Analyse Statique par Interprétation Abstraite

Xavier Rival

EP INRIA ABSTRACTION
45, Rue d'Ulm
75 230, Paris

RÉSUMÉ. *L'interprétation abstraite a été proposée comme un cadre générique permettant de formaliser, dériver et prouver des analyses statiques par approximation conservative, c'est à dire capables de démontrer une propriété donnée pour un sous-ensemble des programmes vérifiant celle-ci. Tout d'abord, nous décrivons pas à pas les principales étapes dans la conception de telles analyses, du choix d'un modèle des programmes à étudier à la formalisation des algorithmes d'analyse, en passant par le choix d'une ensemble de prédicats à utiliser. Ensuite, nous présentons quelques applications récentes de ces techniques, dans divers domaines de l'informatique*

ABSTRACT. *Abstract interpretation was introduced as a generic framework, which allows to formalize, derive and prove static analyses, which are based on conservative approximation, that is, which are able to establish a fixed property for a subset of the programs which actually satisfy it. First, we provide a step by step description of the main steps in the design of such analyses, from the choice of a model for the programs to study, to the formalization of the analysis algorithm, including the choice of a set of predicates to use for the analysis. Then, we present a panel of recent applications of these techniques to various areas of computer science.*

MOTS-CLÉS : *Interprétation Abstraite, Analyse Statique, Méthodes Formelles*

KEYWORDS: *Abstract Interpretation, Static Analysis, Formal Methods*

Technique et science informatiques, pages 1 à 15

1. Introduction

Depuis l'apparatition des premiers ordinateurs, se pose la question de prouver, de manière aussi automatique que possible, la correction de programmes informatiques, en particuliers lorsque ceux-ci sont integrés à des systèmes critiques (e.g., embarqués). Parmi les propriétés de correction les plus importantes, on peut citer l'absence d'erreurs à l'exécution (division par zéro, erreurs arithmétiques ou d'accés à la mémoire). La plupart de ces propriétés sont indécidables, et il est donc impossible de trouver un programme P_a à même de décider pour, n'importe quel programme P, si celui-ci est sûr. Par conséquent, plusieurs compromis ont été proposés : on peut par exemple se limiter à des systèmes finis (et donc décidables), ou bien renoncer à l'automatisation des preuves. Une autre solution consiste à tolérer des résultats qui ne correspondent pas exactement à "P est sûr" ou "P n'est pas sûr" :

– une technique *complète* détecte tous les programmes qui sont sûrs ;

– une technique *correcte* détecte tous les programmes qui ne sont pas sûrs.

Dans la plupart des cas, comme par exemple, pour les programmes critiques, il est important de procéder à des analyses *correctes*, mais la complètude n'est pas primordiale : une *fausse alarme* (i.e., opération élémentaire que P_a ne peut prouver sûre) entraînerait tout au plus un retard dans la certification du programme considéré. Par contre, la correction est cruciale, car tout comportement incorrect pourrait par exemple entraîner une catastrophe, dans le cas d'un logiciel critique embarqué.

La théorie de l'interprétation abstraite (Cousot *et al.*, 1977; Cousot, 1978) permet de formaliser le calcul de telles approximations, dites "sûres", par *analyse statique*. Les principes fondamentaux de cette théorie sont les suivants :

– tout d'abord, une sémantique de référence, dite *sémantique concrête* doit être choisie, afin d'exprimer les propriétés à prouver ;

– ensuite, on fixe une *sémantique abstraite* et on décrit formellement le rapport de celle-ci avec la sémantique abstraite, de manière à exprimer que telle propriété abstraite entraîne telle propriété concrête ;

– enfin, on met au point un algorithme de calcul d'une approximation de la sémantique abstraite, en utilisant des techniques d'*approximation sûre* et d'*élargissement*.

À chaque étape, on s'efforce de préserver un bon rapport précision/efficacité, afin d'obtenir des analyses permettant de prouver les propriétés considérées.

L'objectif de cet article est de décrire la conception d'une telle analyse, et de présenter des exemples.

Dans la première partie de cet article, nous décrivons les principales étapes de la conception d'un analyseur statique par interprétation abstraite visant à prouver l'absence d'erreurs arithmétiques dans des programmes impératifs simples, du choix de la sémantique concrête dans la Section 2 jusqu'à la mise au point de l'outil d'analyse en Section 6. Par ailleurs, nous présenterons dans la Section 7 quelques applications récentes de ce cadre de travail.

