# Calcul de régions de tableaux exactes

**Béatrice Apvrille-Creusillet**

*Centre de Recherche en Informatique*
*École des Mines de Paris*
*35, rue Saint-Honoré*
*77305 FONTAINEBLEAU Cedex*
`creusillet@cri.ensmp.fr`

RÉSUMÉ. *Un nouvel algorithme de calcul exact de régions de tableaux est présenté. Il s'appuie sur une analyse préalable des effets des instructions et des procédures sur les valeurs des variables scalaires entières. Ceci permet de préserver localement l'exactitude de l'analyse, même lorsque des variables intervenant dans les expressions des régions sont modifiées par le programme. Cet algorithme a été implanté dans PIPS, un paralléliseur de programmes scientifiques. Il constitue une première étape vers l'implantation d'un algorithme de privatisation des sections de tableau, qui n'est qu'une des utilisations potentielles des régions dans le domaine de l'optimisation de programmes.*

ABSTRACT. *A new algorithm for exact array region analysis, that relies on a preliminary analysis of the effects of statements and calls upon the values of integer scalar variables, is introduced. The accuracy of the analysis is locally ensured, even when variables occuring in region expressions are modified by the program. It has been implemented in PIPS, a parallelizing compiler for scientific programs. Among the numerous potential applications in the field of program optimization, it provides a first step towards array section privatization.*

MOTS-CLÉS : *compilation, analyse de flot de données, régions de tableaux, optimisation de programme.*

KEY WORDS : *compilation, array dataflow analysis, array regions, program optimization.*

## 1. Introduction

Blume et Eigenmann [BLU 92] ont récemment mené une étude sur les performances des paralléliseurs et les améliorations à leur apporter. Ils ont notamment montré la nécessité de détecter les ensembles d'éléments de tableaux dont le calcul est local

```
      SUBROUTINE FORMM(M)
      PARAMETER (NDDF = 5, NNPED = 9)
      REAL M(NDDF,NNPED,NDDF,NNPED), SUU(NDDF,NDDF)
      COMMON /MASSC/ RHO(5,5)
      DO 30 J = 1,NNPED
         DO 25 I = 1, J
            CALL ZEROV(SUU, NDDF*NDDF)
            .....
            DO 15 JJ = 1, NDDF
               SUU(JJ,JJ) = RHO(JJ,JJ)*SUMNN
15          CONTINUE
            DO 20 II = 1, NDDF
            DO 20 JJ = 1, NDDF
               M(II,I,JJ,J) = SUU(II,JJ)
               M(JJ,J,II,I) = SUU(II,JJ)
20          CONTINUE
25       CONTINUE
30    CONTINUE
      END

      SUBROUTINE ZEROV(VECTOR, LENGTH)
      DIMENSION VECTOR(LENGTH)
      DO 100 I = 1,LENGTH
         VECTOR(I) = 0
100   CONTINUE
      END
```

**Figure 1.** *Extrait de code*

à une boucle, et qui sont dits privatisables. Ceci permet d'éliminer des dépendances entre itérations, et donc de rendre certaines boucles parallélisables.

Prenons l'exemple du programme de la figure 1. Au cours d'une itération de la boucle `25`, tous les éléments du tableau `SUU` sont initialisés à zéro par un appel à `ZEROV`. Puis les éléments de la diagonale sont réinitialisés par la boucle `15`. Enfin, tous les éléments de `SUU` sont référencés en lecture dans le nid de boucle `20`. Ce sont les mêmes éléments qui sont redéfinis d'une itération de la boucle `25` sur l'autre, ce qui introduit des dépendances directes, et empêche la parallélisation de cette boucle. Si l'on arrive à montrer que toute référence en lecture à un élément de `SUU` est précédée d'une écriture à la même itération, le tableau `SUU` pourra être déclaré local à la boucle `25` ; les dépendances entre itérations seront alors éliminées, et la boucle sera parallélisée.

Pour effectuer cette transformation de programme, il est donc nécessaire de connaître avec exactitude les éléments de tableaux qui sont référencés en lecture et en écriture par chaque itération.

Mais ce n'est pas la seule utilisation possible des résultats de l'analyse exacte des ensembles d'éléments de tableau référencés. Un domaine d'application envisageable est la vérification de programme : il est en effet intéressant de pouvoir vérifier à la compilation si les accès aux éléments de tableaux ont bien lieu à l'intérieur des bornes déclarées. Une approche basée sur l'analyse présentée dans cet article serait certainement plus coûteuse qu'une analyse spécifique [GUP 93], mais un compilateur ayant par ailleurs besoin d'informations sur les accès aux éléments de tableaux pourrait tirer parti des résultats de cette analyse pour intégrer cette optimisation.

La génération des communications dans le cadre de la compilation pour machines à mémoire distribuée est un autre domaine dans lequel la connaissance exacte des

références aux éléments de tableau peut permettre d'optimiser le code généré en évitant des communications inutiles.

Par exemple, Fabien Coelho [COE 95] utilise ces informations pour traiter les entrées/sorties lors de la compilation du langage HPF, en mode 2PMD, c'est-à-dire avec un processeur jouant le rôle de l'hôte sur lequel les entrées/sorties sont effectuées, les autres processeurs étant des nœuds de calcul. Ce modèle est représenté figure 3. Prenons pour exemple le programme de la figure 2 qui comporte une instruction d'entrée/sortie (`READ`) concernant les éléments 1 à `n` du tableau `A`.

