RenPar'9,
Lausanne. Suisse, 20–23 mai 1997.

Analyse de flot de données de programmes récursifs à l'aide de grammaires hors-contexte

Albert Cohen[a]

[a]PRiSM, Université de Versailles, 45 Avenue des États-Unis, 78035 Versailles, France.

Résumé
Cet article présente une application originale des langages algébriques à l'analyse de flot de données de programmes récursifs. La technique présentée autorise une description globale et précise du flot, en montrant d'importantes propriétés sémantiques.

Mots clés Analyse de flot, parallélisation, compilation, récursivité, langages algébriques.

1. Introduction

Les techniques de parallélisation automatique pour programmes impératifs reposent presque toujours sur des analyses statiques d'accès à la mémoire. En effet, l'exécution dans un ordre quelconque de deux accès à la mémoire nécessite que ceux-ci ne soient pas en conflit.

Ainsi les *analyses d'alias* recherchent des couples d'accès à la même case mémoire, et les *analyses de dépendance* expriment les accès conflictuels comme relations entre instances d'instructions [1]. L'*analyse de flot de données* raffine ces relations en déterminant, pour une lecture d'une cellule mémoire — le *puits* — la *dernière* écriture dans cette même cellule — la *source*.

Sur les nids de boucles, on distingue les programmes à *contrôle statique* dont l'exécution fixée dès la compilation est propice à une analyse *exacte* [2], des programmes à *contrôle dynamique* dont l'ensemble des opérations est imprédictible statiquement, on parle d'analyse *floue* [3].

La distinction s'étend à la récursion et dans un travail précédent [4] des restrictions garantissent le contrôle statique. On étudie ici l'analyse *floue* de programmes récursifs à contrôle dynamique. Les analyses abstraites classiques calculent un flot de données trop imprécis pour valider les transformations de la parallélisation automatique. L'approche de ce papier est différente : on présente une modélisation par langages algébriques puis une discussion des résultats.

2. Exemples

On utilisera une syntaxe Pascal mais la méthode présentée ici est indépendante du langage. Considérons la procédure `F` fig. 1, initiée par l'appel `F (0, 0)`. À l'aide d'un étiquetage l, s, r, on identifie l'instance d'une instruction au *mot de contrôle* formé de la concaténation des étiquettes des instructions menant à cette instance dans l'*arbre d'appel* : e.g. l'instance ○ de s fig. 2.a est exécutée après l'appel à r, l, r et l ; on la dénote par $rlrls$. $\lhd$ dénote l'ordre textuel des instructions : $l \lhd s \lhd r$ (infixe). L'ordre d'exécution est alors l'ordre *lexicographique* $\ll$ des mots de contrôle. On ne fait *aucune hypothèse sur les instructions conditionnelles* (P_l et P_r).

```
    procedure F (l, r) begin
l     if (Pl) then F (l+1, r);
s     A[l+r] := ··· A[l+r-1] ···;
r     if (Pr) then F (l, r+1)
    end
```

```
    procedure G (l, r) begin
l     if (Pl) then G (l+1, r);
s     A[l-r] := ··· A[l-r]+1 ···;
r     if (Pr) then G (l, r+1)
    end
```

FIG. 1 - *Procédures `F` et `G`.*

Puisque l'argument `l` (resp. `r`) est incrémenté à chaque appel suivant l (resp. r), l'instance $rlrls$ lit la valeur `A[3]`. Les sources de $rlrls$ possibles sont donc toutes les écritures *précédentes* dans `A[3]`, non oblitérées entre temps par une autre écriture dans `A[3]`. Ne pouvant garantir l'exécution de ces instances à la compilation, toutes les écritures dans `A[3]` sont sources potentielles ; désignées par ■ fig. 2.a (les instances de même profondeur accèdent au même élément).

rllrs lit aussi `A[3]`, et les instances précédentes assignant `A[3]` sont les mêmes que pour *rlrls*. Mais désormais, *rlls* (précédant le puits *rllrs*) *est exécutée*, en tant qu'*ancêtre* [1] de *rllrs* :

Lemme 1 *Soit s une instruction d'une procédure* `F` *n'appartenant pas à une conditionnelle. Chaque appel à* `F` *exécutant s, l'exécution d'une instance implique celle de tous ses ancêtres* [1].