2. Le point de départ : la sémantique concrête

Avant de parler de l'analyse statique à proprement parler, nous devons introduire le langage étudié, sa sémantique concrête, ainsi que les propriétés que nous souhaitons étudier. Étant donné un ensemble fini $\mathbb{V}$ de variables (nous supposerons que $\mathbb{V}$ a N éléments), nous nous proposons de considérer le langage suivant :

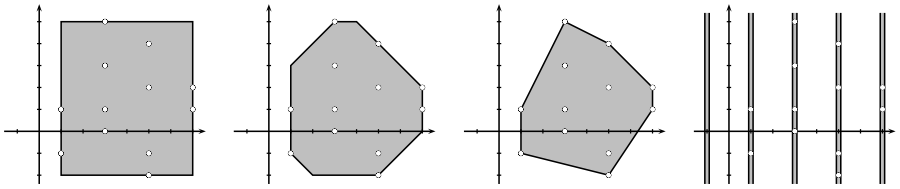
$\mathtt{e}$	$::=$	$n\,(n \in \mathbb{Z}) \mid x\,(x \in \mathbb{V}) \mid \mathtt{e} + \mathtt{e} \mid \mathtt{e} - \mathtt{e} \mid \mathtt{e} \star \mathtt{e}$	expression arithmétique
$\mathtt{c}$	$::=$	$\mathtt{e} = 0 \mid \mathtt{e} \neq 0 \mid \mathtt{e} \geq 0 \mid \ldots$	condition
$\mathtt{i}$	$::=$	$x := \mathtt{e} \mid$ **if** $\mathtt{c}$ **then** $\mathtt{b}$ **else** $\mathtt{b} \mid$ **while** $\mathtt{c}$ **do** $\mathtt{b}$	instuction
$\mathtt{b}$	$::=$	$\mathtt{i}; \ldots; \mathtt{i};$	bloc de code

Nous considérerons des programmes de la forme **assume**(F_{pre}); $\mathtt{b}$; **assert**(F_{post}), où F_{pre} et F_{post} sont des formules logiques sur les valeurs des variables du programme, spécifiant respectivement la *pré-condition* (i.e., hypothèse sur l'état initial) et la *post-condition* (i.e., propriété devant être vérifiée à la fin de l'exécution du programme pour que celle-ci soit considérée comme correcte). Par abus de notation, nous assimilons F_{pre} et F_{post} aux ensembles d'états mémoire qui les satisfont. Dans la suite, note objectif sera justement, étant donné un tel programme, de prouver que la post-condition est vérifiée pour tout état final. Pour exprimer cette propriété, nous définissons tout d'abord la sémantique concrête, qui prendra la forme d'une sémantique *dénotationnelle*, c'est à dire décrivant les comportements des instructions à l'aide de fonctions prenant un état initial et renvoyant un état final.

Un état mémoire est une fonction $\rho : \mathbb{V} \to \mathbb{Z}$; nous notons $\mathbb{M}$ l'ensemble de ces états, défini par $\mathbb{M} = \mathcal{P}(\mathbb{V} \to \mathbb{Z})$. L'évaluation d'une expression $\mathtt{e}$ se fera à l'aide d'une fonction $[\![\mathtt{e}]\!]$ qui prend en argument un état mémoire ρ et renvoie la valeur de $\mathtt{e}$ dans l'environnement ρ. Idéalement, nous voudrions définir la sémantique d'une instruction $\mathtt{i}$ comme une fonction $[\![\mathtt{i}]\!] : \mathbb{M} \to \mathbb{M}$ prenant un état initial et retournant l'état final, néanmoins ce choix ne convient pas, car certaines instructions, comme les boucles peuvent ne pas terminer. Par conséquent, nous optons pour une définition légérement plus générale, où la émantique d'une instruction $\mathtt{i}$ (resp., d'un bloc $\mathtt{b}$) est une fonction $[\![\mathtt{i}]\!] : \mathcal{P}(\mathbb{M}) \to \mathcal{P}(\mathbb{M})$ (resp., $[\![\mathtt{b}]\!] : \mathcal{P}(\mathbb{M}) \to \mathcal{P}(\mathbb{M})$) prenant un ensemble d'état initiaux $\mathcal{E}$ et retournant l'ensemble des états finaux atteignables à partir de $\mathcal{E}$. Les principales règles permettant de décrire ces fonctions sont les suivantes :