Idéalement, l'hôte effectue la lecture, puis envoie les éléments lus aux nœuds sur lesquels ils doivent résider. Parallèlement, chacun des nœuds reçoit les éléments qui lui sont destinés. Le code généré est celui modélisé par la figure 4. Reste à déterminer l'ensemble des éléments concernés, pour pouvoir générer les communications. Si cet ensemble est surestimé, l'hôte envoie aux nœuds des éléments qui n'ont pas forcément été redéfinis par l'entrée/sortie, et dont la valeur est obsolète ou indéfinie. Il faut donc commencer par mettre à jour sur le processeur hôte tous les éléments potentiellement concernés par la lecture, avant de réaliser l'entrée/sortie (voir figure 5). Les valeurs des éléments qui sont renvoyés vers les nœuds sont alors bien à jour. Par rapport au modèle idéal, on a donc généré des communications supplémentaires avant et après l'entrée/sortie. Par contre, si l'ensemble des éléments concernés par l'entrée/sortie est représenté de manière exacte, la phase de récupération devient inutile, et l'on rejoint le modèle idéal.

Dans le projet PIPS (Paralléliseur Interprocédural de Programmes Scientifiques ([IRI 92, IRI 91]), le concept de *régions* [TRI 84] a été choisi pour représenter les ensembles d'éléments de tableaux référencés. Cet article présente un nouvel algorithme de calcul des régions. Son originalité réside dans l'utilisation des effets des instructions sur les valeurs des variables scalaires entières pour propager les régions intraprocéduralement. Ceci permet de préserver l'exactitude de l'analyse, dans un grand nombre de cas, par exemple lorsque des variables scalaires entières sont modifiées par le programme.

Cet article est organisé de la façon suivante. La section 2 présente rapidement les caractéristiques de PIPS nécessaires à la compréhension des algorithmes présentés ultérieurement. La section 3 définit de manière informelle les régions et les opérateurs permettant de les manipuler, puis décrit leur calcul intraprocédural. Enfin, la section 4 situe notre approche par rapport aux travaux d'autres auteurs.

## 2. Présentation de PIPS

PIPS est un **P**aralléliseur **I**nterprocédural de **P**rogrammes **S**cientifiques. Il transforme des programmes écrits en Fortran 77 en remplaçant les boucles `DO` parallélisables par des instructions vectorielles à la Fortran 90, des instructions `DOALL`, ou encore des directives CRAY. Aucune architecture particulière n'est visée, mais si le projet originel concernait les machines à mémoire partagée, les études actuelles s'orientent vers les machines à mémoire distribuée (génération de code, compilation de HPF, privatisation de tableaux).

```
      program iotest
      integer n, i
      parameter(n=100)
      real A(n)
chpf$ template t(n)
chpf$ align A(i) with t(i)
chpf$ processors p(4)
chpf$ distribute t(block) onto p
      ...
      do i=1, n
         READ *, A(i)
      enddo
      ...
      end
```

**Figure 2.** *Programme* HPF

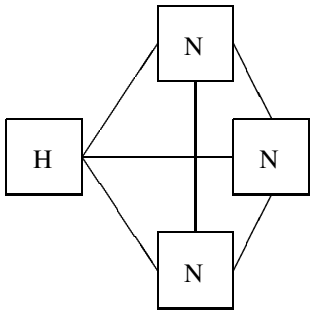



**Figure 3.** *Modèle 2PMD*

| **Hôte** | **Nœud** |
|---|---|
| `do i=1, n`<br>`  READ *, A(i)`<br>`enddo` | `...` |
| `C envoi des elts de A`<br>`C vers les nœuds` | `C récupération des elts`<br>`C locaux de A` |

**Figure 4.** *Code idéalement généré*

| **Hôte** | **Nœud** |
|---|---|
| `C récupération des elts`<br>`C de A` | `C envoi des elts locaux`<br>`C de A à l'hôte` |
| `do i=1, n`<br>`  READ *, A(i)`<br>`enddo` | `...` |
| `C envoi des elts de A`<br>`C vers les nœuds` | `C récupération des elts`<br>`C locaux de A` |

**Figure 5.** *Code généré en cas de surestimation de l'ensemble des éléments concernés par l'entrée/sortie*

```
C  T(K) {K == 3}
   K = 3
C  T(K) {}
   K = FOO(N)
C  T(K) {K == K + 1}
   CALL INC1(K)
```

```
C  P() {}
   K = 3
C  P(K) {K==3}
   K = FOO()
C  P(K) {}
   CALL INC1(K)
C  P(K) {}
```

**Figure 6.** *Transformeurs et préconditions*

Le processus de parallélisation est divisé en différentes phases, chacune des phases correspondant à une analyse ou une transformation de programme particulière.

Les *analyses intraprocédurales* sont effectuées sur le *graphe de contrôle hiérarchisé* du module concerné. Les nœuds de ce graphe correspondent aux structures de contrôle du langage (boucle `DO`, `IF`, séquence d'instructions, instruction simple, ...), ou sont de petits graphes de contrôle qui permettent de décrire les fragments de code non structurés (utilisation de `GOTO`). Ces nœuds sont annotés par les résultats des analyses.

Les *analyses interprocédurales* propagent les informations sur le *graphe des appels*. La récursivité n'étant pas traitée, ce graphe est acyclique, et est donc parcouru de manière ascendante ou descendante.

