DEA Systèmes Informatiques - Université Paris VI
Parallélisation et Vectorisation Automatique

RÉGIONS EXACTES ET PRIVATISATION DE TABLEAUX

Béatrice Apvrille-Creusillet

31 août 1994

Résumé

De nombreuses optimisations de programme nécessitent la connaissance exacte des ensembles d'éléments de tableaux référencés par les instructions ou les procédures. C'est le cas par exemple de la privatisation de tableaux, de l'optimisation des communications, et de la vérification des accès aux éléments de tableaux.

Nous présentons dans ce rapport un nouvel algorithme de calcul des régions de tableaux exactes. 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 l'exactitude de l'analyse, même lorsque des variables intervenant dans les expressions des régions sont modifiées par le programme. Pour pouvoir effectuer ce calcul, les principaux opérateurs permettant de manipuler les régions (union, intersection, différence, et projection) sont définis avec le souci de préserver les ensembles d'éléments de tableaux initialement représentés.

Un algorithme de privatisation des sections de tableaux est ensuite décrit. Pour cela, deux nouveaux types de régions sont introduits : les régions `IN`, qui contiennent les éléments de tableaux importés par l'instruction considérée ; et les régions `OUT`, qui contiennent les éléments de tableaux vivants en sortie d'instruction. Une autre utilisation possible de ces régions serait la génération des commmunications dans le cadre de la compilation pour multiprocesseurs à mémoire distribuée.

Mots-clefs : compilation, analyse de flot de données, optimisation de programmes, privatisation de tableaux.

Abstract

Many program optimizations require the exact knowledge of the sets of array elements that are referenced by statements or procedures. Some examples are array privatization, communication optimization and array range checking.

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 in this report. The accuracy of the analysis is ensured, even when variables occuring in region expressions are modified by the program. In order to make this computation, we define the main operators on regions (union, intersection, difference, and projection), with the purpose of preserving an attribute (`MUST`) that reflects the accuracy of the representation.

An algorithm for array privatization is also presented. For that purpose, two new types of regions are defined : `IN` regions, that contain the array elements imported by the current statement ; and `OUT` regions, that represent the set of live array elements. Another application of these regions would be the generation of communications in a compiler for distributed memory machines.

Key words : compilation, array data-flow analysis, program optimization, array privatization.

Remerciements

Je voudrais adresser mes remerciements au Professeur Robert Mahl pour m'avoir accueillie au sein du Centre de Recherche en Informatique de L'École des Mines de Paris.

Je souhaite également exprimer ma reconnaissance au Professeur Michel Lenci, ancien directeur du CRI, pour m'avoir fait connaître ce centre de recherche et ses domaines d'intérêt.

Je tiens à remercier vivement François Irigoin pour m'avoir proposé ce sujet d'étude, et m'avoir guidée dans ce travail. Je lui doit notamment l'idée initiale de cette étude, c'est-à-dire le calcul des régions d'une séquence d'instructions à l'aide des transformeurs, ainsi que de nombreuses remarques tant sur le fond que sur la forme de ce rapport.

Je tiens à exprimer ma gratitude à Corinne Ancourt pour sa disponibilité, et les fructueuses discussions que nous avons eues au sujet des polyèdres, de leur projection, et de la représentation des régions sous forme de polyèdres.

Je voudrais aussi remercier Fabien Coelho, qui m'a fournit l'exemple de l'introduction sur la compilation des entrées/sorties dans le langage HPF*. Son aide m'a également été très précieuse lors de la définition de l'opérateur de projection d'une région selon une variable du programme.*

Pierre Jouvelot a également beaucoup contribué à ce travail par la relecture attentive et critique des chapitres les plus théoriques. Ces remarques m'ont notamment permis d'apporter plus de concision dans la formulation des définitions, théorèmes et autres propriétés.

Enfin, j'adresse à l'ensemble des membres du CRI mes plus vifs remerciements pour leur accueil chaleureux, et plus particulièrement à Jacqueline Altimira, Annie Pech-Ripaud, et Jean-Marie Kessis pour le support logistique qu'ils m'ont apporté.

Table des matières

Chapitre 1

Introduction

Blume et Eigenmann [14] 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 à une boucle, et qui sont dits privatisables. Ceci permet d'éliminer des dépendances entre itérations, et donc de rendre certaines boucles parallélisables. Pour effectuer cette transformation de programme, il est nécessaire de connaître avec exactitude les éléments de tableaux qui sont lus et écrits à l'intérieur du corps de la boucle.

Prenons l'exemple du programme de la figure 1.1. Au cours de toute itération de la boucle 25, l'ensemble du tableau `SUU` est initialisé à zéro par un appel à `ZEROV`. Puis sa diagonale est réinitialisée par la boucle 15. Enfin, l'ensemble du tableau est lu dans le nid de boucles 20. Ce sont les mêmes éléments qui sont redéfinis d'une itération sur l'autre, ce qui introduit des dépendances directes, et empêche la parallélisation de la boucle 25. Si l'on arrive à montrer que toutes les lectures d'un élément de `SUU` sont précédées d'une écriture à la même itération, le tableau `SUU` pourra être déclaré local à la boucle, et les dépendances précédentes seront éliminées. Pour cela, il est nécessaire de connaître avec exactitude les éléments de tableaux écrits et importés 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 ce rapport serait certainement plus coûteuse qu'une analyse dédiée [56, 44, 29], mais un compilateur ayant par ailleurs besoin d'informations sur les accés 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.

Fabien Coelho utilise déjà 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 ont lieu, les autres processeurs étant des nœuds de calcul. Ce modèle est représenté figure 1.3. Considérons par exemple le programme de la figure 1.2 dans lequel les éléments `A[1:n]` sont lus par l'instruction `READ`.

Idéalement, l'hôte effectue la lecture, puis envoie les éléments lus aux nœuds sur lesquels ils doivent résider. Le code généré est celui modélisé par la figure 1.4. Symétriquement, chacun des nœuds reçoit les éléments qui lui sont destinés. Reste à déterminer l'ensemble des éléments concernés, pour pouvoir générer les communications. Si cet ensemble est sur-

```
      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
```

FIG. 1.1 - *Extrait d'un code de l'*ONERA.

estimé, l'hôte va envoyer aux nœuds des éléments qui n'auront pas forcément été redéfinis par l'entrée/sortie, et dont la valeur sera 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 1.5). Les éléments qui seront renvoyés vers les nœuds seront 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 ([35, 36])), le concept de *régions* [60] a été choisi pour représenter les références aux éléments des tableaux. Ces régions étaient initialement utilisées pour exprimer les effets des procédures sur les ensembles d'éléments de tableaux, et étaient donc considérées comme des sur-approximations des ensembles réellement référencés. Le but de cette étude est de modifier ce calcul des régions pour préserver l'exactitude de l'analyse dans le plus grand nombre de cas possibles, puis d'utiliser les régions exactes pour privatiser les sections de tableaux. L'originalité de ce travail réside dans l'utilisation des effets sur les valeurs des variables scalaires entières pour propager les régions intraprocéduralement, afin de préserver l'exactitude de l'analyse en cas de modification d'une variable scalaire entière par le programme.

Ce rapport est organisé de la façon suivante. Le chapitre 2 présente une bibliographie sur l'analyse de flot de données et la privatisation de tableaux, et permet de situer le problème dans son contexte historique, et d'introduire le vocabulaire et les notions principales utilisées dans ce domaine. Le chapitre 3 donne une vision générale du paralléliseur PIPS et présente certaines de ses caractéristiques qui seront nécessaires à la compréhension des algorithmes de calcul des régions exactes. Le chapitre 4 définit les régions et les opérateurs permettant de les manipuler. Le chapitre 5 présente de manière apporfondie le calcul intraprocédural des effets exacts des instructions sur les éléments de tableaux sous forme de régions `READ/WRITE`. Le chapitre 6 introduit deux nouveaux types de régions, les régions `IN` et `OUT`, qui seront nécessaires au chapitre 7 pour effectuer la 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
```

FIG. 1.2 - *Programme* HPF.

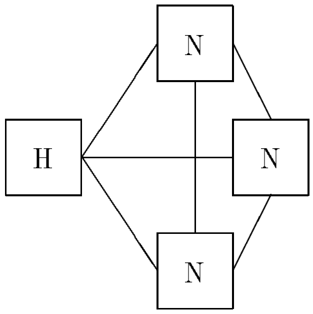


FIG. 1.3 - *Modèle 2PMD.*

Hôte	**Nœud**

`do i=1, n`
`   READ *, A(i)`
`enddo`
	`...`

`C envoi des elts de A` `C vers les nœuds`	`C récupération des elts` `C locaux de A`

FIG. 1.4 - *Code idéalement généré.*

Hôte	**Nœud**
`C récupération des elts` `C de A`	`C envoi des elts locaux` `C de A à l'hôte`

`do i=1, n`
`   READ *, A(i)`
`enddo`
	`...`

`C envoi des elts de A` `C vers les nœuds`	`C récupération des elts` `C locaux de A`

FIG. 1.5 - *Code généré en cas de surestimation de l'ensemble des éléments concernés par l'entrée/sortie.*

Chapitre 2

Étude Bibliographique

Le calcul des régions de tableaux est un moyen de collecter des informations sur les accès aux éléments de tableaux effectués par tel ou tel fragment de programme. Il rentre dans le cadre beaucoup plus genéral de l'*analyse de flot de données*.

La première section de ce chapitre présente une bibliographie chronologique sur l'analyse de flot de données ; elle définit également le vocabulaire employé dans ce domaine, et propose une taxinomie des problèmes d'analyse de flot de données. La deuxième section s'intéresse au problème plus récent de la *privatisation de tableaux*. Les principaux travaux y sont présentés et comparés. Nous essayons également, dans ces deux parties, de situer notre travail par rapport aux études déjà réalisées.

2.1 Analyse de flot de données

L'optimisation d'un programme à la compilation pour le transformer en une forme sémantiquement équivalente nécessite la connaissance d'informations sur la manière dont les données sont définies et utilisées dans le programme. La détermination de ces informations est appelée *analyse de flot de données*.

Par exemple, pour effectuer la propagation des constantes, il est nécessaire de connaître quelles sont les définitions de variables qui atteignent leurs différentes utilisations (problème des *reaching definitions* en anglais). On peut aussi citer le calcul des effets des procédures sur leurs paramètres et sur les variables globales du programme. Les résultats sont utilisés dans les environnements interprocéduraux pour le calcul des dépendances.

Il faut distinguer le problème d'analyse de flot de données dont on cherche les solutions, de la méthode utilisée pour le résoudre. Les premières méthodes de résolution de ces problèmes sont apparues dans la littérature au début des années 1970. Cependant, Kam et Ullman [38] indiquent que leur origine remonte au moins à 1961. Allen [1] a introduit en 1970 une méthode par élimination, dite d'*analyse par intervalles*, dans laquelle l'analyse globale du programme est ramenée à l'analyse de fractions du programme. Elle n'est utilisable que lorsque le graphe de contrôle du programme est réductible. En 1973, Kildall [39] a proposé une approche plus générale, dite *itérative* car les informations sont propagées sur le graphe de contrôle jusqu'à atteindre la stabilité de la solution.

A l'origine, ces deux méthodes étaient utilisées pour l'analyse du flot des données scalaires, seuls les noms des variables étant propagés. Les accès à des éléments d'un tableau étaient donc considérés comme des accès au tableau entier. Ce manque de précision, crucial dans l'analyse des programmes scientifiques, a conduit à adapter les méthodes précédentes pour résoudre les problèmes d'analyse du flot des éléments de tableaux. Un des problèmes qui se posent est alors la représentation des accès à des portions de tableaux. De nombreux travaux [60, 16, 9, 33, 41, 57] s'y sont intéressés dans les années 80 et le début des années 90.

Enfin, une nouvelle approche à été introduite par Feautrier [23] en 1991. Elle s'appuie sur les méthodes utilisées pour l'analyse des dépendences pour résoudre les problèmes d'analyse de flot des données scalaires ou des tableaux.

Cette synthèse est organisée de la façon suivante. La section 2.1.1 propose une taxinomie des problèmes d'analyse de flot de données. Les méthodes de résolution des problèmes d'analyse du flot de données scalaires (itératives et d'élimination) sont ensuite décrites dans la section 2.1.2. La section 2.1.3 est consacrée à l'analyse du flot des éléments de tableaux ; les adaptations des méthodes précédentes puis l'approche de Feautrier sont décrites. Enfin, la section 2.1.4 aborde brièvement quelques axes de recherche en rapport avec l'analyse de flot de données.

2.1.1 Taxinomie des problèmes d'analyse de flot de données

Il existe de nombreux problèmes d'analyse de flot de données. Leur résolution s'appuie sur une représentation abstraite du programme, généralement l'arbre de syntaxe abstrait ou le graphe de contrôle de flot [1] pour une analyse intraprocédurale, ou le graphe des appels pour une analyse interprocédurale. La classification proposée dans cette section distingue les problèmes selon le type de représentation abstraite sur lesquelles on peut les résoudre, et sur la manière dont les informations sont naturellement propagées sur ces graphes.

Analyses intraprocédurales et interprocédurales

Les compilateurs traditionnels effectuent ce que l'on appelle une compilation séparée, c'est-à-dire que les procédures et les fonctions sont analysées indépendamment les unes des autres. Toutefois, ce modèle s'avère insuffisant [14, 35], notamment pour l'analyse des dépendances dans le cadre de la parallélisation automatique. La solution la plus précise consiste à expanser le corps des procédures au niveau de leurs appels. La conséquence de cette approche est l'augmentation importante de la taille du programme, et donc de la taille des problèmes d'analyse de flot de données. Une approche intermédiaire, appelée *analyse interprocédurale*, consiste à propager les informations d'une procédure à une autre, sous forme de résumés. Le programme n'est donc pas modifié, mais la contrepartie est une perte de précision dans les informations propagées. On va donc distinguer deux types d'analyses de flot de données. Les analyses intraprocédurales déterminent et propagent les informations à l'intérieur du corps des procédures, tandis que les analyses interprocédurales propagent les informations entre les procédures, par l'intermédiaire des sites d'appel. Les premiers algorithmes d'analyse de flot de données étaient intraprocéduraux [1, 39], mais l'analyse interprocédurale a été introduite par Allen [2] peu de temps après. Les recherches actuelles concernent les deux problèmes. Sans chercher à être exhaustif, on peut citer [45, 7, 24] pour l'analyse intraprocédurale, et [31, 57] pour l'analyse interprocédurale.

Analyses *descendantes* ou *ascendantes*

Les problèmes d'analyses de flot de données sont appelés *descendantes* [2] ou *ascendantes* [3] selon la direction dans laquelle l'information *coule* dans le graphe de contrôle de flot ou dans le graphe des appels. Pour clarifier ces deux notions, reprenons les deux exemples de l'introduction : les *reaching definitions* et les effets des procédures.

1. Les nœuds de ce graphe sont les séquences linéaires d'instructions simples du programme ; les arcs indiquent les transferts de contrôle entre ces séquences.

2. Forward ou top-down en anglais.

3. Backward ou bottom-up en anglais.

Reaching definitions Considérons le fragment de programme de la figure 2.1 [54]. Le problème est de savoir quelles valeurs y peut prendre lors de son utilisation dans l'instruction S_3, c'est-à-dire quelles sont les définitions de y qui peuvent atteindre S_3. On constate sur son graphe de contrôle que la solution du problème au nœud 6 dépend uniquement des définitions y_1, y_2 et y_3 qui ont lieu sur des nœuds précédant le nœud 6. Le problème des *reaching definitions* est donc un problème d'analyse *descendante*, puisque l'information coule des nœuds 1, 3 et 5 vers le nœud 6.

Effets des procédures Le calcul des effets des procédures est un problème interprocédural. L'information se propage donc sur le graphe des appels. Dans le programme de la figure 2.2, la procédure `ACCES` appelle la procédure `ECRIT`. Si l'on ne connaît pas les effets de `ECRIT` sur la variable `I`, il est impossible de déterminer les effets de `ACCES` sur cette même variable, sauf à adopter une attitude prudente en supposant que `ECRIT` lit et écrit la variable `I`. En regardant son graphe des appels, on s'aperçoit que la solution du problème au nœud `ACCES` dépend de la solution du problème au nœud qui lui succède. L'information doit donc *remonter* le graphe des appels ; c'est bien un problème d'analyse *ascendante*.

Analyses sensibles ou insensibles au flot de contrôle

Les analyses statiques du flot des données supposent que tous les chemins du graphe de contrôle peuvent être empruntés lors de l'exécution du programme. Deux types de problèmes [10] peuvent alors se poser. Un problème est dit *sensible au flot de contrôle*[4], si l'information recherchée doit être exacte sur tous les chemins du graphe de contrôle. Par exemple pour savoir si un tableau est privatisable [61] pour une boucle donnée, il est (entre autres) nécessaire de connaître si les éléments lus sont couverts par les éléments précédemment écrits lors de la même itération. Si cela ne peut être montré, le tableau ne peut pas être privatisé. Un problème est dit *insensible au flot de contrôle*[5], si l'information recherchée doit seulement être vraie sur au moins un des chemins du graphe de contrôle. Le calcul des effets des procédures fait partie de ces problèmes [12, 47]. Considérons pour cela le programme de la figure 2.3 et son graphe des appels. Sur l'un des chemins du graphe, la variable `I` est lue, tandis que sur l'autre, elle est écrite. Les effets de `ACCES` sur la variable `I` s'exprimeront donc de la manière suivante : `I` *peut* être lue ou écrite.

2.1.2 Analyse du flot des données scalaires

Cette section décrit deux méthodes d'analyse du flot des données scalaires, introduites au début des années 1970 : les *méthodes par élimination*, et les *méthodes itératives*. Elles s'appuient toutes les deux sur le graphe de contrôle du programme. Pour les méthodes d'élimination, ce dernier doit être réductible, c'est-à-dire que chaque nœud doit avoir un seul point d'entrée. Cela correspond aux structures de contrôle de la plupart des langages de haut niveau.

A chaque nœud du graphe est associé une équation, ou plus généralement une fonction, qui modélise la manière dont le code associé au nœud influe sur le problème traité. Les méthodes d'élimination cherchent à résoudre directement le système d'équations obtenu, tandis que la méthode itérative se déplace sur le graphe de contrôle en appliquant les fonctions à la solution courante, jusqu'à atteindre la stabilité.

Dans la suite de cette section, nous présentons plus en détail les deux méthodes. Nous prendrons comme exemple celui de la figure 2.4 (issu de [54]), et chercherons à résoudre le problème des *reaching definitions*.

4. *Flow-sensitive problem* ou *must problem* en anglais.
5. *Flow-insensitive problem* ou *may problem* en anglais.

S_1: $y := 9999$
S_2: if $x > z$ then $y := a$
        else if $z > 3 * w$ then $y := b$
S_3: $q := 2 * y$

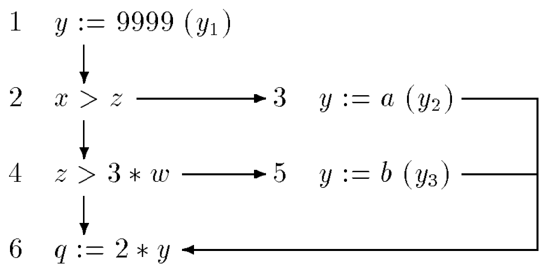


FIG. 2.1 - *Fragment de programme et son graphe de contrôle.*

```
SUBROUTINE ACCES(I)
CALL ECRIT(I)
END

SUBROUTINE ECRIT(I)
I = FOO()
END
```

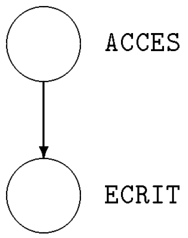


FIG. 2.2 - *Programme bi-procédural et son graphe des appels.*