$$
\begin{array}{l}
[\![n]\!](\rho) = n \qquad [\![x]\!](\rho) = \rho(x) \qquad [\![\mathtt{e}_0 + \mathtt{e}_1]\!](\rho) = [\![\mathtt{e}_0]\!](\rho) + [\![\mathtt{e}_1]\!](\rho) \\
[\![x := \mathtt{e}]\!](\mathcal{E}) = \{\rho[x \leftarrow [\![\mathtt{e}]\!](\rho)] \mid \rho \in \mathcal{E}\} \qquad [\![\mathtt{i}_0; \mathtt{i}_1]\!](\mathcal{E}) = [\![\mathtt{i}_1]\!]([\![\mathtt{i}_0]\!](\mathcal{E})) \\
\quad [\![\textbf{if } \mathtt{e} = 0 \textbf{ then } \mathtt{i}_0 \textbf{ else } \mathtt{i}_1]\!](\mathcal{E}) = \\
\qquad [\![\mathtt{i}_0]\!](\{\rho \in \mathcal{E} \mid [\![\mathtt{e}]\!](\rho) = 0\}) \cup [\![\mathtt{i}_1]\!](\{\rho \in \mathcal{E} \mid [\![\mathtt{e}]\!](\rho) \neq 0\}) \\
[\![\textbf{while } \mathtt{e} \neq 0 \textbf{ do } \mathtt{b}]\!](\mathcal{E}) = \\
\quad \{\rho_n \in \mathbb{M} \mid \exists \rho_0, \rho_1, \ldots, \rho_{n-1} \in \mathbb{M}, \text{ t. q. } \rho_0 \in \mathcal{E}, \\
\qquad [\![\mathtt{e}]\!](\rho_n) = 0 \text{ et } \forall k < n, [\![\mathtt{e}]\!](\rho_k) \neq 0 \text{ et } \rho_{k+1} \in [\![\mathtt{b}]\!](\rho_k)\}
\end{array}
$$

Cette sémantique définit une infinité d'exécutions. À présent, il devient possible de formaliser la propriété que l'on veut prouver, en utilisant notre sémantique concrête.



(a) Intervalles (b) Octogones (c) Polyhèdres (d) Congruences

Figure 1. *Quelques domaines abstraits numériques (*$\mathbb{V} = \{x, y\}$*)*

Ainsi, soit un programme **assume**(F_{pre}); $\mathtt{b}$; **assert**(F_{post}). Alors, on souhaite prouver la propriété de *correction partielle* (c'est à dire qui ignore les exécutions qui ne terminent pas) :

$$[\![\mathtt{b}]\!](\mathbb{M}_{\mathrm{pre}}) \subseteq \mathbb{M}_{\mathrm{post}}. \qquad [1]$$

3. Abstraction et preuve de propriétés concrêtes

La propriété 1 est généralement indécidable. Nous allons donc mettre en oeuvre le processus suivant : tout d'abord, nous choisirons un ensemble d'état $\mathbb{M}^{\mathrm{app}}_{\mathrm{pre}}$ contenant tous les états initiaux (i.e., $\mathbb{M}_{\mathrm{pre}} \subseteq \mathbb{M}^{\mathrm{app}}_{\mathrm{pre}}$) ; ensuite, nous calculerons à partir de $\mathbb{M}^{\mathrm{app}}_{\mathrm{pre}}$ un sur-ensemble $\mathbb{M}^{\mathrm{app}}_{\mathrm{post}}$ des états finaux du programme (i.e., tel que $[\![\mathtt{b}]\!](\mathbb{M}^{\mathrm{app}}_{\mathrm{pre}}) \subseteq \mathbb{M}^{\mathrm{app}}_{\mathrm{post}}$) ; enfin, nous montrerons que $\mathbb{M}^{\mathrm{app}}_{\mathrm{post}}$ est inclus dans $\mathbb{M}_{\mathrm{post}}$. Cela constitue une preuve valide que le programme est correct au sens de 1.

En pratique, ces trois étapes seront effectuées automatiquement, une *abstraction* ayant été fixée au préalable. Plus précisément, on choisira un *domaine abstrait*, c'est à dire un ensemble $\mathbb{M}^\sharp$ de prédicats (ou de représentations en machine de prédicats), muni de structures et d'algorithmes adaptés. Par exemple, un ensemble d'états pourra être sur-approximé par une fonction associant à chaque variable un intervalle contenant toutes les valeurs que celle-ci peut prendre (Cousot *et al.*, 1977), ou par bien un polyhèdre convexe P en dimension N (où N est le nombre de variables) tel que pour tout ρ dans E, ρ interprété comme un N-uplet corresponde à un point dans P (Cousot *et al.*, 1978). De très nombreux tels domaines ont été définis, tels que le domaine des congruences (Granger, 1989) ou les octogones (Miné, 2006). Quelques exemples de domaines numériques sont présentés dans la figure 1, dans le cas où nous ne consdiérons que deux variables.

Pour prouver la correction des analyses, il est important d'exprimer la relation entre propriétés "concrêtes" et propriétés "abstraites". Pour cela, nous utiliserons une *fonction de concrétisation* $\gamma : \mathbb{M}^\sharp \to \mathcal{P}(\mathbb{M})$: intuitivement $\gamma(d^\sharp)$ décrit l'ensemble des états concrêts qui satisfont la propriété abstraite $d^\sharp$. Dans certains cas, on utilise aussi une *fonction d'abstraction* (souvent notée α), qui associe à tout élément de $\mathcal{P}(\mathbb{M})$ l'élément abstrait le plus précis qui le sur-approxime ; toutefois, certains domaines

abstraits n'ont pas de fonction d'abstraction, certains éléments concrets n'ayant pas de meilleure approximation dans le domaine (par exemple, un disque, dans le cas du domaine des polyhèdres convexes). en utilisant le domaine des polyhèdres convexes. Par conséquent, nous utiliserons uniquement la fonction de concrétisation.

