RECHERCHE

Analyse de flot de données pour programmes récursifs à l'aide de langages algébriques

Albert Cohen

Laboratoire PRiSM, Université de Versailles
45, Avenue des États-Unis
78035 Versailles

RÉSUMÉ. *La parallélisation de programmes récursifs reste un problème largement ouvert aujourd'hui, faute d'analyses statiques suffisamment précises pour ces programmes. Cet article présente une application originale des langages algébriques à l'analyse de flot de données de programmes récursifs. On montre la nécessité d'imaginer de nouveaux modèles mieux adaptés à la récursivité et on décrit une méthode d'analyse performante sur une classe particulière de programmes. La technique présentée autorise une description globale et précise du flot, en montrant d'importantes propriétés sémantiques.*

ABSTRACT. *Parallelisation of recursive programs is still an open problem today, lacking suitable and precise static analyses. This article presents a novel data-flow analysis framework based on context-free languages. Necessity is shown for developing new models to handle recursivity; we present a powerful analysis technique for a class of recursive programs. This technique achieves a global and precise description of the data flow and discovers important semantic properties of the programs.*

MOTS-CLÉS : *analyse de flot de données, compilation, parallélisation automatique, récursivité, langages algébriques.*

KEY WORDS : *data-flow analysis, compilation, automatic parallelisation, recursive programs, context-free languages.*

Cet article est paru dans Technique et Science Informatiques (TSI) 1999, volume 18, numéro 3.

1. Introduction

Les compilateurs optimiseurs et paralléliseurs modernes requièrent une étude très fine des propriétés statiques des programmes. La génération d'un code efficace par un compilateur est en effet subordonnée à la compréhension par celui-ci de la structure des programmes et de leurs interactions avec les données manipulées. Les techniques de parallélisation automatique pour programmes impératifs reposent 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 : deux écritures ne peuvent généralement pas être échangées sans altérer la sémantique du programme.

De nombreuses techniques d'analyse ont été développées pour découvrir les propriétés statiques des programmes. Depuis une dixaine d'années, des progrès importants ont eu lieu, tant sur la précision que sur la généralité des analyses [AHO 86, MAR 91]. Certaines formes de parallélisation automatique sont possibles avec de telles analyses [HAR 89] mais elles se limitent généralement à la détection d'interférences entre instructions. Plus récemment, la validation de transformations de code plus complexes a nécessité le recours à des techniques d'analyse plus précises que celles couramment utilisées dans les compilateurs optimiseurs [FEA 91, MAY 93]. Celles-ci sont malheureusement beaucoup moins générales que les techniques classiques ; elles sont notamment incapables de traiter convenablement les programmes récursifs. En effet, les premiers programmes que l'on a cherché à paralléliser étaient écrits en Fortran 77, langage n'autorisant pas la récursion. Pourtant, les programmes que l'on souhaite paralléliser aujourd'hui dépassent largement le cadre du calcul scientifique et sont écrits dans des langages de plus haut niveau.

L'organisation de cet article est la suivante. La section 2 présente la problématique liée à l'analyse de flot pour programmes récursifs, la section 3 expose les objectifs sur des exemples, la section 4 formalise les modèles proposés, les sections 5 et 6 présentent le fonctionnement de l'analyse et la section 7 propose un certain nombre d'applications à la compilation et à la parallélisation automatique.

2. Quelle analyse de flot ?

L'analyse de flot de données regroupant des techniques très diverses aux objectifs multiples, il est nécessaire de préciser le cadre dans lequel s'insère cette étude.

2.1. *Historique*

Les analyses statiques utilisées par les compilateurs optimiseurs classiques étudient généralement les propriétés sémantiques d'un programme au niveau de ses *instructions*. Les techniques de propagation de constantes, par exemple, suivent le graphe de flot de contrôle et examinent pour chaque instruction l'ensemble des variables susceptibles d'être modifiées. Les premières analyses de flot de données [AHO 86] ont été introduites pour unifier des techniques fondées sur la propagation de propriétés statiques le long du graphe de flot de contrôle. Elles peuvent également être vues comme des cas particuliers d'interprétation abstraite [COU 78, COU 81]. Le principe de base consiste à identifier pour chaque lecture dans une cellule mémoire (variable, élément de tableau, etc.) l'instruction qui a calculé la valeur qui s'y trouve, c'est-à-dire la dernière écriture dans cette même cellule. On associe ainsi à une variable une succession de *définitions* et d'*utilisations* (*use-def chains*).

Les travaux plus récents en parallélisation automatique ont montré cependant que la connaissance du flot des données au niveau des instructions n'est pas suffisante. Pour restructurer des nids de boucles Fortran opérant sur des tableaux, on s'aperçoit que les conflits d'accès à la mémoire doivent être caractérisés en fonction des *indices des boucles englobantes*. Il devient ainsi nécessaire de distinguer les différentes *instances d'une instruction à l'exécution*.

Définition

Une instance particulière d'une instruction au cours de l'exécution sera appelée *opération* (dans le cas des nids de boucles, une opération est simplement identifiée par une instruction et un vecteur constitué des indices des boucles englobantes).

2.2. *L'analyse de flot au niveau des opérations*

Les analyses de dépendances [BAN 88] ont été introduites pour caractériser les accès conflictuels entre *opérations* en termes de *producteurs* et *consommateurs*. Pour obtenir une caractérisation précise du flot des données, on raffine les relations de dépendance en déterminant, pour une lecture d'une cellule mémoire — le *puits* — la *dernière* écriture dans cette même cellule — la *source*. Ce type d'analyse porte naturellement le nom d'*analyse de flot de données* [FEA 91, MAY 93]. Elles décrivent le flot des données de manière beaucoup plus précise que les analyses classiques puisque les relations source-puits sont déterminées au niveau des *opérations*. Ce gain en précision s'effectue néanmoins au détriment de la généralité d'application, au départ réduite aux seuls nids de boucles dits *à contrôle statique* [FEA 91]. Depuis, de nombreuses contraintes ont été levées, autorisant une dégradation progressive de la précision en fonction de la proximité du programme avec le modèle à contrôle statique [BAR 97, WON 95].

Lorsqu'un programme comporte des procédures récursives, le flot de contrôle autour de celles-ci se comporte généralement de manière plus complexe qu'un simple nid de boucles. Il est parfois possible d'utiliser des techniques de dérécursivation et de ramener l'étude à une analyse de flot pour nids de boucles. Dans le cas général — et notamment en présence de récursion « arborescente » (au moins deux appels récursifs dans la même procédure) ou de récursion mutuelle (plusieurs procédures s'appellent mutuellement de manière récursive) — l'identification des différentes instances d'une instruction (les opérations) n'est plus possible avec des compteurs de boucle. Il est nécessaire de définir de nouveaux modèles plus généraux et mieux adaptés à la récursion.

