N° d'ordre

THÈSE

présentée à

L'UNIVERSITE PIERRE ET MARIE CURIE

PARIS VI

pour obtenir

LE DIPLOME DE DOCTEUR - INGENIEUR

par

Rémi TRIOLET

Spécialité : **Mathématiques**

Mention : **Informatique**

Sujet de la thèse

CONTRIBUTION A LA PARALLELISATION AUTOMATIQUE DE PROGRAMMES FORTRAN COMPORTANT DES APPELS DE PROCEDURE

Soutenue le 20 Décembre 1984 devant la Commission composée de :

MM.	C. GIRAULT	**Président**
	P. FEAUTRIER	Rapporteur
	P. COUSOT	
	F. IRIGOIN	
	JF. PERROT	Examinateurs
	G. ROUCAIROL	

R E M E R C I E M E N T S

Je remercie vivement:

Monsieur GIRAULT d'avoir accepté de diriger cette thèse et de m'avoir fait l'honneur d'en présider le jury,

Monsieur FEAUTRIER d'avoir participé à la direction de cette thèse en me consacrant beaucoup de temps pour guider mes travaux et pour en améliorer la présentation,

Monsieur COUSOT pour l'intérêt qu'il a porté à ce travail et pour ses idées concernant l'analyse sémantique des programmes,

Monsieur PERROT d'avoir accepté de faire partie du Jury,

Monsieur ROUCAIROL pour les conseils et encouragements qu'il m'a prodigués tout au long de ce travail et pour ses idées concernant le parallélisme.

J'exprime toute ma reconnaissance à François IRIGOIN qui m'a conseillé et guidé de façon permanente. Je le remercie pour ses idées et pour le temps qu'il m'a consacré. J'ai pris un grand plaisir à travailler avec lui.

Je remercie aussi l'équipe du Centre d'Automatique et d'Informatique de l'Ecole Nationale Supérieure des Mines de Paris, où ce travail a été effectué, et plus particulièrement: Monsieur LENCI, Directeur du C.A.I, qui m'y a accueilli, Madame OLIVIER qui m'a aidé dans la programmation de certains algorithmes, et Madame LE GALLIC qui s'est chargée de la frappe du manuscript.

Mes derniers remerciements seront pour les membres de ma famille qui ont relu la totalité de ce document pour en corriger les diverses fautes.

S O M M A I R E

C H A P I T R E I

I. Introduction

I.1. Comment accélérer l'exécution de gros programmes existants par l'utilisation d'une machine parallèle

Le travail présenté dans cette thèse est une contribution à la résolution du problème suivant: comment diminuer le temps d'exécution d'un gros programme existant, par l'utilisation d'une machine parallèle.

Nous insistons sur les caractéristiques "gros" et "existant" du programme car elles justifient en grande partie notre approche. Par gros programme nous entendons programme totalisant plusieurs milliers de lignes réparties en plusieurs dizaines de modules (fonction ou subroutine). Existant signifie qu'une version séquentielle de ce programme a été écrite et mise au point dans un langage de programmation classique et qu'elle est disponible (i).

Il existe différentes catégories de machines parallèles; nous utilisons dans la suite la classification de D.J. Kuck [Kuck 82] pour les distinguer. Nous nous intéressons plus particulièrement aux machines multiprocesseurs car c'est la classe de machines vers laquelle les différentes équipes de recherche en architecture des calculateurs s'orientent actuellement [Gott 83], [Renv 83], [Gajs 83], [Cytr 83a&b]. Ces machines appartiennent à la classe MES de machines (Multiple Execution on Scalars) ou à la classe MEA/MES (Multiple Execution on Arrays and Scalars) suivant que les processeurs sont capables ou non de manipuler des tableaux (vecteurs). Nous verrons cependant que les techniques qui ont été développées pour les machines capables de parallel processing ou de pileline processing sont utilisables; ces machines appartiennent à la classe de machine SEA (Single Execution on Arrays) ou MEA suivant que les processeurs possèdent ou non plusieurs unités fonctionnelles. Lorsque le niveau de précision nécessaire sera faible, nous parlerons de machines ME* ou *EA.

I.2. Première approche: conversion du programme dans un langage parallèle

Une des solutions qui vient immédiatement à l'esprit est de paralléliser manuellement le programme; c'est à dire de le réécrire, en utilisant un langage parallèle adapté à la machine cible, en extrayant le parallélisme qu'il contient et éventuellement en faisant certaines modifications pour augmenter ce parallélisme.

I.2.1. Choix du langage; choix des modules à paralléliser

De nombreux langages parallèles ont été développés ces dernières années. Certains sont spécialisés dans le calcul vectoriel et sont donc plutôt adaptés aux machines SEA; citons ACTUS [Perr 83] langage parallèle dérivé de Pascal, SL/1 [Knig 83] langage parallèle développé au centre de recherche de la NASA pour le CYBER/203, Fortran-8X [Meis 83] et [Crow 83] version parallèle de Fortran-77 non encore disponible, VECTRAN langage développé par IBM et bien sûr tous les sur-ensembles de Fortran proposés par les constructeurs de machines parallèles: Fortran-CRAY, Fortran-CYBER/205, Fortran-BSP etc....

D'autres langages permettent de définir plusieurs processus parallèles communiquant par des moyens divers: passage de paramètres, entrées/sorties simples ou variables globales: DP (Distributed Processes) [Hans 78], d'autres sont en cours d'implémentation: CSP (Communicating Sequential Processes) [Hoar 78], Ada, enfin certains sont disponibles sur

(i) à noter qu'une bibliothèque scientifique telle que IMSL rentre dans ce cadre.

machines: Fortran-HEP sur le HEP. Par leur conception, ces langages sont adaptés aux machines ME* (ii).

Il n'est pas nécessaire de paralléliser tous les modules du programme. Il est envisageable de ne traduire et modifier que certains modules choisis parmi les plus utilisés ou les plus gourmands en temps machine, étant donné que la réduction du temps d'exécution d'un module ne représentant qu'une faible partie du programme, est globalement inintéressante.