Dans la suite, $\mathbb{M}^{\text{app}}_{\text{pre}}$ sera donc représenté par un élément $d^\sharp_{\text{pre}}$ de $\mathbb{M}^\sharp$; de même $\mathbb{M}^{\text{app}}_{\text{post}}$ sera représenté par un élément $d^\sharp_{\text{post}} \in \mathbb{M}^\sharp$. Nous devons à présent mettre au point un algorithme permettant de calculer $d^\sharp_{\text{post}}$ à partir de $d^\sharp_{\text{pre}}$ tout en garantissant la condition suivante :

$$[\![\mathtt{b}]\!](\gamma(d^\sharp_{\text{pre}})) \subseteq \gamma(d^\sharp_{\text{post}}). \qquad [2]$$

Cette condition traduit le fait que l'analyse est *correcte*, c'est à dire qu'elle n'oublie aucune éxecution du programme. Toutefois, elle est généralement *incomplète*, ce qui signifie qu'elle considérera des état ne correspondant à aucune exécution. La mise au point d'un tel algorithme est le but de la section suivante.

4. Vers une sémantique abstraite calculable

Afin de calculer une sur-approximation des états finaux concrets $d^\sharp_{\text{post}}$, nous allons définir des fonctions d'analyses pour chaque instruction `i` (resp., bloc `b`), que nous noterons $[\![\mathtt{i}]\!]^\sharp$ (resp., $[\![\mathtt{b}]\!]^\sharp$). Ces fonctions devront satisfaire des propriétés de correction similaires à celle exprimée par 2 :

$$\begin{array}{l} \forall \mathtt{i},\ \forall \mathcal{E} \in \mathcal{P}(\mathbb{M}),\ \forall d^\sharp \in \mathbb{M}^\sharp,\ \mathcal{E} \subseteq \gamma(d^\sharp) \Longrightarrow [\![\mathtt{i}]\!](\mathcal{E}) \subseteq \gamma([\![\mathtt{i}]\!]^\sharp(d^\sharp)) \\ \forall \mathtt{b},\ \forall \mathcal{E} \in \mathcal{P}(\mathbb{M}),\ \forall d^\sharp \in \mathbb{M}^\sharp,\ \mathcal{E} \subseteq \gamma(d^\sharp) \Longrightarrow [\![\mathtt{b}]\!](\mathcal{E}) \subseteq \gamma([\![\mathtt{b}]\!]^\sharp(d^\sharp)) \end{array} \qquad [3]$$

Ces fonctions seront définies en suivant étape par étape les calculs effectués par le programme, c'est à dire la définition de la sémantique concrête. Dans un premier temps, nous allons supposer l'existence de quelques fonctions de base, supposées calculables, vérifiant chacune une propriété de correction :

– pour toute variable x et toute expression `e` une fonction $\mathit{affectation}^\sharp_{x,\mathtt{e}} : \mathbb{M}^\sharp \to \mathbb{M}^\sharp$ telle que $\forall d^\sharp \in \mathbb{M}^\sharp, \forall \rho \in \mathbb{M},\ \rho \in \gamma(d^\sharp) \Longrightarrow \rho[x \leftarrow [\![\mathtt{e}]\!](d^\sharp)] \in \gamma(\mathit{affectation}^\sharp_{x,\mathtt{e}}(d^\sharp)$;

– pour toute condition `c`, une fonction $\mathit{test}^\sharp_{\mathtt{c}} : \mathbb{M}^\sharp \to \mathbb{M}^\sharp$ telle que pour tous $d^\sharp \in \mathbb{M}^\sharp$ et $\rho \in \mathbb{M}$, l'évaluation de la condition `c` retourne la valeur logique **true** dans l'environnement ρ, alors $\rho \in \gamma(\mathit{test}^\sharp_{\mathtt{c}}(d^\sharp))$;

– un opérateur d'union abstraite $\sqcup^\sharp$ tel que l'union abstraite sur-approxime l'union concrête, c'est à dire que pour tous $\mathcal{E}_0, \mathcal{E}_1 \in \mathcal{P}(\mathbb{M})$ et $d^\sharp_0, d^\sharp_1 \in \mathbb{M}^\sharp$, si $\mathcal{E}_0 \subseteq \gamma(d^\sharp_0)$ et $\mathcal{E}_1 \subseteq \gamma(d^\sharp_1)$, alors $\mathcal{E}_0 \cup \mathcal{E}_1 \subseteq \gamma(d^\sharp_0 \sqcup^\sharp d^\sharp_1)$.