```
SUBROUTINE ACCES(I)
IF (I.LE.2) THEN
     CALL ECRIT(I)
ELSE
     CALL LIT(I)
END

SUBROUTINE ECRIT(I)
I = FOO()
END

SUBROUTINE LIT(I)
J = I
END
```

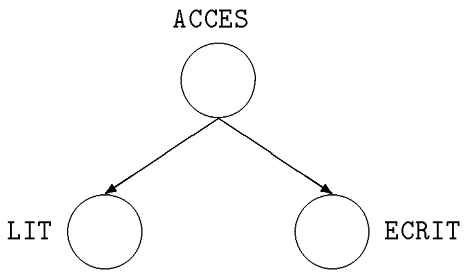


FIG. 2.3 - *Programme tri-procédural et son graphe des appels.*

S_1: $y := 9999$
S_2: if $x > z$ then $x := x - y$; goto S_2
else if $z > 3 * w$ then $y := b$
S_3: $q := 2 * y$

FIG. 2.4 - *Variante du programme de la figure 2.1.*

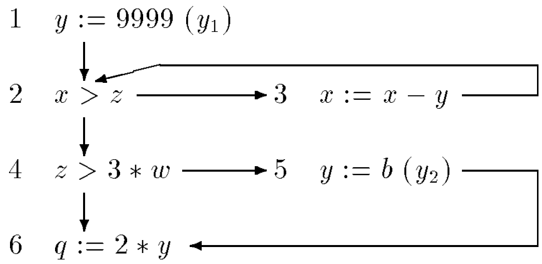


FIG. 2.5 - *Graphe de contrôle.*

$$
\begin{aligned}
X_1 &= \emptyset \\
X_2 &= (p_1 \cap X_1 \cup d_1) \cup (p_3 \cap X_3 \cup d_3) \\
X_3 &= X_4 = (p_2 \cap X_2 \cup d_2) \\
X_5 &= (p_4 \cap X_4 \cup d_4) \\
X_6 &= (p_4 \cap X_4 \cup d_4) \cup (p_5 \cap X_5 \cup d_5)
\end{aligned}
$$

FIG. 2.6 - *Équations décrivant le problème des reaching definitions.*

$$
\begin{aligned}
X_1 &= \emptyset \\
X_2 &= X_3 = X_4 = X_5 = \{d_1\} \\
X_6 &= \{y_1, y_2\}
\end{aligned}
$$

FIG. 2.7 - *Solution.*

Méthodes par élimination

Le problème des *reaching definitions* du programme de la figure 2.4 peut être décrit par des équations. Soit p_j l'ensemble des définitions de y dans le programme si y n'est pas défini au nœud j, et l'ensemble vide sinon. Soit d_j l'ensemble des définitions de y créées au nœud j. L'ensemble X_i des définitions de y qui atteignent le nœud i peut s'exprimer de la manière suivante :

$$
\begin{cases}
X_1 = \emptyset \\
X_i = \bigcup_{j \in Pred(i)} \{p_j \cap X_j \cup d_j\}, i \neq 1
\end{cases}
$$

$Pred(i)$ étant l'ensemble des prédecesseurs du nœud i. Les équations correspondant au graphe de contrôle de la figure 2.5 sont données par la figure 2.6 et leur solution par la figure 2.7.

Le principe des méthodes par élimination consiste à résoudre le système des équations, par une méthode similaire à la méthode d'élimination de Gauss. Lorsque le graphe de contrôle ne contient pas de cycle, cette résolution s'effectue par substitutions successives. Dans le cas contraire, apparaissent des équations dans lesquelles une variable est définie en fonction d'elle-même. C'est le cas pour X_2 dans l'exemple précédent :

$$
\begin{aligned}
X_2 &= (p_1 \cap X_1 \cup d_1) \cup (p_3 \cap X_3 \cup d_3) \\
&= (p_1 \cap X_1 \cup d_1) \cup (p_3 \cap (p_2 \cap X_2 \cup d_2) \cup d_3) \\
&= (\{y_1, y_2\} \cap X_2) \cup \{y_1\}
\end{aligned}
$$

Pour traiter ces équations, on les remplace par d'autres équations de telle sorte qu'une solution du nouveau système d'équations est aussi solution de l'ancien. Ce procédé porte le nom de *cassage de boucle* (*loop breaking*). A chaque problème d'analyse de flot de données correspondent des règles permettant de trouver l'équation qui remplacera l'équation initiale.

Par exemple, pour un problème dont les équations sont de la forme :

$$X = (a \cap X) \cup b$$

l'équation de remplacement sera donnée par :

$$X = b$$

Cela revient à sélectionnner la solution minimale de l'équation initiale parmi l'ensemble de toutes les solutions possibles. Ainsi, dans notre exemple, on remplace l'équation $X_2 = (\{y_1, y_2\} \cap X_2) \cup \{y_1\}$ par $X_2 = \{y_1\}$. A l'aide de ce procédé, on peut résoudre les problèmes d'analyse de flot de données sur les graphes comportant des boucles.

L'inconvénent de cette méthode est sa complexité maximale en $O(n^3)$, n étant le nombre d'équations. Les **méthodes d'analyse par intervalles** permettent de diminuer cette complexité en partitionnant le graphe de contrôle en régions, ou intervalles, qui vont diriger le processus de résolution. Ces techniques comportent généralement trois phases : recherche des intervalles, élimination, propagation. Les deux premières sont parfois regroupées en une seule phase. La phase d'élimination consiste, dans chaque intervalle, à exprimer les équations en fonction de la variable associée au nœud source de l'intervalle auquel elles appartiennent ; ensuite, le système formé des r équations associées aux nœuds sources des intervalles est résolu. La phase de propagation substitue dans les équations internes aux intervalles, l'expression de la variable du nœud source, qui a été trouvée lors de la phase précédente. Les algorithmes d'élimination diffèrent essentiellement sur la manière dont le graphe de contrôle est partitionné, et sur l'ordre de substitution des variables.

Allen Les intervalles d'Allen [1, 3] sont des régions ayant un seul point d'entrée et correspondant approximativement aux boucles du graphe de contrôle. Leur détermination dépend du caractère réductible de ce dernier. Les nœuds sont ajoutés aux intervalles dans un ordre qui préserve l'ordre partiel induit par les dépendances entre les équations. Dans le graphe de la figure 2.5, les intervalles sont :

$$\begin{aligned} I_1 &= \{1\} \\ I_2 &= \{2, 3\} \\ I_3 &= \{4, 5, 6\} \end{aligned}$$

Les nœuds sources des intervalles sont les nœuds 1, 2 et 4. La première étape de la phase d'élimination effectue les substitutions internes à chaque intervalle :

$$\begin{array}{ll} I_1 : & X_1 = \emptyset \\ I_2 : & X_2 = (p_1 \cap X_1 \cup d_1) \cup (p_3 \cap (p_2 \cap X_2 \cup d_2) \cup d_3) \\ & X_3 = (p_2 \cap X_2 \cup d_2) \\ I_3 : & X_4 = (p_2 \cap X_2 \cup d_2) \\ & X_5 = (p_4 \cap X_4 \cup d_4) \\ & X_6 = (p_4 \cap X_4 \cup d_4) \cup (p_5 \cap (p_4 \cap X_4 \cup d_4) \cup d_5) \end{array}$$

On constate que X_3 ne dépend que de X_2, alors que X_5 et X_6 dépendent uniquement de X_4. Par contre, les solutions aux nœuds sources, X_1, X_2 et X_4, déependent les unes des autres. Le système réduit, constitué des équations des nœuds sources des intervalles, est alors résolu par la méthode d'élimination décrite précédemment.

$$\left\{ \begin{array}{l} X_1 = \emptyset \\ X_2 = (p_1 \cap X_1 \cup d_1) \cup (p_3 \cap (p_2 \cap X_2 \cup d_2) \cup d_3) \\ X_4 = (p_2 \cap X_2 \cup d_2) \end{array} \right. \iff \left\{ \begin{array}{l} X_1 = \emptyset \\ X_2 = X_4 = \{y_1\} \end{array} \right.$$

La phase de propagation donne enfin les solutions aux nœuds situés à l'intérieur des intervalles :

$$\begin{array}{ll} I_2: & X_3 = \{y_1\} \\ I_3: & X_5 = \{y_1\} \\ & X_6 = \{y_1, y_2\} \end{array}$$

La complexité de cet algorithme est en $O(n^2)$.

Améliorations D'autres algorithmes ont été proposés, de moindre complexité. Ceci est obtenu en évitant le recalcul des substitutions communes à plusieurs équations.

Hecht et Ullman [62] utilisent des intervalles similaires à ceux de Allen et Cocke. L'ordre des substitutions est non-déterministe, mais elles sont enregistrées dans un arbre pondéré pour éviter leur éventuel recalcul.

Les intervalles de Tarjan [58, 59] sont des sous-graphes fortement connexes à un seul point d'entrée. C'est une définition plus restrictive que celle de Allen et Cocke, mais elle modélise plus fidèlement les structures bouclantes du graphe de contrôle. L'analyse de Tarjan établit un ordre linéaire sur les variables, qui définit l'ordre dans lequel elles seront éliminées. De plus, certaines substitutions sont retardées jusqu'au moment où les facteurs communs peuvent être détectés ; ils sont alors calculés, puis utilisés.

L'algorithme de Graham et Wegman [27] partitionne le graphe de contrôle en *S-sets*, dont les points d'entrée sont les nœuds cibles des arcs arrières du graphe. Les *S-sets* sont aussi des composantes fortement connexes du graphe de contrôle, mais présentent des différences par rapport aux intervalles de Tarjan. L'algorithme utilise un ordre déterministe de substitution différent pour chaque terme individuel des équations.

Les algorithmes de Hecht-Ullman et de Graham-Wegman sont en $O(n \log n)$, et celui de Tarjan en $O(n\alpha(n))$, avec $\alpha(n) \leq 3$ dans la pratique. C'est donc ce dernier qui est le plus performant.

Signalons enfin que l'algorithme de Graham et Wegman s'applique aux graphes de contrôle irréductibles, et que celui de Allen et Cocke peut être étendu de manière à les traiter.

Méthodes itératives

C'est Kildall [39] qui le premier a donné un algorithme permettant de résoudre les problèmes d'analyse de flot de données sur un graphe de contrôle quelconque, c'est-à-dire non réductible.

Informellement, le principe est le suivant. On associe au nœud source du graphe de contrôle un ensemble initialisé, solution du problème avant toute exécution de ce nœud. Cette information est propagée vers les nœuds successeurs à l'aide d'une *fonction d'optimisation*. Les successeurs peuvent alors mettre à jour l'ensemble qui leur est associé. On réitère ce procédé sur le graphe de contrôle tant que la stabilité de la solution n'est pas atteinte.

On peut formaliser cet algorithme à l'aide des théories des graphes et des treillis. Le graphe de contrôle est un graphe orienté fini $G < N, A >$, N étant l'ensemble des nœuds, et A l'ensemble des arcs. G comporte un ensemble S de nœuds sources tels que pour tout nœud du graphe, il existe un chemin allant à ce nœud et partant d'un nœud source. Pour tout nœud n du graphe, on définit l'ensemble des ses successeurs immédiats, $Succ(n)$, et celui de ses prédécesseurs immédiats, $Pred(n)$.

Soit E l'ensemble fini des ensembles de solutions possibles, et soit l'opérateur *meet* $\wedge$, avec les propriétés suivantes :

$$\begin{aligned}
&\wedge : E \times E \rightarrow E, \\
&x \wedge x = x \text{ (idempotens)}, \\
&x \wedge y = y \wedge x \text{ (commutativité)}, \\
&x \wedge (y \wedge z) = (x \wedge y) \wedge z \text{ (associativité)}, \\
&\exists \perp \in E \mid \forall\, x \in E,\ \perp \wedge\, x = \perp \text{ (élément nul)},
\end{aligned}$$

où $x, y, z \in E$. L'ensemble E et l'opérateur $\wedge$ définissent un semi-treillis L. L peut également contenir un élément neutre $\top$ tel que $\forall\, x \in E,\ \top \wedge x = x$. Enfin, l'opérateur $\wedge$ induit un ordre partiel sur E donné par :

$$x \leq y \iff x \wedge y = x,\ \forall\, x, y \in E$$

On associe à chaque nœud du code une fonction qui représente intuitivement la manière dont les données sont transformées lorque le contrôle passe à travers le nœud. Une telle fonction d'optimisation f prend en argument le numéro du nœud considéré (n), ainsi que la solution courante (x). Elle possède les propriétés suivantes :

$$\begin{aligned}
&f : N \times E \rightarrow E, \\
&f(n, x \wedge y) = f(n, x) \wedge f(n, y), n \in N, x, y \in E \text{ (propriété d'homéomorphisme)}.
\end{aligned}$$

L'algorithme peut alors être décrit de la manière suivante. Soit $S_{init} \subseteq S \times E$ un ensemble de paires associées aux nœuds sources, tel que $(s, x) \in S_{init}$ si $s \in S$ est un nœud source auquel on associe l'ensemble solution initial $x \in P$.

1. initialisation	$T \leftarrow S_{init}$
2. terminaison	si $T = \emptyset$ alors fin
3. sélection du nœud	soit $T' \in T,\ T' = (n, e_i),\ n \in N,\ e_i \in E$
	$T \leftarrow T - T'$
4. traversée	soit e_n la valeur courante de la solution au nœud n
	si $e_n \leq e_i$ alors aller en 2
5. nouvelle solution	$e_n \leftarrow e_n \wedge e_i$
	$T \leftarrow T \cup \{(n', f(n, e_n)) \mid n' \in Succ(n)\}$
6. boucle	retourner en 2

Kildall a montré que cet algorithme se termine, et qu'il calcule l'information souhaitée. La complexité en nombre d'itérations est en $O(nh(L))$ où $h(L)$ donne la profondeur du semi-treillis L, qui peut être une fonction de n. Toutefois, Kam et Ullman [38] ont montré que sous certaines conditions, la complexité en nombre d'itérations était de l'ordre de $d+3$, d étant le nombre maximal d'arcs arrières dans un chemin sans cycle du graphe de contrôle.

Appliquons cet algorithme à l'exemple de la figure 2.4. La fonction d'optimisation est la suivante :

$$\begin{aligned}
f(1, e_1) &= e_1 \cup \{y_1\} \\
f(2, e_2) &= e_2 \\
f(3, e_3) &= e_3 \\
f(4, e_4) &= e_4 \\
f(5, e_5) &= e_5 \cup \{y_2\} \\
f(6, e_6) &= e_6
\end{aligned}$$

L'opérateur *meet* utilisé est l'opérateur d'union sur les ensembles ($\wedge = \cup$). Et $x \leq y \iff y \subseteq x$. Le déroulement de l'algorithme est donné par la figure 2.8. On constate que l'on passe deux fois par le nœud 2 ($n \leftarrow 2$ sur la figure). La solution étant constante, c'est celle qui est finalement retenue. On retrouve bien les solutions obtenues avec les méthodes précédentes.

1 $T \leftarrow S_{init} = \{(1, \emptyset)\}$
3 $T' \leftarrow (1, \emptyset),\ n \leftarrow 1,\ e_i \leftarrow \emptyset$
$T \leftarrow \emptyset$
4 $e_1 = \top \not\leq e_i$
5 $e_1 \leftarrow e_1 \cup e_i = \emptyset$
$T \leftarrow T \cup \{(2, f(1, e_1)\} = \{(2, \{y_1\})\}$
3 $T' \leftarrow (2, \{y_1\}),\ \boldsymbol{n \leftarrow 2},\ e_i \leftarrow \{y_1\}$
$T \leftarrow \emptyset$
4 $e_2 = \top \not\leq e_i$
5 $e_2 \leftarrow \{y_1\}$
$T \leftarrow T \cup \{(3, f(2, e_2), (4, f(2, e_2))\} = \{(3, \{y_1\}), (4, \{y_1\})\}$
3 $T' \leftarrow (3, \{y_1\}),\ n \leftarrow 3,\ e_i \leftarrow \{y_1\}$
$T \leftarrow \{(4, \{y_1\})\}$
4 $e_3 = \top \not\leq e_i$
5 $e_3 \leftarrow \{y_1\}$
$T \leftarrow T \cup \{(4, f(3, e_3)\} = \{(4, \{y_1\}), (2, \{y_1\})\}$
3 $T' \leftarrow (4, \{y_1\}),\ n \leftarrow 4,\ e_i \leftarrow \{y_1\}$
$T \leftarrow \{(2, \{y_1\})\}$
4 $e_4 = \top \not\leq e_i$
5 $e_4 \leftarrow \{y_1\}$
$T \leftarrow T \cup \{(5, f(4, e_4), (6, f(4, e_4)\} = \{(2, \{y_1\}), (5, \{y_1\}), (6, \{y_1\})\}$
3 $T' \leftarrow (2, \{y_1\}),\ \boldsymbol{n \leftarrow 2},\ e_i \leftarrow \{y_1\}$
$T \leftarrow \{(5, \{y_1\}), (6, \{y_1\})\}$
4 $e_2 = \{y_1\} = e_i$
3 $T' \leftarrow (5, \{y_1\}),\ n \leftarrow 5,\ e_i \leftarrow \{y_1\}$
$T \leftarrow \{(6, \{y_1\})\}$
4 $e_5 = \top \not\leq e_i$
5 $e_5 \leftarrow \{y_1\}$
$T \leftarrow T \cup \{(6, f(5, e_5)\} = \{(6, \{y_1\}), (6, \{y_1, y_2\})\}$
3 $T' \leftarrow (6, \{y_1\}),\ n \leftarrow 6,\ e_i \leftarrow \{y_1\}$
$T \leftarrow \{(6, \{y_1, y_2\})\}$
4 $e_6 = \top \not\leq e_i$
5 $e_6 \leftarrow \{y_1\}$
$T \leftarrow T \cup \emptyset = \{(6, \{y_1, y_2\})\}$
3 $T' \leftarrow (6, \{y_1, y_2\}),\ n \leftarrow 6,\ e_i \leftarrow \{y_1, y_2\}$
$T \leftarrow \emptyset$
4 $e_6 = \{y_1\} \not\leq e_i$
5 $e_6 \leftarrow \{y_1, y_2\}$
$T \leftarrow T \cup \emptyset = \{(6, \{y_1, y_2\})\}$
2 $T = \emptyset$

FIG. 2.8 - *Résolution du problème des reaching definitions pour le programme de la figure 1 par la méthode itérative.*

2.1.3 Analyse des accès aux éléments de tableaux

Les méthodes classiques que nous venons de présenter résolvent parfaitement les problèmes de flot des données scalaires, mais traitent les accès à des éléments d'un tableau comme des accès au tableau entier. Ceci induit un manque de précision, qui n'est pas acceptable au sein d'un compilateur ou d'un paralléliseur de programmes scientifiques. Cette section présente donc deux techniques qui permettent de traiter les problèmes d'analyse de flot des éléments de tableau. Elles sont très utilisées pour l'analyse interprocédurale des effets des procédures sur les tableaux.

Amélioration des méthodes d'analyse de flot de données scalaires

Une première solution pour analyser les accès aux éléments de tableaux, est d'adapter les méthodes décrites précédemment. Gross et Steenkiste [28] se sont ainsi intéressés à l'adaptation de la méthode d'Allen, tandis que Triolet [60], Callahan, Havlak et Kennedy [16, 34] ont préféré la méthode itérative. Les algorithmes étant très généraux, il suffit, pour pouvoir les réutiliser, de déterminer la manière de représenter les accès aux tableaux par des ensembles, et de définir les opérateurs d'union, d'intersection et de différence sur ces ensembles.

On peut distinguer deux types de représentations des accès aux tableaux. Celles qui donnent une approximation de l'ensemble des éléments réellement référencés. Et celles qui décrivent avec précision cet ensemble.

Représentations approchées En 1984, Triolet [60] a introduit le concept de *région* d'un tableau. Une région est un ensemble d'éléments de tableau décrit par des équations et inéquations affines formant un polyèdre convexe. Les régions permettent de représenter des ensembles de forme arbitrairement complexe, à condition qu'elle soit convexe. Par exemple, il est impossible de représenter l'accès à tous les éléments d'indices pairs ; ou encore l'accès à deux éléments géométriquement distants. De plus, les opérations d'union et de projection ont une complexité théorique élevée, même si, dans la pratique, elle se révèle polynomiale.

Les *Regular Section Descriptors*, ou *RSDs*, introduits par Callahan et Kennedy [16] en 1988, permettent de représenter des accès selon des schémas pré-déterminés : les lignes, les colonnes, les diagonales, et le tableau entier, ainsi que leurs équivalents dans des dimensions d'ordre supérieur. L'intérêt de cette approche est le coût peu élevé des opérations d'union et d'intersection. Plusieurs variantes des RSDs sont apparues par la suite. Balasudaram [9] a proposé les *Data Access Descriptors* qui utilisent des polyèdres convexes dont les bords sont parallèles aux axes ou aux diagonales. C'est une approche intermédiaire entre les RSDs et les régions. Havlak et Kennedy [33] ont enfin introduit les RSDs à bornes et pas.

Représentations exactes D'autres auteurs ont préféré des représentations exactes des éléments de tableaux référencés. Li et Yew conservent des listes d'*atomes*, chaque atome représentant un accès à un élément de tableau. Leur approche revient à ne pas effectuer d'union, ce qui donne à leur méthode une grande précision, mais nécessite plus d'espace mémoire.

Tang [57] représente et propage l'image exacte des références aux éléments de tableaux sous la forme de problèmes en nombres entiers.

Mais ces deux méthodes ne sont *exactes* que dans la mesure où les expressions des indices des tableaux sont linéaires en fonction des bornes de boucle, et ne dépendent que de pseudo-constantes, c'est-à-dire de variables qui ne varient pas dans le graphe de contrôle considéré.

```
s := 0;
for i := 1 to n do
    x[2*i-1] := 1;
for k := 1 to 2*n-1 do
    s := s + x[k]
```

Dépendance directe entre x[2*i-1] et x[k] :

$$q = si\ 2((k+1) \div 2) - (k+1) \geq 0\ alors\ (k+1)\ sinon\ \perp$$

FIG. 2.9 - *Solution du problème des dépendances directes sous forme de quast.*

Méthode basée sur le calcul des dépendances exactes

Présentation du problème et de sa solution Une autre approche pour résoudre les problèmes d'analyse du flot des éléments de tableaux a été introduite par Feautrier [23] en 1991. Elle consiste à trouver une solution exacte au problème suivant : étant donnée une référence en lecture à un élément de tableau, quelle est l'origine de sa valeur ? On dit alors qu'il existe une dépendance directe, ou basée sur la valeur (*value based dependence*), entre les deux références, par opposition à une dépendance relative aux cases mémoire auxquelles on accède.

Si la référence en lecture est dans un nid de boucles, les valeurs des indices du tableau dépendent des indices des boucles. La solution exacte sera donc une fonction des valeurs de ces indices de boucles, car à chaque instance de la référence en lecture doit correspondre une et une seule origine. Cette fonction est appelée *fonction source*. Son calcul est assez complexe, et ne sera pas détaillé ici. Indiquons simplement qu'il utilise le prédicat de séquencement du programme ($\prec$), et un opérateur $max_{\prec}$ basé sur ce prédicat.

Le résultat est présenté sous la forme d'un arbre de décision, le *quast* (pour Quasi Affine Selection Tree). Le qualificatif quasi-affine provient du fait que, en plus des opération $+$, $-$ et $\times$, la division entière peut également être utilisée pour représenter la solution. La figure 2.9 montre un exemple tiré de [23] et sa solution sous forme de *quast*.

Intérêt par rapport aux méthodes précédentes Dans les méthodes adaptées des méthodes d'analyse du flot des données scalaires, les différentes instances d'une référence ne sont pas distinguées les unes des autres, mais agrégées dans un résumé. Il est alors impossible de trouver une solution exacte au problème des dépendances directes. Par contre, elles sont beaucoup plus générales quand aux structures de contrôles qu'elles permettent de traiter. En effet, le point faible de l'approche décrite précédemment est le manque de généralité du langage source accepté. Les seules constructions prises en compte sont les séquences linéaires et les boucles for. De plus, les indices des boucles et des tableaux doivent être affines, et ne doivent pas dépendre de variables dont la valeur n'est pas constante sur la portion de programme analysée. Enfin, l'analyse est intraprocédurale, c'est-à-dire que le code ne doit pas comporter d'appels à des procédures ou à des fonctions.

Améliorations proposées Maydan *et al.* [46] ont proposé une autre version des *quasts*, les *Last Write Tree* ou *LWT*. Leur algorithme est moins général que celui de Feautrier, qu'ils utilisent dans les cas critiques. L'avantage de leur approche est sa rapidité d'exécution.

Maslov [45] propose une manière plus efficace de calculer les fonctions sources, sur un domaine plus vaste que celui de Feautrier, puisque les structures de contrôle conditionnelles et les fragments de programme non-affines sont autorisés. Lors du calcul de la fonction source d'une référence en lecture L donnée, il commence par chercher la solution parmi les références en écriture qui sont lexicographiquement les plus proches de L. Il garde trace des instances de L dont il a déjà trouvé la source, et il élargit son champ de recherche jusqu'à ce qu'il ait couvert toutes les instances de L. Il qualifie son algorithme de *paresseux* car il privilégie la proximité des références. L'avantage de son approche est sa rapidité conjuguée à une plus grande généralité.

Citons également les travaux de Pugh et Wonnacott [51]. Ils calculent d'abord les dépendances basées sur la mémoire. Lors d'une deuxième phase, ils raffinent les résultats trouvés lors de la première phase. Leur approche est de ce fait plus lente que la précédente. Leur justification est que l'information calculée lors de la première phase sera utilisée lors des optimisations. Pour améliorer la vitesse de leur algorithme, ils ont implémenté un mécanisme similaire à celui de Maslov pour garder trace des références déjà couvertes.

Enfin, Feautrier et Collard [17, 24] ont étendu le calcul des *quasts* aux tests dynamiques et aux boucles while.

Bien sûr, l'exactitude de l'analyse n'est pas préservée par les extensions aux fragments de programme non-affines ou à contrôle non-statique proposées par Maslov, Feautrier et Collard.

2.1.4 Autres axes de recherche

Autres méthodes de résolution

D'autre méthodes de résolution des problèmes d'analyse du flot des données ont été proposées. Notre propos n'est pas de les décrire ici. Signalons tout d'abord la conversion des programmes en *Static Single Assignement Form* [21]. Un programme transformé ne contient plus d'affectations multiples d'une même variable. Les analyses de flot de données ultérieures en sont bien évidemment facilitées. Par contre, il semblerait que cette méthode ne permette pas de résoudre de manière efficace certains problèmes comme celui des variables vivantes (*live variables*). Pour pallier ce problème, la même équipe a introduit le concept de *sparse data flow evaluation graph* dont les nœuds sont ceux du graphe de contrôle. Les arcs propagent directement l'information vers les nœuds qui l'utilisent effectivement. L'analyse du flot de données s'effectue alors sur ce graphe. D'autres extensions des *SSA form* incluent des informations sur les conditions sous lesquelles telle variable prend telle valeur : Alpern *et al* ont ainsi introduit les *Gated Static Single Assignment Form* [4], et Havlak les *Thinned Gated Static Single Assignment Form* [32] pour les fragments de code non structurés. Ces graphes portent des informations permettant ensuite d'effectuer des analyses symboliques, qui seront donc plus précises que les analyses traditionnelles.

La théorie de l'interprétation abstraite a été introduite par Cousot et Cousot [19] dans le milieu des années 70. Elle consiste à analyser les programmes en fonction de leur sémantique, en associant à tout objet de la syntaxe abstraite du langage un élément d'un domaine abstrait. Par exemple, Cousot et Halbwachs [20] ont appliqué ce modèle au calcul de contraintes linéaires existant entre les variables scalaires entières d'un programme. Jouvelot [37] a défini une interprétation abstraite donnant une approximation de l'ensemble des cases-mémoire pouvant être concernées par un élément du programme (expression, instruction, ...). Il a nommé ces ensembles *régions* par analogie avec les régions de Triolet.

Environnements de programmation

Le domaine de la compilation se situe dans celui plus vaste des environnements de programmation, et notamment dans celui des environnements interactifs. L'utilisateur doit pouvoir y modifier ses programmes puis les recompiler dans des délais les plus brefs possibles. Deux techniques permettent de réduire le temps passé lors des phases d'analyse de flot de données. Le recalcul incrémental des solutions [53, 15, 18] permet d'éviter de recompiler l'ensemble du programme si les modifications apportées sont facilement modélisables par une modification de la solution de l'analyse. Le problème devient bien sûr encore plus complexe dans un environnement interprocédural. La parallélisation des analyses [8, 55, 63] peut permettre un gain de temps appréciable. Cette parallélisation peut porter sur les analyse interprocédurales ; les modules sont alors analysés en parallèle, autant que leur interdépendance le permet. Mais les analyses intraprocédurales peuvent aussi être parallélisées. Ce sont alors les intervalles qui sont traités en parallèles. La difficulté provient de problèmes d'équilibrage de charge, lorsque les intervalles sont de tailles très inégales.

2.1.5 Conclusion

L'analyse du flot des données est un domaine de recherche dont les origines remontent aux année 1960-70. Malgré cela, il est toujours très actif. En effet, si le problème de l'analyse du flot des données scalaires est bien maîtrisé, celui de l'analyse du flot des éléments de tableaux est loin d'être parfaitement résolu. De nombreuses solutions approchées ont été proposées, sans que l'on puisse établir de comparaison globale. Elles diffèrent par la manière dont sont résumés les accès multiples à des éléments d'un même tableau, et par la complexité des opérateurs permettant de manipuler ces représentations. De plus, leur manque de précision dans la représentation les rend souvent mal adaptées pour traiter les problèmes sensibles au flot de contrôle. Toutefois, elles peuvent bénéficier des résultats d'analyses symboliques antérieures comme Havlak [32] l'a montré pour les *RSD*, et comme nous le montrerons pour les régions dans la suite de ce rapport, et elles paraissent bien adaptées pour représenter les effets des procédures sur leurs paramètres et les variables globales. Des représentations exactes ont été introduites. Elles pèchent soit par leur inefficacité en terme d'occupation mémoire, soit par les restrictions qu'elles imposent sur le langage source (indices affines ...). Tout récemment, une nouvelle approche, basée sur le calcul des dépendances a été proposée. Elle résoud le problème des fonctions sources, qui est un problème sensible au flot de contrôle. Elle donne dores et déjà de bons résultat, mais, si la plupart des restricitions sur le langage source ont été levées, le programme source doit toujours être monoprocédural. Enfin, la recherche est très active dans le domaine de l'intégration de ces techniques dans des environnements de programmation interactifs.

2.2 Privatisation et expansion de tableaux

L'expansion et la privatisation de scalaires sont des techniques bien connues dans le cadre de la parallélisation automatique. L'expansion de scalaire, mieux adaptée aux machines à mémoire partagée consiste à transformer le scalaire en un tableau à une dimension pour distinguer les différentes instances des écritures de cette variables. Des dépendances producteur-producteur sont ainsi éliminées, ainsi que, par transitivité, des dépendances producteur-consommateur, et consommateur-producteur. Ceci rend possible la parallélisation de certaines boucles. L'inconvénient de cette technique est la quantité de mémoire qu'elle nécessite. La privatisation de scalaires permet d'éviter en partie ce surcoût dans le contexte d'une machine à mémoire distribuée. Elle consiste à allouer une version locale de la variable concernée sur chacun des processeurs. Les dépendances sont pareillement éliminées.

En 1986, Padua et Wolfe [48] ont proposé d'étendre ces techniques aux tableaux. Et en 1992, Blume et Eigenmann [14] ont montré expérimentalement la nécessité de privatiser les tableaux pour une parallélisation efficace.

On peut distinguer deux phases dans ce type d'optimisation :

– la détection des *bons* candidats, c'est-à-dire les tableaux dont les éléments interviennent dans des dépendances de type producteur-producteur que l'on ne peut pas éliminer par un simple renommage (dépendances entre deux instructions d'une séquence) ;

- la récupération de la valeur des éléments privatisés ou expansés ; dans le tableau global pour une privatisation (détection de la dernière valeur normalement affectée dans le programme initial) ; aux sites de lecture des éléments initiaux pour une expansion.

Plusieurs travaux se sont déjà penchés sur le problème : Feautrier [22] s'est intéressé à l'expansion de tableaux, tandis que Li [42], Maydan et al. [46], et Tu et Padua [61] ont proposé des algorithmes de privatisation de tableaux.

Les approches de Feautrier et de Maydan s'appuient sur l'analyse fine des dépendances (fonctions sources ou LWT). Feautrier s'en sert pour remplacer les utilisations des éléments du tableau initial par les éléments correspondants du tableau expansé. Maydan les utilise dans les deux phases de l'optimisation. L'originalité de son aprroche réside dans la définition de *vecteurs de dépendance de flot de données* par analogie avec les vecteurs de distance classique. Il les calcule à l'aide des informations portées par les LWT. L'inconvénient de ces deux approches réside dans les restrictions sur les langages source qu'elles imposent (environnement monoprocédural, hypothèses de linéarité, programme à contrôle statique).

Li et Padua s'appuient tous deux sur des analyses de flot de données classiques adaptées à l'étude des ensembles d'éléments de tableaux. Padua représente ceux-ci à l'aide des RSD, tandis que Li reste dans le domaine théorique en ne donnant que les équations générales permettant de résoudre le problème, ce qui lui permet d'aborder la privatisation des sections de tableaux. Ces deux techniques s'appuient sur des graphes de contrôle hiérarchiques qui permettent de tenir compte de la spécificité des boucles DO. Toutefois, Li a étendu son algorithme de manière à prendre en compte les graphes de contrôle les plus généraux. Signalons l'originalité de l'approche de Padua pour déterminer la valeur des éléments de tableaux en sortie de boucle. Lorsque l'analyse statique échoue, le tableau peut tout de même être privatisé grâce à une analyse dynamique ; la contrepartie est le coût de communication élevé que cela induit. Notons que Tu et Padua sont les seuls à proposer une analyse interprocédurale.

Enfin, est paru tout récemment un article proposant une résolution du problème de la privatisation de tableaux à l'exécution [52].

L'algorithme de privatisation que nous proposons dans le chapitre 7 devrait bénéficier, par rapport aux travaux de Li et de Padua, des informations symboliques apportées par les analyses antérieures au calcul des régions, et qui sont utilisées pour leur calcul et leur propagation. Mais seule l'expérience permettra d'en juger. Par contre, notre méthode est difficilement comparable avec celle de Feautrier et de Maydan, puisque le langage source que nous acceptons est beaucoup moins restreint, mais que notre analyse est également moins fine.

Chapitre 3

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).

Ce chapitre présente rapidement l'architecture logicielle de PIPS, ainsi que les analyses et transformations de programme qu'il permet d'effectuer. Pour une description plus complète, le lecteur pourra se référer à [36, 11, 35].

3.1 Architecture logicielle

L'architecture de PIPS est présentée sous forme synoptique dans la figure 3.1. Chaque analyse ou transformation est effectuée à la demande lors d'une phase distincte. Ces phases peuvent dépendre du résultat d'autres phases. Ces dépendances sont décrites dans un fichier sur lequel s'appuie le mécanisme `pips-make` pour assurer l'enchaînement des traitements et le maintien de la cohérence des informations manipulées. Les résultats des différentes phases, encore appelés *ressources*, sont conservés dans une base de données dont la gestion est confiée à l'utilitaire `pipsdbm`. Cette base de données comporte également une copie du programme source.

3.2 Analyses

Notre but n'est pas de décrire ici toutes les phases d'analyse de PIPS, mais uniquement celles qui nous seront utiles lors de la description du calcul des régions. Ces phases peuvent être regroupées sous le terme général d'*analyse sémantique* de programme.

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 non structurés qui permettent de décrire les fragments de code non structurés (utilisation de `GOTO`). Ces nœuds sont annotés par les résultats des analyses qui seront décrites par la suite.

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.

Les résultats des différentes analyses effectuées par PIPS sont utilisées lors de la dernière phase avant la parallélisation proprement dite, l'analyse des dépendances. Elles ont pour

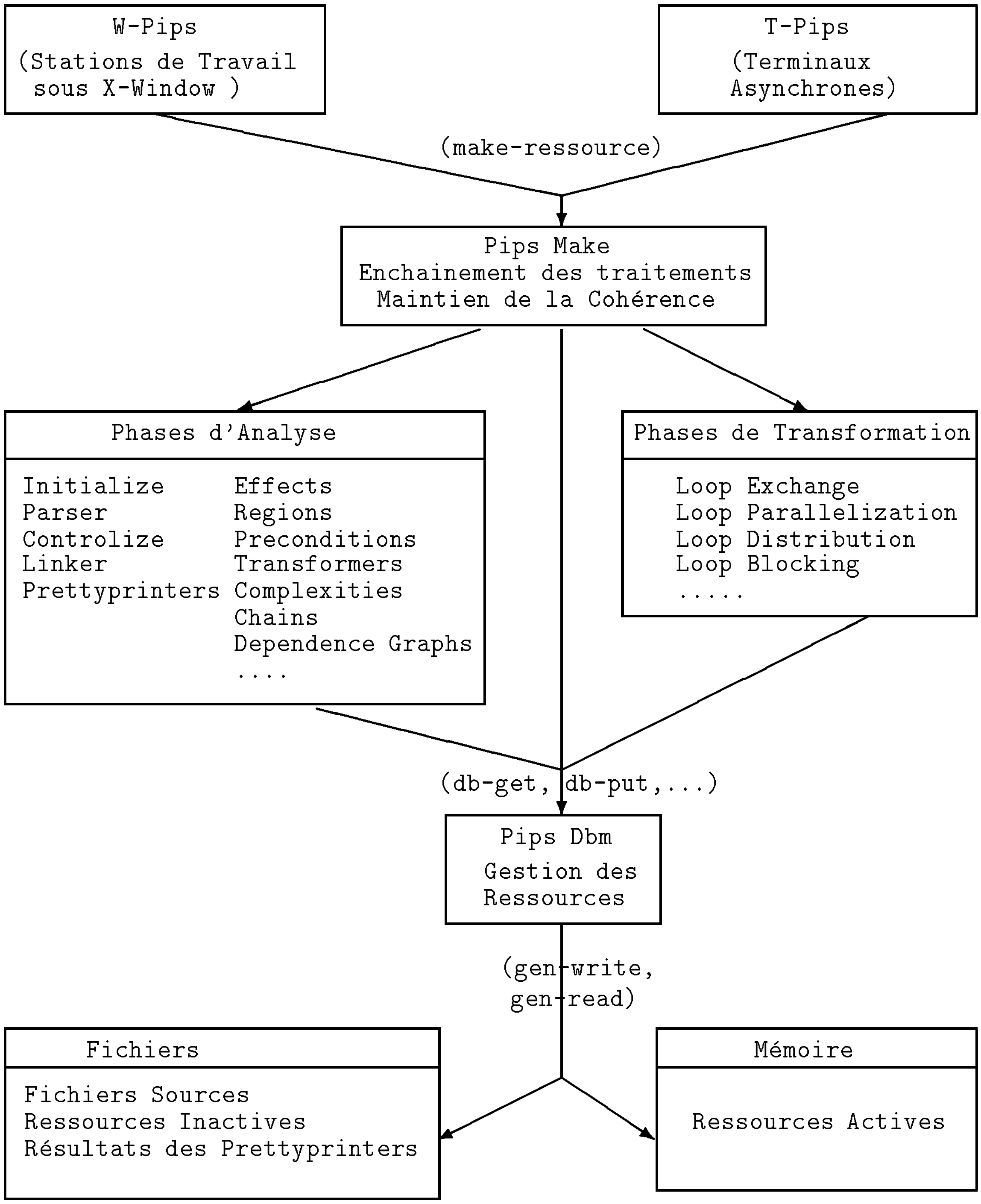


FIG. 3.1 - *Architecture logicielle de PIPS.*

but essentiel de fournir suffisamment d'information pour que dernière phase puisse éliminer le maximum de dépendances pouvant empêcher cette parallélisation.

3.2.1 Effets

Les effets décrivent les opérations effectuées sur la mémoire par une instruction (au sens large) donnée. PIPS distingue les effets en lecture (`READ`) des effets en lecture (`WRITE`), ainis que les effets qui ont toujours lieu (`MUST`) de ceux qui peuvent avoir lieu (`MAY`). Trois types d'effets peuvent être calculés par PIPS :

– les *effets propres* sont les références locales à une instruction ; pour une affectation, ils concerneront toutes les variables accédées en lecture et en écriture, et pour une boucle, ce seront uniquement les effets sur la variable d'index, les bornes de boucle et l'incrément ;

– les *effets cumulés* constituent un résumé des effets d'une instruction complexe et de ses sous-instructions ; notons que les effets sur les éléments de tableaux sont remplacés par des effets sur le tableau lui-même ;

– les *effets résumés* d'un module sont les effets cumulés de son corps dans lesquels on a éliminé les variables locales ; les effets résumés sont utilisés au niveau des sites d'appel pour obtenir leurs effets propres après traduction des paramètres formels en paramètres réels.

3.2.2 Transformeurs

Les transformeurs donnent les relations affines qui existent entre les valeurs des variables scalaires entières dans les états-mémoires suivant et précédant l'exécution d'une instruction ou l'appel d'une procédure. La figure 3.2 donne avant chaque instruction son transformeur sous la forme `T(argument) {prédicat}` ; l'argument du transformeur donne la liste des variables modifiées (effets propres en écriture sur les variables scalaires entières), et le prédicat les relations affines non triviales [1] entre ces variables. On remarquera dans la figure 3.2 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. Plusieurs options de calcul des transformers sont disponibles dans PIPS. Elles correspondent à des degrés d'analyse plus ou moins poussés, et sont présentées dans le tableau ci-dessous :

calcul des invariants de boucle	propagation interprocédurale	
	non	oui
non	`tf` [2]	`tfinter`
oui	`tffull`	`tfinterfull`

En l'absence de propagation interprocédurale, le transformeur associé à un appel de procédure n'apporte pas plus d'information que les effets correspondants. Dans l'exemple ci-dessous, la procédure `INC1(I)` incrémente la variable `K` de 1, mais aucune équation n'apparaît dans la partie prédicat du transformer.