Notre objectif premier étant la parallélisation automatique, on a choisi de prendre les idées des analyses de flot par opérations [FEA 91] comme point de départ, plutôt que d'améliorer des techniques fondées sur l'analyse abstraite [COU 78, AHO 86] à la manière de [HAR 89]. Dans un travail précédent [COH 96] des restrictions fortes sont imposées aux programmes récursifs afin de garantir le contrôle statique. On étudie ici une analyse un peu plus « floue » pour une classe plus générale de programmes récursifs. Il s'agit d'une analyse de flot au niveau des opérations, dans le sens où les relations source-puits concernent des instances d'instructions au cours de l'exécution, et non les instructions elles-mêmes.

La principale contribution de cet article est l'introduction des *langages algébriques* (on dit aussi *langage hors-contexte*) pour décrire le flot de données de programmes récursifs complexes. L'atout majeur de la méthode présentée est sa capacité à distinguer parmi les différentes instances à l'exécution des instructions.

3. Exemples introductifs

Sur deux programmes simples, on va calculer « à la main » les sources de quelques opérations effectuant des accès en lecture à un tableau unidimensionnel `A`. On introduira rapidement des notations et des définitions qui seront reprises plus formellement par la suite.

On ne cherchera pas dans cette étude à donner des sources *exactes*, mais simplement à déterminer une approximation conservative la plus fine possible de l'*ensemble des sources possibles*. C'est pourquoi on ne fera *aucune hypothèse sur les prédicats des instructions conditionnelles et les bornes de boucles*.

La source d'une opération lisant `A`$[c]$ est la dernière opération écrivant dans `A`$[c]$. La recherche de cette opération consiste donc à éliminer le plus possible d'écritures dans `A`$[c]$ dont on peut prédire statiquement qu'elles seront *recouvertes* par des écritures ultérieures. On parlera d'opérations *écrasées* par des sources possibles.

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

    begin
      F (0, 0)
    end
```

Figure 1. *Procédure* `F`

On utilise une syntaxe Pascal mais la méthode présentée ici est indépendante du langage. Considérons la procédure `F` de la figure 1, initiée par l'appel `F (0, 0)`. On étiquette les instructions par l, s et r.

Définition

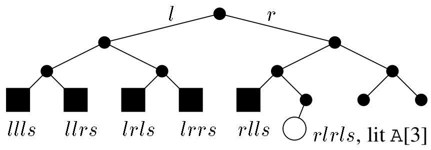
Le *mot de contrôle* d'une opération est formé de la concaténation des étiquettes des instructions de branchement menant à cette opération dans l'*arbre d'appel*. Par exemple l'instance ○ de s figure 2.a est exécutée après l'appel à r, l, r et l ; on la dénote par $rlrls$.

Dans le cas des programmes récursifs, cette notion de mot de contrôle représente l'état de la pile des branchements conduisant à une opération lors d'une exécution quelconque[1]. On va voir sur cet exemple (et on montrera au lemme 2) que l'utilisation des mots de contrôle permet de modéliser simplement l'ordre d'exécution d'un programme. Ceci nous permet d'identifier implicitement chaque opération à son mot de contrôle.

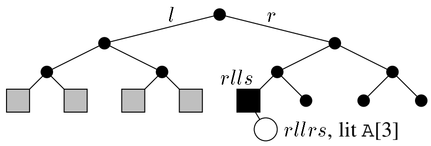
Définition

On note $\lhd$ l'ordre des instructions dans le programme. Ici, $l \lhd s \lhd r$.

Ainsi, l'ordre séquentiel d'exécution est celui d'un parcours d'arbre infixe dans notre exemple. En considérant les mots de contrôle des opérations, on observe que cet ordre infixe est en réalité l'ordre *lexicographique* $\ll$ *induit par* $\lhd$ sur le langage $\{l, s, r\}^*$. Ainsi : $llrs \ll llls$, $rlrs \ll lrrrrs$, etc. et les instances de s correspondantes s'exécutent bien dans le même ordre.



(2.a)
Procédure F, sources de $rlrls$.



(2.b)
Procédure F, sources de $rllrs$.

Figure 2. *Arbres d'appel partiels de* F

Comme on l'a annoncé précédemment, on ne fait aucune hypothèse sur les prédicats des instructions conditionnelles P_l et P_r. Puisque l'argument `l` (respectivement `r`) est incrémenté à chaque appel suivant l (respectivement r), l'instance $rlrls$ lit la valeur A[3]. Les sources de $rlrls$ possibles sont donc toutes les écritures *précédentes* dans A[3], non écrasées 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

1. Ainsi, il n'y a pas de rapport direct entre le mot de contrôle d'une opération et la notion de trace d'exécution.

sources possibles ; désignées par ■ figure 2.a (les instances de même profondeur accèdent au même élément).

L'opération $rllrs$ lit aussi A[3], et les instances précédentes assignant A[3] sont les mêmes que pour $rlrls$. Mais désormais, on sait que $rlls$ (précédant le puits $rllrs$) est exécutée, en tant qu'*ancêtre*[2] de $rllrs$. En effet, l'instruction d'écriture s n'est pas englobée par une conditionnelle, donc chaque appel à F exécute s. L'exécution d'une instance de s implique donc celle de tous ses ancêtres.

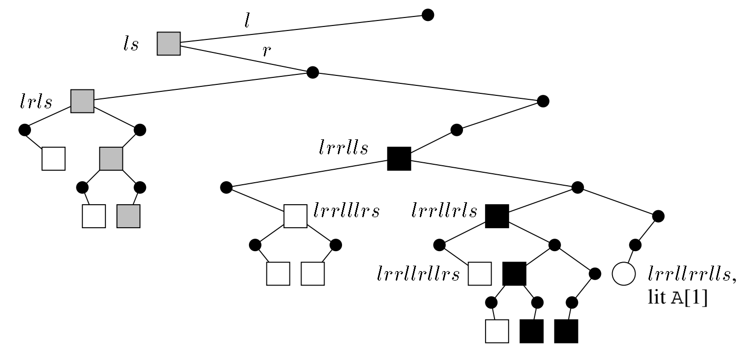
Les écritures précédant $rlls$ sont donc « écrasées », elles sont notées □ sur la figure 2.b. En conclusion, $rlls$ est la seule source possible de $rllrs$.

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

     begin
       G (0, 0)
     end
```

Figure 3. *Procédure* G

Considérons à présent la procédure G figure 3, initiée par G (0, 0). L'index de A étant r-l, les sources de $lrrllrrlls$ se situent parmi les écritures dans A[1].