Les écritures précédant *rlls* sont donc «écrasées» (notées ▩, fig. 2.b). La source de *rllrs* est *rlls*.

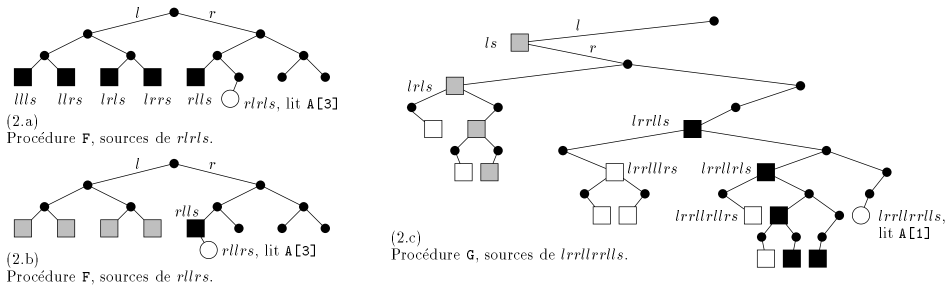

FIG. 2 - *Arbres d'appel partiels de* `F` *et* `G`.

Considérons à présent la procédure `G` fig. 1, initiée par `G (0, 0)`. L'index de `A` étant `r-1`, les sources de *lrrllrrlls* se situent parmi les écritures dans `A[1]`. Il apparaît que *lrrlls*, ancêtre de *lrrllrrlls*, écrit dans `A[1]` (fig. 2.c). Donc toutes les écritures précédant *lrrlls* sont écrasées (*ls*, *lrls*, etc.). De plus, certaines écritures potentielles dans `A[1]` — représentées par □ — s'exécutent *avant* un de leurs ancêtres écrivant dans `A[1]` : ainsi lorsque *lrrllrllrs* est exécutée, *lrrllrls* aussi ; *lrrllrllrs* n'est donc pas source potentielle. Les sources possibles sont donc les ■.

L'objectif est de décrire l'ensemble des ■ à l'aide d'une expression *paramétrée* par le mot de contrôle du puits, *indépendante* de la taille des données ou des calculs.

3. Caractérisation des sources à l'aide de langages algébriques

Lorsque l'instruction d'écriture appartient à une conditionnelle on ne peut en garantir l'exécution, et toute instance conflictuelle précédant le puits sera source potentielle. Sinon, on raffine l'analyse en distinguant trois types d'instances : ■ désigne les sources potentielles ; □ les écritures en conflit dont un ancêtre postérieur est source potentielle ; ▩ celles qui sont écrasées par un ancêtre du *puits* mais n'ont pas d'ancêtre postérieur source potentielle.

3.1. Modèle

On considère un programme composé d'une unique procédure récursive, les variables inductives indexant des tableaux sont translatées lors des appels (pour `F` : l et r sont incrémentées), les tableaux sont uni-dimensionnels et les indexations affines.

Les exemples précédents indiquent que les sources sont caractérisées par des équations affines et des critères liés à l'ordre lexicographique. D'où l'idée d'un automate doté d'un *compteur* (simulé par une pile [5, 6]) capable de repérer les conflits et de garantir l'ordre lexicographique.

Notations : sur la procédure `G` on note $\Sigma = \{l, r\}$, w le mot de contrôle du puits et w' celui d'une source potentielle ; $w, w' \in \Sigma^* \cdot s$. L'index de `A` s'exprime en fonction de w comme «le nombre de l : $|l|_w$, moins le nombre de r : $|r|_w$» . La fonction d'accès est donc $\varphi(w) = |l|_w - |r|_w$. On écrit que w et w' accèdent au même élément : $\varphi(w) = \varphi(w')$ [2], c'est l'*équation de conflit*.