```
C T(K) {}                  SUBROUTINE INC1(I)
  CALL INC1(K)             I = I + 1
                           END
```

1. Une relation triviale est de la forme `I == I#INIT`.
2. *tf* est une abréviation de *transformeur*.

```
C  T(K) {K == 3}
  K = 3
C  T(K) {}
  K = FOO(N)
C  T(L) {L==M+N}
  L = M + N
C  T(M) {0<=M, M<=3}
  M = MOD(N, 4)
```

```
C  P() {}
  K = 3
C  P(K) {K==3}
  K = FOO()
C  P(K) {}
  L = M + N
C  P(K,L) {L==M+N}
  M = MOD(N, 4)
C  P(K,L,M) {0<=M, M<=3, L==M#INIT+N}
```

FIG. 3.2 - *Transformeurs et préconditions*

Au contraire, si la propagation est interprocédurale, le transformer résumé de la procédure appelée (pour notre exemple `T(I) {I==I#INIT+1}`) est réutilisé au niveau du site d'appel après traduction des paramètres formels (`I`) en paramètres réels (`K`) :

```
C T(K) {K==K#INIT+1}
  CALL INC1(K)
```

Si le calcul de l'invariant de boucle n'est pas effectué, le transformer associé à la boucle est réduit aux effets sur l'indice de boucle et sur les variables scalaires éventuellement modifiées par celle-ci :

```
C T(I,K) {}
  DO I = 1, N
     K = K + 1
     ...
  ENDDO
```

Dans le cas où l'invariant de boucle est calculé, et si son expression est affine, on récupère cette dernière dans le prédicat du transformeur :

```
C T(I,K) {K+1==K#INIT+I}
  DO I = 1, N
     K = K + 1
     ...
  ENDDO
```

La précision des analyses ultérieures s'appuyant sur les transformeurs dépendra bien évidemment de la précision choisie pour le calcul de ceux-ci.

3.2.3 Préconditions

Les préconditions sont des prédicats sur les variables scalaires entières, vrais avant exécution de l'instruction. L'exemple de la figure 3.2 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. Ce dernier point apparaît nettement dans la précondition de la dernière instruction : la valeur de `L` dépend de celle de `M#INIT` qui est la valeur de `M` en entrée de la procédure, et non de la valeur courante de `M` qui vient d'être modifiée de manière non linéaire.

Les préconditions sont propagées de manière descendante sur le graphe de contrôle. Elles sont calculées à l'aide des transformeurs, 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 $P_2 = T_1(P_1)$. De même que les transformeurs, elles peuvent, à la demande de l'utilisateur de PIPS être ou non propagées interprocéduralement. La précondition d'un module est en effet l'enveloppe convexe de toutes les préconditions de ses sites d'appels, après traduction des paramètres d'appel en paramètres formels. Si il n'y a pas propagation interprocédurale, la précondition du module a un prédicat vide signifiant que l'on ne dispose d'aucune information sur les valeurs des paramètres d'entrée.

3.3 Transformations de programme

Certaines transformations de code permettent d'éliminer des dépendances intrinsèques du programme. Plusieurs sont déjà implantées dans PIPS ou sont en phase de mise au point :

- le *déroulage de boucle* remplace le corps de la boucle par plusieurs itération successives de celle-ci ;
- l'*inversion de boucles* permet, dans un nid de boucles, d'échanger l'ordre des boucles ;
- *strip mining* est une transformation non-linéaire qui associe à une boucle donnée un nid de deux boucles ;
- la *distribution de boucles* consiste à remplacer une boucle dont le corps est constitué de plusieurs instructions par plusieurs boucles à une seule instruction ;
- l'*évaluation partielle* peut rendre linéaires certaines expressions qui ne l'étaient pas en remplaçant certaines variables par leurs valeurs ;
- l'*élimination de code mort* ;
- la *privatisation de scalaires* qui déclare certaines variables locales à un processeur de manière à éliminer les dépendances entre itérations ;
- la *détection des réductions*, pour les remplacer ensuite par des instructions vectorielles ou des appels à des procédures optimisées.

L'avantage de l'environnement interactif de PIPS est que l'utilisateur peut choisir le type de transformation le mieux adapté à l'architecture de sa machine, *strip mining* par exemple pour une architecture vectorielle.

Chapitre 4

Régions : définition et opérateurs

4.1 Définition d'une région

Une région est définie par Rémi Triolet dans [60] comme étant un ensemble d'éléments de tableau décrit par des équations et inéquation affines formant un polyèdre convexe. Par exemple, la région :

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

contient les éléments `A(1,1)`, `A(2,2)` et `A(3,3)`, `PHI1` et `PHI2` représentant respectivement les éléments de la première et la deuxième dimension de `A`. Une région est généralement associée à une instruction ou a un fragment de code, dont elle modélise les effets sur le tableau considéré. Elle est donc exprimée dans l'état-mémoire précédant son exécution.

Définition 4.1 *Une région R établit une relation $\widehat{R}$ entre les variables $\varphi = (\varphi_1, \ldots, \varphi_n)$ représentant les indices des différentes dimensions du tableau, et tout état-mémoire σ précédant l'exécution du fragment de code auquel elle correspond.*

On note $R(\sigma)$ la fonction qui, à un état-mémoire σ, associe l'ensemble des φ définis par la région :

$$\forall\, \sigma, \quad R(\sigma) = \{\varphi \mid \widehat{R}(\varphi, \sigma)\}$$

Notation 4.1 *Soit une instruction S (au sens général du terme) définissant ou utilisant des éléments d'un tableau.*

On note $action_S(\varphi, \sigma)$ la relation entre les variables φ du tableau considéré, et tout état-mémoire σ précédant l'exécution de l'instruction S, telle que $\{\varphi \mid action_S(\varphi, \sigma)\}$ est l'ensemble des éléments de tableau lus ou écrits par S.

Définition 4.2 *Une région R correspondant à une instruction S étant une approximation des éléments de tableau accédés lors de l'exécution de cette instruction, la relation $\widehat{R}$ associée à la région R vérifie :*

$$\forall\, \sigma \;\; action(\varphi, S, \sigma) \Longrightarrow \widehat{R}(\varphi, \sigma)$$

Les régions étant à l'origine utilisées au sein du projet PIPS dans le cadre du calcul des dépendances, pour exprimer les effets des instructions et des procédures [49], deux autres notions ont été introduites :

- l'*action* réalisée sur les éléments de la région :

  `READ (R)` pour une utilisation (use) ;

`WRITE (W)` pour une définition (def) ;

– l'*approximation* de la région :

`MAY` si les éléments de la région sont seulement potentiellement référencés sur au moins un chemin du graphe de contrôle du fragment de code considéré, et pour tout état-mémoire possible avant exécution de ce fragment de code.

`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 possible avant exécution de ce fragment de code ;

Définition 4.3 *La relation $\widehat{R}$ associée à une région R d'approximation `MUST` et correspondant à une instruction S, vérifie :*

$$\forall \sigma \ \ \widehat{R}(\varphi, \sigma) \Longrightarrow action(\varphi, S, \sigma)$$

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)`. Et les régions correspondant aux instructions suivantes :

```
A(I) = 0
M = A(I-1) + A(I+1)
```

sont respectivement :

```
<A(PHI1)-W-MUST-{PHI1==I}>
```

et

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

La première région est une région `MUST` car elle ne contient que l'élément `A(I)` qui est celui effectivement écrit lors de l'exécution de l'instruction. La deuxième région est une région `MAY` car la fermeture convexe des éléments effectivement lus contient l'élément `A(I)` alors que celui-ci n'est pas concerné par l'action `READ`.

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 étant toujours vraies dans tout état-mémoire σ précédant l'exécution du fragment de programme considéré. Elles 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.

Les régions étant des ensembles d'éléments de tableaux, on pourrait définir tous les opérateurs habituels sur les ensembles. Seuls trois d'entre eux nous intéressent dans le cadre du calcul des régions `MUST` : l'*union*, l'*intersection* et la *différence*. A ces opérateurs, nous ajouterons celui de *projection* selon une variable. Les ensembles que représentent les régions étant décrits par des polyèdres convexes en nombres entiers, nous allons dans un premier temps définir ces derniers ainsi que les principaux opérateurs permettant de les manipuler. Nous utiliserons ensuite ceux-ci pour définir les opérations sur les régions.

4.2 Polyèdres convexes en nombres entiers

4.2.1 Définitions

Définition 4.4 *On appelle* polyèdre convexe en nombres entiers *un ensemble de points $P \subseteq Z^n$ défini par :*

$$P = \{x \in Z^n \mid A.x \leq b\}$$

où A est une matrice de $Z^m \times Z^n$ et b un vecteur de Z^m.

Un polyèdre P est donc caractérisé par un système d'inéquations linéaires. Ces inéquations définissent des demi-espaces affines dont P est l'intersection. Un polyèdre peut également être représenté par un système générateur [5, 20].

Définition 4.5 *Soit un polyèdre P. Son* système générateur *est constitué de trois ensembles : $S_P = \{s_1, \ldots, s_\sigma\}$, un ensemble de sommets, $R_P = \{r_1, \ldots, r_\rho\}$, un ensemble de rayons, et $D_P = \{d_1, \ldots, d_\delta\}$, un ensemble de droites.*

- *Un* sommet *de P est un point $s \in P$ qui n'est pas combinaison linéaire d'autres points de P :*

$$\left\{ \begin{array}{l} s = \sum_{i=1}^{p} \lambda_i x_i \\ \forall\, i \in [1,p] \;\; x_i \in P \\ \forall\, i \in [1,p] \;\; \lambda_i \geq 0 \\ \forall\, i,j \in [1,p] \;\; i \neq j \Longrightarrow x_i \neq x_j \\ \sum_{i=1}^{p} \lambda_i = 1 \end{array} \right. \Longrightarrow \; (\; \forall\, i \in [1,p] \quad \lambda_i = 0 \quad ou \quad s = x_i \;)$$

- *Un* rayon *de P est un vecteur $r \in R^n$ tel qu'il existe une demi-droite de vecteur directeur r et entièrement contenue dans P :*

$$\forall\, x \in P \quad \forall\, \mu \in Z^+ \quad x + \mu r \in P$$

- *Une* droite *de P est un vecteur $d \in R^n$ tel que d et $-d$ sont des rayons de P :*

$$\forall\, x \in P \quad \forall\, \nu \in Z \quad x + \nu r \in P$$

P est alors défini par :

$$x \in P \iff \left\{ \begin{array}{l} \exists\, \lambda_1, \ldots, \lambda_\sigma \in [0,1] \\ \exists\, \mu_1, \ldots, \mu_\rho \in Z^+ \\ \exists\, \nu_1, \ldots, \nu_\delta \in Z \end{array} \right. \quad tels\ que \left\{ \begin{array}{l} \sum_{i=1}^{\sigma} \lambda_i = 1 \\ x = \sum_{i=1}^{\sigma} \lambda_i s_i + \sum_{i=1}^{\rho} \mu_i r_i + \sum_{i=1}^{\delta} \nu_i d_i \end{array} \right.$$

Des algorithmes [30, 25] permettent de passer d'une représentation à l'autre. Le fait de disposer de deux représentations différentes n'est pas un simple gadget mathématique. Certaines opérations sont en effet plus faciles à réaliser en manipulant les polyèdres sous forme de systèmes d'inéquations affines (intersection par exemple), tandis que d'autres sont moins complexes lorsqu'elles sont réalisées à partir des systèmes générateurs.

4.2.2 Opérations

Les opérations sur les polyèdres convexes en nombres entiers sont presque les mêmes que celles habituellement définies pour les ensembles. Nous allons définir l'intersection, l'enveloppe convexe (qui remplace l'opérateur d'union), la différence, la projection, et les tests d'inclusion, d'égalité et d'égalité de l'union et de l'enveloppe convexe. D'autres opérateurs existent, et sont utilisés dans l'implantation du calcul des régions, mais leur description n'est pas utile à la compréhension des chapitres qui suivent, et nous ne les décrirons donc pas.

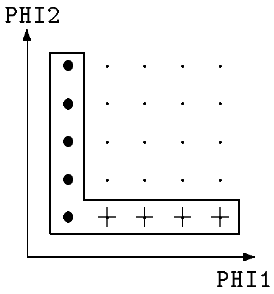


FIG. 4.1 - *Union de deux polyèdres.*

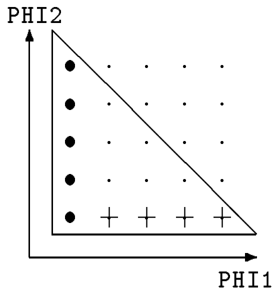


FIG. 4.2 - *Enveloppe convexe de deux polyèdres.*

L'*intersection* de deux polyèdres ne pose pas de problème particulier. En effet, un polyèdre est défini comme étant l'intersection de demi-plans affines. Le polyèdre résultat est donc trivialement l'intersection des demi-plans affines provenant des deux polyèdres initiaux. La seule contrainte est que ces deux polyèdres soient définis sur le même domaine, i.e. Z^n (et non Z^n pour l'un et Z^m pour l'autre) et la même base. Mathématiquement, on obtient :

$$\left\{ \begin{array}{l} P_1 = \{x \mid A_1.x \leq b_1\} \\ P_2 = \{x \mid A_2.x \leq b_2\} \end{array} \right. \Longrightarrow P_1 \cap P_2 = \{x \mid \left[\begin{array}{c} A_1 \\ A_2 \end{array} \right].x = \left[\begin{array}{c} b_1 \\ b_2 \end{array} \right]\}$$

avec $A_1 \in Z^{m_1} \times Z^n$, $A_2 \in Z^{m_2} \times Z^n$, $b_1 \in Z^{m_1}$ et $b_2 \in Z^{m_2}$.

L'*union* de deux polyèdres n'est pas, dans le cas général, un polyèdre. L'exemple de la figure 4.1 donne une illustration de cette propriété : les deux polyèdres initiaux sont représentés par des $\bullet$ et des $+$, et leur union est manifestement concave. L'opérateur utilisé pour fusionner deux polyèdres est donc leur *enveloppe convexe*. C'est une approximation de leur union : c'est le plus petit polyèdre convexe contenant les deux polyèdres initiaux. Il peut donc contenir des points qui ne faisaient pas partie des ensembles de départ (voir la figure 4.2). L'enveloppe convexe de deux polyèdres est plus facile à calculer lorsque ceux-ci sont sous la forme de systèmes générateurs dont il suffit alors de calculer l'union. Si P_1 et P_2 sont deux polyèdres de systèmes générateurs respectifs :

$$\left\{ \begin{array}{l} S_{P_1} = \{s_1^1, \ldots, s_{\sigma_1}^1\} \\ R_{P_1} = \{r_1^1, \ldots, r_{\rho_1}^1\} \\ D_{P_1} = \{d_1^1, \ldots, d_{\delta_1}^1\} \end{array} \right. \text{ et } \left\{ \begin{array}{l} S_{P_2} = \{s_1^2, \ldots, s_{\sigma_2}^2\} \\ R_{P_2} = \{r_1^2, \ldots, r_{\rho_2}^2\} \\ D_{P_2} = \{d_1^2, \ldots, d_{\delta_2}^2\} \end{array} \right.$$

alors $env\_conv(P_1, P_2)$ est défini par le système générateur :

$$\left\{ \begin{array}{l} S = \{s_1^1, \ldots, s_{\sigma_1}^1, s_1^2, \ldots, s_{\sigma_2}^2\} \\ R = \{r_1^1, \ldots, r_{\rho_1}^1, r_1^2, \ldots, r_{\rho_2}^2\} \\ D = \{d_1^1, \ldots, d_{\delta_1}^1, d_1^2, \ldots, d_{\delta_2}^2\} \end{array} \right.$$

La *différence* de deux polyèdres convexes n'est généralement pas un polyèdre convexe. Pour s'en convaincre, il suffit de regarder l'exemple de la figure 4.3. les deux polyèdres initiaux sont représentés par des $\bullet$ et des $\odot$, et leur différence ($\bullet - \odot$) est manifestement concave. Corinne Ancourt a donné dans sa thèse [5] un algorithme permettant de calculer cette différence sous forme d'une union de polyèdres convexes. Pour des raisons de rapidité d'implantation, nous avons préféré utiliser un algorithme basé sur les structures de

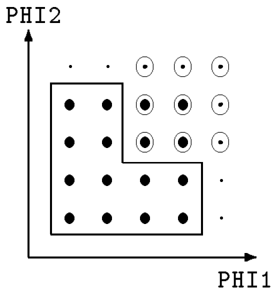


FIG. 4.3 - *Différence de deux polyèdres.*

données introduites par Arnauld Leservot dans [40], à savoir les *disjonctions* et les *chemins*. Il a également défini un algorithme de conversion d'un chemin en une disjonction (*pa_path_to_few_disjunct*) et des algorithmes permettant de tester la faisabilité des chemins et des disjonctions, c'est-à-dire l'existence d'une solution.

Définition 4.6 *Une* disjonction D *est une union de polyèdres :*

$$D = \bigcup_{i=1}^{n} P_i$$

Définition 4.7 *Un* chemin C *est l'intersection d'un polyèdre avec une suite de complémentaires de polyèdres :*

$$C = P_0 \bigcap_{i=1}^{n} \neg P_i$$

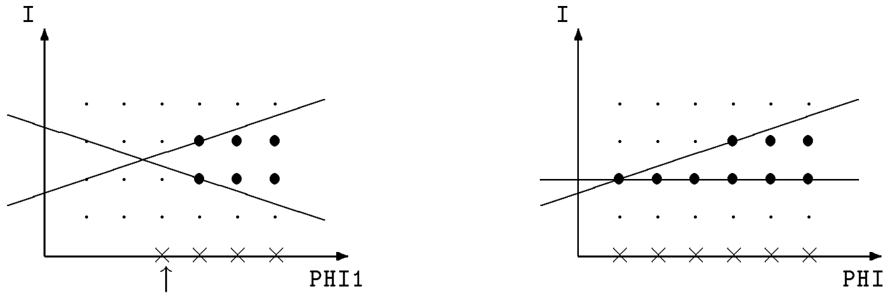
Sachant que $A - B = A \cap \neg B$, on peut alors définir un algorithme de calcul de la différence de deux polyèdres :

Algorithme 4.1 *Soient P_1 et P_2 deux polyèdres convexes. Leur intersection peut être otenue sous forme d'une union de polyèdres à l'aide de l'algorithme suivant :*

$$chemin = P_1 \cap \neg P_2$$
$$disjonction = pa\_path\_to\_few\_disjunct(chemin)$$

Si l'on désire obtenir le résultat sous forme d'un unique polyèdre convexe, il suffit de calculer l'enveloppe convexe de l'union de polyèdres ainsi générée. Mais elle pourra contenir des points n'appartenant pas à la différence.

La *projection* d'un polyèdre selon une variable se calcule en éliminant celle-ci du système de contrainte qui définit le polyèdre. L'algorithme de Fourier-Motzkin [26] permet de réaliser cette élimination. Cet algorithme permet en réalité de calculer l'enveloppe convexe de l'ensemble des projections des points entiers appartenant au polyèdre de départ [5]. Le polyèdre résultat peut donc contenir des points qui ne sont pas la projection d'un point du polyèdre initial. On peut s'en convaincre à l'aide de la figure 4.4. Dans le cas a), un des points de la projection ne correspond à aucun des points du polyèdre constitué des •. Par contre, la projection du polyèdre du cas b) est exacte. Des conditions suffisantes pour que la projection soit exacte ont été données dans [6, 50].



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.*

FIG. 4.4 - *Projection d'un polyèdre.*

Les tests d'*inclusion* d'un polyèdre dans un autre, d'*égalité* de deux polyèdres, et d'*égalité de l'union et de l'enveloppe convexe* de deux polyèdres ont pu être implantés rapidement grâce aux structures de données et fonctions développées par Arnauld Leservot (voir page 33).

Algorithme 4.2 *Soient P_1 et P_2 deux polyèdres convexes en nombre entier. Soit C un chemin tel que $C = P_1 \cap \neg P_2$.*

Si C est non faisable, alors $P_1 \subseteq P_2$, sinon $P_1 \not\subseteq P_2$.

Démonstration : Il suffit de constater que $P_1 \subseteq P_2 \iff P_1 \cap \neg P_2 = \emptyset$, et que, par définition, C est non faisable si et seulement si $C = \emptyset$. □

Le test d'égalité de deux polyèdres se déduit naturellement du test d'inclusion précédent.

Algorithme 4.3 *Soient P_1 et P_2 deux polyèdres convexes en nombre entier.*

$$P_1 = P_2 \iff P_1 \subseteq P_2 \wedge P_2 \subseteq P_1$$

Algorithme 4.4 *Soient P_1 et P_2 deux polyèdres convexes en nombre entier. Soit P_c leur enveloppe convexe. Soit C un chemin tel que $C = P_c \cap \neg P_1 \cap \neg P_2$.*

Si C est non faisable, alors $P_c = P_1 \cup P_2$, sinon $P_c \neq P_1 \cup P_2$.

Démonstration : $C = P_c \cap \neg P_1 \cap \neg P_2 \iff C = P_c \cap \neg(P_1 \cup P_2)$.

Si C est non faisable, alors $P_c \subseteq (P_1 \cup P_2)$. Or, par définition de l'enveloppe convexe, $(P_1 \cup P_2) \subseteq P_c$. On a donc bien $P_c = P_1 \cup P_2$.

Si C est faisable, alors $P_c \not\subseteq (P_1 \cup P_2)$ et donc $P_c \neq (P_1 \cup P_2)$. □

4.3 Opérations sur les régions

Les opérations sur les régions ne peuvent se réduirent aux opérations de même nom sur les polyèdres décrivant les ensembles qu'elles représentent. Nous venons en effet de voir que plusieurs opérations sur les polyèdres (enveloppe convexe, différence, projection) introduisaient des points n'appartenant pas à la solution recherchée. L'approximation (`MAY` ou `MUST`) de la région résultante ne peut, dans ces conditions, être uniquement fonction des approximations des régions initiales. Parfois, elle devra également dépendre de l'exactitude de la solution trouvée, lorsque celle-ci pourra être démontrée. De plus, les polyèdres peuvent prendre deux valeurs remarquables :

- `sc_rn` est un polyèdre représentant l'espace entier, soit Z^n ;
- `sc_empty` est un polyèdre ne contenant aucun point de l'espace considéré ; il est généralement décrit par une unique équation sans solution, {`0==1`}.

Nous avons attribué aux régions dont les prédicats possèdent ces valeurs une sémantique particulière :

- `sc_rn` représente l'ensemble du tableau, et non l'ensemble de l'espace mémoire ; les bornes des tableaux n'étant pas prises en compte, ce type de région est assez fréquent. Pour pouvoir le traiter de façon satisfaisante, on suppose que le programme est correct et donc que les accès aux tableaux sont cohérents vis-à-vis de leur définition.
- `sc_empty` représente un ensemble ne contenant aucun élément de tableau.

Il conviendra donc de définir le comportement des opérateurs lorsque l'un des opérandes prend l'une de ces valeurs. *A priori*, une région `sc_empty MAY` n'a pas de signification ; de plus, comme elle ne contient aucun élément de tableau, toute région `sc_empty` peut être omise. Nous considérerons tout de même ces valeurs pour la définition des opérateurs, dans un souci d'exhaustivité. Enfin, les régions sont réunies sous forme de listes. Une liste ne peut contenir qu'une seule région relative à un tableau donné et pour une action (`READ` ou `WRITE`) donnée. Par exemple, la liste de régions suivante ne peut pas apparaître :

```
C  <A(PHI1)-W-MUST-{PHI1==1}>
C  <A(PHI1)-W-MUST-{PHI1==2}>
C  <B(PHI1,PHI2)-W-MAY-{}>
...
```

Elle serait représentée par la liste :

```
C  <A(PHI1)-W-MUST-{1<=PHI1,PHI1<=2}>
C  <B(PHI1,PHI2)-W-MAY-{}>
...
```

Les opérateurs manipulant ces listes seront décrits en même temps que leurs homologues manipulant les régions *individuelles*.

Dans la suite nous désignerons souvent le prédicat de la région par le mot *région* lorsqu'aucune ambiguïté ne sera possible.