Procédure G, sources de $lrrllrrlls$.

Figure 4. *Arbre d'appel partiel de* G

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

Il apparaît que $lrrlls$, ancêtre de $lrrllrrlls$, écrit dans `A[1]` (figure 4). Donc toutes les écritures précédant $lrrlls$ sont écrasées (ls, $lrls$, etc.). De plus, certaines écritures dans `A[1]` — représentées par □ — s'exécutent *avant* un de leurs ancêtres écrivant dans `A[1]`. Ainsi lorsque $lrrllrllrs$ est exécutée, $lrrllrls$ l'est aussi ; $lrrllrllrs$ n'est donc pas une source possible. On en déduit que les sources possibles sont les ■.

L'objectif de notre analyse de flot de données 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.

4. Formalisation du problème

Le modèle d'exécution des nids de boucle et l'identification des opérations par les compteurs de boucles ne sont pas transposables aux programmes récursifs. On présente donc un nouveau modèle de programme ainsi qu'une stratégie d'identification des opérations (généralisant les notions vues sur les exemples de la section 3).

4.1. *Modèle de programme*

On passe en revue dans les paragraphes suivants la liste des contraintes imposées aux programmes Pascal considérés.

Structures de contrôle

On autorise les appels de procédure récursifs ou non, les boucles, et les instructions conditionnelles quelconques. Les `goto`s et les exceptions ne sont pas autorisés.

Structures de données

On considère des variables scalaires globales (vues comme tableaux à un seul élément) et des tableaux de scalaires *globaux*.

Dans un premier temps, les tableaux sont supposés unidimensionnels.

Variables inductives

On appelle *variable inductive* tout compteur de boucle ou tout argument de procédure récursive vérifiant les deux contraintes suivantes :

- L'initialisation d'une variable inductive — à l'entrée d'une boucle ou lors du premier appel à une procédure récursive — doit être une fonction affine de variables inductives et de constantes symboliques. Par exemple, la procédure `F` est appelée initialement avec $l = r = 0$.

- Les variables inductives intervenant dans les index de tableaux ne peuvent être que *translatées* (incrémentées d'une constante) lors des appels *récursifs*[3]. Par exemple, la procédure `F` comporte deux variables inductives apparaissant dans l'index du tableau `A` : l et r ; elles sont toutes deux incrémentées lors des appels récursifs.

Accès aux tableaux

Les index de tableaux sont fonctions affines de variables inductives et de constantes symboliques.

Pour simplifier, on supposera que les instructions accédant aux tableaux ne sont pas les mêmes que les instructions effectuant des appels de procédure ou les instructions de boucle (l'introduction de variables temporaires permet de mettre un programme quelconque sous cette forme).

On notera que les nids de boucles dont les index de tableaux sont des fonctions affines des compteurs de boucles rentrent bien dans le cadre de ce modèle.

Remarque

On ne fera par la suite *aucune hypothèse sur les prédicats des conditionnelles et les bornes de boucles*.

4.2. *Modélisation du flot de contrôle*

On suppose que les instructions du programme sont étiquetées dans un ensemble Σ. On a $\Sigma = \{l, r, s\}$ sur les procédures `F` et `G`. Pour simplifier, on suppose que les conflits d'accès entre instructions s'effectuent tous sur le même tableau `A`. Dans le cas général, il suffira de répéter l'analyse pour chaque structure de données.

Pour identifier les différentes instances d'une instruction, on généralise la définition de *mot de contrôle* présentée dans les exemples. Lorsque les seules structures de contrôle sont des appels de procédure (gardés par des instructions conditionnelles), il suffit de concaténer les étiquettes des appels menant à une instance donnée (la pile des appels de procédure en quelque sorte) avec l'étiquette de l'instruction considérée.

En présence d'une boucle **for** ou **while**, on tire parti de l'existence d'une écriture récursive équivalente de la boucle en concaténant l'étiquette de celle-ci autant de fois que la valeur de l'indice de boucle. Concrètement, on associe deux étiquettes à chaque boucle :

- la première désigne l'entrée dans la boucle ; elle est analogue au premier appel à une procédure récursive ;

- la deuxième désigne l'itération de la boucle ; elle est analogue à un *appel récur-*

3. En réalité — pour simplifier l'exposé du modèle du programme — cette définition ne convient que pour les boucles et la récursion simple (impliquant une seule procédure). En cas de récursion mutuelle, il faut prendre garde aux phénomènes de « renommage » des variables par passage de paramètres. Il est alors nécessaire d'imposer une contrainte supplémentaire portant sur la « circulation » des variables inductives le long des *cycles* du graphe de flot de *contrôle*

sif s'effectuant après toutes les instructions du corps de la boucle.

Enfin, on note L l'ensemble des mots de contrôle étiquetant les instances possibles des instructions *qui accèdent au tableau* A.

```
      program P;

      procedure X (i) begin
a       if (Pa) then Y (N);
b       if (Pb) then X (i+1);
s       A[i] := ···
      end
      procedure Y (j) begin
C, c    for k = 0 to N do
t         ··· := ··· A[i] ···;
d       if (Pd) then Y (j-1)
      end

      begin
        X (0)
      end
```

Figure 5. *Un exemple plus complexe*

Le programme de la figure 5 permet d'illustrer ces définitions. La notation C, c précédant la boucle **for** signifie que C est l'étiquette de l'entrée dans boucle, et c celle de l'itération ($\Sigma = \{a, b, s, C, c, t, d\}$ pour le programme P).

- L'instruction s fait partie du corps de la procédure X, laquelle n'est appelée que par elle-même et par le programme principal ; les mots de contrôle des instances de l'instruction s sont donc de la forme b^*s.

- La boucle C fait partie du corps de la procédure Y qui est appelée par elle même et par X ; les instances des instructions du corps de cette boucle sont donc identifiées par des mots de contrôle commençant par b^*ad^*C. De plus, la k-ième itération de la boucle est identifiée par c^k ; donc les mots de contrôle des instances de l'instruction t sont de la forme $b^*ad^*Cc^*t$.

