p_1, p_2) = \diamond^\triangleleft (\text{Faexp}^\triangleright \llbracket A_1 \rrbracket r, \text{Faexp}^\triangleright \llbracket A_2 \rrbracket r, p)$

avec $\diamond^\triangleleft (q_1, q_2, p) := \left(\alpha \left(\{c_1 \in q_1 \mid \exists c_2 \in q_2, c_1 \diamond c_2 \in p\} \right), \right.$

$\left. \alpha \left(\{c_2 \in q_2 \mid \exists c_1 \in q_1, c_1 \diamond c_2 \in p\} \right) \right)$

Pour les pointeurs on va considérer que les valeurs abstraites peuvent être un ensemble de variables pointées. Par exemple, si on analyse ce bout de programme :

```

int a = [1, 2];
if (a == 1) {
    a = &b;
} else {
    a = &c;
}
*a = 1;
```

à la fin de l'analyse on aura la valeur abstraite de a qui sera $\text{AbstrPtr}(\{b, c\})$ et celles de b et de c seront l'union de leurs valeurs abstraites précédentes respectives et de $\llbracket 1, 1 \rrbracket = \{1\}$.

Je n'ai pas détaillé ici l'analyse abstraite de l'appel de fonctions mais cela sera fait dans la partie suivante.

2.2 Ajout des blocs assembleurs

Comme on l'a mentionné dans l'introduction, le but que l'on souhaite atteindre est maintenant d'ajouter le support des blocs assembleurs à l'analyseur. Pour pouvoir décrire une nouvelle sémantique ainsi que pour comprendre le fonctionnement des blocs assembleurs et des interactions entre C et assembleur j'ai utilisé les 6 premiers chapitres de [Int97a] ainsi que les pages de [Int97b] correspondant aux instructions que j'ai utilisées.

Pour ajouter le support des blocs assembleur, on va commencer par modifier le contexte. En effet, en assembleur, on peut avoir accès à des registres qui sont en fait les variables implémentées dans le processeur et qui permettent de faire les calculs. Étant donné qu'on se limite aux types C de 32 bits (car on considère un processeur 32 bits) : `int` et pointeurs, on ne va considérer que les registres `EAX`, `EBX`, `ECX`, `EDX`. Pour un bref rappel, le registre `EAX` est souvent utilisé pour stocker le résultat lors de l'appel d'une fonction tandis que les autres sont plus généralistes. On considérera ici que cette convention pour les résultats d'appels de fonction est suivie. On pourrait également stocker dans le contexte certains *flags* qui représentent des états du processeur (par exemple : est-ce que le dernier calcul arithmétique réalisé à renvoyé zéro?). Dans les vrais processeurs, ces *flags* sont stockés dans le registre `EFLAGS`. On ne va pas gérer ici les *flags* pour des raisons de simplicité.

On retrouvera également dans ce nouveau contexte deux piles différentes : la pile mémoire (ou memory stack) qui contient par exemple les valeurs des arguments passés lors de l'appel d'une fonction assembleur, et la pile d'appels (ou call stack) qui contient les adresses de retour des fonctions, pour qu'une fois que la fonction renvoie un résultat, le processeur puisse savoir où revenir pour continuer l'exécution. Pour des raisons pratiques, dans l'implémentation de l'analyseur abstrait que j'ai fait, les piles sont séparées du contexte mais se comportent de la même façon que si elles étaient incluses.

On va donc modifier un peu notre syntaxe pour inclure les blocs assembleurs en commençant par l'ajout des points mémoires assembleurs qui sont dans l'ensemble \mathcal{L}_a .