```
C <A(PHI1)-R-MAY-{I<=PHI1+1, PHI1<=I}
   M = A(I-1) + A(I)
```

FIG. 4.5 - *Enveloppe convexe de deux régions.*

4.3.1 Fusion

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 4.1). 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. 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`.

L'approche la plus prudente, et la plus efficace en terme de temps d'exécution, consiste donc à donner une approximation `MAY` à toute région résultant de l'enveloppe convexe de deux régions. L'inconvénient de cette méthode est son manque de précision. Dans l'exemple de la figure 4.5, la région représente exactement les éléments de `A` qui sont lus par l'instruction ; mais son approximation est `MAY` car elle est le résultat de l'enveloppe convexe de deux régions, `<A(PHI1)-R-MUST-{PHI1==I-1}>` et `<A(PHI1)-R-MUST-{PHI1==I}>`. La préservation de l'attribut `MUST` étant un paramètre critique dans le cadre du calcul de régions exactes, il est nécessaire d'utiliser une approche plus précise. Le maximum de précision est atteint lorsque l'on ne calcule jamais de région résumée. Une liste de régions peut alors contenir plusieurs régions concernant un tableau pour une action donnée. Cela se fait au détriment de l'encombrement mémoire et de la rapidité de toutes les opérations manipulant les régions. Deux approches intermédiaires peuvent être envisagées :

1. la première examine les cas dans lesquels une approximation `MUST` pourrait être conservée si l'enveloppe convexe était exacte, ou si les régions initiales possédaient certaines caratéristiques ;

2. la deuxième, plus coûteuse en mémoire, n'effectue la fusion des régions que lorsque l'union est égale à l'enveloppe convexe ; on peut aussi ajouter la condition que l'une au moins des régions conservées ait une approximation `MUST`; cela réduit vraisemblablement l'espace mémoire nécessaire de manière non négligeable.

Nous ne développerons ici que la première approche. Seule l'expérience pourra montrer l'utilité ou l'inutilité de la deuxième.

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`) ou d'une portion de code non structurée *unstructured*. 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 composée `IF`. De même pour les nœuds composant un *unstructured*.

Le tableau 4.1 donne l'approximation de la région résultante en fonction des valeurs que les régions initiales peuvent prendre et de leurs approximations respectives. Dans le cas

région 1 (R_1)	région 2 (R_2)	$R_1 \cup_{must} R_2$	$R_1 \cup_{may} R_2$
= sc_rn	= sc_rn	= sc_rn	= sc_rn
must	must	must	must
must	may	must	may
may	may	may	may
= sc_rn	≠ sc_rn	= sc_rn	= sc_rn
must	must	must	may
must	may	must	may
may	must	may	may
may	may	may	may
= sc_empty	= sc_empty	= sc_empty (ou rien)	= sc_empty (ou rien)
must	must	must	must
must	may	must	may
may	may	may	may
= sc_empty	≠ sc_empty	= R_2	= R_2
must	must	must	must
must	may	may	may
may	must	must	may
may	may	may	may
dans les autres cas :		= $env\_conv(R_1, R_2)$	
must	must	must, si env. conv. exacte may, sinon	must, si $R_1 = R_2$ may, sinon
must	may	must, si $R_2 \subseteq R_1$ may, sinon	may
may	may	may	may

TAB. 4.1 - *Enveloppe convexe de deux régions.*

liste 1 (l_1)	liste 2 (l_2)	$l_1 \cup_{must} l_2$	$l_1 \cup_{may} l_2$
(nil)	(nil)	(nil)	(nil)
(R)	(nil)	(R)	(R_{may})
(R_1)	(R_2)	($R_1 \cup_{must} R_2$)	($R_1 \cup_{may} R_2$)

TAB. 4.2 - *Union de deux listes de régions.*

général, l'approximation de la région résultante est `MAY`, sauf si l'on arrive à démontrer qu'il existe une relation particulière entre les deux régions initiales. Par exemple, si ces dernières sont des régions `MUST`, et que l'on cherche leur union *must*, l'approximation de la région résultante ne peut être `MUST` que si l'enveloppe convexe est égale à l'union ; dans le cas d'une union *may*, il faut en plus que les deux branches du `IF` (par exemple) accèdent exactement aux même éléments pour que l'approximation `MUST` soit conservée, c'est-à-dire que les régions initiales contiennent les mêmes éléments.

Il s'agit maintenant de calculer l'union de deux listes de régions. Une liste contient au plus une région par tableau et par type d'accès. Il suffit donc de définir l'union de deux listes de régions contenant chacune au plus une région, accèdant au même tableau avec la même action. Le tableau 4.2 définit cet opérateur en fonction du contenu des listes : (nil) représente la liste vide, R_1 et R_2 sont des régions relatives à un même tableau pour une même action. Remarquons que $(R) \cup_{may} (nil) = (R_{may})$, ce qui signifie : *la région R avec une approximation `MAY`.* Cette approximation provient du fait que R n'est référencée que dans l'une des branches d'un `IF` (par exemple), et n'est donc que potentiellement référencée au cours d'une exécution du programme.

Remarque : L'opérateur $\cup_{must}$ ainsi défini n'est pas associatif, c'est-à-dire que

$$(R_1 \cup_{must} R_2) \cup_{must} R_3 \neq R_1 \cup_{must} (R_2 \cup_{must} R_3)$$

Par exemple, la région correpondant à l'expression `A(I-1)+A(I+1)+A(I)` sera :

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

Le `MAY` provient du fait que l'enveloppe convexe des deux polyèdres engendrés par les premiers éléments (`A(I-1)` et `A(I+1)`) n'est pas exacte. La région correspondant à l'expression `A(I-1)+A(I)+A(I+1)`, sera au contraire :

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

4.3.2 Intersection

L'intersection de deux régions correspondant au même tableau, et pour des actions données mais pas forcément identiques, est beaucoup plus simple à calculer que leur union puisque l'intersection de deux polyèdres convexes est un polyèdre convexe. Le prédicat de la région résultante sera donc l'intersection des deux prédicats des régions initiales, à une élimination de contraintes redondantes près. L'opérateur est défini par le tableau 4.3 pour les régions, et par le tableau 4.4 pour les listes de régions.

Contrairement à l'union, il est inutile de disposer de deux opérateurs d'intersection. Le seul cas qui pourrait différer est celui dans lequel R_1 est une régions `MUST` tandis que R_2 est une région `MAY`. Cette dernière contient éventuellement des points qui ne font pas partie de la région effectivement référencé. Ces points peuvent appartenir à l'intersection calculée, mais ne pas faire partie de l'intersection des régions réellement accédées lors de l'exécution. L'approximation de la région résultante doit donc être `MAY` quelques soient les chemins du graphe de contrôle auxquels correspondent les régions R_1 et R_2.

4.3.3 Différence

Nous avons vu page 32 que la différence de deux polyèdres n'était pas, dans le cas général, un polyèdre convexe, et qu'on pouvait la calculer sous forme d'une union de polyèdres convexes. Une région devant, par définition, être représentée par un polyèdre convexe, le

région 1 (R_1)	région 2 (R_2)	intersection
`sc_empty`	may ou must	`sc_empty` must (ou rien)
must	must	must
must	may	may
may	may	may

TAB. 4.3 - *Intersection de deux régions.*

liste 1 (l_1)	liste 2 (R_2)	intersection must
(nil)	(nil)	(nil)
(R)	(nil)	(nil)
(R_1)	(R_2)	$R_1 \cap R_2$

TAB. 4.4 - *Intersection de deux listes de régions.*

prédicat de la différence de deux régions devra donc être une approximation convexe de la différence réelle. En réalité, nous aurons besoin dans la suite de ce rapport de deux opérateurs différents :

- l'opérateur $-_{sup}$ donne une sur-estimation convexe de la différence réelle : il est défini comme étant l'enveloppe convexe des polyèdres résultant de la différence des deux prédicats initiaux ;
- l'opérateur $-_{inf}$ donne une sous-estimation convexe de l'ensemble des éléments qui appartiennent certainement à la différence réelle ; si la différence des deux régions est exacte, le résultat est cette différence ; si elle est inexacte, le résultat est la région vide ; ce n'est pas l'ensemble convexe le plus proche de la différence réelle que l'on puisse calculer, mais cela nous suffira pour l'usage que nous voulons en faire.

L'approximation de la région résultat est donc, dans les deux cas, fonction de l'exactitude de la différence, ainsi que des approximations des régions initiales. Les tableaux 4.5 et 4.6 définissent ces deux opérateurs de différence pour deux régions ($R_1 - R_2$) et pour deux listes de régions ($l_1 - l_2$). Il est important de noter qu'ils ne sont pas commutatifs.

Pour l'opérateur $-_{sup}$, le cas où R_1 est une région `MUST` et où R_2 est une région `MAY` mérite quelques explications. R_2 comporte éventuellement des éléments qui ne sont pas réellement référencés puisque son approximation est `MAY`. Si $R_1 - R_2 \neq \emptyset$, $(R_1 - R_2)_{calculé}$ ne comprendra pas ces éléments et sera donc un sous-ensemble de $(R_1 - R_2)_{réel}$, alors que l'on cherche une solution contenant $(R_1 - R_2)_{réel}$. Pour être sûr de ne perdre aucun élément de R_1, on prendra donc R_1 comme résultat, mais avec une approximation `MAY` car $(R_1 - R_2)_{réel} \subseteq R_1$. Au contraire, si $R_1 \cap R_2 = \emptyset$, $R_1 - R_2 = R_1$ représente la solution exacte, et l'approximation `MUST` de R_1 peut être préservée.

4.3.4 Projection d'une région selon une variable

La projection d'une région selon une variable consiste à éliminer cette variable de son prédicat. Nous avons vu page 33 que cette opération pouvait introduire dans l'ensemble résultat des points n'appartenant pas à la projection réelle. Si l'ajout de ces points ne modifie pas l'ensemble des `PHIs` représenté par le polyèdre, l'approximation des régions `MUST` peut être conservée. Dans le cas contraire, les régions ne représentent plus les mêmes sous-ensembles de tableaux, et leur approximation devient obligatoirement `MAY`. Dans la suite

<table>
<tr><th>région 1 (R_1)</th><th>région 2 (R_2)</th><th>$R_1 -_{sup} R_2$</th><th>$R_1 -_{inf} R_2$</th></tr>
<tr><td colspan="4">cas particuliers :</td></tr>
<tr><td>sc_rn, must</td><td>sc_rn, must</td><td colspan="2">sc_empty, must (ou rien)</td></tr>
<tr><td>sc_rn, may</td><td>sc_rn, must</td><td colspan="2">sc_empty, must (ou rien)</td></tr>
<tr><td>sc_rn</td><td>sc_rn, may</td><td>sc_rn, may</td><td>sc_empty, must (ou rien)</td></tr>
<tr><td>sc_empty</td><td>sc_empty</td><td colspan="2">sc_empty, must (ou rien)</td></tr>
<tr><td>sc_empty</td><td>$\neq$ sc_empty</td><td colspan="2">sc_empty, must (ou rien)</td></tr>
<tr><td colspan="4">cas général :</td></tr>
<tr><td>must</td><td>must</td><td>$R_1 - R_2$, must si diff. exacte
may sinon</td><td rowspan="2">si $R_1 - R_2$ exacte
alors $R_1 - R_2$, must
sinon sc_empty, must
(ou rien)</td></tr>
<tr><td>must</td><td>may</td><td>R_1, must si $R_1 \cap R_2 = \emptyset$
may sinon</td></tr>
<tr><td>may</td><td>must</td><td>$R_1 - R_2$, may</td><td rowspan="2">si $R_1 - R_2$ exacte
alors $R_1 - R_2$, may
sinon sc_empty, must
(ou rien)</td></tr>
<tr><td>may</td><td>may</td><td>R_1, may</td></tr>
</table>

TAB. 4.5 - *Différence de deux régions.*

liste 1 (l_1)	liste 2 (l_2)	$l_1 -_{sup/inf} l_2$
(nil)	(nil)	(nil)
(R)	(nil)	(R)
(nil)	(R)	(nil)
(R_1)	(R_2)	($R_1 -_{sup/inf} R_2$)

TAB. 4.6 - *Différence de deux listes de régions.*

de ce rapport, nous appellerons *projection exacte* l'élimination d'une variable préservant l'ensemble des `PHIs`.

Définition 4.8 *La projection d'une région $R(\sigma)$ selon une variable k est exacte si et seulement si $proj_k(R(\sigma)) = R(\sigma)$.*

Reste à déterminer un critère permettant de savoir si l'ensemble des `PHIs` (dans l'état-mémoire courant) a été modifié par l'élimination de variable. Les conditions présentées dans [6, 50] ne sont pas applicables ici, car elles permettent de distinguer certaines variables des autres. Pour illustrer cette dernière affirmation, prenons l'exemple de la région suivante et projetons-la selon la variable `K` :

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

Dans l'état-mémoire dans lequel la région est exprimée, on sait que `PHI1` vaut exactement `K`, et que, de surcroit, la valeur de `K` est supérieure à 1. Le prédicat de la région projetée est `1<=PHI1`, ce qui signifie que `PHI1` prend (ou plutôt peut prendre) toutes les valeurs entre 1 et l'infini (ou du moins sa borne supérieure). On voit que la sémantique n'est pas la même. Dans un cas, `PHI1` prend une valeur unique, inconnue mais déterminée avec exactitude par celle de `K`. Dans l'autre, la valeur de `PHI1` n'est plus définie avec exactitude en fonction des variables du programme. L'approximation de la région résultant de la projection doit donc être `MAY`.

Les polyèdres définissant les régions comportent plusieurs types d'égalités et d'inégalités. Certaines d'entre elles relient directement les variables `PHIs` à des variables du programmes, comme `PHI1==I+1` ou `PHI1<=K+I` par exemple. D'autres donnent les relations qui existent entre les variables du programme, et proviennent des préconditions. Ces dernières relations introduisent donc dans le polyèdre des variables qui sont indirectement liées aux variables `PHIs`, comme dans `{PHI1==K,K=L+M}` où `PHI1` est indirectement lié à `L` et `M`. Mais certaines variables introduites peuvent ne pas être liées aux variables `PHIs`, même indirectement, comme c'est le cas pour `J` dans le polyèdre `{PHI1==K, 1<=J, J<=N}`.

Un premier critère de projection exacte découle directement de cette dernière remarque :

Définition 4.9 *Soit un polyèdre P, contenant un ensemble de contraintes C restreignant les valeurs des variables appartenant à un ensemble V. On définit la relation* est liée à *entre deux variables de la manière suivante :*

$$x \in V \text{ est liée à } y \in V \iff \left| \begin{array}{l} \exists\, c \in C \mid x \in c \text{ et } y \in c \\ \text{ou } \exists\, c \in C,\ \exists\, z \in V \mid x \in c \text{ et } z \text{ est liée à } y \end{array} \right.$$

Définition 4.10 *Soit une région de polyèdre $\widehat{R}(\varphi, \sigma)$. On appelle* variable liée aux variables *φ une variable appartenant à la fermeture transitive à partir des φ de la relation* est liée à *définie précédemment.*

Théorème 4.1 *La projection d'une région selon une variable du programme à laquelle les variables φ ne sont pas liées est exacte.*

Notation 4.2 *Soient un état-mémoire σ et une variable ou un ensemble de variables v. Nous noterons σ_v l'état-mémoire indéfini sur le domaine v, et identique à σ sur le reste de son domaine.*

Démonstration : Soit R une région décrite par une relation $\widehat{R}(\varphi,\sigma)$. Soit v_{indep} l'ensemble des variables du programme auxquelles les variables φ ne sont pas liées. Soit v_{dep} l'ensemble des variables auxquelles les variables φ sont liées.

La relation $\widehat{R}(\varphi,\sigma)$ peut alors se décomposer en deux relations :

$$\widehat{R}(\varphi,\sigma) = \widehat{R}'(\varphi,\sigma_{v_{indep}}) \wedge P(\sigma_{v_{dep}})$$

On a donc :

$$\begin{aligned} R(\sigma) &= \{\varphi \mid \widehat{R}(\varphi,\sigma)\} \\ &= \{\varphi \mid \widehat{R}'(\varphi,\sigma_{v_{indep}}) \wedge P(\sigma_{v_{dep}})\} \\ &= \{\varphi \mid \widehat{R}'(\varphi,\sigma_{v_{indep}})\} \cap \{\varphi \mid P(\sigma_{v_{dep}})\} \\ &= \{\varphi \mid \widehat{R}'(\varphi,\sigma_{v_{indep}})\} \cap \{\varphi\} \\ &= \{\varphi \mid \widehat{R}'(\varphi,\sigma_{v_{indep}})\} \end{aligned}$$

Introduisons $R'(\sigma_{v_{indep}}) = \{\varphi \mid \widehat{R}'(\varphi,\sigma_{v_{indep}})\}$, et soit $k \in v_{indep}$ la variable selon laquelle on projette. De ce qui précède, nous pouvons déduire :

$$proj_{\sigma_k}(R(\sigma)) = proj_{\sigma_k}(R'(\sigma_{v_{indep}}))$$

Or $k \in v_{indep} \Longrightarrow \sigma_{v_{indep}} \subseteq \sigma_k$, et donc $proj_{\sigma_k}(R'(\sigma_{v_{indep}})) = R'(\sigma_{v_{indep}})$, ce qui implique :

$$proj_{\sigma_k}(R(\sigma)) = R'(\sigma_{v_{indep}}) = R(\sigma)$$

Donc la projection selon une variable dont l'expression des `PHIs` ne dépend pas, même indirectement, conserve l'ensemble des `PHIs` de la région initiale, et est donc exacte au sens des régions. □

Intéressons-nous maintenant à la projection d'une région selon une variable du programme dont dépendent les variables `PHIs`. Il s'agit de trouver les conditions qui caractérisent une projection exacte au sens des régions.

Théorème 4.2 *Soit une région décrite par un polyèdre $\widehat{R}(\varphi,\sigma)$. Soit k une variable du programme liée directement ou indirectement aux variables φ par $\widehat{R}(\varphi,\sigma)$.*

- *Si $\widehat{R}(\varphi,\sigma)$ contient (explicitement ou implicitement) une équation liant la valeur de k dans l'état-mémoire σ à celles des autres variables du programme, et si la substitution de k dans les contraintes de $\widehat{R}(\varphi,\sigma)$ est effectuée à l'aide de cette équation ;*
- *ou, si $\widehat{R}(\varphi,\sigma)$ ne contient aucune équation contenant la variable k, et aucune inéquation liant directement k aux variables φ;*

alors la projection de la région selon k est exacte.

Démonstration : $\widehat{R}(\varphi,\sigma)$ peut être vu comme étant la description de l'espace des φ en fonction de paramètres qui sont les valeurs de variables du programme dans l'état-mémoire σ. k fait partie de ces paramètres.

Si une équation lie la valeur de k à celles des autre variables dans l'état-mémoire σ, l'ensemble des φ décrit par $\widehat{R}(\varphi,\sigma)$ peut être entièrement décrit par la donnée des seules variables autres que k. Donc le polyèdre $\widehat{R}'(\varphi,\sigma_k)$, obtenu à partir de $\widehat{R}(\varphi,\sigma)$ en substituant k

à l'aide de l'équation décrivant sa valeur en fonction de celle des autres variables, est tel que $\{\varphi \,|\, \widehat{R}(\varphi,\sigma)\} = \{\varphi \,|\, \widehat{R}'(\varphi,\sigma_k)\}$. Or, par construction, $\{\varphi \,|\, \widehat{R}(\varphi,\sigma)\} = proj_{\sigma_k}(\{\varphi \,|\, \widehat{R}'(\varphi,\sigma_k)\})$. Donc $proj_{\sigma_k}(\{\varphi \mid \widehat{R}'(\varphi,\sigma_k)\}) = \{\varphi \mid \widehat{R}'(\varphi,\sigma_k)\}$, ce qui est la définition d'une projection exacte.

Si $\widehat{R}(\varphi,\sigma)$ ne contient aucune équation contenant la variable k, et aucune inéquation liant directement k aux variables φ, alors les variables φ sont entièrement déterminées en fonction des valeurs des autres variables, les inéquations dans lesquelles k apparait n'apportant d'information que sur la valeur de k par rapport aux autres variables. La projection selon k ne fera donc perdre aucune information sur l'espace des φ, et elle est de ce fait exacte. □

Remarques :

- La réciproque est fausse. En effet, la projection selon `K` de la région {`PHI1<=K, PHI1==L`} est exacte ({`PHI1==L`}), mais le polyèdre ne vérifie aucune des deux conditions du théorème ci-dessus.

- Si le polyèdre contient des contraintes redondantes, il peut ne pas vérifier les conditions suffisantes décrites précédemment, alors que la projection est exacte. Un exemple est celui du polyèdre {`PHI1<=J,J<=K,PHI1<=K`}.

- La première condition de projection exacte s'apparente aux conditions caractérisant une affectation *inversible* introduites dans [20] dans le cadre de la transformation des contraintes linéaires par les structures de contrôle.

- Ces conditions sont très différentes de celles de [6, 50]. En effet elles peuvent très bien être vérifiées, et que la projection rajoute des points par rapport au polyèdre de départ. La différence provient du fait que la représentation des régions sous forme de polyèdre ne permet pas de tenir compte de la différence entre les variables du programme, dont la valeur dépend de l'état-mémoire considéré, et les variables `PHIs` dont la valeur ne dépend pas de l'état-mémoire, mais qui sont exprimées en fonction des valeurs des variables du programme

Le théorème 4.2 ne peut s'appliquer directement à tous les polyèdres, à cause de la première condition. En effet, on ne peut pas toujours déduire de la forme du polyèdre qu'il existe une relation implicite entre la variable à éliminer et les autres variables du programme. Par exemple, dans le polyèdre {`PHI1==L, PHI1==K`} n'apparait aucune équation liant directement `L` et `K`, alors qu'il contient implicitement l'équation `K==L`. Si l'on se réfère uniquement au polyèdre de départ, la projection ne peut pas être qualifiée d'exacte, alors qu'elle l'est. Il faut donc trouver un moyen de faire apparaître les équations implicites ou au moins de montrer qu'il n'en existe pas.

Définition 4.11 *Soit une région R. Le polyèdre $\widehat{R}(\varphi,\sigma)$ la décrivant est appelé* système minimal *s'il contient le plus petit nombre possible d'équations contenant des variables de φ.*

Remarque : Il n'y a pas unicité du système minimal. En effet, les systèmes {`PHI1==K,K==L`} et {`PHI1==L,K==L`} sont tous deux minimaux, et décrivent le même ensemble de `PHIs`.

Théorème 4.3 *Soit une région R décrite par un polyèdre $\widehat{R}(\varphi, \sigma)$ non redondant.*

Si ce polyèdre est un système minimal, alors les équations dans lesquelles des variables φ apparaissent ne permettent pas de générer de relation linéaire entre les variables du programme qu'elles contiennent.

Démonstration : Soit E l'ensemble d'équations dans lesquelles des variables $\varphi_{eq} \subseteq \varphi$ apparaissent. E peut s'exprimer par :

$$A \begin{bmatrix} \varphi_{eq} \\ V \end{bmatrix} = C$$

Le polyèdre étant non redondant, A est de rang plein.

Supposons que E est minimal et qu'il peut engendrer une relation linéaire implicite entre les variables du programme apparaissant dans ces équations. Cette relation, de la forme $d^tV = e$, peut s'obtenir par combinaison linéaire de certaines lignes de A, et peut donc remplacer l'une de celles-ci. Supposons que c'est la première ligne de A (sinon il suffit de permuter des lignes). Soit A' la matrice A privée de sa première ligne. Soit C' le vecteur C privé de son premier élément. E peut donc être réécrit sous la forme :

$$\left[\begin{array}{c|c} 0 & d^t \\ \hline \multicolumn{2}{c}{A'} \end{array}\right] \begin{bmatrix} \varphi_{eq} \\ V \end{bmatrix} = \left[\begin{array}{c} e \\ \hline C' \end{array}\right]$$

Un système E' des équations contenant des variables de φ est donc donné par :

$$A' \begin{bmatrix} \varphi_{eq} \\ V \end{bmatrix} = C'$$

Or E' contient une équation de moins que E. Ceci est en contradiction avec le fait que E est minimal. Donc si E est minimal, il ne peut engendrer de relation implicite entre les variables du programme. □

Théorème 4.4 *Soit une région R décrite par un polyèdre $\widehat{R}(\varphi, \sigma)$ comportant m équations non redondantes. Soit φ_{eq} le sous-vecteur de dimension n_φ des variables de φ apparaissant dans les équations de $\widehat{R}(\varphi, \sigma)$. Soit V_{eq} le sous-vecteur de dimension n_V des variables du programme apparaissant dans les équations de $\widehat{R}(\varphi, \sigma)$.*

Les équations du polyèdre $\widehat{R}(\varphi, \sigma)$ peuvent alors être mises sous la forme :

$$\left[\begin{array}{c|c} A_\varphi & A_V \end{array}\right] \begin{bmatrix} \varphi_{eq} \\ V_{eq} \end{bmatrix} = C$$

A_φ étant une matrice $m \times n_\varphi$ de rang r, A_V une matrice $m \times n_V$, et C un vecteur de dimension m.

Soit H la forme de Hermite de A^t_φ. H vérifie :

$$H = PA^t_\varphi Q$$

P et Q étant unimodulaires, et P étant une matrice de permutation.

Alors, le système décrit par :

$$\left[\begin{array}{c|c} H^tP & Q^tA_V \end{array}\right] \begin{bmatrix} \varphi_{eq} \\ V_{eq} \end{bmatrix} = Q^tC$$

est équivalent au système initial et est un système minimal.

Démonstration : Montrons tout d'abord que le système décrit par :

$$\left[\begin{array}{c|c} A_\varphi & A_V \end{array}\right] \left[\begin{array}{c} \varphi_{eq} \\ V_{eq} \end{array}\right] = C \tag{4.1}$$

est équivalent au système décrit par :

$$\left[\begin{array}{c|c} H^tP & Q^tA_V \end{array}\right] \left[\begin{array}{c} \varphi_{eq} \\ V_{eq} \end{array}\right] = Q^tC \tag{4.2}$$

Nous savons que $H = PA_\varphi^tQ$. En prenant la transposée de cette expression, nous obtenons donc l'expression équivalente $H^t = Q^tA_\varphi P^t$. Or P est une matrice de permutation ; elle vérifie donc $PP^t = I$. En multipliant à droite $H^t = Q^tA_\varphi P^t$ par P, nous obtenons alors :

$$H^tP = Q^tA_\varphi \tag{4.3}$$

En multipliant à gauche l'équation 4.1 par Q^t, en remplaçant H^tP par l'expression équivalente donnée par l'équation 4.3, et en remarquant que Q^t est inversible, nous obtenons :

$$\left[\begin{array}{c|c} A_\varphi & A_V \end{array}\right] \left[\begin{array}{c} \varphi_{eq} \\ V_{eq} \end{array}\right] = C \quad \Longleftrightarrow \quad \left[\begin{array}{c|c} H^tP & Q^tA_V \end{array}\right] \left[\begin{array}{c} \varphi_{eq} \\ V_{eq} \end{array}\right] = Q^tC$$

Les systèmes 4.1 et 4.2 sont donc bien équivalents.

Montrons maintenant que le nombre d'équations du système 4.2 contenant des variables de φ est minimal.

Remarquons tout d'abord que l'on peut supposer que $P = I$ sans perte de généralité. Il suffit, dans le système original, de permuter les variables φ_{eq} et les colonnes correspondantes de A_φ de manière *ad-hoc*.

H est la forme de Hermite de A_φ^t. Elle est donc de dimension $n_\varphi \times m$ et de rang r, et est de la forme :

$$H = \left[\begin{array}{cc} L & 0 \\ S & 0 \end{array}\right]$$

L est une matrice triangulaire inférieure, de dimension $r \times r$, de rang r ; ses éléments diagonaux sont donc non nuls. S est une matrice de dimension $(n_\varphi - r) \times m$ si $n_\varphi > r$, ou 0×0 si $n_\varphi \leq r$. Le système 4.2 peut donc s'écrire :

$$\left[\begin{array}{c|c} \begin{array}{c|c} L^t & S^t \end{array} & \\ \hline & Q^tA_V \\ 0 & \end{array}\right] \left[\begin{array}{c} \varphi_{eq} \\ V_{eq} \end{array}\right] = Q^tC$$

Ce système comporte r équations contenant des variables de φ, car le nombre de lignes dont les coefficients des variables de φ sont non nuls est égal au nombre de lignes de L, soit r. Le fait que les éléments diagonaux de L, et donc de L^t, soient non nuls garantit que l'on ne peut combiner linéairement les équations contenant des variables de φ pour trouver des équations ne contenant plus de variables de φ. Le système 4.2 contient donc bien le nombre minimal d'équations comportant des variables de φ. □

Corollaire 4.1 *Soit une région R décrite par un polyèdre $\widehat{R}(\varphi, \sigma)$. Soit k une variable du programme apparaissant dans les équations de $\widehat{R}$.*

Si il existe une équation reliant la valeur de k à celles des variables du programme et ne faisant pas intervenir les variables de φ, alors elle peut être reconstituée à partir des équations ne comportant pas de variables de φ du système minimal correspondant à $\widehat{R}$.

Démonstration : D'après le théorème 4.4, le système minimal comporte le nombre minimal d'équations comportant des variables de φ. D'après le théorème 4.3, ces équations ne permettent pas de générer d'équation implicite ne comportant que des variables du programme. Donc, si il existe une relation entre k et les autres variables du programme, elle ne peut être obtenue que par combinaison linéaire des équations du système minimal ne comportant pas de variables de φ. □

Des théorèmes 4.2, 4.3, 4.4, et du dernier corollaire, nous pouvons déduire un algorithme de projection exacte des régions.

Algorithme 4.5 (projection exacte d'une région)
Soit une région R décrite par un polyèdre $\widehat{R}(\varphi, \sigma)$.
Soit k la variable selon laquelle on désire projeter R.

1. *Si R est une région* **`MAY`***, la projeter selon k à l'aide de l'algorithme de Fourier-Motzkin. L'approximation de la région résultante est* **`MAY`***.*

2. *Si R est une région* **`MUST`** *et si les variables de φ ne sont pas liées à k, alors la projection est exacte ; l'effectuer à l'aide de l'algorithme de Fourier-Motzkin. L'approximation de la région résultante est* **`MUST`***.*

3. *Si R est une région* **`MUST`** *et si les variables de φ sont liées à k :*

   (a) *chercher les équations contenant k :*

      i. *il y en a une ne contenant pas de variable de φ :*
         $\longrightarrow$ *effectuer la projection à l'aide de cette équation ; la projection est exacte, et l'approximation de la région résultante est donc* **`MUST`***.*

      ii. *il y en a, mais elles contiennent des variables de φ :*
         $\longrightarrow$ *soit $\widehat{R}'(\varphi, \sigma)$ le système minimal associé à $\widehat{R}(\varphi, \sigma)$; appliquer l'algorithme 4.5 à $\widehat{R}'$ à partir de l'étape 3, et en sautant l'étape 3(a)ii.*

      iii. *sinon, aller en 3b.*

   (b) *chercher les inéquations contenant k et contenant au moins une variable de φ :*

      i. *il n'y en a pas:*
         $\longrightarrow$ *effectuer la projection à l'aide d'une inéquation contenant k ; la projection est exacte, et l'approximation de la région résultante est donc* **`MUST`***.*

      ii. *il y en a une :*
         $\longrightarrow$ *effectuer la projection selon k à l'aide de l'algorithme de Fourier-Motzkin. L'approximation de la région résultante est* **`MAY`***.*

4.4 Conclusion

En définissant les opérateurs permettant de manipuler les régions, nous nous sommes heurtés à divers problèmes provenant de leur représentation sous forme de polyèdre convexe :

1. les opérateurs classiques de manipulation des ensembles (union, projection) ne conservent pas forcément la convexité ; il est donc nécessaire d'employer des opérateurs conservant cette propriété, mais donnant une approximation de l'ensemble recherché ;

2. la représentation ne permet pas de distinguer les rôles des variables `PHIs` et des variables du programme ; ces dernières devraient être considérées comme des paramètres permettant de définir les variables `PHIs` ; ainsi, les contraintes liant les variables du programme entre elles ne devraient pas être considérées de la même manière que les contraintes liant les variables `PHIs` entre elles et aux variables du programme ; ce que la représentation sous forme de polyèdre ne permet pas de réaliser directement.

Une solution partielle au premier problème pourrait être d'utiliser des disjonctions de régions (à intersection vide) pour représenter leur union. L'opérateur d'union serait alors évidemment exact. Mais le surcoût en terme de temps de calcul, et d'utilisation de la mémoire risque d'être important, et seule l'expérience pourra démontrer la nécessité d'une telle précision. Une autre amélioration pourrait être d'introduire des variables *muettes* pour remplacer les variables projetées. Par exmple, la projection de la région :