Ces opérateurs étant fixés, nous pouvons donner une définition pour $[\![.]\!]^\sharp$ excepté pour les boucles, qui feront l'objet de la section 5 :

$$\begin{array}{rcl} [\![x := \mathtt{e}]\!]^\sharp(d^\sharp) & = & \mathit{affectation}^\sharp_{x,\mathtt{e}}(d^\sharp) \\ [\![\textbf{if } \mathtt{e} = 0 \textbf{ then } \mathtt{i}_0 \textbf{ else } \mathtt{i}_1]\!]^\sharp(d^\sharp) & = & [\![\mathtt{i}_0]\!]^\sharp(\mathit{test}^\sharp_{\mathtt{e}=0}(d^\sharp)) \sqcup^\sharp [\![\mathtt{i}_1]\!]^\sharp(\mathit{test}^\sharp_{\mathtt{e}\neq 0}(d^\sharp)) \\ [\![\mathtt{i}_0; \mathtt{i}_1]\!]^\sharp(d^\sharp) & = & [\![\mathtt{i}_1]\!]^\sharp([\![\mathtt{i}_0]\!]^\sharp(d^\sharp)) \end{array}$$

On peut alors prouver par induction sur la syntaxe que l'analyse obtenue est effectivement correcte à savoir que cette définition de $[\![.]\!]^\sharp$ vérifie la propriété 3.

La définition des opérateurs $\mathit{affectation}^\sharp$, $\mathit{test}^\sharp$ et $\sqcup^\sharp$ dépend du domaine abstrait choisi. Par exemple, si l'on considére le cas du domaine des intervalles où une valeur abstraite est une fonction associant un intervalle à chaque variable :

– étant donné un élément abstrait $d^\sharp$, on peut calculer par induction sur la syntaxe un intervalle $[\![\mathtt{e}]\!]^\sharp(d^\sharp)$ contenant toutes les valeurs vers lesquelles peut s'évaluer une expression $\mathtt{e}$, à l'aide de règles telles que $[\![n]\!]^\sharp(d^\sharp) = [n, n]$; $[\![x]\!]^\sharp(d^\sharp) = d^\sharp(x)$; si $[\![\mathtt{e}_0]\!]^\sharp(d^\sharp) = [a_0, b_0]$ et $[\![\mathtt{e}_1]\!]^\sharp(d^\sharp) = [a_1, b_1]$ alors $[\![\mathtt{e}_0 + \mathtt{e}_1]\!]^\sharp(d^\sharp) = [a_0 + a_1, b_0 + b_1]$;

– l'union abstraite $d_0^\sharp \sqcup^\sharp d_1^\sharp$ peut être défini en associant à toute variable x l'intervalle le plus petit contenant les intervalles $d_0^\sharp(x)$ et $d_1^\sharp(x)$ (l'union abstraite d'une paire d'intervalles $[a_0, b_0], [a_1, b_1]$ sera l'intervalle $[\min(a_0, a_1), \max(b_0, b_1)]$).

Dans le cas d'un domaine tel que les polyhèdres (Cousot *et al.*, 1978), un opérateur $\sqcup^\sharp$ pourra être défini selon un principe analogue (enveloppe convexe), tandis que la définition de $\mathit{affectation}^\sharp$ sera plus complexe mais permettra d'inférer des *relations* entre variables.

5. Choix d'un opérateur d'élargissement

Dans la section précédente, nous avons omis le cas des boucles. Considérons donc la boucle **while** $\mathtt{e} \neq 0$ **do** $\mathtt{b}$, que nous noterons $\mathtt{i}$. Avant de proposer une définition pour $[\![\mathtt{i}]\!]^\sharp$, il nous faut raffiner un peu la définition de $[\![\mathtt{i}]\!]$. Intuitivement, on peut construire $[\![\mathtt{i}]\!]$ en accumulant l'ensemble des états sortant de la boucle après un nombre quelconque d'itérations (ce qui correspond à une définition sous la forme du plus petit point fixe de la sémantique du corps de la boucle) :

$$\begin{array}{l} [\![\textbf{while } \mathtt{e} \neq 0 \textbf{ do } \mathtt{b}]\!](\mathcal{E}) = \{\rho \in \mathcal{E}_0 \mid\mid [\![\mathtt{e}]\!](\rho) = 0\} \\ \text{où} \quad \mathcal{E}_0 = \bigcup_{i\in\mathbb{N}} F_\mathtt{b}^i(\mathcal{E}) = \mathbf{lfp} F_\mathtt{b} \\ \text{et} \quad \left\{ \begin{array}{rccl} F_\mathtt{b} : & \mathcal{P}(\mathbb{M}) & \rightarrow & \mathcal{P}(\mathbb{M}) \\ & \mathcal{E} & \mapsto & [\![\mathtt{b}]\!](\{\rho \in \mathcal{E} \mid [\![\mathtt{e}]\!](\rho) \neq 0\}) \end{array} \right. \end{array} \qquad [4]$$