2.3 Nouvelle syntaxe incluant une partie des blocs assembleurs

${}^l\text{stat_c}$::= ${}^l\text{stat_c}$ ${}^{l_c}\text{asm} \{ {}^{l_a}$ ${}^{l_a}\}; {}^{l_c}$	(Anciens cas) (Entrée dans un bloc assembleur) (Sortie d'un bloc assembleur)
${}^{l_a}\text{stat_a}$::= ${}^{l_a}\text{stat_a}$ ${}^{l_a^1}\text{stat_a}$ l_a ${}^{l_a}\text{label} : {}^{l_a}$ ${}^{l_a}\text{MOV} (dst : \text{asm_mutable}), (src : \text{asm_value}) {}^{l_a}$ ${}^{l_a}\text{JMP} \text{asm_value} {}^{l_a}$ ${}^{l_a}\text{CALL} \text{asm_value} {}^{l_a}$ ${}^{l_a}\text{RET} {}^{l_a}$ ${}^{l_a}\text{PUSH} \text{asm_value} {}^{l_a}$ ${}^{l_a}\text{POP} \text{asm_mutable} {}^{l_a}$	
asm_value ::= $\text{dword} \mid \text{dword_r} \mid \text{name_var} \mid @\text{label} \mid [\text{dword_r}] \mid [\text{name_var}]$	
asm_mutable ::= $\text{dword_r} \mid \text{name_var}$ ${}^{l_a} \in \mathcal{L}_a \subset \mathcal{L}$	(Variable et registres mutables)
label ::= name_label	(Label assembleur)
dword ::= $\llbracket -2^{32}, 2^{32} - 1 \rrbracket$	(Constante de 32 bits)
dword_r ::= $\text{EAX} \mid \text{EBX} \mid \text{ECX} \mid \text{EDX}$	(Registre de 32 bits)
memory_stack ::= D stack	(Pile mémoire)
call_stack ::= $\{0, 1\}$ stack	(Pile contenant le type de l'appelant (1 si C, 0 si assembleur))

2.4 Évaluation des nouvelles instructions

On peut maintenant étendre τ pour que l'on puisse évaluer les instructions que l'on a ajoutées. Pour cela, on va utiliser C_a , qui est le contexte auquel on a rajouté les registres, la pile d'appels et la pile mémoire. On va, de plus, rajouter la pile d'appels et la pile mémoire à C puisque, maintenant, on peut avoir des appels à des fonctions C depuis l'assembleur et inversement.

On définit :

- $\text{get_asm_loc} : \text{asm_value} \rightarrow \mathcal{L} \cup \Omega$ qui à une expression « arithmétique » assembleur associe un point mémoire assembleur ou une erreur si l'expression ne correspond pas à un point mémoire.
- $\text{get_fun_stats} : \mathcal{L} \rightarrow \text{stat_c}$ qui à un point mémoire renvoie l'ensemble des instructions de la fonction correspondante.
- $\text{fill}(\text{get_arguments}(l), \rho)$ renvoie le contexte correspondant au contexte ρ dans lequel on a associé aux arguments de la fonction, située au point mémoire l , les premières valeurs

de pile mémoire (si la pile mémoire est trop petite, renvoyer un état d'erreur). La pile mémoire est donc, au moins en partie, renvoyée vidée.

- $\omega_r : C \times D \rightarrow \Omega$ qui est une exception qui permet de retourner le contexte et le résultat d'une fonction lorsqu'une variable a été retournée avant la fin de l'exécution de toutes les instructions d'une fonction. On pourrait penser que cela n'est pas nécessaire si l'on retourne toujours les valeurs à la fin des fonctions, mais en fait ce n'est pas suffisant. En effet, lorsque l'on utilise l'instruction **JMP**, de multiples instructions sont rajoutées dans la file d'exécution et celles-ci pourraient empêcher le bon fonctionnement du programme si elles étaient exécutées.