```
<A(PHI1)-R-MUST-{PHI1==2K}>
```

selon `K` ne serait pas :

```
<A(PHI1)-R-MAY-{}>
```

mais :

```
<A(PHI1)-R-MAY-{PHI1==2THETA}>
```

`THETA` n'étant pas une variable du programme, ce qui permettrait de garder l'information *`PHI` est pair*. La difficulté est alors de définir la sémantique des régions comportant des variables muettes, ainsi que les opérateurs les manipulant, problème difficile nécessitant une représentation normalisée de ces régions [1].

Le deuxième problème pourrait être partiellement résolu en normalisant la représentation. La mise sous forme de *système minimal*, par exemple, permet de distinguer plus facilement les contraintes impliquant des relations entre les variables du programme, des contraintes liant les variables `PHIs` entre elles et aux variables du programme. Ceci implique encore un surcoût en temps de calcul, puisqu'il sera nécessaire de remettre le système sous forme *normale* après chaque opération le manipulant. De plus, le système minimal ne peut constituer qu'un premier pas vers une normalisation car il n'est pas unique.

Pour finir ce chapitre sur une note optimiste, signalons tout de même que les opérateurs qui viennent d'être définis permettent de conserver les approximations `MUST` dans un grand nombre de cas apparaissant dans les *bons* programmes (lorsque tout est bien défini, linéaire, ...). Là encore, des expériences permettront d'évaluer les gains et pertes qu'ils apportent.

1. Communication personnelle de Fabien Coelho.

Chapitre 5

Calcul de régions exactes

Le calcul des régions comporte deux phases. L'*analyse intraprocédurale* s'appuie sur le graphe de contrôle hiérarchisé des modules, et concerne donc les instructions élémentaires, les blocs d'instructions structurés ou non-structurés, les tests et les boucles. L'*analyse interprocédurale* permet de propager les régions relatives à une routine au niveau des différents sites d'appel en remontant le graphe des appels.

Ce chapitre décrit en détail le calcul intraprocédural des régions, en mettant l'accent sur la préservation de leur attribut `MUST`. Le calcul interprocédural sera brièvement introduit par la suite.

5.1 Calcul intraprocédural

5.1.1 Régions d'une instruction

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 équations 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 est exacte car les expressions des indices représentent exactement — dans l'état-mémoire courant — les éléments référencés. L'approximation de cette région est donc `MUST` et 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`.

5.1.2 Régions d'une séquence d'instructions

Nous supposons connues les régions de chacune des instructions composant la séquence, et nous cherchons à calculer les régions résumant les accès aux éléments de tableaux pour l'ensemble de cette structure de contrôle. Ces régions seront donc exprimées dans l'état-mémoire précédant la séquence.

```
      K = FOO()
      DO I = 1,N
C <WORK(PHI1,PHI2)-W-MUST-{1<=PHI1,PHI1<=N,PHI2==K,1<=I,I>=N}>
         DO J = 1,N
            WORK(J,K) = J + K
         ENDDO
         CALL INC1(K)
C <WORK(PHI1,PHI2)-W-MUST-{1<=PHI1,PHI1<=N,PHI2==K,1<=I,I>=N}>
         DO J = 1,N
            WORK(J,K) = J * J - K * K
         ENDDO
      ENDDO

      SUBROUTINE INC1(I)
      I = I + 1
      END
```

FIG. 5.1 - *Régions d'une séquence : exemple.*

Si aucune variable scalaire entière apparaissant dans les expressions d'indice de tableau n'est modifiée lors de l'exécution de la séquence, les régions des instructions sont alors toutes exprimées dans le même état-mémoire et peuvent être fusionnées à l'aide de l'opérateur $\cup_{must}$ pour obtenir les régions de la séquence. Ceci n'est plus possible lorsqu'une variable scalaire entière intervenant dans une expression d'indice est modifiée. Pour s'en convaincre, considérons l'exemple de la figure 5.1. Nous nous intéressons à la séquence constituant le corps de la boucle d'indice `I`. Deux boucles initialisent certains éléments du tableau `WORK`. Dans les régions correspondant à ces deux instructions complexes, la valeur de `PHI2` dépend de celle de `K`. Or, cette dernière variable est modifiée entre les deux instructions par un appel à `INC1`. La valeur de `K` apparaissant dans les deux régions n'est donc pas la même, et il serait incorrect de les fusionner telles quelles.

Une première solution consisterait à tenir compte des effets en écriture de l'appel à `INC1` sur la variable `K`, en éliminant cette dernière de la deuxième région, qui deviendrait alors :

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

Et la fusion avec la première région donnerait un résultat identique. On constate que l'on perd ainsi toute information sur la valeur de `K` (et donc sur `PHI2`), alors que l'on souhaiterait pouvoir tenir compte du fait que `K` est incrémenté de 1 pour obtenir, pour la séquence, la région :

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

Le principe de l'algorithme sera donc d'exprimer les régions dans l'état-mémoire précédant l'exécution de la séquence, en tenant compte des effets des instructions sur les *valeurs* des variables scalaires entières en fonction desquelles les régions sont exprimées. Pour effectuer cette transformation, nous utiliserons les informations contenues dans les transformeurs.

Formulation du problème

Soit la séquence linéaire B, constituée des instructions $S_1, S_2, \ldots, S_n$. Soient $P_1, \ldots, P_n$, $T_1, \ldots, T_n$, et $R_1, \ldots, R_n$, les préconditions, les transformeurs et les régions associés aux instructions $S_1, S_2, \ldots, S_n$ de B.

On introduit R'_k ($k \in [1,n]$), l'ensemble des régions associées à $S_k, \ldots, S_n$ et exprimées dans l'état-mémoire σ_k précédant S_k.

On cherche R_0 l'ensemble de régions associé à B.

Définition 5.1 *Soit R_{k+1} une région exprimée dans l'état-mémoire σ_{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 σ_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. Elle est effectuée à l'aide de l'algorithme 4.5.

Le transformeur définit les nouvelles valeurs des variables en fonction des anciennes. Pour les distinguer, un suffixe `#INIT` est ajouté au nom de ces dernières. Dans la région recherchée, les nouvelles valeurs auront été éliminées, tandis que les anciennes valeurs seront devenues les nouvelles. Il faut alors renommer les variables suffixées par `#INIT`. $\widetilde{R}_k$ est donc obtenu, dans la pratique, de la manière suivante :

- soit V l'ensemble des variables scalaires entières arguments de T_k ; ce sont les variables modifiées par S_k ;
- on ajoute au prédicat de R_{k+1} celui de T_k, et on projette la région résultante suivant les variables contenues dans V ; la région obtenue contient des variables suffixées par `#INIT` ; prenons un exemple :

  ```
  T(I) {I=I#INIT+1}
   et
  <A(PHI1)-W-MUST-{PHI1==I}>
  ```

  Après ajout du prédicat de `T(I)` dans la région, on obtient :

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

  et après projection selon `I` :

  ```
  <A(PHI1)-W-MUST-{PHI1==I#INIT+1}>
  ```

- il suffit alors de renommer les variables suffixées par `#INIT` ; la région obtenue est la région $\widetilde{R}_k$ recherchée ; dans notre exemple, la région finale est :

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

Algorithme 5.1 *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 :*

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

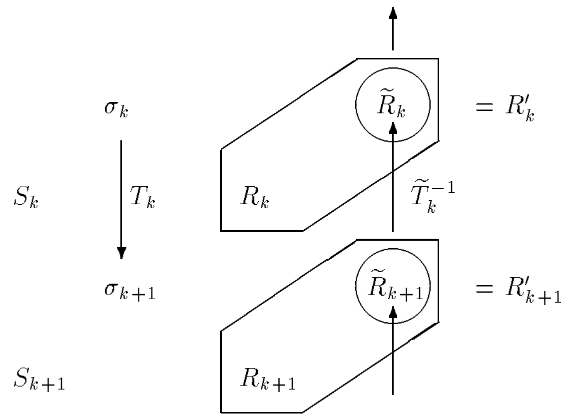


FIG. 5.2 - *Algorithme de calcul des régions d'une séquence.*

La figure 5.2 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 σ_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 σ_k. On a également fait figurer le transformeur T_k qui donne une approximation de la transformation de σ_k en σ_{k+1} par S_k.

Il s'agit maintenant de montrer que l'algorithme présenté précédemment donne un résultat correct, c'est-à dire des régions, que dans certains cas (à caractériser) il permet de préserver les approximations `MUST`, et qu'il donne des régions plus précises que si l'on ne tient compte que des effets sur les variables scalaires entières, comme cela a été montré dans un exemple au début de cette section, et non sur leurs valeurs. L'algorithme 5.1 étant défini récursivement, nous nous limiterons pour ces preuves aux blocs de deux instructions, car l'on peut toujours se ramener à ce cas : il suffit de considérer que la séquence composée des instructions $S_2, \ldots, S_n$ est une unique instruction complexe dont on a déjà calculé les régions. L'algorithme 5.1 étant trivialement correct dans le cas $n = 1$, la preuve par récurrence se déduit immédiatement de la preuve pour deux instructions. De plus, nous supposerons qu'une seule région est associée à chacune des instructions, concernant le même tableau, et avec une même action. La généralisation à des listes de régions est évidente.

Notations et définitions

Soit la séquence linéaire B, constituée des instruction S_1 et S_2. Soient P_1 et P_2, T_1 et T_2, R_1 et R_2, les préconditions, les transformeurs et les régions associés aux instructions de B. Soient σ_1 et σ_2 les états-mémoire précédant respectivement l'exécution de S_1 et S_2.

Soit S une instruction, σ l'état-mémoire la précédant ; on note $\widehat{S}$ la transformation associée à S, qui à un état-mémoire σ fait correspondre l'état-mémoire σ' après exécution de S :

$$\widehat{S} : \sigma \longrightarrow \sigma' \tag{5.1}$$

Soient T_S le transformeur et R_S l'ensemble de régions associés à S ; on note $\widehat{T}_S$ et $\widehat{R}_S$ les relations associées à T_S et R_S. $\widehat{T}_S$ est définie par :

$$\sigma' = \widehat{S}(\sigma) \implies \widehat{T}_S(\sigma, \sigma') \tag{5.2}$$

Pour une région R_S d'approximation `MAY`, la relation $\widehat{R}_S$ est définie par :

$$\forall\, \sigma, \quad action(\varphi, S, \sigma) \implies \widehat{R}_S(\varphi, \sigma)$$

ce qui peut également s'exprimer par :

$$\forall\, \sigma,\ \{\varphi \mid action(\varphi, S, \sigma)\} \subseteq \{\varphi \mid \widehat{R}_S(\varphi, \sigma)\} \tag{5.3}$$

Par contre, pour une région `MUST`, $\widehat{R}_S$ s'exprime de la façon suivante :

$$\forall\, \sigma, \quad action(\varphi, S, \sigma) \iff \widehat{R}_S(\varphi, \sigma)$$

ce qui entraîne :

$$\forall\, \sigma,\ \{\varphi \mid action(\varphi, S, \sigma)\} = \{\varphi \mid \widehat{R}_S(\varphi, \sigma)\} \tag{5.4}$$

Dans les deux cas, la région $R_S(\sigma)$ est l'ensemble $\{\varphi \mid \widehat{R}_S(\varphi, \sigma)\}$.

Correction de l'algorithme pour les régions MAY

Propriété 5.1 *Soit une séquence S_1, S_2 telle que les régions R_1 et R_2 associées à chacune des instructions soient des régions `MAY`. Alors la région R_0 obtenue par l'algorithme 5.1 est une région `MAY` donnant une approximation des éléments de tableau référencés dans la séquence d'instructions S_1, S_2.*

Corollaire 5.1 *L'algorithme 5.1 conserve les régions (i.e., R_0 est une région).*

Démonstration : Dans le cas de l'instruction S_2, l'équation 5.3 devient :

$$\forall \sigma_2,\ \{\varphi \mid action(\varphi, S_2, \sigma_2)\} \subseteq \{\varphi \mid \widehat{R}_2(\varphi, \sigma_2)\}$$

De plus, on sait que $\forall \sigma_2,\ \exists \sigma_1 \ \mid \sigma_2 = \widehat{S}_1(\sigma_1)$. On a donc :

$$\forall \sigma_2\, \exists \sigma_1,\ \{\varphi \mid action(\varphi, S_2, \widehat{S}_1(\sigma_1))\} \subseteq \{\varphi \mid \widehat{R}_2(\varphi, \widehat{S}_1(\sigma_1))\}$$

Or,

$$\{\varphi \mid \widehat{R}_2(\varphi, \widehat{S}_1(\sigma_1))\} = \{\varphi \mid \exists\, \sigma\ \sigma = \widehat{S}_1(\sigma_1) \wedge \widehat{R}_2(\varphi, \sigma)\}$$

ce qui, en tenant compte de la définition 5.2, donne :

$$\{\varphi \mid \widehat{R}_2(\varphi, \widehat{S}_1(\sigma_1))\} \subseteq \{\varphi \mid \exists\, \sigma\ \widehat{T}_1(\sigma_1, \sigma) \wedge \widehat{R}_2(\varphi, \sigma)\}$$

Donc :

$$\begin{aligned} \forall \sigma_2\, \exists \sigma_1,\ \{\varphi \mid \widehat{R}_2(\varphi, \widehat{S}_1(\sigma_1))\} &\subseteq \{\varphi \mid \exists\, \sigma\ \widehat{T}_1(\sigma_1, \sigma) \wedge \widehat{R}_2(\varphi, \sigma)\} \\ &\subseteq \{\varphi, \sigma \mid \widehat{T}_1(\sigma_1, \sigma) \wedge \widehat{R}_2(\varphi, \sigma)\} \end{aligned} \tag{5.5}$$

L'algorithme consiste alors à projeter le deuxième membre de l'équation 5.5 le long de σ. Nous avons vu que l'opérateur de projection d'une région conduit dans le cas général à une approximation. L'équation 5.5 se réécrit donc :

$$\forall \sigma_2\, \exists \sigma_1,\ \{\varphi \mid \widehat{R}_2(\varphi, \widehat{S}_1(\sigma_1))\} \subseteq proj_\sigma\{\varphi, \sigma \mid \widehat{T}_1(\sigma_1, \sigma) \wedge \widehat{R}_2(\varphi, \sigma)\} \tag{5.6}$$

Introduisons $\widetilde{R}_2$ défini par :

$$\widetilde{R}_2(\sigma_1) = proj_\sigma\{\varphi, \sigma \mid \widehat{T}_1(\sigma_1, \sigma) \wedge \widehat{R}_2(\varphi, \sigma)\}$$

$\widetilde{R}_2$ est la projection le long du domaine de l'état-mémoire σ de l'ensemble des φ décrit par l'intersection du polyèdre T_1 et du polyèdre de la région initiale R_2. En utilisant les relations, ceci s'écrit $\widehat{T}_1(\sigma_1, \sigma) \wedge \widehat{R}_2(\varphi, \sigma)$. Par définition de $\widetilde{T}_k^{-1}$, on a donc $\widetilde{R}_2(\sigma_1) = \widetilde{T}_1^{-1}(R_1(\sigma_1))$.

Pour ne pas alourdir les notations, nous noterons $\widetilde{R}_2$ la relation associée à $\widetilde{R}_2$:

$$\widetilde{R}_2(\sigma_1) = \{\varphi \mid \widetilde{R}_2(\varphi, \sigma_1)\}$$

Le but est de montrer que $\widetilde{R}_2$ est une région ; pour cela, réécrivons l'équation 5.6 à l'aide de la définition précédente :

$$\forall \sigma_2 \, \exists \sigma_1, \; \{\varphi \mid \widehat{R}_2(\varphi, \widehat{S}_1(\sigma_1))\} \subseteq \{\varphi \mid \widetilde{R}_2(\varphi, \sigma_1)\}$$

soit :

$$\forall \sigma_2 \, \exists \sigma_1, \; \{\varphi \mid action(\varphi, S_2, \sigma_2)\} \subseteq \{\varphi \mid \widetilde{R}_2(\varphi, \sigma_1)\}$$

On reconnait bien l'expression d'une région `MAY` exprimée dans l'état-mémoire σ_1 telle qu'elle est donnée par l'équation 5.3. De plus, c'est bien une approximation de l'ensemble des éléments référencés dans l'instruction S_2.

R_1 et $\widetilde{R}_2$ étant toutes deux exprimées dans le même état-mémoire, il est légitime de poser :

$$R_0 = \cup_{must}(R_1, \widetilde{R}_2)$$

R_0 est l'enveloppe convexe de deux régions donnant respectivement une approximation de l'ensemble des éléments référencés par S_1 et par S_2. L'opérateur $\cup_{must}$ conservant les régions `MAY`, R_0 est donc bien une approximation de l'ensemble des éléments référencés dans la séquence S_1, S_2.

La démonstration du corollaire est immédiate. Nous venons de démontrer que l'algorithme 5.1 conservait les régions `MAY`. Or toute région `MUST` est également une région `MAY`. L'algorithme 5.1 conserve donc les régions. □

Intérêt de l'algorithme

Propriété 5.2 *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 U_{must}.*

Démonstration : R_2 étant une région `MUST`, l'équation 5.4 donne :

$$\forall \sigma_2, \; \{\varphi \mid action(\varphi, S_2, \sigma_2)\} = \{\varphi \mid \widehat{R}_2(\varphi, \sigma_2)\}$$

On sait d'autre part que $\forall \sigma_2 \, \exists \sigma_1, \; \sigma_2 = \widehat{S}_1(\sigma_1)$, ce qui entraîne :

$$\forall \sigma_2 \, \exists \sigma_1, \; \{\varphi \mid action(\varphi, S_2, \widehat{S}_1(\sigma_1))\} = \{\varphi \mid \widehat{R}_2(\varphi, \widehat{S}_1(\sigma_1))\}$$

On s'intéresse au cas où les valeurs des variables scalaires entières modifiées par S_1 s'expriment (dans l'état σ_2) linéairement en fonction des valeurs des variables scalaires entières dans l'état σ_1. Le transformeur modélise alors exactement le comportement de l'instruction. C'est le cas, par exemple, si T_1 vaut `T(I) {I = I#INIT + J -1}`. La définition 5.2 devient alors :

$$\widehat{T}_S : \quad \sigma' = \widehat{S}(\sigma) \quad \Longleftrightarrow \quad \widehat{T}_S(\sigma, \sigma')$$

On a de ce fait :

$$\begin{aligned} \forall \sigma_2 \, \exists \sigma_1, \; \{\varphi \mid \widehat{R}_2(\varphi, \widehat{S}_1(\sigma_1))\} &= \{\varphi \mid \exists \, \sigma, \; (\sigma = \widehat{S}_1(\sigma_1)) \wedge \widehat{R}_2(\varphi, \sigma)\} \\ &= \{\varphi \mid \exists \, \sigma, \; \widehat{T}_1(\sigma_1, \sigma) \wedge \widehat{R}_2(\varphi, \sigma)\} \end{aligned}$$

La projection le long de σ du deuxième membre de l'équation précédente est obligatoirement exacte puisque l'on a rajouté les contraintes du transformeur à celles de la région. Or ces contraintes sont toutes des égalités, et il y en a au moins une par variable à éliminer. Donc, si l'on note comme précédemment $\widetilde{R}_2$ la région définie par :

$$\widetilde{R}_2(\sigma_1) = proj_\sigma\{\varphi, \sigma \mid \widehat{T}_1(\sigma_1, \sigma) \wedge \widehat{R}_2(\varphi, \sigma)\} = \{\varphi \mid \widetilde{R}_2(\varphi, \sigma_1)\}$$

on obtient :

$$\forall \sigma_2 \, \exists \sigma_1, \; \{\varphi \mid \widehat{R}_2(\varphi, \widehat{S}_1(\sigma_1))\} = \{\varphi \mid \widetilde{R}_2(\varphi, \sigma_1)\}$$

soit finalement :

$$\forall \sigma_2 \, \exists \sigma_1, \; \{\varphi \mid action(\varphi, S_2, \sigma_2)\} = \{\varphi \mid \widetilde{R}_2(\varphi, \sigma_1)\}$$

ce qui est bien la définition d'une région `MUST`. Dans les hypothèses de la propriété, l'opérateur $\widetilde{T}_1^{-1}$ conserve donc bien les approximations `MUST`.

La région recherchée étant $R_0 = \cup_{must}(R_1, \widetilde{R}_2)$ l'approximation `MUST` ne peut être préservée que si l'enveloppe convexe est strictement égale à l'union des régions R_1 et $\widetilde{R}_2$. L'approximation de la région R_0 ne dépend donc que de cette dernière opération.

□

Propriété 5.3 *L'algorithme 5.1 calcule des régions plus précises*[1] *que lorsque seuls les effets sur les variables scalaires entières sont utilisés pour éliminer les variables modifiées par une instruction.*

Démonstration : Nous noterons E_S les effets d'une instruction S sur les variables scalaires entières du programme. Leur définition générale est identique à celle des transformeurs :

$$\widehat{E}_S : \quad \sigma' = \widehat{S}(\sigma) \quad \Longrightarrow \quad \widehat{E}_S(\sigma, \sigma') \tag{5.7}$$

La démonstration comporte deux sous-parties ; dans un premier temps, nous montrerons que :

$$\forall \, \sigma, \sigma' \; \widehat{T}_S(\sigma, \sigma') \Longrightarrow \widehat{E}_S(\sigma, \sigma')$$

mais que la réciproque n'est pas nécessairement vraie ; puis nous reprendrons la démonstration de la page 53 pour montrer que la région $\widetilde{R}_2$ obtenue avec les effets est moins précise que celle obtenue avec les transformeurs, c'est-à-dire que :

$$\{\varphi \mid \widetilde{R}_{2_T}(\varphi, \sigma_1)\} \subseteq \{\varphi \mid \widetilde{R}_{2_E}(\varphi, \sigma_1)\}$$

1. Dans le sens où elles contiennent moins de points n'appartenant pas aux régions initiales.

Montrer que

$$\forall\, \sigma, \sigma',\ \widehat{T}_S(\sigma, \sigma') \Longrightarrow \widehat{E}_S(\sigma, \sigma')$$

est strictement équivalent à montrer que

$$\{\sigma, \sigma' \mid \widehat{T}_S(\sigma, \sigma')\} \subseteq \{\sigma, \sigma' \mid \widehat{E}_S(\sigma, \sigma')\}$$

Par construction, le transformeur $\widehat{T}_S$ donne l'ensemble des variables modifiées par l'instruction S, ainsi qu'un ensemble de prédicats $\widehat{P}_S(\sigma, \sigma')$ reliant les valeurs des arguments de $\widehat{T}_S$ dans l'état σ précédant S avec leurs valeurs dans l'état σ' suivant l'exécution de S.

> T_S fournit donc des équations sur toutes les variables scalaires entières sûrement non modifiées (`I==I#INIT`) qui ne font pas partie de ces arguments, mais donne aussi éventuellement des précisions sur les nouvelles valeurs des variables modifiées (`L==N+1`).

Par construction, les effets $\widehat{E}_S$ donnent l'ensemble des variables scalaires entières sûrement non modifiées par l'exécution de S, sans précision supplémentaire.

> E_S fournit donc des équations sur toutes les variables scalaires entières sûrement non modifiées (`I==I#INIT`), mais n'apporte aucune information supplémentaire sur la valeur des variables modifiées.

On peut donc en déduire :

$$\begin{array}{rcl}\{\sigma, \sigma' \mid \widehat{T}_S(\sigma, \sigma')\} & = & \{\sigma, \sigma' \mid \widehat{E}_S(\sigma, \sigma') \wedge \widehat{P}_S(\sigma, \sigma')\} \\ & = & \{\sigma, \sigma' \mid \widehat{E}_S(\sigma, \sigma')\} \cap \{\sigma, \sigma' \mid \widehat{P}_S(\sigma, \sigma')\}\end{array}$$

ce qui implique :

$$\{\sigma, \sigma' \mid \widehat{T}_S(\sigma, \sigma')\} \subseteq \{\sigma, \sigma' \mid \widehat{E}_S(\sigma, \sigma')\} \tag{5.8}$$

Remarque : Les deux ensembles sont égaux si et seulement si $|\widehat{P}_S| = 0$, c'est-à-dire s'il n'y a pas de contrainte sur les nouvelles valeurs des variables scalaires ; c'est le cas, par exemple, lorsque la transformation effectuée par S n'est pas linéaire.

Reprenons maintenant les étapes qui permettent d'obtenir la région $\widetilde{R}_2$. Les définitions générales de $\widehat{T}_S$ et de $\widehat{E}_S$ étant similaires, la démonstration de la page 53 serait en effet identique en utilisant les transformeurs ou les effets. Dans le cas où l'on utilise les transformeurs, l'équation 5.5 ne change pas :

$$\{\varphi \mid \widehat{R}_2(\varphi, \widehat{S}_1(\sigma_1))\} \subseteq \{\varphi, \sigma \mid \widehat{T}_1(\sigma_1, \sigma) \wedge \widehat{R}_2(\varphi, \sigma)\}$$

tandis que, dans le cas où l'on utilise les effets, elle s'écrit :

$$\{\varphi \mid \widehat{R}_2(\varphi, \widehat{S}_1(\sigma_1))\} \subseteq \{\varphi, \sigma \mid \widehat{E}_1(\sigma_1, \sigma) \wedge \widehat{R}_2(\varphi, \sigma)\}$$

Or,

$$\{\varphi, \sigma \mid \widehat{T}_1(\sigma_1, \sigma) \wedge \widehat{R}_2(\varphi, \sigma)\} = \{\varphi, \sigma \mid \widehat{T}_1(\sigma_1, \sigma)\} \cap \{\varphi, \sigma \mid \widehat{R}_2(\varphi, \sigma)\}$$

et, d'après (5.8),

$$\{\sigma \mid \widehat{T}_1(\sigma_1, \sigma)\} \subseteq \{\sigma \mid \widehat{E}_1(\sigma_1, \sigma)\}$$

On en déduit donc :

$$\{\varphi, \sigma \mid \widehat{T}_1(\sigma_1, \sigma)\} \cap \{\varphi, \sigma \mid \widehat{R}_2(\varphi, \sigma)\} \subseteq \{\varphi, \sigma \mid \widehat{E}_1(\sigma_1, \sigma)\} \cap \{\varphi, \sigma \mid \widehat{R}_2(\varphi, \sigma)\}$$

soit :

$$\{\varphi,\sigma \mid \widehat{T_1}(\sigma_1,\sigma) \wedge \widehat{R}_2(\varphi,\sigma)\} \subseteq \{\varphi,\sigma \mid \widehat{E_1}(\sigma_1,\sigma) \wedge \widehat{R}_2(\varphi,\sigma)\}$$

On projette ensuite le long de σ :

$$proj_\sigma\{\varphi,\sigma \mid \widehat{T_1}(\sigma_1,\sigma) \wedge \widehat{R}_2(\varphi,\sigma)\} \subseteq proj_\sigma\{\varphi,\sigma \mid \widehat{E_1}(\sigma_1,\sigma) \wedge \widehat{R}_2(\varphi,\sigma)\} \tag{5.9}$$

et l'on introduit $\widetilde{R}_{2_T}$ et $\widetilde{R}_{2_E}$ tels que :

$$\begin{aligned} proj_\sigma\{\varphi,\sigma \mid \widehat{T_1}(\sigma_1,\sigma) \wedge \widehat{R}_2(\varphi,\sigma)\} &= \{\varphi \mid \widetilde{R}_{2_T}(\varphi,\sigma_1)\} \\ proj_\sigma\{\varphi,\sigma \mid \widehat{E_1}(\sigma_1,\sigma) \wedge \widehat{R}_2(\varphi,\sigma)\} &= \{\varphi \mid \widetilde{R}_{2_E}(\varphi,\sigma_1)\} \end{aligned}$$

L'équation 5.9 peut alors se réécrire :

$$\{\varphi \mid \widetilde{R}_{2_T}(\varphi,\sigma_1)\} \subseteq \{\varphi \mid \widetilde{R}_{2_E}(\varphi,\sigma_1)\}$$

La région obtenue en appliquant la transformation inverse du transformeur est donc incluse dans celle obtenue en appliquant la transformation inverse induite par les effets sur les variables scalaires entières.

□

5.1.3 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 va consister 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 cette itération, y compris l'indice de boucle. Dans les exemples de la figure 5.3, la variable `K` est modifiée, et 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[2]. Ainsi, dans l'exemple *a)* de la figure 5.3, le transformeur de la boucle (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, on obtient la région R_A suivante :

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

`K` représente ici la valeur de la variable `K` avant exécution de la boucle. Par contre, `I` représente la valeur de la variable `I` au cours d'une itération. Cette région n'a donc pas grande signification. C'est juste un intermédiaire de calcul qui n'est montré ici que dans un souci de clarification.

2. Il faut sélectionner les options *tffull* ou *tfinterfull* de PIPS pour obtenir ces relations ; avec les autres options, le prédicat du transformeur ne contient aucune équation.