Ainsi J. Dongarra montre dans [Dong 83] comment, en parallélisant seulement 3 fonctions de bas niveau, il a obtenu des accélérations allant de 6.7 à 8.5 sur des algorithmes matriciels tournant sur un HEP (Produit de matrices, décomposition de Choleski d'une matrice, LU factorisation etc...).

I.2.2. Avantages de cette approche

La parallélisation manuelle permet de construire une version du programme tout à fait adaptée à la machine cible; par exemple certaines machines contiennent le matériel nécessaire à l'exécution d'instructions de haut niveau telles que: recherche du maximum d'un vecteur, somme des éléments d'un vecteur, produit scalaire de deux vecteurs, itération du ler ordre ($A_{i+1} = A_i * B_i + C_i$) etc..., le S-820 (Hitachi) propose ces instructions ainsi que beaucoup d'autres (iii). Le programmeur chargé de la traduction peut, grâce à sa connaissance de la machine, tenir compte de ces caractéristiques assez facilement.

Le principal avantage de la parallélisation manuelle réside dans la connaissance de la sémantique du programme. Des transformations profondes du programme peuvent être entreprises si le programmeur sait qu'elles seront sans effet sur les résultats; ainsi les accès à une table des symboles peuvent être effectués dans n'importe quel ordre et peuvent donc être parallélisés. De même la connaissance du programme permet de savoir quels sont les modules gourmands qu'il convient de traiter dans le cadre d'une traduction partielle.

I.2.3. Inconvénients de cette approche

Le travail demandé par la parallélisation d'un gros programme est important, répétitif et fastidieux. Citons le travail de l'équipe de A. Lichnevski à l'INRIA pour paralléliser manuellement un programme d'éléments finis en vue de l'exécution sur CRAY-1. Une des transformations pratiquées consiste à inclure une boucle dans une procédure au lieu de boucler sur ses appels:

```
      DO 10 I=1,N                    SUBROUTINE P
10       CALL P(I)                   DO 10 I=1,N

                                10       ---          ancien corps
                                         ---             de P
```

cette transformation implique l'expansion de tous les scalaires de P et l'ajout d'une dimension à ses tableaux; cette parallélisation a duré 6 mois/homme, mais a permis de passer de quelques MFLOPS à 76 MFLOPS.

Ce problème est d'autant plus important que ce travail est généralement effectué

(ii) le HEP de Denelcor est un multipipeline multiprocesseur (MES/MEA)
(iii) Ces caractéristiques ont été recueillies dans les photocopies des transparents utilisés par T. Block lors de sa conférence sur les calculateurs japonais (Club Calcul Parallèle de l'INRIA).

par des non-informaticiens qui n'ont pas forcément envie d'apprendre un nouveau langage et l'architecture de leur machine.

La parallélisation manuelle peut conduire à des erreurs; les transformations effectuées sur le programme ne sont pas garanties. Ce risque augmente avec la taille du programme.

Enfin cette méthode est dangereuse dans la mesure où elle compromet la portabilité. Ainsi les primitives du Fortran-Cray permettant par exemple d'améliorer le traitement des tests dans les boucles (CUMGT, CVMGZ,...) ne sont reconnues que par CFT (CRAY Fortran Translator). Il en est de même de tous les sur-ensembles de Fortran.

I.3. Deuxième approche: parallélisation automatique

I.3.1. Présentation de cette approche

Une deuxième approche consiste à utiliser un paralléliseur pour traiter le programme. Un paralléliseur est un outil qui produit une version parallèle d'un programme à partir d'une version séquentielle, généralement le source.

Une paralléliseur autorise l'exécution simultanée d'instructions différentes et/ou d'occurrences différentes de la même instruction à la condition que celles-ci ne soient pas en conflit pour l'accès à des variables, c'est à dire que les calculs qu'elles effectuent soient indépendants.

Certains paralléliseurs diminuent le nombre de conflits -nommés dépendances- en effectuant des transformations de programme, ce qui a pour conséquence d'augmenter le parallélisme potentiel du programme.

Le résultat d'un paralléliseur peut revêtir différentes formes. Certains produisent directement du code objet pour une machine spécifiée par avance, d'autres régénèrent un programme source dans lequel le parallélisme est exprimé à l'aide de constructions syntaxiques particulières, d'autres savent faire les deux.

I.3.2. Avantages de cette approche

L'effort imposé au programmeur par la première approche disparait avec la parallélisation automatique; c'est à notre avis le principal avantage. D'autre part si le paralléliseur a été conçu pour la machine utilisée ou si son action est paramétrable par la description de cette machine, la version parallèle fournie lui est adaptée. Enfin le problème de la portabilité est supprimé puisque le programme initial n'est pas traduit dans un autre langage. Tout au plus faut-il le modifier pour supprimer les constructions mal acceptées par le paralléliseur utilisé -s'il y en a- ou pour ajouter des DIRECTIVES: sortes de conseils fournis au paralléliseur par le programmeur.

I.3.3. Problèmes posés par cette approche

Ils sont essentiellement liés à la perte de la connaissance de la sémantique du programme. Une partie de celle-ci peut être retrouvée par le paralléliseur, ce qui lui permet d'effectuer un meilleur traitement. Cependant, des transformations profondes comme celles envisagées au §I.2.2. ne peuvent pas être effectuées automatiquement dans l'état actuel de la connaissance de la sémantique d'un programme.

Nous reviendrons plus en détails sur les méthodes de parallélisation automatique aux chapitres II et IV.

I.4. Présentation sur exemple de notre contribution à la parallélisation automatique

I.4.1. Améliorations proposées

Nous proposons d'améliorer les méthodes actuelles de parallélisation automatique sur les trois points suivants.

(1) Traitement de l'aliasing: il y a aliasing quand les espaces mémoires de plusieurs paramètres formels et/ou variables globales sont associés dynamiquement. Ce problème est bien connu mais généralement passé sous silence, ce qui peut conduire à une parallélisation incorrecte.

(2) Parallélisation d'instructions comportant des appels de sous-programmes: de telles instructions sont actuellement laissées de côté par les paralléliseurs. Il en résulte une perte de parallélisme surtout pour les machines ME*.