Pour simplifier l'écriture des valeurs de τ , on va considérer `other_in_error` définie telle que si l'on a

$$\left\{ \left(\langle l, \rho \rangle, \langle \dots, \dots \rangle \right) \mid l \in \mathcal{L}_{defined}, \rho \in C_{defined} \right\} \cup \text{other_in_error}$$

alors

$$\text{other_in_error} \triangleq \left\{ \left(\langle l, \rho \rangle, \omega \right) \mid l \in \mathcal{L}_{defined}, \rho \in C \setminus C_{defined} \right\}$$

Plus intuitivement, on ajoute à un système de transitions les transitions partant des mêmes points de programme mais pour d'autres contextes, vers un état d'erreur. Cela permet de ne pas avoir à définir explicitement tous les cas d'erreur mais à retourner un cas d'erreur générique.

On peut, maintenant que l'on a défini certains outils, expliciter l'extension de τ aux blocs assembleurs.

$$(15) \quad \tau \left[{}^{l_c} \text{asm} \{ {}^{l_a} \} \right] = \left\{ \left(\langle l_c, \rho_a|_C \rangle, \langle l_a, \rho_a \rangle \right) \mid \rho_a \in C_a \right\}$$

Pour l'entrée dans un bloc assembleur, on va seulement rajouter au précédent contexte les registres.

$$(16) \quad \tau \left[{}^{l_a}; {}^{l_c} \right] = \left\{ \left(\langle l_a, \rho_a \rangle, \langle l_c, \rho_a|_C \rangle \right) \mid \rho_a \in C_a \right\}$$

Pour la sortie d'un bloc assembleur, on va juste supprimer du contexte les registres. En n'autorisant pas les déclarations de variables après les blocs assembleurs on peut se rendre compte que la pile de mémoire ne changera pas dans le code C qui suivra.

$$(17) \quad \tau \left[{}^{l_a} \text{MOV} (dst : \text{asm_mutable}), (src : \text{asm_value}) {}^{l'_a} \right] = \left\{ \left(\langle l_a, \rho_a \rangle, \langle l'_a, \rho_a[dst \leftarrow v] \rangle \right) \mid \rho_a \in C_a, dst \in \text{Dom}(\rho_a) \right\}$$

\cup `other_in_error`

(où v est l'évaluation de src)

Pour la copie d'une valeur, on va seulement vérifier que la variable de destination est bien dans le contexte, puis seulement, mettre l'évaluation de l'expression source dans la destination.

$$(18) \quad \tau \left[{}^{l_a} \text{JMP} (dst : \text{asm_value}) {}^{l'_a} \right] = \left\{ \left(\langle l_a, \rho_a \rangle, \langle l'_a, \rho_a \rangle \right) \mid \rho_a \in C_a, (l'_a := \text{get_asm_loc}(dst)) \notin \Omega \right\} \\ \cup \left(l'_a := \text{get_asm_loc}(dst) \in \Omega \right) ? 1_{\mathcal{L}_a \times C_a} \text{ ; } \tau \left[\text{get_fun_stats}(l'_a) \right]$$

$$\circ \{(\omega_r(\rho), \omega_r(\rho)) \mid \rho \in C \cup C_a\}$$

∪ `other_in_error`

Pour le saut, on va d'abord vérifier que l'on a bien en argument un point de programme (on ne peut sauter que vers un point de programme assembleur car on ne peut pas savoir à l'avance la taille que le code C peut prendre et donc où l'on sauterait). On impose ensuite la condition de passer par un retour de fonction pour que la suite normale du code ne soit pas considérée. C'est pour cela qu'on utilise la composition et non pas l'union.

$$(19) \quad \tau^{[l_a \mathbf{CALL} \text{ dest } l'_a]} =$$

$$\left\{ \left(\langle l_a, \rho_a \rangle, \langle l^1, \rho^1 \rangle \right) \mid \rho_a \in C_a, \right.$$