```
      K = FOO()

C  T(I,K) {K#INIT+I==1+K}
C  <A(PHI1)-W-MUST-{1+K<=PHI1, PHI1<=N+K, 10==N, 5==M, 3==L}>

      DO I = 1, N

C  regions du corps de la boucle
C  <A(PHI1)-W-MUST-{PHI1==K+1, 1<=I, I<=N, 10==N, 5==M, 3==L}>

         K = K+1
         A(K) = B(K)
      ENDDO
```

a) variable modifiée avant l'accès à un élément de tableau.

```
      K = FOO()

C  T(I,K) {K#INIT+I==1+K}
C  <A(PHI1)-W-MUST-{K<=PHI1, PHI1+1<=N+K, 3==L, 5==M, 10==N}>

      DO I = 1, N

C  regions du corps de la boucle
C  <A(PHI1)-W-MUST-{PHI1==K, I<=N, 1<=I, 10==N, 5==M, 3==L}>

         A(K) = B(K)
         K = K+1
      ENDDO
```

b) variable modifiée après l'accès à un élément de tableau.

FIG. 5.3 - *Transformers et régions d'une boucle DO.*

Reste encore à éliminer l'indice de boucle de la région R_A. 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 équations de R_A qui décrivent I : {1<=I, I<=N, PHI1==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 équations de R_A signifient : $\exists\ i \in [1,n] \mid \varphi_1 = k + i$. Mais vu de l'extérieur de la boucle, comme I est un indice de boucle, la sémantique donnée à ces équations devient la suivante : $\forall i \in [1,n]\ \ \varphi_1 = k + i$. 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 $[1,n]$. En l'éliminant, on perd de l'information.

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. la boucle est exécutée au moins une fois.

La figure 5.3 donne les régions après élimination de l'indice de boucle. On constate que le résultat est correct, que la variable K soit modifiée avant ou après l'accès à l'élément de tableau (exemples a) et b)).

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 $\cup_{must}$.

5.1.4 Régions d'une instruction conditionnelle IF

Une instruction IF est composée 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 étant exprimées dans le même état-mémoire, leur union est calculée en utilisant l'opérateur d'union $\cup_{may}$. Ces régions sont alors 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 : l'instruction conditionnelle, puis l'une des deux branches.

La figure 5.4 donne divers exemples de régions d'une instruction conditionnelle. Le premier fragment de code contient un IF à une seule branche. L'approximation de la région résultante est donc MAY car l'élément de tableau ne sera lu que si la condition est vraie à l'exécution. Un problème apparaît dans le deuxième exemple. Une région MUST est attendue, puisque c'est le même élément symbolique de tableau qui est référencé ; mais on obtient une région MAY. Ceci provient du fait que l'on ne peut pas montrer que les prédicats des régions initiales, {PHI1==N,N==3} et {PHI1==N}, sont égaux, car ils représentent des ensembles différents. Ce problème n'apparait plus dans le troisième exemple car la condition du test porte sur un réel R ; les prédicats des régions des deux branches sont donc identiques puisqu'ils représentent les mêmes éléments. Dans le quatrième fragment de programme, l'enveloppe convexe des régions des deux branches est exacte. Mais on obtient une région MAY car c'est l'opérateur $\cup_{may}$ qui est utilisé. Le résultat est correct puique lors de l'exécution on n'accèdera qu'à un seul des deux éléments de la région. Chacun des deux éléments est donc seulement potentiellement référencé. Dans le dernier exemple, un élément de tableau B(N)

```
C  une seule branche
C  <A(PHI1)-W-MAY-{PHI1==N, 4<=N}>
      IF (N.GT.3) THEN
         A(N) = N-1
      ENDIF
```

```
C  deux branches
C  <A(PHI1)-W-MAY-{PHI1==N}>
      IF (N.GT.3) THEN
         A(N) = N-1
      ELSE
         A(N) = N+1
      ENDIF
```

```
C  un reel apparait dans la condition
C  <B(PHI1)-W-MUST-{PHI1==N}>
      IF (R.GT.0) THEN
         B(N) = N/R
      ELSE
         B(N) = A(N)*R
      ENDIF

C  deux elements de tableau differents references dans
C  les deux branches
C  <A(PHI1)-W-MAY-{N<=PHI1, PHI1<=N+1}>
      IF (N.GT.3) THEN
         A(N) = N-1
      ELSE
         A(N+1) = N+1
      ENDIF

C  la variable du test est un element de tableau, mais
C  n'est pas referencee dans les branches
C  <A(PHI1)-W-MAY-{PHI1==N}>
C  <B(PHI1)-R-MUST-{PHI1==N}>
      IF (B(N).EQ.N) THEN
         A(N) = N + 1
      ENDIF

C  la variable de test est un element de tableau et
C  est referencee dans une des deux branches
C  <A(PHI1)-W-MUST-{PHI1==N}>
C  <B(PHI1)-R-MUST-{PHI1==N}>
      IF (B(N).EQ.N) THEN
         A(N) = B(N)+1
      ELSE
         A(N) = N-1
      ENDIF
```

FIG. 5.4 - *Régions d'un IF : exemples.*

apparait dans une seule des deux branches. L'approximation de la région qui lui correspond est toutefois `MUST` car il apparait également dans la condition, et sera donc toujours lu.

5.1.5 Régions d'une partie de code non structurée

Contrairement aux parties de code structurées, les parties de code non structurées ne sont pas représentées dans PIPS par un arbre, mais par un graphe de contrôle [36]. Chaque nœud du graphe est lui-même une instruction complexe (boucle, test ou séquence linéaire). Les régions de cette partie de code sont calculées en fusionnant avec l'opérateur $\cup_{may}$ les régions de tous les nœuds du graphe, qui sont obtenues grâce aux algorithmes présentés précédemment. Puis les variables qui sont modifiées sont éliminées des régions obtenues. Si le graphe ne contient qu'un seul nœud[3], aucune union et aucune élimination ne sont effectuées, et les approximations sont conservées.

5.2 Calcul interprocédural

Summary regions

L'ensemble des régions associées au corps d'une procédure est filtré en masquant les régions correspondant à des variables désactivées en sortie de procédure, et en projetant les variables locales qui ne l'auraient pas déjà été. L'ensemble de régions résultant est appelé *summary regions* : ce sont les régions qui fournissent un résumé des effets de la procédure sur les éléments de tableaux lors d'un appel. La figure 5.5 donne les *summary regions* de la procédure `SUMREG`. On remarque que les régions internes relatives à la variable dynamique `C` n'apparaissent pas dans les *summary regions*.

Conversion des régions

Les *summary regions* sont utilisées à chaque site d'appel, après substitution des arguments formels par les arguments réels lorsque c'est possible [35], aussi bien dans la référence de la région concernée (`A(PHI1)`), que dans son prédicat. Les variables globales sont renommées conformément à leur nom au niveau du site d'appel. La figure 5.6 montre la conversion de la *summary region* de la figure 5.5 qui a lieu à l'appel de la procédure `SUMREG`.

5.3 Qualité des résultats

Le calcul des régions exactes est déjà implanté dans PIPS. Sa robustesse a été testée sur les programmes du Perfect Club [13]. Son comportement s'est révélé très satisfaisant, malgré quelques imperfections au niveau du calcul interprocédural, qui ne faisait pas partie de cette étude. D'autres expériences sont encore à réaliser pour tester notamment l'efficacité des opérateurs d'union et de projection, ainsi que l'adéquation de la représentation des ensembles de tableaux sous forme de régions, dans le cadre de l'analyse des programmes scientifiques.

3. Un tel graphe est utilisé dans PIPS pour représenter le corps d'une procédure ou d'une fonction.

```
C        summary regions :
C        <A(PHI1)-W-MUST-{1<=PHI1, PHI1<=5}>
C        <B(PHI1)-W-MUST-{PHI1==1}>

       SUBROUTINE SUMREG(A,B)
       INTEGER A(5), B(5), C(5)

C        <C(PHI1)-W-MUST-{PHI1==1}>
       C(1) = 0
C        <B(PHI1)-W-MUST-{PHI1==1}>
       B(1) = 0

C        <A(PHI1)-W-MUST-{1<=PHI1, PHI1<=5}>
       DO I = 1,5
C            <A(PHI1)-W-MUST-{PHI1==I, 1<=I, I<=5}>
          A(I) = 0
       ENDDO
       END
```

Fig. 5.5 - *summary regions d'une procédure*

```
       PROGRAM APPEL
       INTEGER D(5), F(5)

C        utilisation des summary regions de SUMREG :
C        <D(PHI1)-W-MUST-{1<=PHI1, PHI1<=5}>
C        <F(PHI1)-W-MUST-{PHI1==1}>
       CALL SUMREG(D,F)

       END
```

Fig. 5.6 - *utilisation des summary regions*

Chapitre 6

Calcul de régions *IN* et *OUT*

Les régions `IN` correspondant à un fragment de programme sont les régions dont les éléments sont lus avant d'être éventuellement réinitialisés au sein de ce même fragment. Ce sont donc les éléments de tableau *importés* par l'instruction considérée. Par exemple, dans le programme suivant :

```
A(I) = A(I-1)
A(I-1) = A(I)
```

la région `IN` correspondant au tableau `A` est :

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

On remarque qu'elle ne comporte pas l'élément `A(I)`, qui est écrit avant d'être lu.

Une région `OUT` correspondant à un fragment de programme contient les éléments définis par ce fragment et utilisé dans la *suite* du programme. Ce sont donc les éléments de tableaux *vivants* ou *exportés* vers le reste du programme.

Les régions `IN` et `OUT` peuvent avoir de nombreuses applications. Nous verrons dans le chapitre suivant leur utilisation pour privatiser les portions de tableaux. Elles peuvent également servir à générer des communications car elles correspondent aux ensembles SEND et RECEIVE de la portion de programme considérée. Enfin, elles peuvent permettre d'accélérer l'analyse des dépendances, car elles ne reflètent que les dépendances directes, les autres dépendances se déduisant par transitivité.

Le calcul interprocédural des régions `IN` est similaire à celui décrit dans le chapitre précédent pour les régions `READ/WRITE`. Il consiste à traduire les régions `IN` exprimées à l'aide des paramètres formels en fonction des paramètres d'appel. Par contre, les régions `OUT` sont propagées de manière descendante sur le graphe des appels. La traduction consiste alors à exprimer les paramètres réels en paramètres formels, et à fusionner les régions `OUT` des différents sites d'appel. Nous nous limiterons, dans le cadre de cette étude, à la description du calcul intraprocédural.

Tout au long de ce chapitre, nous illustrerons notre propos à l'aide de l'exemple de la figure 6.1. Certains éléments du tableau `WORK` sont initialisés dans la première boucle `J`. Dans la deuxième, d'autres éléments sont initialisés. Enfin, dans cette même boucle, les deux ensembles d'éléments sont lus.

6.1 Régions IN

6.1.1 Régions IN d'une instruction

Les régions `IN` d'une instruction simple sont identiques aux régions `READ` décrites précédemment, sauf en ce qui concerne les appels à des procédures externes, dont le traitement

```
SUBROUTINE INIT(A,N,K,M)
INTEGER N,K,M,A(N)
DIMENSION WORK(100,100)
K = M * M
DO I = 1,N
   DO J = 1,N
      WORK(J,K) = J + K
   ENDDO
   CALL INC1(K)
   DO J = 1,N
      WORK(J,K) = J * J - K * K
      A(I) = A(I) + WORK(J,K) + WORK(J,K-1)
   ENDDO
ENDDO
END

SUBROUTINE INC1(I)
I = I + 1
END
```

FIG. 6.1 - *Procédure INIT.*

relève de l'analyse interprocédurale. Considérons une instruction de la forme $s : lhs = rhs$, rhs ne modifiant pas ses arguments. Par définition, les régions READ de cette instruction sont :

$$READ(s) = READ(rhs)$$

Or, rhs est évalué avant que l'affectation à lhs de la valeur résultante ne soit effectuée. Les accès en écriture effectués pas s ne peuvent donc intervenir avant les accès en lecture. Les régions IN correspondant à s sont donc données par :

$$IN(s) = READ(rhs) = READ(s)$$

6.1.2 Régions IN d'une séquence d'instructions

Supposons connues les régions IN de chacune des instructions composant la séquence $S_1, \ldots, S_n$. Nous nous intéressons à l'ensemble IN des éléments de tableau qui sont lus avant d'être éventuellement réinitialisés par les instructions suivantes dans l'ordre de la séquence. Par définition, et en réutilisant les opérateurs introduits dans les chapitres précédents, nous avons, pour tout $k \in [1, n-1]$:

$$IN(S_k, \ldots, S_n) = IN(S_k) \cup_{must} [\widetilde{T}_k^{-1}(IN(S_k+1, \ldots, S_n)) -_{sup} WRITE(S_k)]$$

L'opérateur $\widetilde{T}_k^{-1}$ permet ici d'exprimer les régions IN correspondant à $S_{k+1}, \ldots, S_n$ dans le même état-mémoire σ_k que les régions WRITE et IN de S_k. $\widetilde{T}_k^{-1}(IN(S_k+1, \ldots, S_n)) -_{sup} WRITE(S_k)$ représente les régions importées par la sous-séquence $S_{k+1}, \ldots, S_n$, mais non définies par l'instruction S_k. La fusion avec $IN(S_k)$ donne donc bien l'ensemble des éléments lus par la séquence $S_k, \ldots, S_n$ avant d'être redéfinis par cette même séquence.

Prenons l'exemple du corps de la deuxième boucle en `J` de la procédure `INIT` (figure 6.1), et intéressons nous aux régions `IN` correspondant au tableau `WORK`. Nous avons :

```
C S1 :
C <WORK(PHI1,PHI2))-WRITE-MUST-{PHI1==J, PHI2==K, 1<=J, J<=N, 1<=I, I<=N}>
     WORK(J,K) = J * J - K * K

C S2 :
C <WORK(PHI1,PHI2)-IN-MUST-{PHI1==J, K<=PHI2+1, PHI2<=K, 1<=J, J<=N, 1<=I, I<=N}>
     A(I) = A(I) + WORK(J,K) + WORK(J,K-1)
```

Le calcul de $IN(S_2) -_{sup} WRITE(S_1)$ donne :

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

Comme $IN(S_1) = \emptyset$, cette région est bien la région `IN` correspondant à la séquence d'instructions S_1, S_2 pour le tableau `WORK`. Nous obtenons finalement pour tous les tableaux concernés, les régions `IN` suivantes :

```
C <WORK(PHI1,PHI2)-IN-MUST-{PHI1==J, PHI2+1==K, 1<=J, J<=N, 1<=I, I<=N}>
C <A(PHI1)-IN-MUST-{PHI1==I, 1<=J, J<=N, 1<=I, I<=N}>
```

6.1.3 Régions IN d'une boucle DO

Nous suppposons connues les régions `IN` du corps de la boucle. Si des variables scalaires entières dont dépendent les expressions d'indices des tableaux sont modifiées par la boucle, on peut s'en affranchir à l'aide de l'opérateur $\widetilde{T}^{-1}_{boucle}$. Nous pouvons donc considérer que ces régions `IN` ne dépendent que de l'indice de la boucle, et de variables scalaires entières constantes sur le fragment de code considéré.

Tout se passe donc comme si le corps de la boucle était composé d'une unique instruction simple, $s(i)$, i appartenant à l'espace d'itération. Les régions `IN` et `WRITE` correspondantes seront notées $IN(i)$ et $W(i)$. La contribution de $IN(i)$ à l'ensemble $IN(boucle)$, est limitée par les écritures qui ont lieu aux itérations précédentes. Celles-ci ne seront modélisables sous forme de régions (i.e. sous forme de polyèdre convexe, et uniquement en fonction des variables du programme) que si la valeur absolue de l'incrément vaut 1 :

– si l'incrément de la boucle vaut ± 1 et si les bornes de boucles sont affines,

  si l'incrément est positif,

  $$IN(boucle) = proj_i(IN(i) -_{sup} proj_{i'}(W(i')_{i'<i})) \cup_{must} IN(i = borne\_inf)$$

  si il est négatif,

  $$IN(boucle) = proj_i(IN(i) -_{sup} proj_{i'}(W(i')_{i'>i})) \cup_{must} IN(i = borne\_inf)$$

  $IN(i) -_{sup} proj_{i'}(W(i')_{i'>i})$ représente les éléments de tableau écrits par l'itération i et non définis aux itérations précédantes, à l'exclusion de la première. L'union avec $IN(i = borne\_inf)$ permet de ne pas oublier les éléments importés par la première itération. Nous illustrerons ceci ultérieurement par un exemple.

– et dans le cas général,

  $$IN(boucle) = IN(en\_tete) \cup_{must} proj_i(IN(i)_{may})$$

  $IN(i)_{may}$ étant l'ensemble de régions $IN(i)$ dans lequel toutes les approximations sont `MAY`. On ne peut pas savoir quels sont exactement les éléments écrits aux itérations précédantes ; l'approximation la plus sure est donc de considérer que tous les éléments de l'itération considérée peuvent être exportés vers l'extérieur de la boucle ; comme c'est une éventualité, les régions `IN` résultantes sont forcément des régions `MAY`.

Nous allons maintenant illustrer les détails de l'implantation en reprennant l'exemple de la figure 6.1. Nous nous intéressons à la deuxième boucle `J` :

```
      DO J = 1, N
C regions IN et WRITE du corps de la boucle
C <WORK(PHI1,PHI2)-IN-MUST-{PHI1==J, PHI2+1==K, 1<=J, J<=N, 1<=I, I<=N}>
C <WORK(PHI1,PHI2))-W-MUST-{PHI1==J, PHI2==K, 1<=J, J<=N, 1<=I, I<=N}>
C <A(PHI1)-IN-MUST-{PHI1==I, 1<=J, J<=N, 1<=I, I<=N}>
C <A(PHI1)-W-MUST-{PHI1==I, 1<=J, J<=N, 1<=I, I<=N}>
      ENDDO
```

Nous allons tout d'abord calculer les régions `IN` correspondant au tableau `WORK`. La variable j' sera notée `J#INT`. Nous avons :

```
IN(J) = {PHI1==J,PHI2+1==K,1<=J,J<=N, <=I,I<=N}, MUST
W(J#INT) = {PHI1==J#INT,PHI2==K,1<=J#INT,J#INT<=N,1<=I,I<=N,J#INT+1<=J}, MUST
```

L'inéquation `J#INT+1<=J` dans `W(J#INT)` correspond à $j' < j$. Calculer la différence de ces deux régions n'aurait aucun sens car elles ne sont pas exprimées sur le même domaine. Le domaine de `IN(J)` ne comprend pas la variable `J#INT` ; le domaine de `W(J#INT)` comporte bien la variable `J`, mais il n'est pas restreint par {`1<=J, J<=N`}. Il faut donc ajouter à `W(J#INT)` les contraintes {`1<=J, J<=N`} et à `IN(J)` les contraintes {`1<=J#INT, J#INT<=N, J#INT+1<=J`}. On peut alors calculer la différence `IN(J) - W(J#INT)` qui vaut, après élimination de `J#INT` :

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

On calcule ensuite l'union avec `IN(J=1)` :