(3) Calcul et utilisation d'assertions sur les variables du programme: nous proposons de calculer puis d'utiliser des assertions pour améliorer différents traitements effectués par les paralléliseurs. Ces traitements concernent généralement les accès aux éléments de tableau.

I.4.2. Exemple

Dans [Dong 83] J. Dongarra propose de calculer un produit matriciel en utilisant 3 procédures:

(1) SSDOT ajoute le produit scalaire d'une ligne d'une matrice M par un vecteur X à la ième composante d'un vecteur Y;

(2) SMXPY calcule Y=Y+M*X par appels successifs à SSDOT;

(3) MM calcule les n colonnes d'une matrice A=B*C par appel à SMXPY avec Y égal à une colonne de A, M égale à B et X égal à une colonne de C.

Les accélérations annoncées au §II.2.1 sont obtenues en parallélisant dans SMXPY les appels à SSDOT. Nous allons montrer comment il est possible de paralléliser dans MM les appels à SMXPY de façon automatique. Cet exemple est l'objet de l'annexe, où nous montrons comment obtenir les résultats que nous allons exposés. Nous transformons, pour la clarté de l'exposé, la procédure SMXPY en expansant l'appel à SSDOT. D'où:

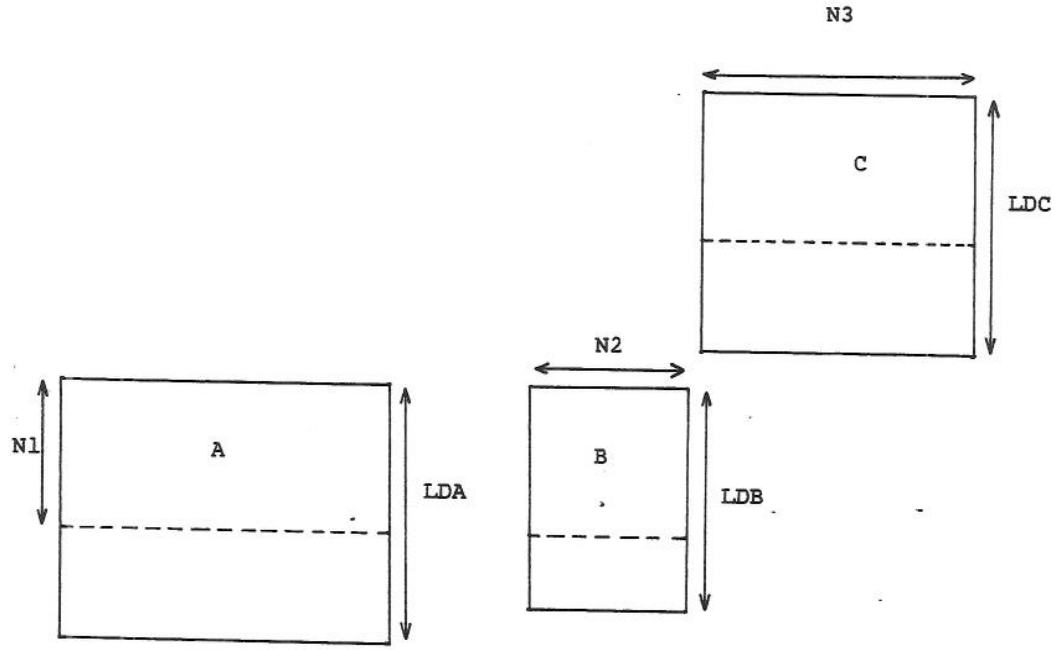


Figure I.1

```
                SUBROUTINE MM(A,LDA,N1,N3,B,LDB,N2,C,LDC)
                REAL A(LDA,*),B(LDB,*),C(LDC,*)
        C
(m1)            DO 10 I = 1,N3
(m2)    10         CALL SMXPY(N2,A(1,I),N1,LDB,C(1,I),B)
        C
                RETURN
                END

                SUBROUTINE SMXPY(N2,Y,N1,LDM,X,M)
                REAL X(*),Y(*),M(LDM,*)
        C
(s1)            DO 10 J=1,N1
(s2)               Y(J)=0
(s3)               DO 10 K=1,N2
(s4)    10            Y(J)=Y(J)+X(K)*M(J,K)
                RETURN
                END
```

Remarque I-1: à noter que le fonctionnement de ce programme est basé sur la connaissance de l'ordonnancement des éléments d'un tableau en Fortran: stockage par colonne.

Les différentes itérations du corps de la boucle m1 peuvent être effectuées en parallèle à condition que les variables manipulées par deux itérations quelconques ne génèrent pas de conflit lecture/écriture ou écriture/écriture (iv). Nous devons donc

(iv) Cette condition est trop restrictive mais nous l'utilisons pour simplifier cet exemple.

analyser le programme pour connaître l'ensemble des variables manipulées par SMXPY.

I.4.2.1. Analyse "pessimiste"

Une lère analyse peut être effectuée: elle consiste à supposer que SMXPY lit et écrit toutes les variables auxquelles elle a accès: paramètres et variables globales (v). Cette analyse donne comme résultat:

m2 lit et écrit N2,A,N1,LDB,C,B

ce qui implique que m1 est non-parallélisable.

I.4.2.2. Analyse "tableau = entité"

Cette analyse consiste à rechercher les variables manipulées par chaque instruction de l'appelante en considérant que toute manipulation d'un élément de tableau implique la manipulation de la totalité du tableau. Après avoir fait l'union de ces manipulations, une traduction est effectuée visant à les exprimer en fonction des variables de l'appelante. Durant ce processus, les variables locales de l'appelée sont ignorées:

s1	lit	{N1}	et écrit	ø
s2	"	ø	"	{Y}
s3	"	{N2}	"	ø
s4	"	{Y,X,M}	"	{Y}
SMXPY	"	{N2,N1,Y,X,M}	"	{Y}
m2	"	{N2,N1,A,C,B}	"	{A}

Bien que le résultat soit amélioré par rapport à I.4.2.1, la boucle m2 demeure non-parallélisable.