$$\left. \begin{aligned} & (l' \in \mathcal{L}_c ? \rho^1 := \text{fill}(\text{get_arguments}(l^1), \rho^1), (\langle l_a, \rho_a \rangle, \langle l^1, \rho^1 \rangle) \in \tau^{[l_a]; l^1}), \\ & \mathbf{i} \rho^1 := \rho_a), \\ & \rho^1[\text{call_stack}].\text{push}(0, l'_a), \rho^1[\text{memory_stack}].\text{push}(\text{AbstrLoc}(l'_a)) \\ & \left. \right\} \\ & \cup \tau[\text{get_fun_stats}(l^1)] \\ & \circ \left(\left\{ (\omega_r(\rho), \langle l'_a, \rho \rangle) \mid \rho[\text{memory_stack}].\text{pop}() = \text{AbstrLoc}(l'_a) \right\} \right. \\ & \quad \left. \cup \left\{ (\omega_r(\rho), \langle l'_a, \rho \rangle) \mid (l''_a := \rho[\text{memory_stack}].\text{pop}()) \neq \text{AbstrLoc}(l'_a) \right\} \right) \\ & \quad \circ \tau[\text{get_fun_stats}(l''_a)] \\ & \cup \left(\left\{ (\langle l_a, \rho_a \rangle, \langle l_a, \rho_a \rangle) \mid \rho_a \in C_a \right\} \right. \\ & \quad \left. \cup \left\{ (\langle l, \rho_c \rangle, \langle l, \rho_a \rangle) \mid l \in \mathcal{L}_c, (\langle l, \rho_c \rangle, \langle l, \rho_a \rangle) \in \tau^{[l \mathbf{asm}(l)]} \right\} \right) \\ & \text{si } (l^1 := \text{get_asm_loc}(\text{dest})) \notin \Omega \\ & \cup \text{other_in_error} \end{aligned}$$

Pour l'appel de fonctions depuis l'assembleur, on va commencer par considérer deux cas : si la fonction appelée est une fonction C, on va vider la pile pour remplir le contexte avec les arguments et on va quitter le bloc assembleur. Sinon on va juste transmettre le contexte actuel. Puis, on va remplir les deux piles avec l'appel de fonction en cours. Ensuite, on exécute la fonction et on vérifie à la fin que l'adresse de retour est bien l'_a . Si ce n'est pas le cas, on saute à l'autre adresse (seulement dans le cas d'une fonction assembleur car sinon on ne peut pas modifier l'adresse de retour, puisqu'on n'a pas fait d'hypothèse sur la pile mémoire dans le cas du code C).

On utilise la composition et non pas l'union à certains endroits pour pouvoir retrouver l'appel correspondant à un retour de fonction.

En réalité, certains cas ne sont pas gérés par cette définition mais ils sont un peu trop complexes pour que l'on puisse écrire la fonction de manière formelle. Un exemple de cas non-géré est le fait que l'on ne supprime pas les variables créées à la fin de l'appel d'une fonction C. De plus, on ne gère pas ici le cas où une variable a le même nom qu'une autre déjà existante (par exemple avec les fonctions récursives) et le programme permet d'accéder aux variables définies auparavant alors qu'elles ne sont normalement pas disponibles (si on vidait le contexte, en l'état, on empêcherait l'accès aux variables pointées par les arguments de la fonction). En revanche, tous ces cas sont gérés par le programme que j'ai conçu et dont vous pouvez retrouver les

sources en annexe.

Le cas des appels de fonctions en C n'est pas traité ici car il ressemble dans sa quasi-globalité au précédent.

$$(20) \quad \tau \left[{}^{l_c} \mathbf{return} \text{ name_var} {}^{l_c} \right] = \\ \left\{ \left(\langle l_c, \rho_c \rangle, \omega_r(\rho_c, \rho_c[\text{name_var}]) \right) \mid \rho_c \in C_c, \text{call_stack.front}() = 1 \right\} \\ \cup \text{other_in_error}$$

Lorsqu'une fonction renvoie un résultat et que la fonction a été appelée depuis du code C, on va retourner un état d'erreur qui sera rattrapé par τ de l'instruction appelante ou par un **if then else** avant d'être renvoyée (on ne considère ici que le cas où les deux branches renvoient en même temps, même si le cas général est traité par le programme que j'ai réalisé et dont le code source se trouve en annexe).

