Algèbre linéaire pour invariants polynomiaux

Steven De Oliveira, Saddek Bensalem, Virgile Prévosto

Algèbre linéaire pour invariants polynomiaux

Steven de Oliveira[1], Saddek Bensalem[2], Virgile Prevosto[1]

1 : CEA, List ; 2 : Université Grenoble Alpes

Résumé

Nous présentons dans ce papier une nouvelle technique de génération d'invariants dans le contexte d'un certain type de boucles polynomiales. Notre méthode a l'avantage d'être plus rapide que les méthodes existantes pour des boucles équivalentes et plus simple à implanter car elle repose sur des algorithmes d'algèbre linéaire de complexité polynomiale. Un outil implémentant cette méthode est en cours de développement dans Frama-C [2], une plateforme open-source, extensible et collaborative dédiée à l'analyse de programmes C.

1 Introduction

La phase de développement d'un programme requiert plus que la simple connaissance de la syntaxe d'un langage mais aussi de sa sémantique, parfois complexe. Aussi il est largement admis que la vérification en est une étape importante. Les techniques de vérification déductive permettent d'établir formellement les propriétés fonctionnelles d'un programme. Cependant, elles nécessitent un effort important de la part de l'utilisateur, qui doit entre autres choses fournir des invariants (propriétés qui restent vraies à chaque tour) pour chacune des boucles présentes dans le programme. De ce fait, de nombreux travaux visent à automatiser la découverte de certains de ces invariants. C'est dans cette optique que nous avons développé une nouvelle technique de génération d'invariants de boucle dans le contexte d'un certain type d'affectations définies dans [5]. Notre méthode a l'avantage d'être plus rapide que les méthodes existantes pour des boucles équivalentes et est plus simple à implanter car elle repose sur des algorithmes d'algèbre linéaire de complexité polynomiale. Un outil implantant cette procédure est en cours de développement dans Frama-C [2], une plateforme open-source, extensible et collaborative dédiée à l'analyse de programmes C.

Structure de l'article Cet article sera centré sur l'analyse de l'exemple en figure 1, sur lequel nous allons appliquer en détail chaque étape de notre méthode jusqu'à la génération de l'invariant propre à cette boucle. L'affectation de x et de y est simultanée. Notre méthode est composée de deux opérations distinctes :

```
(x,y) := (0,0);
while (y < N) do
   (x,y) := (x + y*y, y + 1);
done
```

Figure 1: Programme calculant la somme de carrés successifs. Cette boucle admet l'invariant $-6.x + y - 3.y^2 + 2y^3 = 0$ pour tout N.

1. la transformation des affectations polynomiales de la boucle en affectations affines (Section 2.1);

2. la génération d'invariants sur la nouvelle boucle affine (Section 2.2).

Nous présenterons ensuite notre procédure pour prendre en compte les conditions présentes dans les boucles dans la section 2.3.

Remarque Toutes les propriétés présentées, notamment la complétude et la complexité de notre méthode ont été prouvées. Par souci de place, elles ne sont pas présentées dans cet article et feront partie d'une version plus longue.

2 Génération automatique d'invariants polynomiaux

2.1 Linéarisation de boucles polynomiales

2.1.1 Réduction

Nous allons chercher à trouver une affectation affine dont le comportement est équivalent à celui de l'affectation P de la boucle, avec $P(x, y) = (x + y^2, y + 1)$. Nous allons avoir besoin de nous débarasser du monôme y^2 qui est la seule opération non-linéaire. Plutôt que de considérer y^2 comme un calcul à effectuer, on peut chercher à exprimer l'évolution de sa valeur après l'affectation comme proposé par [4]. Cela nous conduit à remarquer que si $y' = y + 1$, alors $(y^2)' = y^2 + 2y + 1$. Notons que l'évolution de y^2 s'exprime comme une combinaison affine de y^2 et de y. Posons donc une variable y_2 que l'on initialise à y^2 avant l'exécution de la boucle : elle représentera le monome y^2 le long de l'exécution de la boucle. Dès lors $f(x, y, y_2) = (x + y_2, y + 1, y_2 + 2y + 1)$. f est une application affine et a exactement le même comportement que P sur x et y. Nous avons ainsi *réduit* l'analyse de l'affectation polynomiale P à celle de f, plus simple.