I.4.2.3. Analyse avec utilisation d'assertions

La méthode que nous développons dans les chapitres suivants consiste à calculer puis à utiliser des assertions sur les variables du programme pour affiner les traitements des tableaux. Ces assertions ne peuvent être calculées qu'à la condition de disposer d'un programme complet. Nous supposons donc que MM est appelée depuis un programme principal comportant les déclarations de trois matrices M1, M2 et M3 et un appel à MM:

```
DIM M1(10,20),M2(50,70),M3(70,20)
CALL MM(M1,10,10,20,M2,50,70,M3,70)  (pp1)
```

Dans ces conditions, notre méthode permet d'aboutir aux résultats suivants.

(1) Les boucles s1 et s2 possèdent des propriétés qui nous permettent d'établir les assertions suivantes:

(v) Cette analyse présente l'avantage de ne pas avoir besoin du texte de la procédure appelée et sera donc utilisée lorsque celui-ci n'est pas disponible (bibliothèques)

en s2 $\{1 \leq J \leq N1\}$

en s4 $\{1 \leq J \leq N1,\ 1 \leq K \leq N2\}$

(2) N1 et N2 sont deux variables paramètres formels non modifiées par l'exécution de SMXPY.

(3) Les manipulations effectuées par une occurence quelconque d'une instruction sont décrites par ce que nous appelons des régions: une région est un couple (V,A) où V est un identificateur de variable et A un ensemble d'assertions définissant les variations des indices, notées ρ_i (vi). Les régions manipulées par les occurences de s2 et s4 sont:

s2 écrit $\{(Y,\{\rho_1 = J,\ 1 \leq J \leq N1\})\}$ et lit $\emptyset$

s4 écrit $\{(Y,\{\rho_1 = J,\ 1 \leq J \leq N1\})\}$ et lit $\{(Y,\{\rho_1 = J,\ 1 \leq J \leq N1\}),$ $(X,\{\rho_1 = K,\ 1 \leq K \leq N2\}),\ (M,\{\rho_1 = J,\ \rho_2 = K,\ 1 \leq J \leq N1, 1 \leq K \leq N2\})\}$

(4) les régions manipulées par une instruction sont déduites de celles manipulées par une occurence d'instruction en ne conservant dans les assertions que les variables "constantes" (non modifiées) dans la procédure:

s2 écrit $\{(Y,\{1\leq\rho_1\leq N1\})\}$ et lit $\emptyset$

s4 écrit $\{(Y,\{1\leq\rho_1\leq N1\})\}$ et lit $\{(Y,\{1\leq\rho_1\leq N1\}),$ $(X,\{1\leq\rho_1\leq N2\}),\ (M,\{1\leq\rho_1\leq N1,\ 1\leq\rho_2\leq N2\})\}$

(5) la traduction de ces régions est complexe; la méthode que nous décrivons au chapitre VI permet d'aboutir au résultat souhaité:

m2 écrit et lit $\{(A,\{1\leq\rho_1\leq N1,\ \rho_2 = I,\ 1 \leq I \leq N3\})\}$

m2 lit $\{N2,N1,(C,\{1\leq\rho_1\leq N2,\ \rho_2 = I,\ 1 \leq I \leq N3\}),$ $(B,\{1\leq\rho_1\leq N1,\ 1\leq\rho_2\leq N2\})\}$

à la condition de vérifier que

$$N1 \leq LDA \wedge N1 \leq LDB \wedge N2<LDC \qquad (AV)$$

La méthode de calcul d'assertions interprocédurale que nous décrivons aux chapitres VII et X et qui nous a permit d'établir les points (1) et (2) nous permet de même d'associer à m2, pour l'appel (ppl), les assertions suivantes:

$$\{LDA = 10,\ N1 = 10,\ \ldots,\ LDC = 70\}$$

elles nous suffisent pour prouver les trois assertions (AV). Dans ces conditions, nous pouvons générer une version parallèle de la procédure MM où la boucle m1 est parallélisée; en effet chaque itération manipule une colonne différente de A.

(6) Il est important de noter que la version parallèle obtenue n'est utilisable que pour l'appel (ppl), ou tout appel vérifiant les conditions (AV). Notre méthode permet de paralléliser différemment une procédure en fonction des appels statiques qui en sont

(vi) une définition précise d'une région est donnée dans le chapitre V.

faits.

Remarque I-2: nous n'avons pas tenu compte jusqu'ici d'un aliasing éventuel entre 2 ou plusieurs des variables de SMXPY qui aurait empéché la parallélisation de la boucle m1. Rappelons que ce problème est traité au chapitre IX.

I.4.2.4. Analyse par les méthodes existantes de parallélisation

Ces méthodes, que nous présentons au chapitre II, ne sont pas capables de paralléliser la boucle m2, justement à cause de l'appel de procédure. Leurs auteurs ne se sont pas intéressées au traitement des appels de procédure autrement que par la méthode de recopie du corps de l'appelée dans celui de l'appelante. Ceci est sans doute dû au fait que ces méthodes ont été développées plutôt pour des machines vectorielles (ILLIAC IV, CRAY-1, etc...) sur lesquelles la boucle m2 serait, dans tous les cas, exécutée séquentiellement.

I.5. Le langage traité: FORTRAN-77

Toute l'étude que nous exposons dans les chapitres suivants a été menée pour Fortran-77. Les programmes scientifiques sont parmi ceux pour lesquels l'exécution sur une machine parallèle est intéressante; ces programmes sont généralement écrits en Fortran et c'est donc la raison de notre choix.

Fortran-77 comprend la plupart des constructions des langages de programmation structurés. Il faut cependant noter que la récursivité qui est interdite en Fortran-77 (vii) et autorisée dans la plupart des langages modernes (Pascal, Ada etc....) et la notion de pointeurs, poseraient de sérieux problèmes pour l'adaptation de notre méthode à l'un de ces langages.

Fortran-77 comprend aussi quelques constructions spécifiques posant des problèmes (EQUIVALENCES, ENTRYS, représentations différentes d'un COMMON, etc ...) qui ne peuvent être ignorées puisqu'un programme existant peut les utiliser.

I.6. Présentation du plan suivi

Voici une briève description du contenu de chaque chapitre:

II : Nous y présentons quelques-unes des méthodes de parallélisation automatique existantes.