Le cas du ${}^{l_a} \mathbf{RET} {}^{l_a}$ n'est pas décrit ici puisqu'il ressemble dans sa quasi-globalité au cas précédent.

On a donc décrit ici une grande partie des opérations des blocs assembleurs étant donné que la plupart des autres opérations peuvent être simplement décrites comme composition des fonctions déjà décrites (en particulier **JMP** et **MOV**). Pour trouver leurs versions abstraites, il suffit d'utiliser la même technique que précédemment, vous trouverez en particulier le cas du **JMP** dans les annexes page 23.

2.5 Fonctionnalités supplémentaires implémentées dans mon analyseur

J'ai ajouté à mon analyseur abstrait certaines fonctionnalités qui n'ont pas été décrites dans les parties ci-dessus, leur présentation rendant plus complexe la compréhension des concepts mis en jeux.

Tout d'abord, il est possible dans mon analyseur de renvoyer un résultat avant la fin de la fonction, ou seulement d'un côté d'un **if then else**, ou bien dans un **while**, et tout cela en gardant la propriété de terminaison en temps fini. En effet, je ne garde pas seulement le type de l'appelant dans la pile d'appels, mais aussi le résultat renvoyé et le contexte associé.

Une syntaxe plus riche que celle présentée ici est disponible avec, par exemple, la présence d'opérateurs ternaires, de boucles **for** ou d'opérations plus complexes sur les pointeurs. En revanche, lors de la lecture du programme, certaines instructions sont transformées en instructions équivalentes déjà implémentées. Cela permet de réduire le nombre de cas à considérer et de limiter le nombre d'erreurs de l'analyseur.

Il est possible qu'une erreur trouvée par mon analyseur soit en fait un faux positif. Alors, pour que l'exécution puisse continuer, il vaut mieux afficher une alarme puis continuer le calcul avec une approximation correcte des transitions entre points de programme sans erreurs.

Je permets aussi au programme de modifier grandement le flot du programme, comme les exemples ci-dessous le montrent :

Listing 1 – Saut dans une branche éloignée

```
int a;
if (a <= 0) {
    if (a >= -3) {
        return 0;
    } else {
        a = 1;
        asm { JMP @autreBranche };
    }
} else {
    a = 2;
    asm { autreBranche: };
}
return a;
```

Listing 2 – Changement du point de retour

```
int test() {
    asm {
        otherFunction://(int a)
        POP eax
        JMP @finAppel
    };
    asm {
        maFonction://(int a)
        POP edx
        PUSH @otherFunction
        RET
    };
    asm {
        maFonctionPrincipale://()
        PUSH 42
        CALL @maFonction
        finAppel:
        RET
    };
}

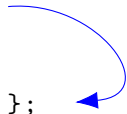
int main() {
    int a;
    a = maFonctionPrincipale();
    return a;
}
```

Listing 3 – Saut dans une boucle

```

int main() {
    int a;
    a = [0,2];
    asm { JMP @dansWhile };
    a = a / 0;
    while (a < 2) {
        asm { dansWhile: };
        if (a == 0) {
            return 1;
        }
        a = a + 1;
    }
    return 0;
}

```



2.6 Contraintes de mon analyseur

J'ai quelques fois évoqué dans ce rapport des contraintes que je m'imposais, mais je ne les ai pas décrites exhaustivement. Les voici :