```
IN(J=1) = {PHI1==J,PHI2+1==K,J==1,1<=I,I<=N}, MUST
```

Le résultat est alors :

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

Cette région n'est pas différente de celle obtenue par le calcul de `IN(J) - W(J#INT)`. Pour se convaincre de la nécessité de rajouter les éléments de `IN(J=1)`, détaillons le calcul des régions `IN` correspondant au tableau `A`. Nous savons que :

```
IN(J) = {PHI1==I,1<=J,J<=N,1<=I,I<=N,1<=J#INT,J#INT<=N,J#INT+1<=J}, MUST
W(J#INT) = {PHI1==I,1<=J#INT,J#INT<=N,1<=J,J<=N,1<=I,I<=N,J#INT+1<=J}, MUST
```

On constate que `IN(J) = W(J#INT)`, et donc que la différence `IN(J) - W(J#INT)` vaut :

```
C <A(PHI1)-IN-MUST-{0==1}>
```

Cet ensemble ne contient aucun élément du tableau. Pourtant, il est aisé de se convaincre que le résultat attendu est :

```
C <A(PHI1)-IN-MUST-{PHI1==I,1<=J,J<=N,1<=I,I<=N}>
```

Ceci provient du fait que lorsque l'on calcule `IN(J) - W(J#INT)`, la première valeur de `J` n'est pas prise en compte. Dans l'exemple précédent, les contraintes sur les valeurs de `J` et `J#INT` sont données par le système :

```
1<=J, J<=N,
1<=J#INT, J#INT<=N
J#INT+1<=J
```

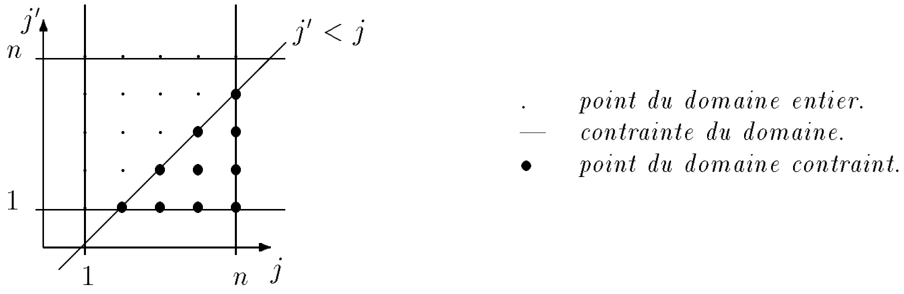


FIG. 6.2 - *Domaine de résolution.*

La figure 6.2 montre bien que les valeurs de `J` sont contraintes par {`2<=J, J<=N`} et non par {`1<=J, J<=N`}. Il faut donc ajouter à l'ensemble `IN(J) = W(J#INT)` les points de l'ensemble `IN(1)`. Après élimination de `I`, on obtient bien la région :

```
C <A(PHI1)-IN-MUST-{PHI1==I,1<=J,J<=N,1<=I,I<=N}>
```

Finalement, les régions `IN` de cette boucle sont :

```
C <WORK(PHI1,PHI2)-IN-MUST-{1<=PHI1, PHI1<=N, PHI2+1==K, 1<=J, J<=N, 1<=I, I<=N}>
C <A(PHI1)-IN-MUST-{PHI1==I, 1<=I, I<=N}>
```

Les régions `IN` complètes de l'exemple de la figure 6.1 sont données par la figure 6.3. On constate que la boucle `I` n'importe aucun des éléments du tableau `WORK` car la région `IN` correspondante (`WORK(PHI1,PHI2)-IN-MUST-`{`1==0`}) est vide. Si les régions `IN` du corps de cette boucle figuraient également sur cette figure, on constaterait qu'il en est de même pour le corps de la boucle. On pourrait donc déclarer le calcul de cette portion de tableau comme étant local à chaque itération.

6.1.4 Régions IN d'une instruction conditionnelle IF

On suppose connues les régions `IN` de chacune des branches (vraie et fausse) de l'instruction `IF`. L'évaluation de l'une de ces deux branches ne pouvant précéder celle de l'autre, les régions `IN` de la deuxième ne peuvent être écrasés par les régions `WRITE` de la première. On a donc comme pour les régions `READ` :

$$IN(test) = IN(condition) \cup_{must} (IN(vraie) \cup_{may} IN(fausse))$$

6.1.5 Régions IN d'une partie de code non structurée

Nous supposons connues les régions `IN` de chacun des nœuds du graphe de l'*unstructured*. Il n'est pas possible de calculer les régions `IN` comme les régions `READ`, car l'ordre dans lequel les nœuds seront exécutés n'est pas connu *a priori*, et donc nous ne savons pas quelles sont les écritures qui précèdent les lectures. Dans un tel contexte, nous avons choisi la solution la plus simple, qui consiste à calculer les régions `IN` de la même manière que les régions `READ` mais à partir des régions `IN` des nœuds, puis à les transformer systématiquement en régions `MAY`. L'ensemble trouvé contient ainsi toutes les éléments qui peuvent éventuellement être en entrée de l'*unstructured*. Nous pourrions envisager une stratégie dans laquelle plus d'information serait préservée, mais le coût en calcul risquerait d'être très important par rapport au bénéfice que l'on en retirerait, les fragments de code non-structurés étant très

```
C <A(PHI1)-IN-MUST-{1<=PHI1, PHI1<=N}>
      SUBROUTINE INIT(A,N,K,M)
      INTEGER N,K,M,A(N)
      DIMENSION WORK(100,100)
      K = M * M

C <WORK(PHI1,PHI2)-IN-MUST-{1==0}>
C <A(PHI1)-IN-MUST-{1<=PHI1, PHI1<=N}>
      DO I = 1,N
         DO J = 1,N
            WORK(J,K) = J + K
         ENDDO
         CALL INC1(K)

C <WORK(PHI1,PHI2)-IN-MUST-{1<=PHI1, PHI1<=N, PHI2+1==K, 1<=J, J<=N, 1<=I, I<=N}>
C <A(PHI1)-IN-MUST-{PHI1==I, 1<=I, I<=N}>
         DO J = 1,N
            WORK(J,K) = J * J - K * K
            A(I) = A(I) + WORK(J,K) + WORK(J,K-1)
         ENDDO
      ENDDO
      END
```

FIG. 6.3 - *Régions IN de la procédure INIT.*

difficiles voire impossibles à paralléliser. L'équation permettant de calculer les régions `IN` d'un *unstructured* est finalement :

$$IN(unstructured) = (\cup_{may}\ proj_{\sigma_u}(IN(noeuds)))_{may}$$

σ_u étant l'état-mémoire précédant l'exécution de l'*unstructured*, l'utilisation de l'opérateur $\cup_{may}$ n'ayant de sens que lorsque toutes les régions sont exprimées dans le même état-mémoire. Le dernier *may* signifie que les régions résultantes sont toutes des régions `MAY` quelque soit le résultat donné par l'opérateur $\cup_{may}$.

6.2 Régions OUT

Les régions `OUT`, rappelons-le, sont les régions définies par le fragment de code considéré, et qui seront lues ultérieurement dans le déroulement du programme: ce sont les régions *exportées*. L'intuition voudrait alors que ces régions soient exprimées dans l'état-mémoire succédant la portion de code considérée. Trois arguments s'opposent à ce choix :

1. tout d'abord, un argument de confort pour l'utilisateur, qui préférera voir les régions `OUT` exprimées dans le même état-mémoire que les régions `IN` ou `READ/WRITE` correspondant à la même structure du programme, pour éviter d'avoir à faire de tête les transformations de variables nécessaires à leur comparaison ;

2. le deuxième argument ressemble au premier, mais du point de vue du programmeur ; en effet, comme nous le verrons par la suite, les régions `OUT` sont définies en fonction des régions `IN` et `WRITE`; il sera beaucoup plus facile de les combiner si elles sont exprimées dans le même état-mémoire; et le travail sera également facilité pour les phases ultérieure de PIPS qui utiliseront ces différents types de régions ;

3. enfin, passer d'un état-mémoire à un autre est certes coûteux, mais pas impossible ; une phase ayant besoin des régions `OUT` exprimées dans l'état-mémoire suivant la structure à laquelle elles correspondent pourra toujours traduire les régions `OUT` déjà calculées.

Une autre remarque s'impose. Contrairement aux régions `READ/WRITE`, les régions `OUT` sont propagées de la racine du graphe de contrôle hiérarchisé vers les feuilles. Par exemple, les régions `OUT` des instructions composant une séquence d'instruction seront calculées à partir des régions `OUT` correspondant à cette dernière. Nous n'aurons donc pas, par exemple, à décrire le calcul des régions d'une instruction, mais celui des régions `OUT` d'une instruction faisant partie d'une séquence linéaire.

6.2.1 Régions OUT des instructions d'une séquence

Conformément aux remarques précédentes, nous supposons connues les régions `OUT` correspondant à la séquence linéaire, et exprimées dans l'état-mémoire précédant celle-ci. Nous cherchons les régions `OUT` associées à chacune des instructions composant la séquence. Ces régions sont les régions écrites par l'instruction considérée, et exportées soit vers l'extérieur de la séquence, soit vers les instructions suivantes dans l'ordre de la séquence, *exportées* signifiant *utilisées avant d'être éventuellement redéfinies.*

Formulation du problème

Soit la séquence linéaire B, constituée des instructions $S_1, S_2, \ldots, S_n$. Soient $P_1, \ldots, P_n$, $T_1, \ldots, T_n$, $W_1, \ldots, W_n$ et $IN_1, \ldots, IN_n$ les préconditions, les transformeurs et les régions `WRITE` et `IN` associés aux instructions $S_1, \ldots, S_n$ de B.

On cherche à calculer les régions $OUT_1, \ldots, OUT_n$ associées aux instructions $S_1, \ldots, S_n$ de B.

Définition 6.1 *Soit R_k une région exprimée dans l'état-mémoire σ_k.*

De même que l'on avait défini la transformation inverse de T_k, on définit la transformation associée à T_k, *$\widetilde{T}_k$ par :*

$$\widetilde{T}_k : R_k \rightarrow \widetilde{R}_{k+1} = proj_{\sigma_{k+1}}(intersection(R_k, T_k))$$

$\widetilde{R}_{k+1}$ est la projection sur l'état-mémoire σ_{k+1} de la région dont le prédicat est l'intersection du prédicat initial de R_k 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. Elle est effectuée à l'aide de l'algorithme 4.5.

Algorithme 6.1 *Les équations récursives suivantes fournissent un algorithme permettant de calculer les régions OUT_k ($k \in [1, n]$).*

On note OUT_B les régions `OUT` associées à B, et exprimées dans l'état-mémoire précédant l'exécution de la séquence.

On introduit IN'_k les régions `IN` associées à $S_k, \ldots, S_n$ (dont le calcul a été présenté page 64), ainsi que OUT'_k ($k \in [1, n]$), la restriction à $S_1, \ldots, S_k$ des régions OUT_B et exprimées dans l'état-mémoire σ_k précédant S_k.

$$\begin{cases} OUT'_n = \widetilde{T}_n^{-1}(\widetilde{T}_B(OUT_B)) \\ OUT_n = W_n \cap OUT'_n \\ OUT'_k = \widetilde{T}_k^{-1}(OUT'_{k+1} -_{sup} W_{k+1}),\ si\ 1 \leq k < n \\ OUT_k = W_k \cap [\, OUT'_k \cup_{must} \widetilde{T}_k^{-1}(IN'_{k+1})\,],\ si\ 1 \leq k < n \end{cases}$$

Avec :

$$\begin{cases} IN'_n = IN_n \\ IN'_k = IN_k \cup_{must} [\, \widetilde{T}_k^{-1}(IN'_{k+1}) -_{sup} W_k \,],\ si\ 1 \leq k < n \end{cases}$$

Démonstration : Il s'agit de démontrer que les régions OUT'_k et OUT_k calculées sont bien celles recherchées.

OUT_B est l'ensemble de régions `OUT` associées à B, et donc à $S_1, \ldots, S_N$, et exprimées dans l'état-mémoire précédant l'exécution de la séquence. $\widetilde{T}_B(OUT_B)$ represente les régions `OUT` de $S_1, \ldots, S_N$ dans l'état-mémoire σ_{k+1} suivant B, car $\widetilde{T}_B$ est la transformation associée au transformeur de la séquence B. L'opérateur $\widetilde{T}_n^{-1}$ permettant de traduire une région exprimée dans l'état σ_{n+1} en une région similaire exprimée dans l'état σ_n, $\widetilde{T}_n^{-1}(\widetilde{T}_B(OUT_B))$ est donc l'ensemble de régions associées à $S_1, \ldots, S_N$, mais exprimées dans l'état-mémoire σ_n précédant S_n, ce qui est la définition de OUT'_n. Nous avons donc bien :

$$OUT'_n = \widetilde{T}_n^{-1}(\widetilde{T}_B(OUT_B))$$

La restriction à la sous-séquence $S1, \ldots, S_k$ des régions `OUT` de B , sont les régions exportées à l'extérieur de B par la séquence $S1, \ldots, S_{k+1}$, moins les régions déjà exportées par l'instruction S_{k+1} ; le tout exprimé dans l'état-mémoire σ_k, soit :

$$OUT'_k = \widetilde{T}_k^{-1}(OUT'_{k+1} -_{sup} W_{k+1})$$

Enfin, les régions `OUT` de l'instruction S sont les régions écrites par S et exportées vers l'extérieur de B, $W_k \cap OUT'_k$, auxquelles il faut ajouter les régions écrites par S et exportées vers $S_{k+1}, \ldots, S_n$, soit $W_k \cap \widetilde{T}_k^{-1}(IN'_{k+1})$. Ce qui donne finalement :

$$OUT_k = W_k \cap [\, OUT'_k \cup_{must} \widetilde{T}_k^{-1}(IN'_{k+1}) \,]$$

□

Reprenons l'exemple de la procédure `INIT` (figure 6.1), et supposons que les régions `OUT` de la séquence d'instructions constituant le corps de la boucle la plus externe sont :

```
<A(PHI1)-OUT-MUST-{PHI1==I}>
```

`N` n'étant pas modifié par la séquence, l'expression de cette région dans l'état-mémoire précédant la dernière instruction de la séquence (i.e. la deuxième boucle en `J`) ne change pas. Les régions `WRITE` de cette instruction étant :

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

on en déduit, à l'aide de l'équation $OUT_n = OUT_3 = W_n \cap OUT'_n$, les régions `OUT` associées à cette instruction :

```
      <A(PHI1)-W-MUST-{PHI1==I}>
inter <A(PHI1)-OUT-MUST-{PHI1==I}>
=     <A(PHI1)-OUT-MUST-{PHI1==I}>
```

ce qui correspond bien au fait que le tableau `WORK` n'est pas exporté par cette deuxième boucle en `J` vers le reste du programme. La contribution des instructions précédentes aux régions `OUT` de la séquence est égale à l'ensemble vide ($OUT'_1 = OUT'_2 = \emptyset$) puisque toutes les régions exportées vers l'extérieur de la séquence sont définies dans cette dernière instruction. De plus, les régions `IN` associées aux deux dernières instructions, et exprimées dans l'état-mémoire précédant la première boucle en `J`, soit $\widetilde{T}_1^{-1}(IN'_2)$, sont :

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

Les régions `WRITE` de la première instruction de la séquence étant :

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

ses régions `OUT`, calculées à l'aide de l'équation $OUT_1 = W_1 \cap [OUT_1' \cup_{must} \widetilde{T}_1^{-1}(IN_2')]$, sont finalement :

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

6.2.2 Régions OUT du corps d'une boucle DO

Étant données les régions `OUT` de la boucle, nous cherchons à calculer les régions `OUT` correspondant au corps de la boucle. Le corps de la boucle représentant une itération, ses régions `OUT` seront les régions définies au cours de cette itération et exportées soit vers l'extérieur de la boucle, soit vers les itérations ultérieures.

Nous pouvons en déduire l'équation permettant de calculer les régions $OUT(i)$ du corps d'une boucle, i appartenant à l'espace d'itération, et connaissant les régions OUT_B de la boucle et les régions `WRITE` du corps de celle-ci ($W(i)$) :

– si l'incrément de la boucle vaut ± 1 et si les bornes de boucles sont affines,

si l'incrément est positif,

$$OUT(i) =$$
$$\{\, [W(i) -_{sup} proj_{i'}(W(i')_{i'>i})] \,\cap\, \widetilde{T}_B(OUT_B) \,\} \,\cup_{must}$$
$$\{\, W(i) \,\cap\, [\, proj_{i'}(\, IN(i')_{i'>i} -_{sup} proj_{i''}(W(i'')_{i<i''<i'})\,) \,\cup_{must}\, IN(i+1) \,]\,\}$$

si il est négatif,

$$OUT(i) =$$
$$\{\, [W(i) -_{sup} proj_{i'}(W(i')_{i'<i})] \,\cap\, \widetilde{T}_B(OUT_B) \,\} \,\cup_{must}$$
$$\{\, W(i) \,\cap\, [\, proj_{i'}(\, IN(i')_{i'<i} -_{sup} proj_{i''}(W(i'')_{i'<i''<i})\,) \,\cup_{must}\, IN(i-1) \,]\,\}$$

remarque : les projections utilisées sont les mêmes que pour un indice de boucle.

La figure 6.4 donne une vision intuitive de cette équation. La boucle B y est représentée déroulée, ainsi que certaines itérations, désignées par i, i' et i''. Verticalement se trouvent les régions correspondant à un tableau à une dimension et six éléments ; à chaque fois, l'intégralité du tableau est représentée, seuls les éléments grisés faisant partie de la région.

Le schéma du haut illustre la première partie de l'équation, c'est-à-dire le calcul des régions définies par une itération i et exportées vers l'extérieur de la boucle. Si un élément est défini à une itération i, et si cette définition est masquée par une redéfinition à une itération i' postérieure, alors l'itération i ne peut exporter cet élément vers l'extérieur de la boucle. Ici c'est le cas pour le 4-ième élément du tableau. Enfin, pour que l'élément soit réellement exporté vers l'extérieur de la boucle, il faut qu'il soit exporté par la boucle ; c'est pourquoi nous avons fait figurer les régions $OUT(B) = OUT_B$ de la boucle sur le schéma.

Le shéma du bas détaille la deuxième partie de l'équation, à savoir le calcul des régions définies par une itération i et exportées vers une itération i' postérieure. Comme précédemment, un élément ne peut être exporté d'une itération i vers une itération i' que s'il n'est pas redéfini par une itération i'' située entre i et i'.

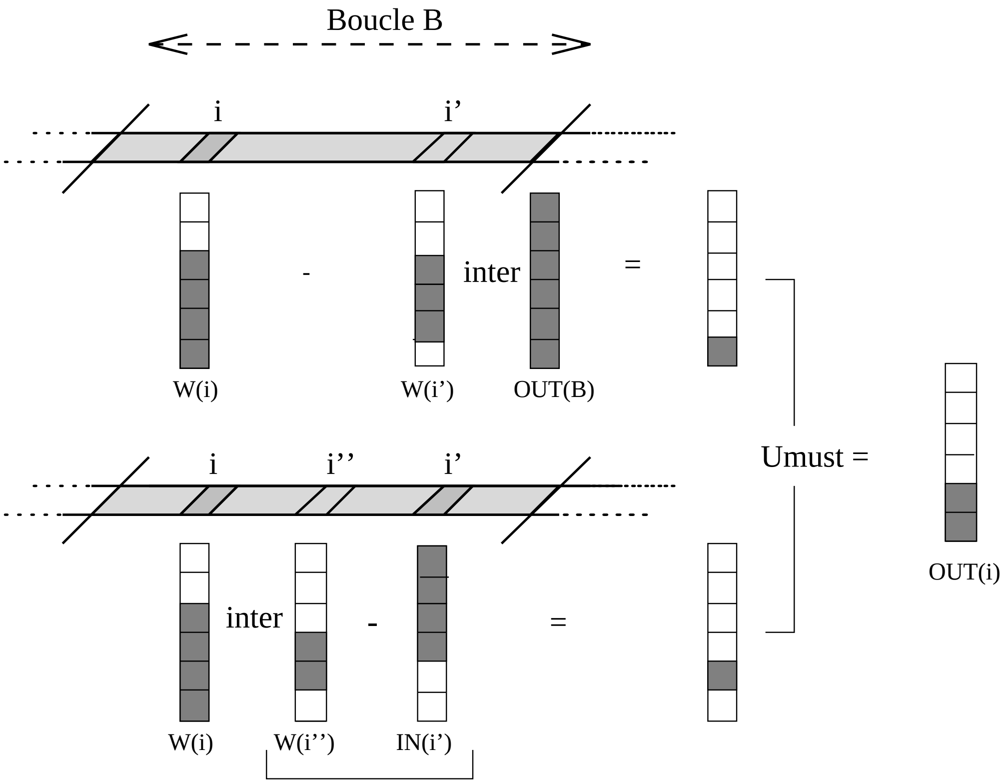


FIG. 6.4 - *Régions OUT d'une itération i d'une boucle B.*

Nous allons maintenant détailler les différents éléments de la première équation.

$W(i) -_{sup} proj_{i'}(W(i')_{i'<i})$ représente les éléments de tableau écrits par l'itération i et non redéfinis aux itérations suivantes. $\widetilde{T}_B(OUT_B)$ est l'expression dans l'état-mémoire précédant le corps de la boucle de OUT_B. $(W(i) -_{sup} proj_{i'}(W(i')_{i'<i})) \cap \widetilde{T}_B(OUT_B)$ contient donc les éléments définis dans l'itération i et exportés vers l'extérieur de la boucle.

$IN(i')_{i'<i} -_{sup} proj_{i''}(W(i'')_{i'<i''<i})$ représente les éléments importés par l'itération i' mais non définis aux itérations i'' suivant i et précédant i'. L'ensemble $proj_{i'}(IN(i')_{i'<i} -_{sup} proj_{i''}(W(i'')_{i<i''<i'})) \cup_{must} IN(i+1)$ contient donc les éléments lus par les itérations suivant celle considérée, et venant soit de cette itération et des précédentes, soit de l'extérieur de la boucle. Il est nécessaire d'ajouter les éléments de $IN(i+1)$ car i' est contraint par $i' < i'' < i$ et ne peut prendre la valeur $i+1$ (voir le calcul des régions `IN` d'une boucle `DO` pour plus de précisions). L'intersection avec $W(i)$ ne conserve que les éléments définis dans l'itération i, qui sont donc ceux exportés vers les itérations suivantes.

– et dans le cas général,

$$OUT(i) = (\widetilde{T}_B(OUT_B) \cap W(i))_{may}$$

On ne peut pas savoir quels sont exactement les éléments écrits ou importés par les itérations suivantes ; l'approximation la plus sûre est donc de considérer que tous les éléments de l'itération considérée peuvent être exportés vers l'extérieur de la boucle ; comme c'est une éventualité, les régions `OUT` résultantes sont forcément des régions `MAY`.

Remarque : Au prédicat de $\widetilde{T}_B(OUT_B)$, il est nécessaire d'ajouter les préconditions propres de la boucle, qui sont les contraintes sur le domaine de variation de celui-ci, pour que les opérations ultérieures sur les polyèdres aient un sens.

Reprenons notre exemple de la figure 6.1, et supposons que les régions `OUT` de la boucle la plus externe sont :

```
<A(PHI1)-OUT-MUST-{1<=PHI1, PHI1<=N}>
```

Le transformeur de la boucle est `T(I,K){K#INIT+I==1+K}`. En appliquant à la région `OUT` précédente la transformation directe associée à ce transformeur, et en ajoutant les préconditions propres de la boucle, soit `{1<=I, I<=N}`, nous obtenons :

```
<A(PHI1)-OUT-MUST-{1<=PHI1, PHI1<=N, 1<=I, I<=N}>
```

Les régions `WRITE` de ce corps de boucle sont :

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

Seule la région concernant `A` nous intéresse, car aucun élément du tableau `WORK` n'est exporté par la boucle vers le reste du programme, ou importé par les autres itération. Calculons $W(i) -_{sup} proj_{i'} W(i')_{i'>i}$. i' étant représenté par `I#INT`, $W(i')_{i'>i}$ vaut :

```
<A(PHI1)-W-MUST-{PHI1==I#INT, 1<=I, I<=N, 1<=I#INT, I#INT<=N, I+1<=I#INT}>
```

l'élimination de `I#INT` donne :

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

et $W(i) -_{sup} proj_{i'} W(i')_{i'>i}$ vaut finalement :

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

Donc $(W(i) -_{sup} proj_{i'} W(i')_{i'>i}) = W(i)$. Il faut maintenant calculer $W(i) \cap \widetilde{T}_B(OUT_B)$:

```
      <A(PHI1)-W-MUST-{PHI1==I, 1<=I, I<=N}>
inter <A(PHI1)-OUT-MUST-{1<=PHI1, PHI1<=N, 1<=I, I<=N}>
=     <A(PHI1)-OUT-MUST-{PHI1==I, 1<=I, I<=N}>
```

i'' étant représenté par `I#INT2`, $W(i'')_{i<i''<i'}$ s'exprime facilement à partir de $W(i)$:

```
<A(PHI1)-W-MUST-{PHI1==I#INT2, 1<=I, I<=N, 1<=I#INT, I#INT<=N,
                 1<=I#INT2, I#INT2<=N, I+1<=I#INT2, I#INT2+1<=I#INT}>
```

en éliminant `I#INT2`, nous obtenons :

```
<A(PHI1)-W-MUST-{1<=I, I<=N, 1<=I#INT, I#INT<=N,
                 1<=PHI1, PHI1<=N, I+1<=PHI1, PHI1+1<=I#INT}>
```

Les régions `IN` du corps de la boucle sont, d'après la section précédente :

```
<A(PHI1)-IN-MUST-{PHI1==I, 1<=I, I<=N}>
```

$IN(i')_{i'>i} -_{sup} proj_{i''}(W(i'')_{i<i''<i'}))$ vaut alors :

```
  <A(PHI1)-IN-MUST-{PHI1==I#INT, 1<=I, I<=N,
                    1<=I#INT, I#INT<=N, I+1<=I#INT}>
- <A(PHI1)-W-MUST-{1<=I, I<=N, 1<=I#INT, I#INT<=N,
                   1<=PHI1, PHI1<=N, I+1<=PHI1, PHI1+1<=I#INT}>
= <A(PHI1)-IN-MUST-{0==1}>
```

l'équation `0==1` permettant de représenter la région vide. Il faut ensuite éliminer `I#INT`, ce qui est immédiat, et ajouter $IN(i+1)$, soit :

```
      <A(PHI1)-IN-MUST-{0==1}>
Umust <A(PHI1)-IN-MUST-{PHI1==I+1, 1<=I, I<=N}>
=     <A(PHI1)-IN-MUST-{PHI1==I+1, 1<=I, I<=N}>
```

Les régions exportées vers les itérations suivantes sont alors celles qui sont également définies par l'itération considérée :

```
      <A(PHI1)-W-MUST-{PHI1==I, 1<=I, I<=N}>
inter <A(PHI1)-IN-MUST-{PHI1==I+1, 1<=I, I<=N}>
=     <A(PHI1)-OUT-MUST-{0==1}>
```

Les régions `OUT` du corps de la boucle la plus externe sont donc finalement :

```
      <A(PHI1)-OUT-MUST-{PHI1==I, 1<=I, I<=N}>
Umust <A(PHI1)-OUT-MUST-{0==1}>
=     <A(PHI1)-OUT-MUST-{PHI1==I, 1<=I, I<=N}>
```

On retrouve la région `OUT` que l'on avait supposée être exportée par le corps de cette boucle dans l'exemple choisi pour illustrer le calcul des régions `OUT` d'une séquence d'instructions.

Les régions `OUT` complètes de l'exemple de la figure 6.1 sont données par la figure 6.5. On constate que la boucle `I` n'exporte aucun des éléments du tableau `WORK` car les régions `OUT` correspondantes ne comprennent pas d'éléments de `WORK`. Si les régions du corps de cette boucle figuraient également sur cette figure, on constaterait qu'il en est de même pour le corps de la boucle. Dans le cadre de la privatisation de tableau, cela signifie qu'il serait inutile de mettre à jour le tableau global à partir des valeurs des tableaux locaux.