Notre but n'est pas de décrire ici toutes les phases d'analyse et de transformations de PIPS, mais uniquement celles qui nous seront utiles lors de la description du calcul des régions : les *transformeurs* et les *préconditions* [IRI 92].

Les transformeurs donnent les relations affines qui existent entre les valeurs des variables scalaires entières avant et après exécution d'une instruction ou d'un appel de procédure. Ils expriment donc les effets des instructions sur les valeurs des variables scalaires entières. La figure 6 donne avant chaque instruction son transformeur sous la forme `T(argument) {prédicat}` ; l'argument du transformeur donne la liste des variables modifiées, et le prédicat les relations affines non triviales [1] entre ces variables. On remarquera dans la figure 6 que l'expression de `K` dans l'instruction `K = FOO(N)` n'étant pas affine, aucune équation ou inéquation n'apparaît dans le prédicat du transformeur correspondant.

Les préconditions sont des prédicats sur les variables scalaires entières, vrais avant exécution de l'instruction. L'exemple de la figure 6 donne pour chaque instruction ses préconditions, sous la forme `P(var) {...}`, qui exprime l'effet du module courant sur son état mémoire, entre son point d'entrée et l'instruction concernée.

Les transformeurs sont calculés de manière ascendante, c'est-à-dire de la fin du module courant vers le début, et les préconditions sont propagées dans le sens contraire, de telle sorte que si $T_1$ et $P_1$ sont respectivement le transformeur et la précondition associés à l'instruction $S_1$, $P_2$ la précondition associée à l'instruction $S_2$ suivant $S_1$, alors $P_2 = T_1(P_1)$.

1. Une relation est triviale si elle est vérifiée quelles que soient les valeurs que peuvent prendre les variables. Elle est de la forme `nouvelle_valeur(I) == ancienne_valeur(I)`.

## 3. Calcul des régions

Le calcul des régions comporte deux phases : (1) le calcul intraprocédural, qui s'appuie sur le graphe de contrôle hiérarchisé du corps de la procédure pour calculer un résumé de ses effets sur les éléments de tableaux ; (2) la propagation interprocédurale ascendante, qui élimine des régions du corps de la procédure les variables locales, et masque les régions correspondant à des variables désactivées en sortie de procédure ; le résumé obtenu est ensuite utilisé au niveau des différents appels de la procédure après traduction des paramètres formels en paramètres réels.

Cette section s'intéresse à la première phase, la deuxième phase étant décrite plus en détail dans [IRI 92]. Dans un premier temps, nous définirons les régions et les opérateurs permettant de les manipuler. Puis nous aborderons le calcul des régions pour les principales structures du langage FORTRAN c'est-à-dire la séquence d'instructions, la boucle `DO`, et le test `IF`. Nous illustrerons notre propos à l'aide du programme de la figure 7.

```
K = FOO()                                SUBROUTINE INC1(I)
DO I = 1,N                               I = I + 1
   DO J = 1,N                            END
      WORK(J,K) = J + K
  ENDDO
   CALL INC1(K)
   DO J = 1,N
      WORK(J,K) = J * J - K * K
   ENDDO
ENDDO
```

**Figure 7.** *Exemple d'application*

### 3.1. *Définition et opérateurs*

Une région est définie dans [TRI 84] comme étant un ensemble d'éléments de tableaux décrit par des équations et inéquations affines formant un polyèdre convexe. Les régions étant utilisées dans PIPS pour exprimer les effets des instructions et des procédures, deux autres notions ont été introduites : (1) l'*action* réalisée sur les éléments de la région (`READ` (`R`) pour une utilisation ou `WRITE` (`W`) pour une définition) ; (2) l'*approximation* de la région (`MUST` si tous les éléments de la région sont effectivement concernés par l'action, pour tous les chemins du graphe de contrôle du fragment de code considéré et pour tout état mémoire [2] possible avant exécution de ce fragment de code ; la région est alors dite *exacte* ; ou `MAY` si les éléments de la région sont seulement potentiellement atteints). Par exemple, la région :

`<A(PHI1,PHI2)-W-MUST-{1<=PHI1, PHI1<=3, PHI1==PHI2}>`

où `PHI1` et `PHI2` représentent respectivement la première et la deuxième dimension de `A`, correspond à une référence en écriture aux éléments `A(1,1)`, `A(2,2)` et `A(3,3)`.

2. On rappelle qu'un état mémoire associe une valeur à chaque variable du programme pour une exécution donnée, et pour un instant donné de cette exécution.

Les contraintes affines définissant une région peuvent comporter des variables du programme dont la valeur dépend de l'état mémoire considéré (ex. `PHI1==I`). Nous dirons alors que la region est exprimée dans cet état mémoire. Par exemple, si avant l'exécution de l'instruction `I=I+1` le polyèdre définissant la région est `PHI1==I`, il doit être `PHI1==I-1` après exécution de l'instruction.

Pour apporter des informations plus précises sur les variables `PHIs`, on peut également ajouter au prédicat de la région celui des préconditions. Cela ne change en rien la définition des régions. Les préconditions apportent simplement des précisions sur les relations qui existent entre les paramètres en fonction desquels les régions sont définies, et qui sont les variables du programme. Ceci peut faciliter les opérations ultérieures.