1. La notion d'ancêtre diffère de celle de préfixe : si u est *préfixe* d'un mot de contrôle w, us est *ancêtre* de w.

3.2. Automate

Un compteur requiert trois symboles de pile : le symbole Z de fond de pile détecte l'annulation, I code en unaire les nombres positifs, D les nombres négatifs. Mais on ne peut garantir l'ordre d'exécution avec un automate reconnaissant ww', on doit décomposer l'ordre lexicographique :

$$w' \ll w \quad \Longleftrightarrow \quad w = u\alpha v, \quad w' = u\alpha' v', \quad \alpha' \lhd \alpha, \quad u \in \Sigma^*, \quad \alpha v, \alpha' v' \in \Sigma^* s. \qquad (1)$$

Ainsi on cherche à caractériser des mots de la forme $u \bullet \alpha v \alpha' v'$ où $\bullet$ est un symbole de séparation, tels que $u\alpha' v'$ soit source potentielle de $u\alpha v$. Pour w et w' donnés, il existe un unique mot $u \bullet \alpha v \alpha' v'$ vérifiant $\alpha' \lhd \alpha$. Réciproquement, la détermination de w et w' est rendue possible par $\bullet$ et le s qui termine αv. La transformation est donc bijective, et on notera $\langle w \bullet w' \rangle$ le mot $u \bullet \alpha v \alpha' v'$. L'équation de conflit devient [2] $\varphi(\alpha) + \varphi(v) = \varphi(\alpha') + \varphi(v')$.

Le fonctionnement de l'automate proposé à la figure 3 s'énonce relativement simplement :

1. la pile contenant initialement le symbole Z, lire le préfixe u ($\mathbf{q_0}$) et le symbole $\bullet$ [3] ($\mathbf{q_1}$) ;
2. lire α et v en «mémorisant» $\varphi(\alpha) + \varphi(v)$ sur la pile ($\mathbf{q_2}$), jusqu'à ce que s soit lu ($\mathbf{q_3}$ et $\mathbf{q_4}$) ;
3. lire α' avec $\alpha' \lhd \alpha$ ($\mathbf{q_3}$ et $\mathbf{q_4}$) et v' ($\mathbf{q_5}$) en mémorisant $\varphi(\alpha) + \varphi(v) - \varphi(\alpha') + \varphi(v')$; en s'assurant de surcroît que l'on *ne lit pas* l lorsque la fonction de conflit est nulle, jusqu'à ce que s soit lu dans une configuration où la fonction de conflit est nulle ($\mathbf{q_6}$).

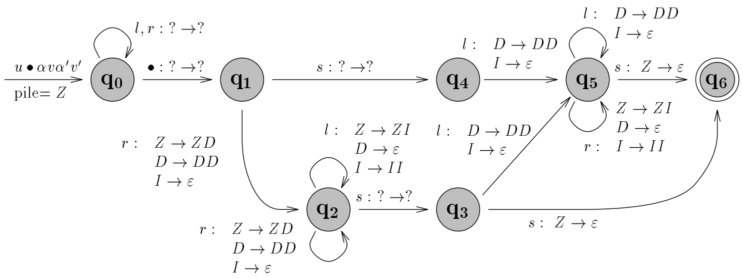

FIG. 3 - *Automate à pile décrivant le flot de la procédure* `G`.

L'automate reconnaît toutes les sources potentielles. On montre notamment que les écritures de type □ (dont un ancêtre *postérieur* est source potentielle) sont éliminées, car une transition lisant l n'est pas valide lorsque la pile est Z ($\mathbf{q_3} \to \mathbf{q_5}$, $\mathbf{q_4} \to \mathbf{q_5}$ et $\mathbf{q_5} \to \mathbf{q_5}$). Pourtant, les écritures de type ▩ précédant un ancêtre de w ne sont pas éliminées. Les langages algébriques ne permettent pas de décrire ces instances ; cela fera l'objet de travaux ultérieurs.

3.3. Grammaire hors-contexte

Donner l'expression complète de la grammaire présente peu d'intérêt, son calcul est simple [5]. On note tout de même que les sources du sous-arbre d'une source potentielle w' (les ■ issus de w') constituent le langage de Lukasiewicz [6] : $\mathrm{S} \vdash w' \cdot s \,\big|\, w' \cdot rLs \quad \mathrm{L} \vdash l \,\big|\, r\mathrm{LL}$.

4. Généralisation

On montre qu'il est possible d'étendre la *construction* de l'automate de la procédure `G` aux programmes respectant les hypothèses énoncées précédemment, et l'algorithme reste simple : le nombre d'états de l'automate croît linéairement avec le nombre d'instructions. Formellement :

Théorème 1 *Pour* deux *instructions de lecture et d'écriture données, on* construit *un automate à pile (ou une grammaire) acceptant $\langle w \bullet w' \rangle$ lorsque l'une des conditions suivantes est vérifiée :*
(i) w' est une source potentielle de w ;
(ii) w' écrit dans la même cellule que w, et une source potentielle ancêtre de w, mais pas de w', est effectivement exécutée.