- Le langage des mots de contrôle des instructions affectant A est donc

$$L = b^*s + b^*ad^*Cc^*t.$$

On vérifie alors facilement le résultat suivant.

Lemme 1 *En l'absence d'hypothèse sur les bornes de boucles et les prédicats des expressions conditionnelles, L est un langage rationnel (on dit aussi langage régulier).*

Pour tout programme respectant le modèle de la section 4.1, on construit ainsi une *injection* naturelle des *opérations* du programme vers son langage des mots de contrôle

L.

On dispose d'un ordre naturel sur les opérations du programme : l'ordre séquentiel d'exécution. Afin de préserver cet ordre sur les mots de contrôle, on introduit un ordre *partiel* noté $\lhd$ sur les *instructions* du programme. On suppose que les instructions sont ordonnées à l'intérieur d'un même bloc (de procédure, boucle, conditionnelle, etc.) suivant l'ordre textuel du programme. Dans le programme de la figure 5 on a ainsi : $a \lhd b \lhd s$ et $C \lhd d$ (il s'agit bien de C et non pas c !). De plus, on suppose que les instructions d'un bloc de boucle précèdent l'instruction d'itération elle même [4]. Dans le programme de la figure 5 on a ainsi : $t \lhd c$. Les instructions entre blocs différents ou dans des branches distinctes d'une conditionnelle sont supposées incomparables. Cet ordre partiel permet de définir un ordre *lexicographique* partiel $\ll$ sur Σ^*, et donc sur le langage L des mots de contrôle. Cet ordre n'est en général pas total sur L puisque peuvent coexister deux mots de contrôle d'opérations situées dans deux branches distinctes d'une conditionnelle. Par construction de $\lhd$ on peut énoncer le résultat suivant.

Lemme 2 *Une opération ω_1 s'exécute avant une opération ω_2 si et seulement si leurs mots de contrôle respectifs w_1 et w_2 vérifient $w_1 \ll w_2$.*

On peut désormais identifier l'ensemble des opérations ordonné par l'ordre séquentiel d'exécution au langage L des mots de contrôle muni de l'ordre lexicographique $\ll$. On confondra donc l'opération ω avec son mot de contrôle w.

Remarque

Comme par hypothèse les appels de procédure et les instructions de boucle ne comportent pas d'accès aux tableaux, on peut affirmer que les mots de L se terminent nécessairement par l'étiquette d'une instruction d'affectation simple.

4.3. *Paramétrage des index de tableaux*

Dans les nids de boucles à contrôle statique [FEA 91], les index de tableaux sont des fonctions affines des compteurs de boucles. De la même manière, on souhaite relier les index de tableaux accédés par une opération au mot de contrôle de celle-ci.

On montre que les restrictions sur les arguments des appels de procédure imposées par le modèle de programme permettent de *prévoir statiquement* le comportement des variables inductives. En effet, la présence de l'étiquette d'un appel de procédure ou d'une boucle dans un mot de contrôle est synonyme de translation des variables inductives ou d'initialisation de celles-ci à une fonction affine [5].

Lemme 3 *Soit α étiquetant une instruction autre qu'une affectation, β étiquetant un appel récursif et* `i` *une variable inductive « visible » à la fois dans le bloc syntaxique* [6] *représenté par α et le bloc syntaxique de la procédure appelée par β. Soient u et v deux*

4. Cette définition qui semble contre-intuitive vient du fait que l'on considère que l'itération de la boucle a lieu *après* l'exécution du bloc.

5. L'initialisation n'est pas permise dans les appels *récursifs*, il n'y a donc pas de *réinitialisation*.

6. Un corps de boucle, de procédure, de conditionnelle...

mots tels que $u\alpha\beta v$ soit le mot de contrôle d'une opération valide.

Alors les valeurs de `i` au points $u\alpha$ et $u\alpha\beta$ diffèrent d'une constante indépendante de u et v.

Preuve

Il suffit de remarquer que β — instruction du bloc syntaxique α — est un appel récursif. L'appel à β ne peut donc que translater la variable d'induction `i`. □

On en déduit, par induction sur le mot de contrôle, que les variables inductives sont des fonctions affines du nombre d'occurrences des étiquettes des instructions dans le mot de contrôle d'une opération. En d'autres termes, les variables inductives sont des fonctions affines du vecteur de Parikh [PAR 66] du mot de contrôle courant.

Un index de tableau est donc une fonction affine du nombre d'occurrences des étiquettes dans le mot de contrôle d'une opération.

Définition

On peut définir une fonction de l'ensemble des opérations du programme dans la mémoire, associant à chaque mot de contrôle un indice de tableau. Cette fonction est appelée *fonction d'accès*. On parle aussi de fonction d'accès en lecture (resp. écriture), restreinte aux opérations de lecture (resp. d'écriture) au cours de l'exécution du programme.

On notera $|w|_\alpha$ le nombre d'occurrences de l'étiquette α dans le mot de contrôle w. Pour la procédure `G` par exemple, la fonction d'accès en lecture au tableau `A` dans l'instruction s est $\varphi(w) = |w|_l - |w|_r$. On en déduit le résultat important suivant.

Lemme 4 *Considérons une procédure récursive* $\mathtt{Z}\,(a_1, \ldots, a_n)$*, appelée initialement par* $\mathtt{Z}\,(c_1, \ldots, c_n)$*, comportant un appel récursif* α *de la forme* $\mathtt{Z}\,(a_1 + d_1, \ldots, a_n + d_n)$*, et un index de tableau* $\mathtt{A}[a_0 + \lambda_1 a_1 + \cdots + \lambda_n a_n]$*. Si le mot de contrôle est* w*, le coefficient de* $|w|_\alpha$ *de la fonction d'accès sera*

$$\lambda_1 d_1 + \cdots + \lambda_n d_n,$$

et le terme constant de la fonction d'accès sera

$$a_0 + \lambda_1 c_1 + \cdots + \lambda_n c_n.$$

Le modèle de programme retenu permet ainsi d'écrire tous les accès au tableaux en fonction du mot de contrôle et de constantes symboliques.

4.4. *Caractérisation des sources*

Étant donné une opération w lisant le tableau `A` avec la fonction d'accès φ ; une opération w' écrivant `A` avec la fonction d'accès φ' est la source de w si et seulement si elle est la *plus grande opération* au sens de $\ll$ qui vérifie les trois prédicats suivants.

Prédicat de séquencement

« La source est exécutée avant le puits ». w' s'exécute avant w s'écrit simplement $w' \ll w$, d'après le lemme 2. On remarquera que la connaissance de l'ordre d'exécution des opérations ne préjuge pas de l'exécution effective de celles-ci.