III : Nous y introduisons trois propositions d'amélioration de ces méthodes: prise en compte de l'aliasing, traitement des appels de procédure par recherche de leurs effets et calcul et utilisation d'assertions pour l'affinement du traitement des tableaux. Nous montrons parallèlement l'intérêt de ces améliorations.

IV : Nous montrons que les appels de procédure ne peuvent être éliminés sans perte d'une certaine forme de parallélisme et qu'ils posent des problèmes quand ils sont imbriqués sur plus d'un niveau. Nous montrons quelles en sont les conséquences sur nos propositions.

(vii) Une étude menée par les auteurs du langage SL/1 parmi les scientifiques de la NASA montre que ceux-ci ne se servent pas de la récursivité [Knig 83].

V & VI : Nous montrons comment calculer les effets de l'exécution d'une procédure sur ses propres variables (V) et comment reporter ces effets sur l'appel correspondant de la procédure appelante, c'est ce que nous appelons traduction des effets (VI).

VII & VIII : Nous y présentons différentes méthodes permettant d'associer un ensemble d'assertions à chaque instruction d'une procédure (VII) et d'utiliser ces assertions pour affiner le traitement des tableaux et évaluer ou comparer des expressions (VIII). Les principes de ces méthodes ont pour la plupart déjà été publiés; leur utilisation pratique dans notre cadre nécessite cependant une adaptation.

IX : Nous y proposons une méthode de calcul de l'aliasing.

X : Nous proposons un ordonnancement des divers traitements exposés auparavant dans le cas d'un programme complet.

Annexe : Nous traitons manuellement l'exemple présenté dans l'introduction afin de montrer comment les résultats seraient automatiquement obtenus.

CHAPITRE II

II. Présentation de quelques méthodes de parallélisation existantes

II.1. Introduction

II.2. La méthode de G. Roucairol

II.2.1. Présentation

II.2.2. Définition d'un schéma de programme parallèle

II.2.3. Calcul d'un schéma

II.2.4. La K-équivalence

II.2.5. Réalisations à files généralisées associée à un programme pour la K-équivalence

II.2.6. Points forts et points faibles de cette méthode

II.3. Présentation des méthodes locales de parallélisation; cas des logiciels VESTA et PARAFRASE

II.3.1. Introduction

II.3.2. Présentation

II.3.2.1. Principes généraux

II.3.2.2. Le graphe de dépendances

II.3.2.3. Les transformations de programme

II.3.2.4. Analyse et parallélisation du programme

II.3.3. Remarques sur les choix à effectuer pendant la parallélisation

II.3.4. Etat d'avancement des logiciels, autres projets

II.4. Conclusion

II. Présentation de quelques méthodes de parallélisation existantes

II.1. Introduction

Dans ce chapitre, nous allons présenter plusieurs méthodes de parallélisation automatique de programmes. G. Roucairol [Rouc 82] classe ces méthodes en deux catégories.

(1) Les méthodes globales, qui considèrent un programme comme un ensemble d'opérateurs et qui, à partir du graphe de contrôle du programme séquentiel et d'une relation de dépendance sur les opérateurs construisent un automate de contrôle autorisant des exécutions parallèles des différents opérateurs du programme.

(2) Les méthodes locales qui utilisent la sémantique de certains opérateurs des langages de programmation pour en extraire le parallélisme: boucle DO Fortran (PARAFRASE, VESTA), expressions arithmétiques (D.J. Kuck, J.L. Baer), accès à des hyperplans parallèles d'un tableau (D.J. Kuck, L. Lamport).

Nous illustrons la première catégorie en présentant la méthode de G. Roucairol et la deuxième en présentant, en parallèle, les logiciels PARAFRASE et VESTA.

II.2. La méthode de G. Roucairol

II.2.1. Présentation

L'essentiel de ce paragraphe est extrait de [Rouc 82]; il est organisé de la façon suivante:

(1) Nous définissons tout d'abord la notion de schéma de programme parallèle: objet définissant les opérateurs d'un programme, les variables en entrée et en sortie de chaque opérateur et l'ordre partiel d'exécution de ces opérateurs.

(2) Nous montrons comment, à partir d'une relation d'équivalence sur les schémas, G. Roucairol compare le parallélisme offert par différents schémas.

(3) Nous présentons la K-équivalence, une des relations d'équivalence proposées dans [Rouc 82].

(4) Nous montrons comment G. Roucairol propose de construire un schéma parallèle qui définisse des exécutions parallèles K-équivalentes à l'exécution séquentielle d'un programme.

II.2.2. Définition d'un schéma de programme parallèle

Un schéma de programme parallèle est un couple $S = (S_C, S_I)$ où

(a) S_I est un schéma de données qui définit les opérateurs du programme et les variables qu'ils utilisent; il se compose de

---

(i) Dans ce modèle, l'exécution d'un opérateur est supposée instantanée; Cette restriction peut être évitée en décomposant chaque opérateur en un opérateur début et un opérateur fin.

un ensemble A d'opérateurs A = {a,b,c,... } (i)

un ensemble V de variables V = $\{v_1, v_2, \ldots\}$ (ii)

il permet d'autre part d'associer à chaque opérateur a ϵ A l'ensemble des variables utilisées (données) par a:

$$D(a) = \{d_1, \ldots, d_{i_a}\}$$

et l'ensemble des variables modifiées (résultats) par a:

$$R(a) = \{r_1, \ldots, r_{j_a}\}$$

(b) S_C est un schéma de contrôle définissant l'ordre partiel d'exécution des opérateurs; c'est un quadruplet $S_C = (Q, q_o, \Sigma, \tau)$ où

Q est un ensemble d'états,
q_o est un état initial,
$\Sigma = \cup \Sigma(a)$, a ϵ A est un alphabet,
$\Sigma(a) = \{a_1, \ldots, a_{k_a}\}$ est l'alphabet des terminaisons possibles de l'opérateur a (k_a = 1 pour les opérateurs qui ne sont pas des tests),
τ est une fonction partielle $\tau: Q \times \Sigma \rightarrow Q$, appelée fonction de transition.