---

[2] Les accès en écriture et en lecture sont les mêmes sur cet exemple, mais l'analyse ne fait pas cette hypothèse.

[3] Si les accès en lecture et en écriture n'étaient pas les mêmes, il faudrait mettre à jour la pile en conséquence.

Remarque : lorsque l'instruction source considérée est englobée par une instruction conditionnelle il est impossible de garantir qu'un ancêtre du puits est effectivement exécuté, la condition (*ii*) n'a donc plus lieu d'être. Dans ce cas, la caractérisation à l'aide de langages algébriques assure une *précision optimale* (sans hypothèse sur les prédicats des instructions conditionnelles).

Programmes réels : les restrictions (sec. 3) sont trop sévères. Les mots de contrôle ainsi que l'équivalence des ordres lexicographique et d'exécution se conservent avec des procédures mutuellement récursives. En contraignant les variables d'induction, la méthode s'étend ainsi à de plus gros programmes, comprenant notamment les *nids de boucles*. Enfin les tableaux multidimensionnels posent problème car la classe des langages algébriques *n'est pas* close pour l'intersection ; une solution peu satisfaisante consiste à «linéariser» les tableaux.

5. Évaluation et développements

De nombreuses propriétés sémantiques se traduisent en problèmes décidables sur des grammaires hors-contexte. On peut décider en temps linéaire si w' est une source potentielle de w, vérifier la validité des accès, rechercher les lectures de non-initialisées, montrer que l'ensemble des sources potentielles est fini, etc. Ces propriétés peuvent être montrées pour un puits donné ou pour toutes les instances d'une instruction d'assignation. L'intersection d'un langage algébrique avec un langage rationnel étant algébrique, on montre même que la décidabilité d'une propriété pour un puits implique la décidabilité pour tout ensemble rationnel de puits.

Il semble ainsi que cette technique gagne en précision sur les analyses classiques, bien qu'aucune expérimentation ne permette de l'affirmer à présent (l'implémentation actuelle se limite à l'analyse exacte [4]) ; bien sûr, des restrictions sur les programmes analysés sont inévitables.

De tels résultats s'avèrent très utiles pour l'optimisation et la vérification de code, et feront l'objet de travaux futurs. En revanche, l'expansion de tableaux et la déduction automatique d'un ordonnancement parallèle à partir de cette méthode restent ouvertes.

6. Conclusion

Cet article présente une analyse de flot de données des accès aux tableaux dans les programmes récursifs. Le contrôle étant dynamique, on ne peut calculer que des ensembles de sources potentielles. Afin de les décrire on introduit un modèle fondé sur les langages algébriques. La précision obtenue est élevée et les propriétés du flot sont découvertes automatiquement.

Les applications potentielles sont nombreuses, notamment en parallélisation automatique, sachant que l'adaptation des techniques de vérification et de transformation de programme à ce modèle de description du flot requiert sans doute le développement de nouveaux algorithmes.

Bibliographie

1. Pugh W. – A practical algorithm for exact array dependence analysis. *Communications of the ACM*, vol. 35, n° 8, août 1992, pp. 27–47.
2. Feautrier P. – Dataflow analysis of scalar and array references. *Int. Journal of Parallel Programming*, vol. 20, n° 1, février 1991, pp. 23–53.
3. Collard J.-F., Barthou D. et Feautrier P. – Fuzzy array dataflow analysis. *In : Proc. of 5th ACM SIGPLAN, Symp. PPoPP*, pp. 92–102. – Santa Barbara, CA, juillet 1995.
4. Cohen A., Collard J.-F. et Griebl M. – *Array Data-flow Analysis for Imperative Recursive Programs.* – Rapport technique n° 96/035, PRiSM, Université de Versailles, septembre 1996.
5. Hopcroft J. E. et Ullman J. D. – *Introduction to Automata Theory, Languages, and Computation.* – Addison-Wesley, 1979.
6. Berstel J. – *Transductions and Context-Free Languages.* – Teubner, 1979.