Les régions étant des ensembles d'éléments de tableaux, on pourrait définir tous les opérateurs habituels sur les ensembles. Seuls deux d'entre eux nous intéressent dans le cadre du calcul des régions exactes : l'*union* et la *projection selon une variable du programme*.

Le prédicat d'une région devant toujours être un polyèdre convexe, le prédicat de la fusion de deux régions ne peut résulter de l'union des deux prédicats initiaux (voir figure 8). Le résumé de plusieurs accès aux éléments d'un même tableau sera donc représenté par l'enveloppe convexe des régions correspondantes. L'enveloppe convexe de deux polyèdres convexes contient généralement des points qui n'appartiennent pas aux systèmes initiaux (voir la figure 9). Dans ce cas les éléments de la région résultante ne sont pas forcément tous lus ou écrits, et son approximation doit donc être `MAY`, même si les approximations des régions initiales étaient `MUST`. Toutefois, une approche moins grossière permet de préserver l'attribut `MUST` de la région lorsque l'union est égale à l'enveloppe convexe, ou si les régions initiales possédent certaines caractéristiques.

Enfin, il convient de discerner deux types d'union : les unions dites *must* qui sont utilisées pour calculer les résumés correspondant à une instruction simple ou à une séquence linéaire d'instructions, et les unions *may* qui permettent de calculer les régions résumées d'une instruction de branchement (`IF`). En effet, même si une région de tableau est sûrement lue (région `MUST`) dans une branche d'un `IF` et pas dans l'autre, elle ne l'est que potentiellement pour l'ensemble de l'instruction `IF`.

Le tableau 10 définit les opérateurs d'union *must* et *may* ($\cup_{must}$ et $\cup_{may}$) : il donne les caractéristiques de la région résultante en fonction de celles des régions initiales.

La *projection* d'un polyèdre selon une variable se calcule en éliminant celle-ci du système de contraintes qui définit le polyèdre. L'algorithme de Fourier-Motzkin [FOU 27] permet de réaliser cette élimination. En réalité, il calcule l'enveloppe convexe de l'ensemble des projections des points entiers appartenant au polyèdre de départ [ANC 91a]. Le polyèdre résultant peut donc contenir des points qui ne correspondent pas à la projection d'un point du polyèdre initial. On peut s'en convaincre à l'aide de la figure 11. Dans le cas a), un des points de la projection ne correspond à aucun des points du polyèdre constitué des $\bullet$. Par contre, la projection du polyèdre du cas b) est exacte. Une région projetée pourra donc contenir des points n'appartenant pas à l'ensemble de départ. Des conditions suffisantes pour que la projection d'une région selon une variable du programme soit exacte ont été données dans [APV 94]. Ces conditions diffèrent de celles introduites dans [ANC 91b], car les variables du

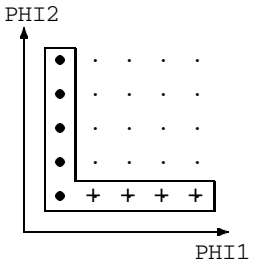



**Figure 8.** *Union de deux polyèdres*

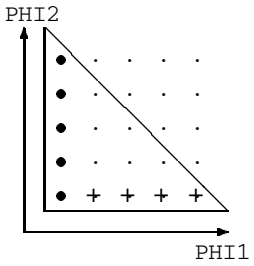
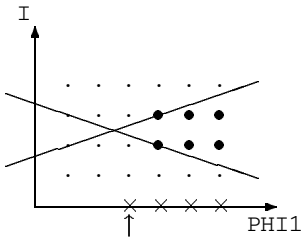


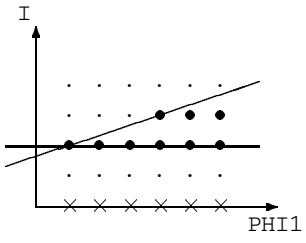

**Figure 9.** *Enveloppe convexe de deux polyèdres*

| région 1 ($R_1$) | région 2 ($R_2$) | $R_1 \cup_{must} R_2$ | $R_1 \cup_{may} R_2$ |
|---|---|---|---|
| must | must | must, si env. conv. exacte<br>may, sinon | must, si $R_1 = R_2$<br>may, sinon |
| must | may | must, si $R_2 \subseteq R_1$<br>may, sinon | may |
| may | may | may | may |

**Figure 10.** *Enveloppe convexe de deux régions*




*a) projection non-exacte*




*b) projection exacte*

— *frontière du polyèdre.*
• *point appartenant au polyèdre.*
× *point appartenant à la projection.*
↑ *point appartenant à la projection, mais pas au polyèdre de départ.*

**Figure 11.** *Projection d'un polyèdre*

programme en fonctions desquelles les variables PHIs sont exprimées doivent être considérées comme des constantes symboliques .

La projection d'une région $R_\sigma$ exprimée dans un état mémoire $\sigma$ sur un état mémoire $\sigma'$ est alors définie comme étant la projection successive de cette région selon les variables dont la valeur est différente dans $\sigma$ et $\sigma'$. Nous la noterons $proj_{\sigma'}(R_\sigma)$.

Enfin, nous définissons par abus de langage l'intersection d'une région et d'un transformeur comme étant la région dont le polyèdre est l'intersection du polyèdre de la région initiale et de celui du transformeur, et dont l'action et l'approximation sont celles de la région de départ.