- Il n'est pas possible d'accéder aux variables de la pile mémoire qui ont été introduites avant un bloc de code C. En effet, le code C peut légitimement écrire dans la pile mémoire.
- Lors d'un **JMP**, il n'est plus possible d'accéder aux variables C. En effet, il n'existe pas de convention pour l'ordre des arguments des fonctions C dans la pile mémoire.
- Une fonction appelée depuis le code C (resp. assembleur) devra être retournée dans du code C (resp. assembleur). En effet, les appels de fonctions en C, dans leur équivalent compilé en assembleur, ajoutent automatiquement un prélude et un postlude qui permettent d'ajouter puis de supprimer les arguments de la pile mémoire. Or, si on ne prend que l'un de ces prélude/postlude, on risque de faire des erreurs dans la pile mémoire. C'est pour cela que dans la pile d'appels, on stocke quel type de code a fait l'appel.
- L'utilisation de la récursivité n'assure pas la terminaison de l'analyse. En général, son usage est interdit dans l'embarqué mais je l'ai tout de même implémentée pour qu'un programme qui l'utilise de façon très modérée puisse tout de même être prouvé sans erreur.
- L'utilisation de **JMP** peut également empêcher la terminaison de l'analyse (c'est de la récursivité cachée). En revanche, on peut considérer que l'on n'a pas besoin de faire de boucles assembleur puisqu'on peut simplement les faire en C puis mettre le contenu de la boucle dans un bloc assembleur.

Conclusion

Mon stage m'a permis de comprendre le formalisme de l'interprétation abstraite puis d'appliquer les principes acquis pour créer du formalisme qui supporte les blocs assembleurs. J'ai pu utiliser ces connaissances pour créer un analyseur prenant en charge ces nouveaux types d'instructions.

En formalisant les différentes opérations des blocs assembleurs dans le code C, j'ai pu montrer que l'analyse abstraite de tels types de code est possible. Je l'ai implémentée de la manière la plus générale possible. La seule restriction est de ne pas utiliser certains types d'instructions bien spécifiques qui ne sont pas indispensables à l'écriture de programmes.

Maintenant que ce travail est fait, on peut penser à formaliser puis implémenter une partie moins restreinte des blocs assembleurs, par exemple la gestion des segments.

Références

- [Cou99] P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999. <https://www.di.ens.fr/~cousot/COUSOTpapers/Marktoberdorf98.shtml>, [Visité pour la dernière fois le 22 juillet 2019].
- [Fer05] Jérôme Feret. The arithmetic-geometric progression abstract domain. 2005. <https://www.di.ens.fr/~feret/publication/vmcai2005.ps.gz>, [Visité pour la dernière fois le 22 juillet 2019].
- [Int97a] Intel. *Intel Architecture Software Developer's Manual, Volume 1 : Basic Architecture*. 1997. <http://www.cs.cornell.edu/courses/cs412/2000SP/resources/IntelArchitectureVol.1.PDF>, [Visité pour la dernière fois le 22 juillet 2019].
- [Int97b] Intel. *Intel Architecture Software Developer's Manual, Volume 2 : Instruction Set Reference*. 1997. <http://www.cs.cornell.edu/courses/cs412/2000SP/resources/IntelArchitectureVol.2.PDF>, [Visité pour la dernière fois le 22 juillet 2019].
- [Min13] Antoine Miné. *Static analysis by abstract interpretation of concurrent programs*. Habilitation à diriger des recherches, Ecole Normale Supérieure de Paris - ENS Paris, November 2013, <https://tel.archives-ouvertes.fr/tel-00903447/file/hdr-compact-col.pdf>. [Visité pour la dernière fois le 22 juillet 2019].

Annexe 1 : Programme réalisé

Tout le code que j'ai produit et qui consiste en un analyseur abstrait d'une partie du code C avec des blocs assembleurs se trouve à l'adresse <https://github.com/jboillot/analyser>. La documentation associée à ce programme et générée automatiquement se trouve à l'adresse <https://jboillot.github.io/analyser/>.