Pour définir un calcul dans $\mathbb{M}^\sharp$ effectuant les mêmes étapes, il pourrait sembler suffisant d'utiliser l'opérateur $\mathit{test}^\sharp$ à la place des tests concrêts, $[\![\mathtt{b}]\!]^\sharp$ à la place de $[\![\mathtt{b}]\!]$, et $\sqcup^\sharp$ en lieu et place de $\cup$. Toutefois, cette solution ne convient pas, puisque la formule ci-dessus consiste en une union *infinie*, et puisque la fonction d'analyse se doit d'être *calculable* en machine, et donc en un temps fini. Pour garantir la terminaison de la suite d'itérés abstraits, nous utiliserons un *opérateur d'élargissement* ∇, en lieu et place de $\sqcup^\sharp$; celui-ci aura pour but de calculer une approximation sûre de ses deux paramètres (de même que $\sqcup^\sharp$), et d'assurer la *terminaison* de toute suite d'itérations abstraites (autrement dit, un tel opérateur garantit que tout suite d'élargissements termine), ce qui permet de donner une définition d'une fonction d'analyse $[\![\mathtt{i}]\!]^\sharp$ calculable pour la boucle ; plus précisément on définira $[\![\mathtt{i}]\!]^\sharp(d^\sharp)$ comme étant la limite de

```
assume(−2 147 483 648 ≤ x
        ∧ x ≤ 2 147 483 647);
if x ≥ 0 then {
      s := 1;
} else {
      s := −1;
}
assert(s ≠ 0);
```

```
x := 50 000; y := 0;
while x ≥ 0 do {
      x := x − 1;
      if random() then {
            y := y + 1;
      }
}
assert(0 ≤ y ≤ 50 000);
```

Figure 2. *Deux programmes simples : signe, et boucle*

la suite suivante, dont on peut prouver qu'elle est stationnaire (c'est à dire qu'à partir d'un certain rang, elle devient stable) :

$$X_0^\sharp = d^\sharp \qquad \forall n \in \mathbb{N},\ X_{n+1}^\sharp = X_n^\sharp \nabla F_{\mathfrak{b}}(X_n^\sharp)$$

Le choix d'un bon opérateur d'élargissement est crucial pour la réussite d'une analyse statique. D'une manière générale, le principe d'un tel opérateur consiste à supprimer les contraintes instables dans l'itéré précédent, chaque élément abstrait pouvant être vu comme un ensemble fini de contraintes. Le nombre de contraintes étant décroissant et positif, celui-ci converge, vers un ensemble stable, correspondant à la limite de la suite d'itérés abstraits. Il n'y a généralement pas d'opérateur d'élargissement unique, ni même meilleur que les autres, donc découvrir un opérateur d'élargissement qui marche bien en pratique est une tâche complexe. On peut considérer une suite d'itérations avec élargissement comme une tentative de preuve par récurrence où l'hypothèse de récurrence ne serait pas connue initialement, et devrait être "découverte" au cours du processus : le procédé généralement suivi par un mathématicien dans un tel cas consiste à étudier le cas des premières valeurs, puis à essayer de deviner une propriété de récurrence $\mathcal{P}$ par affaiblissements successifs, et ensuite à essayer de prouver que $\mathcal{P}_n \Rightarrow \mathcal{P}_{n+1}$; lorsque cette implication ne peut être prouvée, on peut essayer de trouver une propriété plus faible et ainsi de suite. Mais parfois, il faut au contraire renforcer $\mathcal{P}$ pour trouver une propriété inductive, et l'opérateur d'élargissement ne permet pas de faire cela ; nous commenterons ce cas dans la section 6.

Le résultat d'une analyse statique est donc le résultat de plusieurs approximations : tout d'abord, l'analyse utilise un ensemble de prédicats abstraits, qui ne permet pas forcément d'exprimer toute propriété concrête ; ensuite, les fonctions de transfert abstraites $\mathit{test}^\sharp$ et $\mathit{affectation}^\sharp$ ne calculent généralement pas le résultat le plus précis possible (e.g., pour des raisons de coût) ; enfin, l'opérateur d'élargissement lui même induit une perte de précision, afin de garantir la terminaison des analyses.