II.2.3. Calcul d'un schéma

Un calcul d'un schéma S est un mot de Σ^+ accepté par l'automate défini par le schéma de contrôle. Nous notons C(S) l'ensemble des calculs d'un schéma S.

Si R est une relation d'équivalence sur un ensemble de calculs, deux schémas S et S' sont dits R-équivalents si

$$\forall x \in C(S) \quad \exists y \in C(S') \text{ tq } (x,y) \in R$$

et

$$\forall x \in C(S') \quad \exists y \in C(S) \text{ tq } (x,y) \in R.$$

Ceci permet à G. Roucairol de comparer le parallélisme offert par deux schémas. Soient S et S' deux schémas R-équivalents, S sera dit plus parallèle que S' si $C(S') \subset C(S)$.

II.2.4. La K-équivalence

La K-équivalence est basée sur la comparaison de l'ordre d'exécution des opérateurs qui partagent des variables. Soient a et b deux opérateurs, la relation ρ est ainsi définie:

(ii) Dans la partie de la méthode de G. Roucairol que nous décrivons, une variable est une entité dont le contenu est globalement manipulé.

$$D(a) \cap R(b) \neq \emptyset$$
$$\text{ou } R(a) \cap D(b) \neq \emptyset \iff (a,b) \in \rho$$
$$\text{ou } R(a) \cap R(b) \neq \emptyset$$

G.Roucairol définit la K-équivalence à partir de la relation ρ comme suit:

Soit $\bar{\rho}$ la fermeture réflexive de ρ; soient x et y deux calculs:

$$(x,y) \in K \iff \forall (a,b) \in \bar{\rho},\ E(\Sigma(a) \cup \Sigma(b), x) = E(\Sigma(a) \cup \Sigma(b), y)$$

où $E(A,z)$, A Σ décrit le mot obtenu à partir de z en y détruisant les symboles qui n'appartiennent pas à A.

II.2.5. Réalisations à files généralisées associée à un programme pour la K-équivalence

La méthode de G. Roucairol permet d'associer à un programme un schéma parallèle qui autorise des exécutions parallèles K-équivalentes à l'exécution séquentielle. Ce schéma est matérialisé par un dispositif dont la construction et le fonctionnement sont détaillés dans [Rouc 82].

Nous donnons une briève description de ce dispositif; il se compose des éléments suivants:

(1) Un ensemble I de queues; chaque queue contenant un mot construit sur l'alphabet $\Gamma = \Gamma_d \cup \Gamma_f$ avec:

$\Gamma_d = \{\bar{a},\bar{b},\bar{c},\ldots\}$ début d'opérateur,
$\Gamma_f = \{\underset{\sim}{a},\underset{\sim}{b},\underset{\sim}{c},\ldots\}$ fin d'opérateur.

A chaque opérateur a est associé un ensemble I(a) de queues déterminé à partir de la relation $\bar{\rho}$ (chaque queue représente une dépendence entre 2 opérateurs).

(2) Un automate d'états finis P, construit à partir du graphe de contrôle du programme initial. A chaque transition de cet automate et à chaque queue est associé un mot de Γ_d qui est ajouté au contenu de la queue quand la transition est effectuée.

Nous donnons une briève description du fonctionnement de ce dispositif:

(1) Chaque opérateur a scrute les queues de I(a). Il est autorisé à s'exécuter lorsque chacune de ces queues F vérifie la condition suivante:

$\exists x \in (\Gamma - \{\bar{a}\})^*$ et $\pi \in \Gamma_f$ tels que $x\pi\bar{a}$ est un prefixe du contenu de F.

(2) A la fin de l'exécution d'un opérateur a, la première occurence de $\bar{a}$ dans chaque queue de I(a) est remplacée par $\underset{\sim}{a}$.

(3) L'exécution d'un opérateur qui est un test (possédant plusieurs terminaisons), conduit à une transition de l'automate et donc à des allongements de toutes les queues. (chaque allongement traduisant que les opérateurs dont l'exécution était conditionnée par le résultat de ce test peuvent s'exécuter).

Remarque II-1: Le fonctionnement de cet automate implique qu'un opérateur ne peut s'exécuter avant l'opérateur test qui le conditionne et que l'ordre d'exécution de deux opérateurs en conflit selon ρ est identique à l'ordre séquentiel.

Exemple II-1: Le programme suivant calcule dans P la puissance Nième de A. Nous supposons que A et N sont initialisés.

(a)	P = 1	R(a) = {P}	D(a) = ∅
(b)	while (N > 0)	R(b) = ∅	D(b) = {N}
(c)	P = P * A	R(c) = {P}	D(c) = {P,A}
(d)	N = N - 1	R(d) = {N}	D(d) = {N}

ρ = {(a,c), (c,a), (d,b), (b,d), (a,a), (b,b), (c,c), (d,d)}

Les différentes queues sont $F_{a,c}$, $F_{b,d}$, $F_{a,a}$, $F_{b,b}$

D'autre part:

$I(a) = \{F_{a,c}, F_{a,a}\}$,
$I(b) = \{F_{b,d}, F_{b,b}\}$,
$I(c) = \{F_{a,c}\}$ et
$I(d) = \{F_{b,d}\}$.

L'automate d'états finis à le graphe suivant:

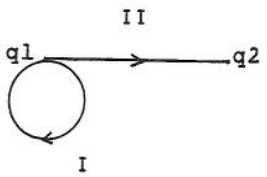


Figure II-1

La transition I est activée par la terminaison b_1 de l'opérateur b, la transition II par la terminaison b_2.

Le tableau suivant donne les mots de Σ qui sont ajoutés aux différentes queues lors de l'activation d'une transition; la lère ligne donne le contenu initial des queues; $\tilde{\epsilon}$ est un symbole spécial de Γ_f.