Équation de conflit

« La source et le puits accèdent la même cellule mémoire (en écriture et en lecture respectivement) ». À l'aide des fonctions d'accès, on obtient

$$\varphi(w) = \varphi'(w'). \tag{1}$$

C'est une équation affine en les nombres d'occurrences des étiquettes des instructions, d'après le lemme 4.

Prédicat d'existence

« La source est effectivement exécutée ». Aucune hypothèse n'étant effectuée sur les prédicats des expressions conditionnelles et les bornes de boucles, les seules informations dont on dispose sont des propriétés structurelles du programme. On présentera celles que l'on prend en considération dans la section 5.

Comme on ne peut connaître exactement le *prédicat d'existence*, on se contentera d'approximations conservatives : par défaut une opération est considérée comme exécutée, à moins qu'on ne puisse prouver le contraire. Une opération puits peut ainsi avoir un *ensemble de sources possibles*.

5. Description des sources à l'aide de langages algébriques

Dans toute cette section on choisit de fixer une instruction de lecture r et une instruction d'écriture s, et on ne cherchera à décrire que les sources des instances de r qui se situent parmi les instances de s.

5.1. *Préliminaires*

On a vu dans la section 4.4 que la reconnaissance des sources nécessite la satisfaction d'une équation affine et de critères liés à l'ordre lexicographique. D'où l'idée d'un automate doté d'un *compteur* capable de repérer les conflits et de garantir l'ordre lexicographique.

D'après [HOP 79, BER 79], un automate à compteur peut être simulé par un automate à pile. Trois symboles de pile suffisent : le symbole Z de fond de pile détecte l'annulation, I code en un-aire les nombres positifs, D les nombres négatifs. Les transitions peuvent ajouter ou soustraire des *constantes* à l'aide d'incrémentations et de décrémentations successives.

Cependant, notre objectif n'est pas de décrire les sources possibles d'une opération puits donnée, mais bien de déterminer une *relation* entre puits et sources. Cette relation, élément de l'ensemble des parties de $L \times L$, peut être vue comme une fonction de L vers les parties de L qui associe à une opération l'ensemble de ses sources possibles. Les *transducteurs à pile* [BER 79] — plus que les automates à pile — apparaissent ainsi comme les objets mathématiques privilégiés pour notre analyse de flot : on peut voir les transducteurs comme des automates dont les transitions peuvent écrire des mots sur un ruban de sortie.

Le résultat de notre analyse pourrait donc être représenté par un transducteur à pile reconnaissant les puits en entrée et produisant les sources possibles correspondantes en sortie. Les analogues fonctionnels de ces transducteurs sont les *transductions algébriques* (on dit aussi *transduction hors-contexte*), qui sont bien dans notre cas des fonctions de L vers les parties de L.

Afin de ne pas compliquer cette présentation par une introduction à la théorie des transductions algébriques, on utilise le théorème de Nivat [EIL 74] (employé comme définition des transductions algébriques dans [BER 79]).

Théorème 1 (Nivat) *Une transduction $\tau : X^* \to Y^*$ est algébrique si et seulement s'il existe un alphabet Z, deux morphismes de monoïdes $\alpha : Z^* \to X^*$ et $\beta : Z^* \to Y^*$, et un langage algébrique $A \subset Z^*$ tels que*

$$\forall w \in X^* \ : \ \tau(w) = \beta(\alpha^{-1}(w) \cap A).$$

Ainsi, on peut décomposer une transduction algébrique en la donnée de deux morphismes et d'un langage algébrique. Dans notre cas, on va pouvoir ramener la définition d'une relation sur des couples de mots (w, w') à la construction d'un automate reconnaissant un *encodage* particulier de ces couples sur un mot unique. Pour cela, on rappelle la définition de l'ordre lexicographique $\ll$:

$$w' \ll w \iff w = u\alpha v, \ w' = u\alpha' v', \ \alpha' \lhd \alpha, \ u \in \Sigma^*, \ \alpha v, \alpha' v' \in \Sigma^* s. \tag{2}$$

Au lieu de caractériser des couples puits-source de la forme (w, w'), on cherchera à caractériser des mots de la forme $u \bullet \alpha v \bullet \alpha' v'$ où $\bullet$ est un symbole de séparation, tels que $w' = u\alpha' v'$ soit source de $w = u\alpha v$.

Pour w et w' donnés, il existe un unique mot $u \bullet \alpha v \bullet \alpha' v'$ vérifiant $w = u\alpha v$, $w' = u\alpha' v'$, et $\alpha' \lhd \alpha$. Réciproquement, la détermination de w et w' est rendue possible par les deux séparateurs $\bullet$. La transformation est donc bijective, et on notera $\langle w \bullet w' \rangle$ le mot $u \bullet \alpha v \bullet \alpha' v'$.

5.2. *Encodage des prédicats caractérisant les sources*

Étant donné un mot $\langle w \bullet w' \rangle = u \bullet \alpha v \bullet \alpha' v'$, la vérification de l'ordre lexicographique $w' \ll w$ est élémentaire : il suffit de garantir que $\alpha' \lhd \alpha$. En termes d'automate, on consacre des états à la mémorisation de l'étiquette α le temps de lire v en entier, puis $\bullet$ puis de comparer avec α'.

L'équation de conflit (1) devient

$$\varphi(u) + \varphi(\alpha) + \varphi(v) = \varphi'(u) + \varphi'(\alpha') + \varphi'(v'). \tag{3}$$

Le rôle du compteur de l'automate consistera par conséquent à *détecter l'annulation* de $\varphi(u) + \varphi(\alpha) + \varphi(v) - \varphi'(u) - \varphi'(\alpha') - \varphi'(v')$. On décompose l'évaluation de l'expression en suivant les trois « modules » de l'encodage $\langle w \bullet w' \rangle$.

1. Lors de la lecture de u, l'automate devra « mémoriser » $\varphi(u) - \varphi'(u)$.
2. Lors de la lecture de αv, il devra mettre à jour $(\varphi(u) - \varphi'(u)) + \varphi(\alpha) + \varphi(v)$.
3. Enfin, lors de la lecture de $\alpha' v'$, il devra détecter l'annulation de $\varphi(u) - \varphi'(u) + \varphi(\alpha) + \varphi(v) - \varphi'(\alpha') - \varphi'(v')$.