Nous considérons à présent les deux très courts programmes présentés dans la figure 2. Tout d'abord, le programme de gauche calcule le signe d'une variable entière, et nous souhaitons que l'analyse statique prouve que le résultat est différent de 0. Sur ce très bref programme, il est possible de voir qu'à la sortie de la conditionnelle s vaut

soit -1 soit 1, mais il n'est pas toujours possible d'énumérer ainsi les valeurs possibles à l'aide d'un ensemble de taille raisonnable. Considérons donc les domaines abstraits présentés dans la figure 1. Tout d'abord, les domaines d'intervalles, de polyhèdres et d'octogones décrivent uniquement des ensembles d'états convexes ; par conséquent, dans ce cas, ils ne permettent pas de prouver que s est non nul. Par contre, le domaine des congruences permet d'inférer que s est impair, donc non nul. Dans le cas du second programme, un domaine *relationnel* tel que les octogones ou polyhèdres permet d'inférer, par itérations avec élargissement que $x + y \leq 50\,000$ en tête de boucle. En effet, initialement $x = 50\,000$ et $y = 0$, puis à la première itération $x = 49\,999$ et $y \in [0, 1]$, donc $x + y \leq 50\,000$ et $y \geq 0$. Ces deux contraintes sont ensuite stables.

6. Mise au point d'un analyseur statique

En pratique, la conception et le développement d'un analyseur statique requièrent que l'on effectue un certain nombre de choix, qui ont tous un impact sur l'efficacité de l'analyse (en termes de temps de calcul et quantité de mémoire utilisée) ainsi que son niveau de précision (i.e., sa capacité à inférer des propriétés non triviales). En particulier, nous avons vu que le domaine abstrait joue un rôle crucial. Pour prouver des propriétés complexes, il est généralement utile de combiner plusieurs domaines, afin d'obtenir un ensemble suffisament expressif de prédicats abstraits ; on utilise alors des constructions telle que le *produit réduit*, qui permet de manipuler des *conjonctions* de propriétés exprimées par des domaines abstraits différents $\mathbb{M}^\sharp_0, \ldots, \mathbb{M}^\sharp_p$. Dans ce cas, s'ajoute alors un nouveau choix de conception : celui de l'opérateur de réduction, dont le rôle est de raffiner un prédicat exprimé dans $\mathbb{M}^\sharp_i$ à partir d'un prédicat exprimé dans $\mathbb{M}^\sharp_j$ (avec $i \neq j$). Remarquons que de nombreux domaines sont à présent disponibles, parfois même sous la forme de librairies comme APRON(Jeannet *et al.*, 2009), ce qui réduit notablement le travail de programmation.

Le domaine abstrait étant complètement fixé, il reste de nombreuses manières d'optimiser la précision et l'efficacité de l'analyse. Le choix de l'opérateur d'élargissement est particulièrement important. Si un opérateur supprimant les contraintes instables permet généralement de garantir la terminaison des analyses, il est souvent nécessaire de "ralentir" ce processus de suppression des contraintes instables afin d'inférer des invariants de boucles précis : on peut par exemple définir un élargissement "avec seuil" (si la contrainte $x \leq n$ n'est pas stable, où n est un entier plus petit que $c_{\mathbf{seuil}}$, on peut remplacer celle-ci par $x \leq c_{\mathbf{seuil}}$ plutôt que de la supprimer) ou bien utiliser une stratégie d'élargissement adaptée (e.g., en "déroulant" les premières itération des boucles). De telles techniques sont utilisées par exemple dans ASTRÉE (Blanchet *et al.*, 2003). Enfin, pour chaque opération élémentaire, il existe souvent plusieurs choix garantissant la correction, mais correspondant à des niveaux différents de précision. Ces choix pourront avantageusement être guidés par une bonne connaissance des programmes qui seront soumis à l'analyseur ; ainsi, le choix de développer une analsye spécifique à certaines familles de programmes peut permettre la réalisation d'analyses précises et efficaces (Blanchet *et al.*, 2003).

7. Quelques applications de l'analyse statique par interprétation abstraite

La technique d'analyse statique décrite dans les sections précédentes a été appliquée à de nombreux domaines de l'informatique, au cours des dernières décennies.

Ainsi, dans le domaine des logiciels embarqués, plusieurs analyses ont été proposées, notamment afin de garantir des propriétés de sûreté. Par exemple, l'analyse proposée dans (Theiling *et al.*, 2000) permet de calculer des bornes supérieures au temps d'exécution de fragments de code assembleur (en prenant en compte le comportement du cache, et du pipeline), ce qui est particulièrement important dans le cas d'applications temps réel ; cet outil fait l'objet d'une diffusion industrielle. Dans le même temps, plusieurs analyses ont permis de calculer un sur-ensemble des erreurs à l'exécution. L'analyseur généraliste Polyspace a été utilisé suite à l'échec du vol inaugural d'Ariane 5 (Lions *et al.*, 1996) pour analyser des applications écrites en ADA. Depuis 2002, l'analyseur ASTRÉE (Blanchet *et al.*, 2003) a été développé afin d'analyser avec précision certaines familles d'applications critiques embarquées, et a effectivement permis de prouver l'absence d'erreurs à l'exécution dans des applications avioniques de taille industrielle ; cet outil est actuellement en cours d'industrialisation par l'entreprise Absint. Par ailleurs, la recherche des sources d'imprécision dans les calculs flottants a également fait l'objet de travaux en analyse statique (Goubault *et al.*, 2002). Des analyses telles que CSSV (Dor *et al.*, 2003) permettent de garantir des propriétés de sécurité, telles que l'absence de dépassement de buffers. D'autres analyses ont permis de prouver des propriétés sur des langages plus haut niveau encore, comme par exemple des langages synchrones comme (Jeannet, 2003). Enfin, de nouvelles problèmatiques sont actuellement considérées, comme la conséquence d'une légère désynchronisation d'horloges entre sous-systèmes syncrhones (Bertrane, 2005).