Nous allons maintenant détailler le calcul des régions pour les principales structures du langage FORTRAN.

### 3.2. *Calcul des régions d'une instruction simple*

On distingue plusieurs types d'instructions simples : les affectations, les appels à des fonctions intrinsèques, les opérations d'entrée/sortie et les appels à des procédures définies par le programme. Nous ne nous intéressons ici qu'aux affectations, les autres cas relevant de traitements particuliers ou de l'analyse interprocédurale.

Chaque référence à un élément de tableau est convertie en une région élémentaire dont le prédicat comprend les contraintes contenues dans la précondition de l'instruction, et les équations liant les variables PHIs aux expressions des indices de l'élément de tableau. Cette région élémentaire est exacte (MUST) si et seulement si les expressions des indices sont des fonctions affines des variables du programme. Son action dépend de la position de la référence dans l'instruction (à gauche ou à droite de l'affectation). Si deux régions élémentaires concernent le même tableau pour des actions identiques, elles sont fusionnées à l'aide de l'opérateur $\cup_{must}$. Leur approximation peut alors devenir MAY. Pour l'instruction `M = A(I) + A(I+1)` par exemple, les régions élémentaires correspondant aux deux références en lecture sont :

```
<A(PHI1)-R-MUST-{PHI1==I}>
<A(PHI1)-R-MUST-{PHI1==I+1}>
```

et la région correspondant à l'instruction est finalement :

```
<A(PHI1)-R-MUST-{I<=PHI1, PHI1<=I+1}>
```

### 3.3. *Calcul des régions d'une séquence d'instructions*

Une région est exprimée en fonction des valeurs des variables scalaires entières dans l'état mémoire précédant l'exécution de l'instruction à laquelle elle correspond. Ces valeurs sont contraintes par les préconditions. Le principe de l'algorithme est d'exprimer les régions dans l'état mémoire précédant l'exécution de la séquence d'instructions, en tenant compte des effets des instructions sur les valeurs des variables scalaires entières en fonction desquelles elles sont exprimées. Pour montrer l'intérêt de cet algorithme, reprenons l'exemple de la figure 7.

Nous nous intéressons aux accès en écriture aux éléments du tableau WORK (régions W). La région $R$ correspondant à la deuxième boucle J est donnée par :

```
<WORK(PHI1,PHI2)-W-MUST-{1<=I,I<=N,1<=PHI1,PHI1<=N,PHI2==K}>
```

L'instruction précédente, CALL INC1(K), modifie K. Si l'on utilise simplement les effets de ce CALL (i.e. K modifié) pour exprimer la région $R$ dans l'état mémoire précédant cette instruction, nous obtenons la région $\widetilde{R}$ :

```
<WORK(PHI1,PHI2)-W-MAY-{1<=I,I<=N,1<=PHI1,PHI1<=N}>
```

qui ne contient plus d'information sur la deuxième dimension des éléments référencés, ce qui explique que ce soit une région MAY. La région $R_0$ obtenue pour l'ensemble de la séquence linéaire est alors :

```
<WORK(PHI1,PHI2)-W-MAY-{1<=I,I<=N,1<=PHI1,PHI1<=N}>
```

Au contraire, en tenant compte des effets de l'instruction CALL INC1(K) sur la valeur de K (qui sont donnés par le transformeur T(K) {K==K+1}), nous obtenons la région $\widetilde{R}$ :

```
<WORK(PHI1,PHI2)-W-MUST-{1<=I,I<=N,1<=PHI1,PHI1<=N,PHI2==K+1}>
```

La région $R_0$ est alors :

```
<WORK(PHI1,PHI2)-W-MUST-
                  {1<=I,I<=N,1<=PHI1,PHI1<=N,K<=PHI2,PHI2<=K+1}>
```

qui représente avec exactitude les éléments de tableaux effectivement référencés.

Nous allons maintenant décrire cet algorithme de manière plus formelle. Soit $B$, la séquence linéaire constituée des instructions $S_1, S_2, \ldots, S_n$. Nous cherchons $R_0$ l'ensemble de régions associé à $B$. Pour toute instruction $S_k$, nous noterons $\sigma_k$ l'état mémoire la précédant, et $T_k$ le transformeur correspondant.

**Définition 1** *Soit $R_{k+1}$ une région exprimée dans l'état mémoire $\sigma_{k+1}$.*

*On définit la* transformation inverse associée à $T_k$, $\widetilde{T}_k^{-1}$ *par :*

$$\widetilde{T}_k^{-1} : R_{k+1} \rightarrow \widetilde{R}_k = proj_{\sigma_k}(intersection(R_{k+1}, T_k))$$

$\widetilde{R}_k$ est la projection sur l'état mémoire $\sigma_k$ de la région dont le prédicat est l'intersection du prédicat initial de $R_{k+1}$ et du prédicat de $T_k$. Cette projection consiste en une élimination des variables modifiées par l'instruction, qui sont données par l'argument du transformeur. Si le transformeur $T_k$ décrit exactement la transformation effectuée par $S_k$, alors les approximations MUST peuvent être conservées.