L'intégration des propriétés structurelles du programme est nécessaire pour garantir ou infirmer l'exécution de certaines opérations. On va reprendre les propriétés de parenté utilisées dans la section 3 de manière plus formelle.

Lorsque l'instruction s d'écriture[7] n'est pas gardée par une conditionnelle, elle est systématiquement exécutée à chaque itération de la boucle englobante ou à chaque appel de la procédure qui la contient. On peut alors partitionner les opérations d'écriture qui vérifient simultanément le prédicat de séquencement et l'équation de conflit en fonction de leur « parenté » avec une instance de s.

□ désigne les écritures en conflit écrasées par un ancêtre « postérieur » en conflit. Formellement : w' est de type □ si et seulement s'il existe $u \in \Sigma^*$ préfixe de w' tel que $w' \ll us \ll w$ et $\varphi'(us) = \varphi(w)$.

▒ désigne les écritures qui sont écrasées par un ancêtre du *puits* mais n'ont pas d'ancêtre postérieur source. Formellement : w' est de type ▒ si et seulement si w' n'est pas de type □ et s'il existe $u \in \Sigma^*$ préfixe de w tel que $w' \ll us \ll w$ et $\varphi'(us) = \varphi(w)$.

■ désigne les autres écritures en conflit.

Une opération w' de type □ est écrasée par une instance de x dont on peut garantir qu'elle est exécutée lorsque w' l'est. Une opération w' de type ▒ est écrasée par une instance de s dont on peut garantir qu'elle est exécutée (la recherche des sources d'une opération w suppose implicitement que celle-ci est exécutée). Les seules sources possibles sont donc à chercher parmi les ■.

L'exploitation d'autres propriétés structurelles (exclusion mutuelle, etc.) est laissée pour des travaux ultérieurs ; on considérera donc que l'ensemble des sources possibles d'une opération w est l'ensemble des w' vérifiant le prédicat de séquencement et l'équation de conflit qui sont de type ■.

Théorème 2 ______________

7. On se limite pour l'instant à une seule instruction d'écriture.

5.3. *Construction de l'automate*

Le résultat fondamental est le suivant.

Pour deux instructions de lecture et d'écriture données r et s, on peut construire un automate à pile (ou une grammaire hors-contexte) acceptant $\langle w \bullet w' \rangle$ lorsque les trois conditions suivantes sont vérifiées :

(i) $w' \ll w$ *;*

(ii) $\varphi(w) = \varphi'(w')$ *;*

(iii) w' *n'est pas de type* $\square$.

Autrement dit, l'automate reconnaît, étant donné un puits w, l'ensemble des opérations de type $\textcolor{gray}{\blacksquare}$ *et* $\blacksquare$.

Preuve

On a vu dans la section 5.2 que l'ordre lexicographique est élémentaire à vérifier sur un encodage $\langle w \bullet w' \rangle$. Aucune pile n'est nécessaire pour cela, l'ensemble des $\langle w \bullet w' \rangle$ tels que $w' \ll w$ est un langage rationnel. L'intersection d'un langage algébrique et d'un langage rationnel étant algébrique, il suffit de montrer que les conditions (ii) et (iii) peuvent être reconnues par un automate à pile.

D'après le lemme 4, le calcul de la différence des fonctions d'accès s'effectue à l'aide de l'addition ou de la soustraction d'une constante à chaque lecture d'une étiquette. Un compteur (simulé par une pile) peut donc parfaitement le réaliser. Comme on l'a vu à la section 5.1, la détection du conflit $\varphi(w) - \varphi'(w') = 0$ peut s'effectuer à la lecture du symbole Z de fond de pile. La condition (ii) peut donc être reconnue par un automate à pile. Lorsque s (l'instruction d'écriture considérée) n'est pas englobée dans une instruction conditionnelle, il peut exister des opérations de type $\square$. L'élimination de celles-ci s'effectue également lors de la détection d'un conflit : il suffit d'interdire la lecture d'une étiquette d'instruction précédant s pour l'ordre $\lhd$. En effet, si $uv_1s \ll w$ et $\varphi(w) = \varphi'(uv_1s)$, toutes les opérations de la forme uv_1xv_2 avec $x \lhd s$ telles que $\varphi(w) = \varphi'(uv_1xv_2)$ seront écrasées par uv_1s. □

On notera que lorsque l'instruction s d'écriture dans `A` est englobée par des conditionnelles, il est impossible de garantir qu'un ancêtre du puits est effectivement exécuté, la condition (*iii*) 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) en décrivant exactement l'ensemble des $\blacksquare$. Inversement, lorsque une instruction d'écriture dans `A` n'est pas gardée par une conditionnelle, il n'est pas possible de reconnaître exactement l'ensemble des $\blacksquare$ à l'aide d'un automate à pile : en effet, éliminer les $\textcolor{gray}{\blacksquare}$ nécessite un deuxième compteur indépendant du compteur « détecteur de conflit », et les automates à deux compteurs ne sont pas en général des automates à pile.

5.4. *Flot des données global*

On avait choisi jusqu'ici de se limiter à deux instructions particulières r et s, mais pour obtenir l'intégralité de l'analyse de flot, il nous faut prendre en compte l'ensemble des instructions du programme.

La solution la plus simple consiste à construire les automates à pile pour tous les couples d'instructions de lecture et d'écriture, puis à calculer l'automate « union » de tous ceux-ci (la classe des langages algébriques est close par union). Cependant, certaines opérations seront ainsi déclarées sources possibles alors qu'il est clair que des instances d'une autre instruction d'écriture les écrasent. On retiendra néanmoins cette solution pour simplifier l'analyse.

Par conséquent, le théorème 2 nous permet d'établir la possibilité de construire un automate à pile décrivant une sur-approximation du flot des données d'un programme.

Une description exhaustive d'un algorithme pour construire l'automate correspondant à l'analyse de flot de données d'un programme quelconque serait trop longue et fastidieuse. On se contentera d'en esquisser les articulations majeures sur un exemple.

6. Exemple de la procédure `G`

Dans le cas de la procédure `G`, on a $\varphi(w) = \varphi'(w) = |w|_l - |w|_r$, donc en particulier $\varphi(u) - \varphi'(u) = 0$. Ceci nous permet d'écrire l'automate à pile décrivant le flot des données de la procédure `G`.