Le cadre de l'interprétation abstraite a également permis de formaliser des transformations de programmes (Cousot *et al.*, 2002) telles que la compilation (Rival, 2004), la décompilation (Chang *et al.*, 2006) ou bien le slicing (Mastroeni *et al.*, 2008). De telles formalisations ont l'avantage de permettre de combiner des analyses statiques et des transformations de programmes, par exemple, pour généraliser les algorithmes de transformation ou bien pour prouver des propriétés sur leur résultats. L'analyse statique est également très souvent utiliser pour permettre des transformations optimisantes : ainsi, le cadre de travail Ciao PP (Hermenegildo *et al.*, 2003) permet de développer et raisonner sur des programmes logiques de type Prolog et intégre des analyses statiques à des fins d'optimisation.

Par ailleurs, les techniques d'analyse statique ont permis d'inférer des propriétés sur de nombreuses familles de langages informatiques ou non. On peut citer les analyses de code mobile (Feret, 2005), qui permettent de prouver de nombreux types de propriétés. Plus récemment, des chaînes de réaction moléculaire en biologie ont pu être formalisées sous la forme de systèmes de réecritures sur des graphes, et ainsi être étudiées à l'aide d'analyses statiques par interprétation abstraite (Danos *et al.*, 2008).

8. Bibliographie

Bertrane J., « Static analysis by abstract interpretation of the quasi-synchronous composition of synchronous programs », *VMCAI*, Springer, 2005.

Blanchet B., Cousot P., Cousot R., Feret J., Mauborgne L., Miné A., Monniaux D., Rival X., « A Static Analyzer for Large Safety Critical Software », *PLDI'03*, ACM, 2003.

Chang B.-Y. E., Harren M., Necula G. C., « Analysis of low-level code using cooperating decompilers », *SAS*, Springer, 2006.

Cousot P., Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes, PhD thesis, Université de Grenoble, 1978.

Cousot P., Cousot R., « Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints », *POPL'77*, ACM, 1977.

Cousot P., Cousot R., « Systematic Design of Program Transformation Frameworks by Abstract Interpretation », *POPL'02*, ACM Press, New York, NY, 2002.

Cousot P., Halbwachs N., « Automatic discovery of linear restraints among variables of a program », *POPL'78*, ACM, 1978.

Danos V., Feret J., Fontana W., Krivine J., « Abstract interpretation of cellular signalling networks », *VMCAI'08*, Springer, 2008.

Dor N., Rodeh M., Sagiv M., « CSSV : Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C », *PLDI'03*, ACM, 2003.

Feret J., « Abstract interpretation of mobile systems », *Journal of Logic and Algebraic Programming*, 2005.

Goubault E., Martel M., Putot S., « Asserting the precision of floating-point computations : a simple abstract interpreter », *ESOP*, Springer, 2002.

Granger P., « Static Analysis of Arithmetical Congruences », *International Journal of Computer Mathematics*, vol. 30, p. 165-190, 1989.

Hermenegildo M., Puebla G., Bueno F., López-García P., « Program Development Using Abstract Interpretation (And The Ciao System Preprocessor) », *SAS*, Springer, 2003.

Jeannet B., « Partitioning in linear relation analysis : application to the verfication of reactive systems », *FMSD*, 2003.

Jeannet B., Miné A., « Apron : A Library of Numerical Abstract Domains for Static Analysis », *CAV*, Springer, 2009.

Lions J. L., al., ARIANE 5, Flight 501 Failure, Technical report, Inquiry Board, 1996.

Mastroeni I., Zanardini D., « Data dependencies and program slicing : from syntax to abstract semantics. », *PEPM*, ACM, 2008.

Miné A., « The Octagon abstract domain », *HOSC*, 2006.

Rival X., « Symbolic transfer functions-based approaches to certified compilation », *POPL'04*, ACM, 2004.

Theiling H., Ferdinand C., Wilhelm R., « Fast and Precise WCET Prediction by Seperate Cache and Path Analyses », *Real-Time Systems*, 2000. Springer.