**Algorithme 1** *L'ensemble $R_0$ des régions associées à la séquence $S_1, \ldots, S_n$, $n \geq 1$ peut être calculé récursivement à l'aide de l'algorithme :*

$$\begin{cases} R'_k = \cup_{must}(R_k, \widetilde{T}_k^{-1}(R'_{k+1})), \forall k \in [1, n-1] \\ R'_n = R_n \\ R_0 = R'_1 \end{cases}$$

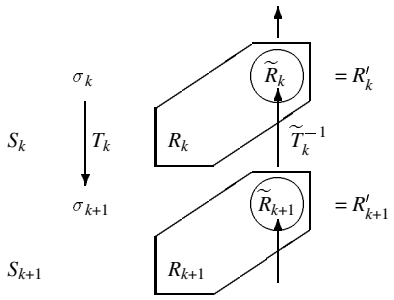



**Figure 12.** *Algorithme*

$R'_k$ est l'ensemble de régions associé à la séquence linéaire $S_k, \ldots, S_n$ et exprimé dans l'état mémoire $\sigma_k$ précédant $S_k$. La figure 12 donne une vision intuitive de cet algorithme. En appliquant $\widetilde{T}_k^{-1}$ à la région $R'_{k+1}$ obtenue à l'étape précédente, on obtient $\widetilde{R}_k$ qui est exprimée dans l'état $\sigma_k$. $R'_k$ est alors l'enveloppe convexe de $\widetilde{R}_k$ et de $R_k$ qui est la région associée à $S_k$ et exprimée dans l'état $\sigma_k$. On a également fait figurer le transformeur $T_k$ qui donne une approximation conservatrice de la transformation de $\sigma_k$ en $\sigma_{k+1}$ par $S_k$.

**Propriété 1** *L'ensemble $R_0$ obtenu à l'aide de l'algorithme précédent est bien une sur-approximation de l'ensemble des éléments de tableaux référencés dans la séquence d'instructions $S_1, \ldots, S_n$.*

**Propriété 2** *Soit une séquence $S_1, S_2$ telle que les régions $R_1$ et $R_2$ associées à chacune des instructions soient des régions* `MUST`*. Si le transformeur $T_1$ comporte autant d'équations que de variables modifiées par $S_1$, alors l'opérateur $\widetilde{T}_1^{-1}$ conserve les approximations* `MUST`*, et l'approximation de la région $R_0$ ne dépend que du résultat de la fusion réalisée à l'aide de l'opérateur $\cup_{must}$.*

**Propriété 3** *L'algorithme précédent calcule des régions plus précises que lorsque seuls les effets sur les variables scalaires entières sont utilisés pour éliminer les variables modifiées par une instruction*[3]*, dans le sens où elles contiennent moins d'éléments n'appartenant pas aux régions initiales.*

Ces propriétés ont été démontrées dans [APV 94].

### 3.4. *Calcul des régions d'une boucle DO*

Une boucle `DO` se compose (1) d'un en-tête donnant le domaine de variation de l'indice de boucle, (2) d'un corps de boucle, qui n'est autre qu'une séquence d'instructions complexes.

La première étape du calcul des régions de la boucle consiste à calculer les régions du bloc d'instructions constituant le corps. Ces régions correspondent à une itération quelconque. Elles sont donc fonction des variables qui peuvent être modifiées lors de

3. Et non les effets sur leur valeur, soit : $\widetilde{T}_k^{-1} : R_{k+1} \longrightarrow \widetilde{R}_k = proj_{\sigma_k}(R_{k+1})$

cette itération, y compris l'indice de boucle. Dans l'exemple de la figure 7, la variable K est modifiée, et nous avons vu que les régions du corps de la boucle en dépendent.

Ces variables sont éliminées lors de la deuxième étape, en utilisant le transformeur de la boucle. Celui-ci fournit en effet l'invariant de boucle, c'est-à-dire des relations affines entre les variables modifiées à chaque itération (indice de boucle compris), et les valeurs des variables avant exécution de la boucle. Ainsi, pour la boucle la plus externe de l'exemple de la figure 7, le transformeur $T_B$ = `T(I,K) {K#INIT+I==1+K}` donne, pour chaque itération, la valeur de K en fonction de la valeur de I à la même itération, et en fonction de la valeur de K avant exécution de la boucle (K#INIT).

Si l'on applique l'opérateur $\widetilde{T}_B^{-1}$ associé au transformeur $T_B$ précédent à la région correspondant au corps de la boucle, mais sans éliminer l'indice de boucle, et après renommage de K#INIT en K, on obtient la région $R$ suivante :

```
<WORK(PHI1,PHI2)-W-MUST-
            {1<=PHI1,PHI1<=N,K+I-1<=PHI2,PHI2<=K+I,1<=I,I<=N}>
```

K représente ici la valeur de la variable K avant exécution de la boucle. Elle ne dépend donc plus de la valeur de I. Par contre, I représente la valeur de la variable I au cours d'une itération.

Reste ensuite à éliminer l'indice de boucle de la région $R$. Lors de cette élimination, l'approximation MUST de la région n'est pas préservée sous les mêmes conditions, à cause de la sémantique particulière de la boucle DO. Considérons les contraintes de $R$ qui contiennent I : {1<=I, I<=N, K+I<=PHI2+1,PHI2<=K+I}. Vue de l'intérieur de la boucle (i.e. au sein d'une itération), I est une variable comme les autres, et les contraintes de $R$ signifient : $\exists\, i \in [1,n] \mid k+i-1 \leq \varphi_2 \wedge \varphi 2 \leq k+i$. Mais vu de l'extérieur de la boucle, comme I est un indice de boucle, la sémantique donnée à ces équations devient la suivante : $\forall i \in [1,n]\ \ k+i-1 \leq \varphi_2 \wedge \varphi 2 \leq k+i$. Ceci n'est pas vrai si l'incrément de l'indice de boucle n'est pas égal, en valeur absolue, à 1. Dans ce cas, l'indice de boucle ne prend pas toutes les valeurs de l'intervalle $[1,n]$, et on perd de l'information en l'éliminant.