Celui-ci comporte 7 états, $\mathbf{q_0}$ est l'état initial et $\mathbf{q_6}$ l'état terminal. La pile contient initialement le symbole Z de fond de pile. L'état $\mathbf{q_0}$ correspond à la lecture du préfixe u commun à l'écriture $w' = u\alpha' v'$ et à la lecture $w = u\alpha v$. Les transitions $\mathbf{q_1} \to \mathbf{q_2}$ et $\mathbf{q_1} \to \mathbf{q_4}$ réalisent les différents choix possibles sur α (r ou s) . $\mathbf{q_2}$ correspond à la lecture de v. Les transitions $\mathbf{q_4} \to \mathbf{q_5}$, $\mathbf{q_3} \to \mathbf{q_5}$ et $\mathbf{q_3} \to \mathbf{q_6}$ réalisent les différents choix possibles sur α' (l lorsque $\alpha = s$; et l ou s lorsque $\alpha = r$). Enfin, $\mathbf{q_5}$ correspond à la lecture de v', et les transitions vers $\mathbf{q_6}$ achèvent la reconnaissance de $\langle w \bullet w' \rangle$.

6.1. *L'automate*

Le fonctionnement de l'automate proposé à la figure 6 s'énonce relativement simplement :

1. lire le préfixe u (état $\mathbf{q_0}$) et le symbole $\bullet$ (état $\mathbf{q_1}$)[8] ;

2. lire α et v en « mémorisant » $\varphi(\alpha) + \varphi(v)$ sur la pile ($\mathbf{q_2}$), jusqu'à ce que s et $\bullet$ soient lus ($\mathbf{q_3}$ 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

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

nulle, jusqu'à ce que s soit lu dans une configuration où la fonction de conflit est nulle ($\mathbf{q_6}$).

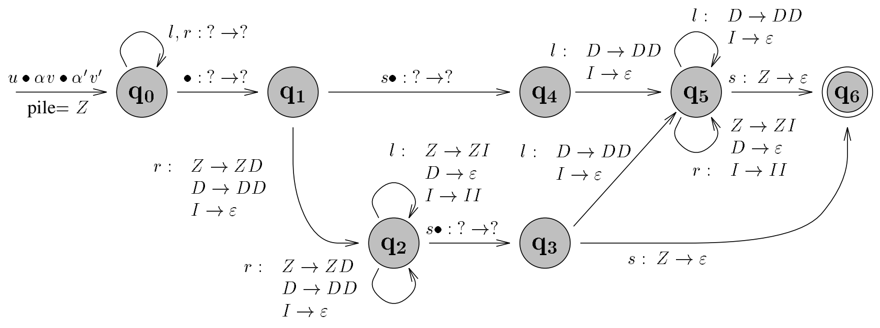


Figure 6. *Automate à pile décrivant le flot de la procédure* `G`

L'automate reconnaît toutes les sources possibles. Les écritures de type □ (dont un ancêtre *postérieur* est source possible) 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, comme on l'a annoncé dans la section précédente, les écritures de type ▒ précédant un ancêtre de w ne sont pas éliminées.

6.2. *Grammaire hors-contexte*

On peut calculer simplement l'expression de la grammaire associée à l'automate [HOP 79]. L'expression générale de la grammaire n'est pas plus informative que celle de l'automate, mais peut rendre plus évidentes certaines propriétés des ensembles de sources.

En particulier, dans le cas de la procédure `G`, on note que les sources du sous-arbre d'une source w' (les ■ issus de w') constituent le langage de Lukasiewicz [BER 79] :

$$\mathrm{S} \vdash w' \cdot s \,|\, w' \cdot rLs \qquad\qquad L \vdash l \,|\, rLL.$$

7. Évaluation et développements

L'étude précédente a montré la possibilité de définir des modèles et des algorithmes se prêtant à l'analyse de flot de données de programmes récursifs particuliers. Il reste cependant à étudier la portée générale ce ceux-ci et les applications que l'on peut faire d'une description du flot des données par une grammaire hors-contexte.

7.1. *Précision de l'analyse*

Le prédicat de séquencement et l'équation de conflit peuvent toujours être reconnus sans approximation. De plus les seules propriétés structurelles que l'on a considérées (existence d'un ancêtre en conflit) ne permettent pas d'infirmer l'existence d'une opération, mais au contraire de garantir l'existence de certaines ; ainsi toutes les opérations du langage L sont par défaut supposées exécutées. On remarquera donc que la relation décrivant le flot des données est toujours incluse dans l'ensemble des dépendances producteur-consommateur (aussi appelées dépendances vraies ou dépendances de flot) du programme : cette analyse de flot est donc équivalente dans le pire cas à une analyse de dépendances producteur-consommateur.

Le problème de la détermination du flot global est assez technique. L'approximation induite par la solution simple de la section 5.4 n'est elle pas trop coûteuse ? Il s'agit de comparer les sources provenant de deux instructions d'écriture s_1 et s_2 différentes. Une instance de s_1 peut être ancêtre postérieur d'une instance de s_2 et écraser celle-ci. On peut détecter ce phénomène avec le compteur unique d'un automate à pile lorsque les fonctions d'accès de s_1 et s_2 sont identiques (ou égales à une constante près). En général, le problème se ramène à l'intersection de deux langages algébriques, qui n'est pas algébrique. Heureusement on peut penser que de tels conflits se produisent rarement si les accès sont quelconques, c'est pourquoi l'approximation consistant à prendre l'union proposée à la section 5.4 est à priori raisonnable.

Enfin, on a vu sur le calcul « à la main » des sources de la procédure `G` que la perte de précision liée aux écritures de type $\blacksquare$ est tout à fait acceptable. Qu'en est-il dans le cas général du modèle de programmes considéré ?

Pour certains programmes, le fait de considérer que les écritures de type $\blacksquare$ sont des sources possibles n'est pas satisfaisant : notamment lorsque les seules sources possibles sont en réalité des ancêtres du puits. On risque alors d'approximer un nombre fini de sources possibles par un ensemble potentiellement infini ! Le résultat de l'analyse reste néanmoins plus précis qu'une analyse de dépendances (qui reconnaîtrait également les écritures de type $\square$).

De plus, la description du flot étant effectuée au niveau des opérations et non au niveau des instructions, la quantité d'information obtenue par notre analyse ne peut être que supérieure à celles des analyses de flot conventionnelles [AHO 86, MAR 91, HAR 89] (même lorsqu'elles sont capables de traiter les programmes récursifs).