	$F_{a,a}$	$F_{b,b}$	$F_{a,c}$	$F_{b,d}$
contenu initial	$\tilde{\epsilon}\bar{a}$	$\tilde{\epsilon}\bar{b}$	$\tilde{\epsilon}\bar{a}$	$\tilde{\epsilon}\bar{b}$
I	-	$\bar{b}$	$\bar{c}$	$\bar{d}\bar{b}$
II	-	-	-	-

Fonctionnement:

au départ a et b peuvent s'exécuter en parallèle. On voit d'autre part que b et d peuvent s'exécuter sans autre contrainte que celle imposée par $F_{d,b}$. Chaque exécution de b conduisant à la terminaison b_1 implique le remplissage de $F_{a,c}$ par un symbole $\bar{c}$; dès que a est exécuté ($\bar{a}$ est alors remplacé par $\tilde{a}$) c peut s'exécuter autant de fois qu'il y a des symboles $\bar{c}$ dans $F_{a,c}$.

Les conséquences sont: à chaque tour de boucle c et d s'exécutent en parallèle (on passe de 3 unités de temps à 2); si de plus, d et b sont plus rapides que c, ils peuvent "prendre de l'avance" sur c.

II.2.6. Points forts et points faibles de cette méthode

Le principal avantage de cette méthode est de permettre la construction d'un dispositif de contrôle prallèle permettant d'atteindre le parallélisme maximum. Elle peut être appliquée moyennant un prétraitement des programmes, à tous les langages de programmation. Cette méthode définie une architecture de machine que nous schématisons de la façon suivante (extension du schéma proposé dans [Kell 73]):

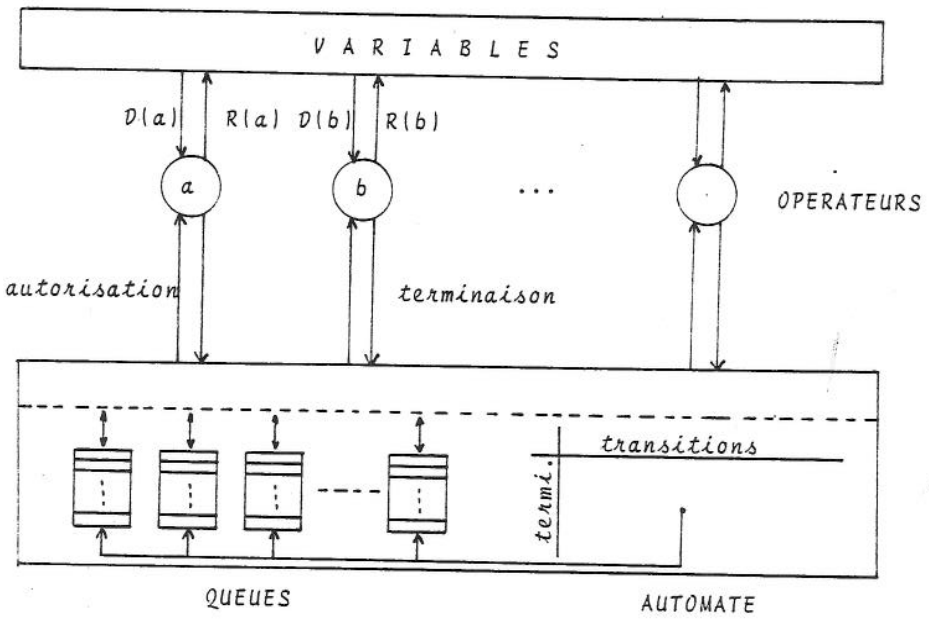


Figure I.2

Indépendamment des problèmes posés par l'adaptation d'un programme Fortran au formalisme de cette méthode (recherche de l'application "instruction → opérateur", recherche des variables manipulées par les appels de procédure, introduction de variables spéciales pour garantir la séquentialité des entrées-sorties, ...) nous pensons que la limitation suivante est importante.

Il est impossible de paralléliser l'exécution des différentes occurences du même opérateur; ceci est normal étant donné qu'un opérateur est censé manipuler les mêmes variables à chaque exécution, ce qui n'est pas vrai en Fortran à cause des accès aux tableaux; mais il est impossible, sans modification de la méthode de les traiter autrement que comme des entités.

II.3. Présentation des méthodes locales de parallélisation; cas des logiciels VESTA et PARAFRASE

II.3.1. Introduction

Les informations relatives à PARAFRASE sont extraites des références suivantes: [Kuck 72], [Kuck 74], [Kuck 79a&b], [Kuck 81a,b&c], [Kuck 82], [Wolf 78], [Cytr 82], [Davi 81], [Leas 76]; celles relatives à VESTA sont extraites de [Bill X] qui regroupe différents

rapports.

Dans le §II.3.2 nous présentons les méthodes locales de parallélisation; nous commençons par exposer les principes généraux de ces méthodes, puis nous montrons ce qu'est le graphe des dépendances (GD); nous énonçons ensuite à l'aide d'exemples les différentes transformations visant à diminuer le nombre d'arcs dans le GD. Enfin nous montrons comment, à partir du GD restreint, il est possible générer une version parallèle du programme initial.

Dans le §II.3.3 nous présentons les points forts et les points faibles de ces méthodes.

II.3.2. Présentation

II.3.2.1. Principes généraux

La parallélisation d'un programme suit les étapes suivantes:

(1) Recherche des dépendances entre instructions causées par les accès aux variables. Cette recherche utilise la sémantique de constructions syntaxiques telles que les boucles, de façon à restreindre les dépendances dues aux tableaux.

(2) Transformation du programme de façon à éliminer des dépendances. Ces transformations modifient le schéma de données et le schéma de contrôle du programme. Elles sont validées par la sémantique du langage utilisé.

(3) Analyse du programme final et de ses dépendances pour générer un programme comportant des constructions syntaxiques parallèles. Il est important de noter que les méthodes locales ne proposent pas de mécanisme de contrôle et s'adaptent à celui de la machine cible.

II.3.2.2. Le graphe de dépendances

C'est un graphe orienté dont les noeuds représentent les instructions du programme et les arcs les dépendances crées par les accès aux variables. Ces dépendances se décomposent en trois catégories; soient S et T deux instructions non nécessairement distinctes, et S_i et T_j deux occurences de ces instructions telles que S_i s'exécute avant T_j lors de l'exécution séquentielle du programme.