Les *conditions de conservation des approximations MUST lors de la projection d'un indice de boucle* sont donc :

1. les bornes de boucle sont affines ; cela signifie qu'elles donnent un encadrement symbolique de l'indice de boucle, encadrement que l'on retrouve dans les préconditions, et donc dans les régions ;
2. la valeur absolue de l'incrément vaut 1 ; l'indice de boucle prend alors toutes les valeurs de son intervalle de variation ;
3. les conditions de projection exacte proposées dans [ANC 91b, PUG 92] sont vérifiées.

Dans notre exemple, l'incrément de la boucle vaut 1, les bornes de boucle sont affines et la projection est exacte. La région correspondant à la boucle est donc :

```
<WORK(PHI1,PHI2)-W-MUST-{1<=PHI1,PHI1<=N,K<=PHI2,PHI2<=K+N}>
```

Si les bornes de boucle ou l'incrément sont des références à des éléments de tableau, les régions de la boucle sont obtenues en fusionnant les régions calculées précédemment et les régions de l'en-tête de la boucle à l'aide de l'opérateur $\cup_{must}$.

### 3.5. *Calcul des régions d'un IF*

Une instruction `IF` est composée d'une instruction conditionnelle et de deux branches, dont l'une au moins est non vide. Chaque branche est traitée comme une séquence linéaire d'instructions complexes. Les régions résultantes sont alors exprimées dans le même état mémoire car l'instruction conditionnelle ne peut modifier les valeurs des variables scalaires entières. Leur union peut donc être calculée directement, en utilisant l'opérateur d'union $\cup_{may}$. Ces régions sont ensuite fusionnées avec les régions correspondant à l'instruction conditionnelle du test, à l'aide de l'opérateur d'union $\cup_{must}$ car tout se passe comme si l'on avait une séquence de deux instructions complexes : l'instruction conditionnelle, puis l'une des deux branches.

## 4. Autres travaux

Triolet [TRI 84] avait initialement introduit les régions pour calculer une approximation conservatrice des effets des procédures sur les éléments de tableaux (régions `MAY`), problème d'analyse de flot de données *insensible au flot de contrôle*. Nous avons étendu son approche de manière à calculer les effets *exacts* des instructions et des procédures sur les éléments de tableaux, problème *sensible au flot de contrôle* et tenant compte des modifications de variables scalaires entières.

D'autres travaux [CAL 88, HAV 91, BAL 89] se sont intéressés au calcul de résumés approchés des effets des procédures sur les éléments de tableaux, ou ont introduit [LI 88, TAN 93] des représentations exactes des références aux éléments de tableaux. Mais aucun de ces auteurs ne donne de précision sur le comportement de leur méthode en présence de variables scalaires entières variant à l'intérieur du corps des boucles et des procédures analysées. Havlak et Kennedy [HAV 91] indiquent juste que si une procédure modifie une variable scalaire entière dont dépendent les indices de tableaux, les ensembles d'éléments de tableaux correspondants prennent une valeur indéfinie.

Une autre approche plus fine a été introduite par Feautrier [FEA 91] en 1991. Elle consiste à trouver une solution exacte au problème des dépendances directes. Le point faible de cette approche réside dans les caractéristiques du langage source accepté : contrôle statique (pas d'instruction conditionnelle), indices de tableaux et bornes de boucle affines, programme monoprocédural. Des études plus récentes [MAS 94, COL 94] ont étendu le domaine d'application de cette méthode de résolution aux fragments de programme non-affines, et au boucles while, mais au prix d'une perte d'exactitude de l'analyse.

Enfin, divers algorithmes de privatisation ou d'expansion de tableaux ont déjà été proposés [FEA 88, LI 92, MAY 93, TU 93]. Mais soit le contexte est monoprocédural, ou à contrôle statique, et donc beaucoup moins général que celui que PIPS peut offrir, soit les auteurs regrettent le manque d'analyse symbolique de leur environnement, problème que devraient résoudre, au moins partiellement, les régions exactes, qui sont une représentation symbolique des éléments de tableaux référencés.

## 5. Conclusion et travaux à venir

Un nouvel algorithme de calcul intraprocédural des régions de tableaux a été présenté dans cet article. Il prend en compte les effets des instructions et des procédures sur les valeurs des variables scalaires entières lors de la propagation intraprocédurale des régions. L'exactitude de l'analyse dépend essentiellement de celle des transformeurs, mais aussi de l'exactitude des opérations d'union et de projection.

L'implantation actuelle couvre l'ensemble de la norme FORTRAN 77, à quelques exceptions mineures près. Des expériences préliminaires sur les programmes du Perfect Club [BER 89] en ont montré la robustesse et la praticabilité, malgré la complexité théorique exponentielle des opérations d'union et de projection de polyèdres. D'autres expériences seront nécessaires pour calculer le *taux de succès* des opérateurs implantés, c'est-à-dire le nombre de régions exactes qu'ils permettent de préserver, par rapport au nombre de régions exactes reçues en entrée. Ceci devrait permettre de déterminer si la représentation des régions sous forme de polyèdres convexes est suffisamment précise pour l'analyse des programmes scientifiques, ou si une représentation sous forme d'union finie de polyèdres est à envisager, sachant que le coût en terme de temps de calcul et d'occupation mémoire sera certainement beaucoup plus important.