Pour des applications classiques de l'analyse de flot (vérification de code, optimisation, mise en assignation unique par renommage des variables, élimination de code mort, compréhension de programmes, gestion optimisée de la mémoire), on peut ainsi penser que la précision de la description du flot par des langages algébriques est bonne pour les programmes récursifs considérés (sachant qu'aucune hypothèse n'est faite sur les bornes de boucles et les prédicats des instructions conditionnelles). En revanche, sur certains programmes, la connaissance des sources risque de ne pas être suffisamment bonne pour valider des restructurations de code complexes (mise en assignation unique, expansion de tableaux, privatisation, ordonnancement) nécessaires à la mise en évidence de parallélisme.

7.2. *Portée du modèle de programme*

Le problème majeur que l'on rencontre pour élargir le modèle de programme est celui de la non clôture par intersection de la classe des langages algébriques. Les tableaux à plusieurs dimensions posent problème en particulier, car il n'est pas possible de mettre à jour simultanément deux compteurs à l'aide d'un automate à pile. Une solution peu satisfaisante consiste néanmoins à « linéariser » les tableaux. De plus il n'est pas possible de prendre en compte les bornes des boucles et les prédicats des instructions conditionnelles sans faire appel à de nouvelles inégalités affines et donc sans sortir de la classe des langages algébriques.

7.3. *Exploitation de l'analyse*

La description du flot des données à l'aide de grammaires hors-contexte n'aurait aucun intérêt si celles-ci n'étaient pas exploitables par des algorithmes de vérification, d'optimisation ou de parallélisation. Fort heureusement, de nombreuses propriétés sémantiques se traduisent en problèmes décidables sur des grammaires hors-contexte.

- On peut décider si w' est une source possible de w. Par ailleurs, on a vu que l'automate à pile décrivant les sources de la procédure `G` est déterministe. Décider si w' est une source possible de w peut donc s'effectuer en temps linéaire sur la taille des mots de contrôle. Nous conjecturons que ce résultat se généralise à tout le modèle de programmes considéré. Cette question est capitale pour exploiter *efficacement* la description du flot.

- Le calcul de l'ensemble des opérations dont l'ensemble des sources associées est vide permet de détecter les lectures de cellules mémoire non initialisées. Mieux encore, on peut identifier exactement opération par opération les accès illégaux. Cette information permet également d'éliminer du code mort (i.e. des écritures en mémoire non suivies de lectures).

- L'intersection d'un langage rationnel et d'un langage algébrique est algébrique. Toute propriété décidable sur les sources d'une opération donnée est donc décidable pour un ensemble rationnel d'opérations. L'exploitation des propriétés sémantiques décrites par l'analyse de flot peut ainsi se faire de manière globale et paramétrique.

Ces résultats de décidabilité prouvent que la description des sources par des langages algébriques met en évidence des propriétés sémantiques importantes. Afin de paralléliser automatiquement des programmes récursifs, il reste à développer des techniques de privatisation de tableau, d'expansion et d'ordonnancement parallèle à partir de grammaires hors-contexte.

8. Conclusion

On a présenté une analyse de flot de données des accès aux tableaux adaptée aux programmes récursifs. Le contrôle étant dynamique, on ne peut calculer que des ensembles de sources possibles. Afin de les décrire on introduit un modèle fondé sur les langages algébriques. Malgré une approximation sur les langages de sources, la précision obtenue est bien supérieure à ce qu'il est possible d'obtenir par une méthode classique sur le modèle de programmes considéré. Il est bien évident que de nombreux travaux seront nécessaires pour étudier l'élargissement du modèle de programmes et pour affiner la précision de l'analyse.

Les propriétés de décidabilité des grammaires hors-contexte permettent de détecter automatiquement de nombreuses propriétés du flot des données, et d'envisager de nombreuses applications, notamment en parallélisation automatique. L'adaptation des techniques de vérification et de transformation de programme à ce modèle de description du flot requiert bien entendu le développement de nouveaux algorithmes.

L'intérêt principal de cette étude est de mettre en évidence la nécessité et la possibilité d'imaginer des modèles alternatifs aux modèles polyédriques utilisés classiquement en parallélisation automatique de nids de boucles. Nous pensons ainsi que ces idées contribueront à l'extension du domaine d'action et des performances des compilateurs paralléliseurs.

9. Bibliographie

[AHO 86] AHO A. V., SETHI R. et ULLMAN J. D., *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Mass, 1986.

[BAN 88] BANERJEE U., *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, 1988.

[BAR 97] BARTHOU D., COLLARD J.-F. et FEAUTRIER P., « Fuzzy Array Dataflow Analysis ». *Journal of Parallel and Distributed Computing*, vol. 40, p. 210–226, 1997.

[BER 79] BERSTEL J., *Transductions and Context-Free Languages*. Teubner, 1979.

[COH 96] COHEN A., COLLARD J.-F. et GRIEBL M., « Array Data-flow Analysis for Imperative Recursive Programs ». Rapport technique 96/035, PRiSM, Université de Versailles, September 1996.

[COU 78] COUSOT P. et COUSOT R., « *Formal descriptions of programming concepts* », Chapitre Static determination of dynamic properties of recursive procedures, p. 237–277. North-Holland, 1978.

[COU 81] COUSOT P., « *Program Flow analysis: theory and applications* », Chapitre Semantic foundations of programs analysis, p. 303–342. Prentice-Hall, 1981.

[EIL 74] EILENBERG S., *Automata, Languages and Machines*. Academic Press, 1974.

[FEA 91] FEAUTRIER P., « Dataflow Analysis of Scalar and Array References ». *Int. Journal of Parallel Programming*, vol. 20, n° 1, p. 23–53, February 1991.

[HAR 89] HARRISON W. L., « The interprocedural analysis and automatic parallelisation of Scheme programs ». *Lisp and Symbolic Computation*, vol. 2, n° 3, p. 176–396, October 1989.

[HOP 79] HOPCROFT J. E. et ULLMAN J. D., *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[MAR 91] MARLOWE T. J. et RYDER B. G., « Properties of data flow frameworks: a unified model ». *Acta Informatica*, vol. 28, p. 121–163, 1991.

[MAY 93] MAYDAN D. E., AMARASINGHE S. P. et LAM M. S., « Array Dataflow Analysis and its Use in Array Privatization ». In *ACM Symp. on Principles of Programming Languages*, p. 2–15, January 1993.

[PAR 66] PARIKH R. J., « On Context-free Languages ». *J. of the ACM*, vol. 13, n° 4, p. 570–581, 1966.

[WON 95] WONNACOTT D. et PUGH W., « Nonlinear array dependence analysis ». In *Proc. Third Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers*, 1995. Troy, New York.