Voici une brève description des différents modules du projet :

- **Analyser** : Module d'entrée du programme. Parse les arguments donnés à l'analyseur puis analyse le programme.
- **Analysis** : Permet de lancer l'analyse.
- **Types** : Ensemble des types utilisés dans le programme.
- **Options** : État des options que l'on spécifie à l'analyseur.
- **SpecificAbstract** : Ensemble de fonctions permettant l'analyse abstraite spécifique au domaine abstrait choisi. On peut notamment retrouver les fonctions correspondant à l'abstraction, au *meet* et au *join*.
- **GenericAbstract** : Ensemble de fonctions permettant l'analyse abstraite non-spécifique au domaine abstrait choisi. On peut notamment retrouver les fonctions correspondant à APost, Faexp, Baexp...
- **Exceptions** : Module permettant de signaler les différentes erreurs potentielles rencontrées.
- **SpecificVariables** : Ensemble de variables correspondant aux registres dans C_a . Celles-ci sont considérées différemment des autres variables car on ne peut pas y avoir accès comme aux variables.
- **Utils** : Module regroupant des fonctions utilitaires. On peut notamment y trouver une fonction permettant d'ajouter des points de programme aux différentes instructions du programme analysé.
- **Parser** : Module permettant de parser les programmes à analyser.
- **Lexer** : Module permettant au **Parser** de comprendre les différents symboles du langage que l'on analyse.

Annexe 2 : Contexte institutionnel et social du stage

Mon stage s'est déroulé dans l'équipe ANTIQUE (pour ANalyse staTIQUE par interprétation abstraite). Le laboratoire est situé à l'ENS de Paris. L'équipe est co-gérée par le CNRS, l'ENS et l'INRIA. Les travaux de l'équipe vont de la création d'un analyseur statique à la bio-informatique, mais avec comme constante l'utilisation de l'analyse abstraite. Ce domaine de recherche a été créé par Radhia et Patrick COUSOT, en particulier dans l'équipe ABSTRACTION dont ANTIQUE est le successeur.

J'ai pu suivre une matinée d'intégration des nouveaux arrivants à l'INRIA Paris. J'ai également assisté à la soutenance de thèse d'Andreea Beica qui avait pour sujet « Parallel Simulation of Modular Complex Systems ».

J'étais dans un bureau dans lequel travaillaient deux doctorants, un ingénieur de recherche ainsi que d'autres stagiaires. Ce cadre était intéressant et permettait d'échanger de manière informelle sur les différents thèmes du laboratoire et de bénéficier de leurs compétences.

Je profite de cette annexe pour remercier mes tuteurs Marc CHEVALIER et Jérôme FERET qui m'ont beaucoup aidé durant le stage, pendant la rédaction du rapport ainsi que pour la préparation de la présentation. J'ai pu, grâce à eux, apprendre de multiples concepts de l'interprétation abstraite qui m'ont beaucoup intéressé.

Annexe 3 : Démonstrations

Démonstration 1

On veut montrer que $\bigcup_{i \leq n} t^i = \lambda r. (1_S \cup t \circ r)^n(\emptyset)$ avec t une relation binaire, élément de $\mathcal{P}(S)$, et $n \in \mathbb{N}$.

Faisons-le par récurrence sur $n \in \mathbb{N}$:

— **Initialisation** : Pour $n = 0$,

$$\begin{aligned} \bigcup_{i \leq n} t^i &= t^0 \\ &\triangleq 1_S \\ &= (\lambda r. 1_S \cup t \circ r)^0(\emptyset) \end{aligned}$$

L'initialisation est vérifiée.

— **Hérédité** : Supposons pour $n \in \mathbb{N}$ fixé que la propriété que l'on veut démontrer est vraie et montrons qu'elle reste vraie au rang suivant :

$$\begin{aligned} (\lambda r. 1_S \cup t \circ r)^{n+1}(\emptyset) &= (\lambda r. 1_S \cup t \circ r) \circ (\lambda r. 1_S \cup t \circ r)^n(\emptyset) && \text{(par composition de fonctions)} \\ &= (1_S \cup t \circ (\lambda r. 1_S \cup t \circ r)^n(\emptyset)) \\ &= (1_S \cup t \circ \bigcup_{i \leq n} t^i) && \text{(par hypothèse de récurrence)} \\ &= (1_S \cup \bigcup_{1 \leq i \leq n+1} t^i) \\ &= \bigcup_{i \leq n+1} t^i && \text{(car } t^0 = 1_S) \end{aligned}$$

L'hérédité est donc vérifiée pour tout $n \in \mathbb{N}$.