Remarque Notre exemple a la particularité d'être linéarisable mais ce n'est pas le cas pour toutes les affectations polynomiales. Par exemple si l'on essaie pour l'application $P(x) = x^2$ d'exprimer le monôme x^2 en fonction de x, il sera nécessaire d'exprimer x^4, x^8, etc. Nous devons ainsi nous restreindre aux affectations

qui *ne transforment pas polynomialement une variable elle-même*. Cette classe d'affectation a déjà été définie dans [5] :

Définition 1 *Soit $g \in \mathbb{Q}[X]^m$ une affectation polynomiale. g est soluble s'il existe une partition de X en sous-vecteurs de variables $x = w_1 \uplus ... \uplus w_k$ telle que $\forall j.\ 1 \leqslant j \leqslant k$ on a*

$$g_{w_j}(x) = M_j w_j^T + P_j(w_1, ..., w_{j-1})$$

En d'autres termes, une affectation soluble transforme une variable :

- soit de manière affine;
- soit par une transformation affine de variables plus un polynôme d'autres variables qui sont elles-mêmes soit transformées linéairement, soit par une affectation soluble *indépendante*.

Sur l'exemple précédent, on peut prendre $w_1 = (y),\ P_1 = 1,\ w_2 = (x)$ et $P_2(y) = y^2$. Dans ce cas, on remarque que $(x', y', P_2(y')) = f(x, y, P_2(y))$. Plus généralement, la propriété suivante est vraie :

Propriété 1 *Pour toute affectation soluble g, il existe une application affine f et un polynôme P tels que $X' = g(X) \Rightarrow (X', P(X')) = f(X, P(X))$.*

2.1.2 Élévation

L'invariant que nous cherchons est de degré 3, c'est à dire qu'il est composé de monômes de degré maximal 3. Notre méthode de génération d'invariant sur les affectations linéraires ne prend en compte que les variables utilisées, donc si nous souhaitons trouver des invariants de degré supérieur à 2, l'étape de réduction n'est pas suffisante car on n'y exprime que l'évolution d'un monôme de degré 2. Nous devons continuer d'appliquer la méthode décrite précédemment pour exprimer de nouveaux monômes de degré *plus élevé*. Dans notre cas, nous aurons besoin de considérer dans notre analyse des monômes de degré 3 et moins, c'est à dire xy et y^3. Considérer x^2 n'est pas nécessaire car l'évolution de ce monôme nécessite une expression de y^4, un monôme de degré superieur à 3. C'est dans le cas de l'élévation d'une application affine que l'on aura le plus de variables, dont une borne supérieure est donnée dans [4].

Propriété 2 *Toute affectation soluble g utilisant n variables est linéarisable par une affectation utilisant au plus $\binom{n+d}{d}$ nouvelles variables, où d est le degré du polynôme utilisé dans la linéarisation. Pour un nombre de variables donné, on a donc besoin d'une quantité polynomiale de nouvelles variables.*

2.2 Génération d'invariants

Nous présentons dans cette partie une technique de génération d'invariants pour les boucles affines, qui étend naturellement l'étape de linéarisation. Par la suite, nous considérerons une application affine comme une application linéaire à laquelle on ajoute une variable $\mathbb{1}$ toujours égale à 1. Les constantes k rendant une application non-linéaire seront alors remplacées par $k.\mathbb{1}$.

Informellement, un invariant est une formule logique telle que

1. elle est valide à l'état initial de la boucle ;
2. après chaque itération de la boucle, elle reste valide.