Enfin, ces régions seront utilisées pour implanter un algorithme de privatisation de tableaux, destiné à améliorer le taux de parallélisation de PIPS [BLU 92, APV 94].

## 6. Bibliographie


[ANC 91a] C. ANCOURT. « Génération de Code pour Multiprocesseurs à Mémoires Locales ». Thèse de doctorat, Université Paris VI, Mars 1991.

[ANC 91b] C. ANCOURT, F. IRIGOIN. « Scanning Polyhedra with DO Loops ». *Symposium on Principles and Practice of Parallel Programming*, April 1991.

[APV 94] B. APVRILLE-CREUSILLET. « Régions exactes et privatisation de tableaux ». Rapport de DEA Systèmes Informatiques, Université Paris VI, septembre 1994. Disponible sous `http://cri.ensmp.fr/~creusil`.

[BAL 89] V. BALASUNDARAM, K. KENNEDY. « A technique for summarizing data access and its use in parallelism enhancing transformations ». *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, p. 41–53, June 1989.

[BER 89] M. BERRY, D. CHEN, P. KOSS, D. KUCK, V. LO, Y. PANG, R. ROLOFF, A. SAMEH, E. CLEMENTI, S. CHIN, D. SCHNEIDER, G. FOX, P. MESSINA, D. WALKER, C. HSIUNG, J. SCHWARZMEIER, K. LUE, S. ORZAG, F. SEIDL, O. JOHNSON, G. SWANSON, R. GOODRUM, J. MARTIN. « The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers ». Technical Report CSRD-827, Center for Supercomputing Research and Development, University of Illinois, May 1989.

[BLU 92] W. BLUME, R. EIGENMANN. « Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs ». *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, November 1992.

[CAL 88] D. CALLAHAN, K. KENNEDY. « Analysis of Interprocedural Side Effects in a Parallel Programming Environment ». *Journal of Parallel and Distributed Computing*, 5:517–550, 1988.

[COE 95] F. COELHO. « Compilation of I/O Communications for HPF ». *Frontiers'95*, February 1995. Disponible sous `http://cri.ensmp.fr/~coelho`.

[COL 94] J.-F. COLLARD. « Parallélisation automatique des boucles while par exécution spéculative ». *Rencontres Francophones du Parallélisme*, p. 77–80, juin 1994.

[FEA 88] P. FEAUTRIER. « Array Expansion ». *International Conference on Supercomputing*, p. 429–441, July 1988.

[FEA 91] P. FEAUTRIER. « Dataflow Analysis of Array and Scalar References ». *International Journal of Parallel Programming*, 20(1):23–53, September 1991.

[FOU 27] J. FOURIER. Analyse des travaux de l'Académie Royale des Sciences pendant l'année 1823, partie mathématiques. *Histoire de l'Académie Royale des Sciences de l'Institut de France*, Vol. 7. 1827.

[GUP 93] R. GUPTA. « Optimizing Array Bound Checks Using Flow Analysis ». *ACM Letters on Programming Languages and Systems*, 2(1–4):135–150, march–december 1993.

[HAV 91] P. HAVLAK, K. KENNEDY. « An Implementation of Interprocedural Bounded Regular Section Analysis ». *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[IRI 91] F. IRIGOIN, P. JOUVELOT, R. TRIOLET. « Semantical Interprocedural Parallelization : An Overview of the PIPS project ». *International Conference on Supercomputing*, June 1991.

[IRI 92] F. IRIGOIN. « Interprocedural Analyses for Programming Environments ». *Workshop on Environments and Tools for Parallel Scientific Computing*, September 1992.

[LI 88] Z. LI, P.-C. YEW. « Efficient Interprocedural Analysis and Program Restructuring for Parallel Programs ». *ACM SIGPLAN Notices*, 23(9):85–99, 1988.

[LI 92] Z. LI. « Array Privatization for Parallel Execution of Loops ». *International Conference on Supercomputing*, p. 313–322, July 1992.

[MAS 94] V. MASLOV. « Lazy Array Data-Flow Analysis ». *Symposium on Principles of Programming Language*, p. 311–325, January 1994.

[MAY 93] D. E. MAYDAN, S. P. AMARASINGHE, M. S. LAM. « Array Data-Flow Analysis and its Use in Array Privatization ». *Symposium on Principles of Programming Language*, January 1993.

[PUG 92] W. PUGH. « A practical Algorithm for Exact Array Dependence Analysis ». *Communications of the ACM*, 35(8):102–114, August 1992.

[TAN 93] P. TANG. « Exact Side Effects for Interprocedural Dependence Analysis ». *International Conference on Supercomputing*, p. 137–146, July 1993.

[TRI 84] R. TRIOLET. « Contribution à la Parallélisation Automatique de Programmes Fortran Comportant des Appels de Procédures ». Thèse de doctorat, Université Paris VI, 1984.

[TU 93] P. TU, D. PADUA. « Automatic Array Privatization ». *Language and Compilers for Parallel Computing*, August 1993.