On a ainsi démontré la propriété voulue.

Démonstration 2

On veut déterminer une expression pour $\text{APost}[[l_a \mathbf{JMP} \text{dst} l'_a]]$:

Commençons par calculer la clôture réflexive transitive de τ associée au saut :

On a pris comme contrainte de ne pas utiliser de récursivité. Ainsi, si l'on est à un point de programme l_a et que l'on saute à un autre point de programme, on n'y retournera pas durant l'exécution de l'instruction et donc on peut considérer que le clôture réflexive transitive de $\tau[[l_a \mathbf{JMP} \text{dst} l'_a]]$ est l'union de celle de $\{(\langle l_a, \rho_a \rangle, \langle l'_a, \rho_a \rangle) \mid \rho_a \in C_a, l'_a := \text{get_asm_loc}(\text{dst}) \notin \Omega\}$ et de $(l'_a := \text{get_asm_loc}(\text{dst}) \in \Omega) \ ? \ 1_{\mathcal{L}_a \times C_a} \ ; \ \tau[\text{get_fun_stat}(l'_a)]$ ainsi que des différentes transitions vers des états d'erreur qui ne pourront être exécutés qu'une seule fois (car il n'existe pas de transition permettant de sortir de l'état d'erreur qui ne soit pas rattrapée avant).

On peut alors poser, d'une façon similaire à la définition de $\text{APost}[[l_a X := v l'_a]]$ à la page 65 de [Cou99] :

$$\begin{aligned} \text{APost}[[l_a \mathbf{JMP} \text{dst} l'_a]] &= \lambda \check{r}^\# . l'_a := \text{get_asm_loc}(\text{dst}) \in \Omega \\ &\quad ? \check{r}^\# \\ &\quad ; \left(\check{r}^\# [l'_a \leftarrow \check{r}^\# [l'_a] \dot{\cup} \check{r}^\# [l_a]] \right) \circ \text{APost}[[\text{get_fun_stat}(l'_a)]] \end{aligned}$$

Glossaire

1_S : Relation binaire correspondant à l'identité : $1_S = \{\langle s, s \rangle \mid s \in S\}$.

$\lambda a. b$: Fonction qui à un argument a associe l'expression b .

$a ? b \text{ ; } c$: Si a est vrai alors cette expression vaut b , sinon c .

Clôture réflexive transitive : Plus petite relation réflexive et transitive qui contient la relation initiale. C'est en fait le lfp d'une certaine fonction associée à la relation binaire. Cette fonction est explicitée dans l'équation 11.

Dom : Retourne l'ensemble sur lequel est défini une fonction.

Fonctions avec le symbole \triangleleft : Fonctions abstraites permettant, à partir de contraintes sur des éléments abstraits, de retrouver un contexte abstrait.

Fonctions avec le symbole \triangleright : Fonctions abstraites permettant, à partir d'un contexte abstrait, de retrouver des éléments abstraits.

lfp : Plus petit point fixe d'une fonction (least fix-point), *i.e.* que l'image de ce point par la fonction est égale à l'antécédent. Pour assurer la terminaison de cette opération, on utilise une fonction que l'on appelle *widening* et qui permet de trouver une sur-approximation du point fixe en temps fini.

Opérateurs et variables avec le symbole \cdot : Opérateurs et variables agissant sur les contextes.

Opérateurs et variables avec le symbole $\ddot{\cdot}$: Opérateurs et variables agissant sur les environnements.

Opérateurs et variables avec le symbole $\dot{\cdot}$: Opérateurs et variables agissant sur les transitions entre environnements.

Opérateurs et variables avec le symbole $\#$: Opérateurs et variables agissant sur des éléments abstraits.

Relation binaire : Proposition qui lie ensemble des éléments de deux ensembles. On la représente sous la forme d'un ensemble de paires d'éléments.