```
C <A(PHI1)-OUT-MUST-{1<=PHI1, PHI1<=N}>
      SUBROUTINE INIT(A,N,K,M)
      INTEGER N,K,M,A(N)
      DIMENSION WORK(100,100)
      K = M * M

C <A(PHI1)-OUT-MUST-{1<=PHI1, PHI1<=N}>
      DO I = 1,N

C regions out du corps de la boucle en I:
C <A(PHI1)-OUT-MUST-{PHI1==I, 1<=I, I<=N}>

C <WORK(PHI1,PHI2)-OUT-MUST-{1<=PHI1, PHI1<=N, PHI2==K}>
         DO J = 1,N
C <WORK(PHI1,PHI2)-OUT-MUST-{PHI1==J, PHI2==K}>
            WORK(J,K) = J + K
         ENDDO

         CALL INC1(K)

C <A(PHI1)-OUT-MUST-{PHI1==I, 1<=I, I<=N}>
         DO J = 1,N

C regions out du corps de la boucle en J:
C <A(PHI1)-OUT-MUST-{PHI1==I, 1<=I, I<=N}>

C <WORK(PHI1,PHI2)-OUT-MUST-{PHI1==J, PHI2==K}>
            WORK(J,K) = J * J - K * K
C <A(PHI1)-OUT-MUST-{PHI1==I, 1<=I, I<=N}>
            A(I) = A(I) + WORK(J,K) + WORK(J,K-1)
         ENDDO
      ENDDO
      END
```

FIG. 6.5 - *Régions OUT de la procédure INIT.*

6.2.3 Régions OUT des branches d'une instruction conditionnelle IF

Les régions `OUT` des branches d'un test `IF` sont beaucoup plus simples à calculer que les régions `OUT` précédentes. Supposons connues les régions `OUT` du `IF`. Les régions `OUT` d'une des deux branches sont les régions exportées par le `IF` et définies dans cette branche. Les équation permettant de calculer ces régions sont donc trivialement :

$$OUT(vraie) = OUT(test) \cap WRITE(vraie)$$
$$OUT(fausse) = OUT(test) \cap WRITE(fausse)$$

Remarque : Dans chacun des cas, il faut rajouter à $OUT(test)$ les préconditions de la branche concernée, pour que l'opération d'intersection ait un sens.

6.2.4 Régions OUT des nœuds d'une partie de code non structurée

Nous supposons connues les régions `OUT` associées à l'*unstructured*. Nous cherchons à calculer les régions `OUT` de chacun des œuds. Nous ignorons dans quel ordre ces nœuds seront parcourus à l'exécution. Donc, pour un nœud donné, il est impossible de calculer les régions qui seront exportées et importées par les nœuds suivants. Ainsi, nous ne pouvons calculer qu'une approximation des régions `OUT` associées à chaque nœud, à savoir un sur-ensemble des régions exportées vers l'extérieur de la structure. De plus, les régions `OUT` de l'*unstructured* sont exprimées dans l'état-mémoire précédant celui-ci, et non dans l'état-mémoire précédant le nœud considéré. Il faut donc éliminer de ces régions toutes les variables susceptibles d'être modifiées par l'*unstructured*, opération que nous noterons $proj_u(OUT(unstructured))$. L'équation permettant de calculer les régions recherchées est finalement :

$$OUT(n\oe ud) = (proj_u(OUT(unstructured)) \cap WRITE(n\oe ud))_{may}$$

6.2.5 Propagation interprocédurale

Les régions `OUT` d'un appel à une procédure sont les régions définies par celle-ci, et qui sont utilisées dans la suite du programme, c'est-à-dire dans les instructions qui suivent le site d'appel. Leur calcul est donc intraprocédural, mais s'appuie sur le résultat de l'analyse interprocédurale des régions `WRITE`.

Les régions `OUT` d'une procédure sont les régions définies par cette procédure et exportées vers la suite du programme, qui est ici constituée de tous les chemins du graphe de contrôle suivants les sites d'appels. Ces régions sont donc l'union des régions `OUT` des sites d'appel (analyse interprocédurale descendante). C'est une union *may* car un élément exporté à un site d'appel peut ne pas l'être à un autre site d'appel. Par conséquent, les régions `OUT` d'une procédure risquent d'être souvent des régions `MAY`. Une solution consiste alors à clôner [31, 43] la procédure concernée, pour en avoir autant de versions que de *contextes d'appel*, un contexte d'appel étant ici constitué par les valeurs des variables scalaires entières, mais également par les éléments contenus dans les régions `OUT`. L'avantage de cette approche est que l'on évite d'effectuer les unions génératrices d'approximations `MAY`, mais ceci se fait au détriment de la taille du programme à analyser, sans toutefois atteindre les proportions de l'expansion totale de la procédure qui a également l'inconvénient de détruire la structure du programme.

6.3 Qualité du calcul des régions IN et OUT

Le calcul des régions `IN` tel qu'il a été décrit précédemment est déjà implanté. Nous ne l'avons testé que sur de petits exemples mettant en relief ses caractéristiques. Celles-ci laissent présager de bons résultats dans la pratique, ce qui reste à confirmer par des expériences sur des programmes de taille réelle.

Le calcul des régions `OUT` n'est pas encore implanté. Et la complexité des équations permettant de les calculer est telle qu'il ne nous est pas possible de prévoir la qualité des résultats que nous pourrons obtenir. Ceux-ci dépendront en grande partie de la précision des opérateurs permettant de manipuler les régions et de leur capacité à préserver les approximations des régions initiales. Ils dépendront également de l'adéquation de la représentation sous forme de polyèdre. Enfin, nous venons de voir que la qualité du calcul interprocédural dépend de l'unicité des contextes d'appels, que seul le clônage permet de garantir, mais qui n'est pas effectué de manière automatique par PIPS.

Chapitre 7

Privatisation de tableau

Une variable privée est une variable qui n'est référencée que par la portée lexicale à laquelle elle appartient. Privatiser une section de tableau pour un fragment de programme donné consiste donc à déterminer si toute référence en lecture à un élément de la section est précédée par une référence en écriture dans le même fragment de programme. Si la dernière définition d'un élément est encore vivante à l'issue de l'exécution du fragment de programme, il convient également de générer la mise à jour de l'élément correspondant de la variable globale originelle.

Nous allons, dans ce chapitre, nous limiter à la privatisation des sections de tableaux dans les boucles, dans le but d'éliminer des dépendances entre itérations. Ainsi, ne sera pas considérée comme privatisable une section dont le calcul serait local à une seule itération. La détection de ces sections pourrait certes permettre un certaine nombre d'optimisations comme la scalarisation, ou l'allocation en registre des éléments privatisés. Mais leur calcul est plus simple que celui que nous allons effectuer, et s'en déduit aisément.

Ce chapitre est organisé de la façon suivante. Nous formulons tout d'abord le problème que pose la privatisation des sections de tableaux dans une boucle. Nous proposons ensuite un algorithme permettant de le résoudre, puis nous l'illustrons en reprenant l'exemple de la figure 5.1.

7.1 Formulation du problème

Nous avons vu page 21 que la détermination des régions de tableaux privatisables pour une boucle posait deux problèmes.

Le premier est la détermination des candidats à la privatisation, c'est-à-dire des éléments de tableaux dont toutes les lectures sont précédées par une écriture à la même itération. Les *bons* candidats sont alors ceux dont le calcul est local à plusieurs itérations (sinon la privatisation n'élimine aucune dépendance), et qui ne sont importés par aucune autre itération (sinon la parallélisation détruirait le déterminisme de l'exécution séquentielle).

Le deuxième problème apparait lorsque des éléments des régions privatisées sont utilisés en dehors de la boucle considérée. Il faut alors déterminer quelles sont les itérations qui mettront à jour les éléments du tableau global à l'aide des valeurs privées qu'elles possèdent. Si ce dernier problème, dit du *copy-out*, n'est pas résolu avec exactitude (régions `MUST`), le tableau ne peut pas être privatisé car sa valeur globale ne peut être mise à jour de manière déterministe.

Nous allons maintenant voir comment résoudre ces deux problèmes à l'aide des régions dont le calcul a été décrit précédemment.

7.2 Algorithme

7.2.1 Détermination des *bons* candidats à la privatisation

Nous allons tout d'abord résoudre le premier problème soulevé précédemment, c'est-à-dire, pour chaque itération, la détermination des éléments de tableaux dont toutes les lectures sont précédées par des écritures à la même itération ($Loc(i)$), dont le calcul est local à plusieurs itérations ($\exists\ i' \mid Loc(i) \cap Loc(i') \neq \emptyset$), et qui ne sont importés, même potentiellement, par aucune autre itération. Nous noterons $Cand(i)$ l'ensemble de ces *bons* candidats.

Par définition, $Loc(i)$ contient les éléments de tableaux qui sont potentiellement écrits, mais certainement pas importés, par l'itération i :

$$Loc(i) = W(i) -_{inf} IN(I)$$

L'ensemble $Cand(i)$ est alors donné par :

$$\begin{array}{ll} si & \left\{ \begin{array}{l} loc(i) \cap Loc(i')_{i'<i} \neq \emptyset \\ ou \\ loc(i) \cap Loc(i')_{i<i'} \neq \emptyset \end{array} \right. \\ & \\ alors & Cand(i) = Loc(i) -_{inf} proj_{i'}(IN(i')) \\ sinon & Cand(i) = \emptyset \end{array}$$

Remarques :

- l'avant-dernière équation devrait être en réalité $Cand(i) = Loc(i) -_{inf} proj_{i'}(IN(i')_{i' \neq i})$, $proj_{i'}(IN(i')_{i' \neq i})$ représentant l'ensemble des éléments de tableaux importés par les autres itérations ; mais comme, par définition, $Loc(i) \cap IN(i' = i) = \emptyset$, l'équation précédente a pu être simplifiée ;

- $Loc(i)$ et $Loc(i')$ ne doivent pas dépendre des variables modifiées dans le corps de la boucle (sauf l'indice de boucle) ; lorsque c'est le cas, l'opérateur $\widetilde{T}^{-1}_{boucle}$ permet d'exprimer les régions uniquement en fonction de l'indice de boucle ;

- si l'on désire détecter tous les éléments de tableaux privatisables, même ceux dont le calcul n'introduit pas de fausse dépendance entre les itérations, il suffit de ne pas tenir compte de la condition précédente, et de déterminer $Cand(i)$ à l'aide de l'équation de la branche *vraie* du test.

7.2.2 Détermination des sections réellement privatisables

Il s'agit maintenant de résoudre le problème du *copy-out* dans l'hypothèse où il existe des candidats à la privatisation. Nous noterons $Out\_Priv(i)$ l'ensemble des éléments de tableaux candidats à la privatisation qui doivent être exportés vers l'extérieur de la boucle. Par construction, nous avons $Cand(i) \cap proj_{i'}(IN(i')) = \emptyset$. Les éléments de $Cand(i)$ ne sont donc pas exportés vers les autres itérations. S'ils sont exportés, ils ne peuvent l'être que vers l'extérieur de la boucle. Nous avons donc :

$$Out\_Priv(i) = Cand(i) \cap OUT(i)$$

Si $Out\_Priv(i)$ est une région vide ou une région `MUST`, la région *Cand* est entièrement privatisable car les éléments qui la constituent sont soit exportés vers l'extérieur de la

```
      SUBROUTINE INIT(A,N,K,M)
      INTEGER N,K,M,A(N)
      DIMENSION WORK(100,100)
      K = M * M

      DO I = 1,N

C regions du corps de la boucle en I:
C <WORK(PHI1,PHI2)-W-MUST-{1<=PHI1,PHI1<=N,K<=PHI2,PHI2<=K+1,1<=I,I>=N}>
C <WORK(PHI1,PHI2)-IN-MUST-{0==1}>
C <WORK(PHI1,PHI2)-OUT-MUST-{0==1}>

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

FIG. 7.1 - *Régions WRITE, IN et OUT de la boucle en I de la procédure INIT, pour le tableau WORK.*

boucle, et nous les connaissons avec exactitude, soit ne sont pas exportés du tout. Nous avons alors :

$$Priv(i) = Cand(i)$$
$$Copy\_Out(i) = Out\_Priv(i)$$

Si $Out\_Priv(i)$ est une région `MAY`, nous ne pouvons pas mettre à jour le tableau global à l'aide des valeurs des éléments qui s'y trouvent, car il est possible que l'itération considérée ne les ait même pas redéfinis. Les seuls éléments privatisables, d'un point de vue formel, sont donc les candidats à la privatisation qui ne sont pas exportés, même potentiellement, par l'itération. Nous avonc dans ce cas :

$$Priv(i) = Cand(i) -_{inf} Out(i)$$
$$Copy\_Out(i) = \emptyset$$

D'un point de vue pratique, il est fort probable que cette définition de l'ensemble $Priv(i)$ n'apporte rien au niveau de la parallélisation de la boucle, car certains candidats potentiels ne seront pas privatisés, et certaines *fausses* dépendances ne pourront pas être éliminées.

7.3 exemple

Pour illustrer l'algorithme précédent, reprenons l'exemple de la figure 5.1. Nous avons fait figurer sur la figure 7.1 les régions du corps de la boucle la plus externe, qui nous seront utiles pour déterminer si le tableau `WORK` est privatisable.

Déterminons tout d'abord l'ensemble $Loc(i)$. Comme $IN(i) = \emptyset$, nous avons l'égalité $Loc(i) = W(i)$, soit :

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

$Loc(i)$ dépend de la valeur de la variable `K` à l'intérieur de l'itération i. Il faut donc appliquer la transformation inverse du transformeur de la boucle (`T(I,K) {K#INIT+I==K+1}`), sans éliminer l'indice de boucle, pour exprimer cette région en fonction de la valeur de `K` avant la boucle :

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

$Loc(i) - Loc(i')_{i'<i}$ vaut alors :

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

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

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

La région résultante n'etant pas vide, et les régions $IN(i')$ étant vides, l'expression de la région $Cand(i)$ est finalement :

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

Et comme nous avons $OUT(i) = \emptyset$, la solution est :

$$\begin{array}{l} Priv(i) = Cand(i) \\ Copy\_Out(i) = \emptyset \end{array}$$

Par contre, si les régions `OUT` de la boucle sont les suivantes :

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

les régions $OUT(i)$ du corps de la boucle sont alors :

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

Calculons $Out\_Priv(i) = Cand(i) \cap OUT(i)$:

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

Cette région est une région `MUST`. Nous avonc donc une solution exacte au problème du copy-out, et l'ensemble de la région candidate peut être privatisée.

7.4 Qualité de la privatisation

L'algorithme décrit dans la section 7.2 n'a pas encore été implanté. Nous ne pouvons donc juger de la qualité des résultats qu'il permettra d'obtenir. Nous espérons toutefois qu'elle bénéficiera de la préservation des informations symboliques apportées par les préconditions, grâce à l'utilisation des transformeurs pour la propagation intraprocédurale des régions. Mais elle dépendra en grande partie de la qualité du calcul des régions `IN` et `OUT`, et donc de la précision des opérateurs permettant de manipuler les régions, ainsi que de l'adéquation de la représentation sous forme de polyèdre. Un défaut lié aux régions apparaît dores et déjà. Les régions `OUT` devant être valables pour toute itération, il est en effet impossible de représenter sous forme de région le fait que seule la dernière itération doit mettre à jour l'intégralité du tableau global, ce qui se produit fréquemment.

Chapitre 8

Conclusion

La parallélisation automatique, et l'optimisation de programmes en général, nécessitent des analyses approfondies du flot des données composées, ou tableaux. Dans cette étude, nous avons repris le calcul des régions telles que les a définies Triolet [60] pour introduire le calcul des régions de tableaux exactes, c'est-à-dire des ensembles d'éléments de tableaux décrits par des polyèdres convexes, et représentant exactement la solution au problème considéré :

- régions `READ/WRITE` traditionnelles, exprimant les effets d'une instruction complexe ou d'une procédure sur un ensemble d'éléments de tableau ;
- régions `IN` représentant les éléments lus avant d'être éventuellement redéfinis, c'est-à'-dire importés par la portion de code considérée ;
- régions `OUT` contenant les éléments de tableau vivants à la sortie de l'instruction à laquelles elles correspondent.

Afin de préserver autant que possible l'exactitude des régions, nous avons tout d'abord défini avec soin les opérateurs permettant de les manipuler, les opérations d'ensemble sur les polyèdres convexes donnant dans la plupart des cas une approximation de l'ensemble recherché. L'intersection ne posait pas de problème particulier, mais nous avons redéfini l'union et la différence de deux régions, ainsi que la projection d'une région selon une variable du programme. La définition de ce dernier opérateur a en outre permis de clarifier le rôle des variables du programme dans l'expression de la région sous forme de polyèdre.

Nous avons ensuite présenté les opérateurs spécifiques et équations de flot de données permettant de calculer les régions `READ/WRITE`, `IN` et `OUT` sur le graphe de contrôle hiérarchisé du programme. L'originalité de l'approche réside dans l'utilisation des effets des instructions sur les valeurs des variables scalaires entières pour préserver l'exactitude de la représentation sous forme de région en cas de modification d'une variable intervenant dans l'expression des indices du tableau. L'expression de la région peut alors être exacte tout en restant symbolique.

Enfin, nous avons montré comment les régions `IN` et `OUT` pouvaient être utilisées pour privatiser les tableaux, c'est-à-dire pour détecter les sections de tableaux qui peuvent être redéclarés comme étant locales à une itération, puis pour générer la redéfinition du tableau global lorsque cela s'avère nécessaire et possible.

Au niveau de la réalisation pratique, tous les opérateurs décrits dans ce rapport sont disponibles, à l'exclusion de l'opérateur de projection exacte qui n'est que partiellement implanté.

La propagation interprocédurale ascendante et le calcul intraprocédural des régions `READ/WRITE MAY` avaient déjà été réalisés dans PIPS [49]. Ce dernier point a nécessité de profondes modifications pour le calcul des régions `MUST`, notamment pour le calcul des régions d'une séquence d'instruction et d'une boucle. Une première série d'expériences sur les codes du Perfect Club [13] en a déjà montré la robustesse. L'analyse intraprocédurale des régions `IN` est également disponible et donne des résultats corrects sur de petits cas d'école, mais n'a pas été testée de manière approfondie.

Par contre, le calcul des régions `OUT` est encore à l'état d'ébauche, l'implantation de la propagation interprocédurale descendante nécessitant un temps de réflexion supplémentaire. Et par conséquent, la privatisation de tableaux reste en suspend, le problème des *copy-out* ne pouvant être résolu.

Quelques expériences seront également 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 de disjonction de polyèdres est à envisager plus sérieusement, sachant que le coût en terme de temps de calcul et d'occupation mémoire sera certainement très important.

Une fois la privatisation de tableaux implantée, il faudra également réaliser une série d'expériences pour en évaluer l'efficacité, et pouvoir comparer les résultats obtenus avec ceux déjà publiés par d'autres équipes.

Bibliographie

[1] F. E. Allen. Control flow analysis. *ACM SIGPLAN Notices*, 5(7):1–19, 1970.

[2] F. E. Allen. Interprocedural data flow analysis. In *Proceedings of the IFIP Congress*, pages 398–402, 1974.

[3] F. E. Allen and J Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–147, March 1977.

[4] B. Alpern, M.N. Wegman, and F.K. Zadeck. Detecting equality of variables in programs. In *Symposium on Principles of Programming Language*, pages 1–11, January 1988.

[5] Corinne Ancourt. *Génération de Code pour Multiprocesseurs à Mémoires Locales.* PhD thesis, Université Paris VI, Mars 1991.

[6] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *Symposium on Principles and Practice of Parallel Programming*, April 1991.

[7] Béatrice Apvrille. Calcul de régions de tableaux exactes. In *Rencontres Francophones du Parallélisme*, pages 65–68, juin 1994.

[8] U. Assmann, H. van Someren, and M. Alt. Compilers for parallel architectures : The COMPARE project. In *Fourth International Workshop on Compilers for Parallel Computers*, pages 451–454, December 1993.

[9] V. Balasundaram and Ken Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 41–53, June 1989.

[10] J. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Symposium on Principles of Programming Language*, January 1979.

[11] Bruno Baron, François Irigoin, and Pierre Jouvelot. *Projet PIPS, Manuel utilisateur du paralléliseur batch.* rapport interne EMP-CRI E144.

[12] J.M. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, September 1978.

[13] 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, and 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.

[14] W. Blume and 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.

[15] Michael Burke and Linda Torczon. Interprocedural optimization : Eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems*, 15(3):367–399, July 1993.

[16] D. Callahan. The program summary graph and flow-sensitive interprocedural dataflow analysis. *ACM SIGPLAN Notices*, 23(7):47–56, July 19988. Also published in the proceedings of the International Conference on Programming Language Design and Implementation.

[17] Jean-François Collard. Parallélisation automatique des boucles while par exécution spéculative. In *Rencontres Francophones du Parallélisme*, pages 77–80, juin 1994.

[18] Keith Cooper, Mary Hall, Robert Hood, Ken Kennedy, Kathrin McKinley, John Mellor-Crummey, Linda Torczon, and Scott Waren. The Parascope parallel programming environment. *Proceedings of the IEEE*, 81(2), February 1993.

[19] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Language*, pages 238–252, 1977.

[20] Patrick Cousot and Nicholas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Symposium on Principles of Programming Language*, pages 84–97, 1978.

[21] Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman, and Kenneth Zadeck. Efficiently computing static single assignement form and the control dependence graph. In *Symposium on Principles of Programming Language*, pages 25–35, January 1989.

[22] Paul Feautrier. Array expansion. In *International Conference on Supercomputing*, pages 429–441, July 1988.

[23] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, September 1991.

[24] Paul Feautrier and Jean-François Collard. Fuzzy array dataflow analysis. Research Report 94–21, ENS-Lyon, July 1994.

[25] F. Fernandez and P. Quinton. Extension of chernikova's algorithm for solving general mixed linear programming problems. Publication Interne 437, IRISA, 1988.

[26] J.B.J. Fourier. Analyse des travaux de l'Académie Royale des Sciences pendant l'année 1823, partie mathématiques. In *Histoire de l'Académie Royale des Sciences de l'Institut de France*, volume 7. 1827.

[27] S. Graham and M. Wegman. Fast and usually linear algorithm for global flow analysis. *Journal of the ACM*, 23(1):172–202, January 1976.

[28] Thomas Gross and Peter Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software : Practice and Experience*, 20(2):133–155, February 1990.

[29] Rajiv Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1–4):135–150, march–december 1993.

[30] Nicholas Halbwachs. *Détermination Automatique de Relations Linéaires Vérifiées par les Variables d'un Programme*. PhD thesis, Institut National Polytechnique de Grenoble, 1979.

[31] Mary W. Hall, John M. Mellor-Crummey, Alan Carle, and René G. Rodríguez. FIAT : A framework for interprocedural analysis and transformation. In *Sixth International Workshop on Compilers for Parallel Computers*, August 1993.

[32] Paul Havlak. *Interprocedural Symbolic Analysis*. PhD thesis, Rice University, Houston, Texas, May 1994.

[33] Paul Havlak and Ken Kennedy. Experience with interprocedural analysis of array side effects. Computer Science Technical Report TR90–124, Rice University, Houston, Texas, 1990.

[34] Paul Havlak and Ken Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[35] François Irigoin. Interprocedural analyses for programming environments. In *Workshop on Environments and Tools for Parallel Scientific Computing*, September 1992.

[36] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization : An overview of the PIPS project. In *International Conference on Supercomputing*, June 1991.

[37] Pierre Jouvelot. *Parallélisation Sémantique : Une Approche Dénotationnelle Non-Standard pour la Parallélisation de Programmes Impératifs Séquentiels*. PhD thesis, Université Paris VI, 1987.

[38] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, January 1976.

[39] Gary A. Kildall. A unified approach to global program optimization. In *Symposium on Principles of Programming Language*, pages 194–206, January 1973.

[40] Arnauld Leservot. Extension de C3 aux unions de polyèdres. Rapport technique, Commissariat à l'Énergie Atomique, janvier 1994.

[41] Z. Li and P.-C. Yew. Efficient interprocedural analysis and program restructuring for parallel programs. *ACM SIGPLAN Notices*, 23(9):85–99, 1988.

[42] Zhiyuan Li. Array privatization for parallel execution of loops. In *International Conference on Supercomputing*, pages 313–322, July 1992.

[43] Jon Loelinger and Robert Metzger. Developing and interprocedural optimizing compiler. *ACM SIGPLAN Notices*, 29(4):41–48, April 1994.

[44] V. Markstein, J. Cocke, and P. Markstein. Optimization of range checking. In *Proceedings of the ACM Symposium on Compiler Construction*, pages 114–119, June 1982.

[45] Vadim Maslov. Lazy array data-flow analysis. In *Symposium on Principles of Programming Language*, pages 311–325, January 1994.

[46] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array data-flow analysis and its use in array privatization. In *Symposium on Principles of Programming Language*, January 1993.

[47] E. Myers. A precise inter-procedural data-flow algorithm. In *Symposium on Principles of Programming Language*, January 1981.

[48] David Padua and Michael Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[49] Alexis Platonoff. Calcul des effets des procédures au moyen des régions. Rapport de Stage EMP-CAI-I 132, CRI, École des Mines de Paris, Septembre 1990.

[50] William Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[51] William Pugh and David Wonnacott. An evaluation of exact methods for analysis of value-based array data dependences. In *Language and Compilers for Parallel Computing*, August 1993.

[52] Lawrence Rauchwerger and David Padua. The privatizing DOALL test : a run-time technique for DOALL loop identification and array privatization. In *International Conference on Supercomputing*, 1994.

[53] Barbara Ryder, William Landi, and H. Pande. Profiling an incremental data-flow analysis algorithm. *IEEE Transactions on Software Engineering*, 16(2), February 1990.

[54] Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data-flow analysis. *ACM Computing Surveys*, 18(3):277–316, September 1986.

[55] B.G. Ryder, M.E. Fiuczynski, and Y-F. Lee. Region analysis : a parallel elimination method for data flow analysis. In *Fourth International Workshop on Compilers for Parallel Computers*, pages 225–245, December 1993.

[56] N. Suzuki and K. Ishihata. Implementation of array biund checker. In *Symposium on Principles of Programming Language*, pages 132–143, 1977.

[57] Peiyi Tang. Exact side effects for interprocedural dependence analysis. In *International Conference on Supercomputing*, pages 137–146, July 1993.

[58] R. E. Tarjan. Testing flow graph reducibility. *J. Comput. Syst. Sci.*, 9:355–365, 1974.

[59] R. E. Tarjan. Fasts algorithms for solving paths problems. *Journal of the ACM*, 28(3):594–614, July 1981.

[60] Rémi Triolet. *Contribution à la Parallélisation Automatique de Programmes Fortran Comportant des Appels de Procédures.* PhD thesis, Université Paris VI, 1984.

[61] Peng Tu and David Padua. Automatic array privatization. In *Language and Compilers for Parallel Computing*, August 1993.

[62] J. D. Ullman. Fast algorithms for the elimination of common subexpressions. *Acta Informatica*, 2(3):191–213, 1973.

[63] A.H. Veen and M. de Lange. Overview of the PREPARE project. In *Fourth International Workshop on Compilers for Parallel Computers*, pages 365–371, December 1993.