Nous nous intéressons d'abord à la recherche de *semi-invariants*, des formules qui satisfont uniquement le second critère sous la forme d'une combinaison linéaire sur X. Dans ce cadre, une telle formule est une forme linéaire φ telle que :

$$\varphi(X) = 0 \Rightarrow \varphi(f(X)) = 0 \tag{1}$$

Par l'algèbre linéaire, l'égalité suivante est toujours vraie

$$\varphi(f(X)) = (f^*(\varphi))(X) \tag{2}$$

où f^* est l'application duale de f. Si φ est un vecteur propre de f^* (i.e. il existe λ tel que $f^*(\varphi) = \lambda\varphi$), l'équation (1) est automatiquement vraie.

Dans notre cas, f^* peut être calculée en transposant la matrice représentant f. Par souci de simplicité, nous allons ignorer le monôme xy car il n'intervient pas dans l'expression de l'invariant que nous cherchons.
Nous avons alors $f^*(x, y, y_2, y_3, \mathbb{1}) = (x, y+2.y_2+3.y_3, x+y_2+3.y_3, y_3, y+y_2+y_3+\mathbb{1}, y+y_2+y_3+\mathbb{1})$. f^* admet la valeur propre 1. L'espace propre de f^* associé à 1 est généré par les vecteurs $e_1 = (-6, 1, -3, 2, 0)^T$ et $e_2 = (0, 0, 0, 0, 1)^T$. On trouve ainsi que $F_{k1,k2} = (k_1.(-6.x + y - 3.y_2 + 2y_3) + k_2.\mathbb{1} = 0)$ est un semi-invariant, avec $k_1, k_2 \in \mathbb{Q}$. En écrivant $k = -\frac{k_2}{k_1}$ et en remplaçant $\mathbb{1}$ par 1, on peut ré-écrire la formule précédente par

$$F_k = (-6.x + y - 3.y_2 + 2y_3 = k)$$

Le paramètre peut être inféré par l'état initial de la boucle. Si par exemple la boucle commence avec $(x = 0, y = 0)$, alors $-6.x + y - 3.y_2 + 2y_3 = 0$ et donc F_0 est un invariant de la boucle. Plus généralement, l'union des espaces propres de f^* est toujours un ensemble de semi-invariants pour l'affectation f.

Complexité La transposition de matrices et la recherche des vecteurs propres sont des problèmes polynomiaux en la taille de la matrice, donc polynomiaux en le nombre de variables. Dans le cas d'une boucle linéarisée par la méthode de la section 2.1 qui génère un nombre polynomial de nouvelles variables, la composition des deux est bien polynomiale.

```
x = a, y = b, z = 0;
while (y > 1) {
    if (y%2 == 1)
    then
      (x,y,z) := (2x, (y−1)/2, x + z)
    else
      (x,y,z) := (2x, y/2, z) }
```

Figure 2: Calcul du produit de a et b (exemple de [5])

2.3 Boucles avec conditions

Une boucle peut contenir plusieurs conditions qui séparent le graphe de flot de contrôle en plusieurs chemins différents. Supposons qu'à chaque itération, la boucle peut choisir aléatoirement chaque condition. Dès lors, à chaque tour de boucle une des affectations va être effectuée, indépendemment des tours précédents. Prenons en exemple le programme de la figure 2 calculant le produit de x et y. Posons $X = (x, y, z, \mathbb{1})$. Même si on considère comme non déterministes les conditions du programme original, nous pouvons trouver des invariants utiles. Nous avons ici deux affectations : $f_1(X) = (2x, (y-1)/2, x+z, \mathbb{1})$ et $f_2(X) = (2x, y/2, z, \mathbb{1})$. L'ensemble des semi-invariants vrais sur les différents corps de boucle possibles est alors l'intersection des semi-invariants vrais sur chaque corps de boucle. Nous venons de voir que l'ensemble des invariants générés peut être représenté comme une union d'espaces vectoriels, leur intersection est donc également une union d'espaces vectoriels. L'élévation au degré 2 de f_1 et de f_2 retourne 10 vecteurs propres pour chacune des applications. Pour simplifier, nous allons nous concentrer sur les vecteurs propres associés à la valeur propre 1. Posons φ_e l'application linéaire induite par e.

f_1^* a 4 vecteurs propres $\{e_i\}_{i\in[1,4]}$ associés à 1 tels que

- $\varphi_{e_1}(X) = -x + xy$;
- $\varphi_{e_2}(X) = x + z$;
- $\varphi_{e_3}(X) = -2.xz + x^2 + z^2$;
- $\varphi_{e_4}(X) = \mathbb{1}$.

f_2^* a également 4 vecteurs propres $\{e'_i\}_{i\in[1,4]}$ associés à 1 tels que

- $\varphi_{e'_1}(X) = xy$;
- $\varphi_{e'_2}(X) = z$;
- $\varphi_{e'_3}(X) = z^2$;
- $\varphi_{e'_4}(X) = \mathbb{1}$.

Premièrement, on remarque que $e_4 = e'_4$. Ensuite, on peut voir que $e_1 + e_2 = e'_1 + e'_2$. Finalement, on remarque que $e_1 + e_2 + k.e_4 \in (Vect(\{e_i\}_{i\in[1,4]}) \cap Vect(\{e'_i\}_{i\in[1,4]}))$. C'est pourquoi $(\varphi_{e_1+e_2+k.e_4}(X) = 0)$ est un semi-invariant pour f_1 et f_2, donc il l'est pour toute la boucle. En remplaçant $\varphi_{k.e_4}(X)$ par $k.\mathbb{1} = -k'$ et $\varphi_{e_1+e_2}(X)$ par $xy + z$, on trouve $xy + z = k'$. Initialement,

$xy + z = ab$, donc on sait que $xy + z = ab$ est un invariant de la boucle. Cet invariant est nécessaire pour prouver que ce programme multiplie bien x et y.

Remarque L'intersection de deux espaces vectoriel est équivalente au calcul du noyau de la matrice dont les colonnes sont les bases des deux espaces vectoriels. C'est donc également un calcul polynomial.

3 Conclusion

Ce nouvel algorithme de génération d'invariants pour boucle affine composé avec nos procédures de linéarisation et de gestion des conditions nous permet de générer des invariants non-triviaux sur des boucles complexes. Il est en réalité complet pour les affectations solubles. Notons également que notre algorithme bénéficie d'une complexité polynomiale qui contraste avec les techniques existantes qui sont soit incomplètes avec [1], soit exponentielles avec les bases de Gröbner [3, 5].

Notre attention se porte désormais sur l'étude des boucles affines admettant des vecteurs propres irrationnels qui ne permettent pas une représentation correcte des variables selon un modèle proche de celui d'un programme C. Nous travaillons sur un formalisme complet pour étendre la précédente méthode et développons plusieurs pistes pour faire face à ce problème.

Remerciements Nous remercions les relecteurs anonymes pour leur suggestions et remarques sur une première version de cet article.

Références

[1] D. Cachera, T. Jensen, A. Jobin, and F. Kirchner. Inference of polynomial invariants for imperative programs: A farewell to Gröbner bases. *Science of Computer Programming*, 93:89–109, 2014.

[2] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.

[3] L. Kovács. Aligator: A Mathematica package for invariant generation (system description). In *Automated Reasoning*, pages 275–282. Springer, 2008.

[4] M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *ACM SIGPLAN Notices*, volume 39, pages 330–341. ACM, 2004.

[5] E. Rodríguez-Carbonell and D. Kapur. Generating all polynomial invariants in simple loops. *Journal of Symbolic Computation*, 42(4):443–476, 2007.