(1) Il y a dépendance de S vers T si S_i modifie une donnée que T_j utilise. Ce type de dépendance est nommé "data-dependence" dans PARAFRASE et "Producteur-Consommateur (PC)" dans VESTA.

(2) Il y a dépendance de S vers T si S_i utilise une donnée que T_j modifie: ce type de dépendance est nommé "anti-dependance" dans PARAFRASE et "Consommateur-Producteur (PC)" dans VESTA.

(3) Il y a dépendance de S vers T si S_i et T_j modifie la même donnée; ce type de dépendance est nommé "outpout-dependance" dans PARAFRASE et "Producteur-Producteur (PP)" dans VESTA.

Nous remarquons que si les tableaux sont traités comme des entités, la relation ρ de la méthode de G. Roucairol peut être calculée à partir des dépendances PC, CP et PP de la façon suivante (a et b sont deux opérateurs):

$$\left.\begin{array}{r} \text{a et b sont en dépendance PP} \\ \text{ou a et b sont en dépendance PC} \\ \text{ou a et b sont en dépendance CP} \end{array}\right\} \Leftrightarrow (a,b) \in \rho$$

VESTA distingue les dépendances existant entre deux instructions en faisant abstraction des boucles, des autres dépendances où la notion d'occurence d'instruction intervient. Ces dernières sont appelées dépendances de fermeture (PC*, CP*, PP*).

Une quatrième sorte de dépendance est utilisée pour matérialiser les tests. Elle est nommée "t-dépendance" dans VESTA et "control-dependence" dans PARAFRASE. Cette représentation concise du contrôle facilite les transformations des tests présentées dans le §II.3.2.3.

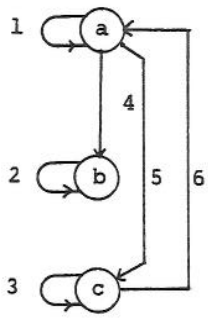
La construction du GD pour une portion de programme quelconque est délicate; c'est pourquoi ces logiciels ne construisent que le GD de la portion de programme à paralléliser, généralement une boucle ou un ensemble de boucles bien imbriquées.

En ce qui concerne les boucles, il nous parait important de noter que VESTA recherche toutes les boucles d'un programme et est capable de détecter les pseudo-boucles DO fabriquées par le programmeur à partir de tests et de branchements.

Exemple II-2: Voici une boucle DO et son GD associé.

```
        DO 10 I = 1,N
a          TMP  = T(I) + ...
b          T(I) = T(I+1) * ...
c          R = R + TMP
   10   CONTINUE
```

1: PP* (TMP)
2: CP* (T(I),T(I+1))
3: PP*, CP*, PC* (R)
4: CP (T(I))
5: PC (TMP)
6: CP (TMP)



II.3.2.3. Les transformations de programme

Les transformations de programmes effectuées par les différents logiciels sont nombreuses, souvent exposées dans des articles différents (PARAFRASE) et correspondant donc à des versions différentes. D'autre part, les mêmes transformations portent des noms différents suivant l'auteur des articles. Aussi est-il difficile de structurer cet ensemble de transformations; nous proposons la décomposition suivante.

(1) Une partie de ces transformations vise à normaliser les variations des indices de boucles et à normaliser les indices de tableaux. Il est important de noter que PARAFRASE impose des restrictions sur les boucles DO contrairement à VESTA qui, grâce à la notion de compte-tour, traite les boucles de façcon plus générale. Nous présentons, par un

exemple, une de ces transformations communes à VESTA et à PARAFRASE; elle se nomme "dérécursivation des scalaires" dans VESTA et "induction variable removal" dans PARAFRASE:

```
      J = 1
      DO 10 I = 1,N                      DO 10 I = 1,N
            J = J + 2          ⟶              J = 2 * I + 1
            ...                              ...
10    CONTINUE                   10    CONTINUE
```

(2) Une partie de ces transformations vise à éliminer les instructions de contrôle et de branchement des corps de boucles. Celles-ci sont genântes dans la mesure où une instruction conditionnée par un test peut ne pas être exécutée à chaque itération de la boucle. Les transformations usuellement proposées sont: le remplacement des tests par des vecteurs de contrôle ou par des affectations conditionnelles:

```
IF (C(I)) THEN
      Z(I) = X(I)
ELSE                    → Z(I) = IF (C(I)) THEN X(I) ELSE Y(I)
      Z(I) = Y(I)
ENDIF
```

Nous signalons que K. Kennedy et son équipe proposent dans [Alle 83] une étude très détaillée de la suppression des instructions de contrôle, cet article étant particulièrement intéressant, par les algorithmes proposés.

(3) Une partie de ces transformations vise à éliminer des arcs du GD. PARAFRASE en propose 4, VESTA 2. Elles sont intéressantes dans la mesure où les dépendances gênent la parallélisation. Nous les présentons grâce à l'exemple suivant:

```
A = Z(I) + X(I)
T(I) = A * C/2
A = K(I-1) + K(I+1)
R(I) = A/2
K(I) = ...
```

"SCALAR RENAMING" (PARAFRASE)

```
A1 = Z(I) + X(I)
T(I) = A1 * C /2
A2 = K(I-1) + K(I+1)
R(I) = A2/2
K(I) = ...
```

"SCALAR EXPANSION"
"EXPANSION DE SCALAIRE"

```
A1(I) = Z(I) + X(I)
T(I) = A1(I) * C/2
A2 = K(I-1) + K(I+1)
R(I) = A2/2
K(I) = ...
```

"FORWARD SUBSTITUTION"
"SUBSTITUTION VARIABLE VALEUR"

```
A1(I) = Z(I) + X(I)
T(I) = A1(I) * C/2
R(I) = (K(I-1) + K(I+1))/2
K(I) = ...
```

"NODE SPLITTING" (PARAFRASE)

```
A1(I) = Z(I) + X(I)
T(I) = A1(I) * C/2
TMP(I) = K(I+1)
R(I) = (K(I-1) + TMP(I))/2
K(I) = ...
```

Cet exemple, bien que construit pour mettre en évidence les diverses transformations, reste réaliste. La version initiale est non parallélisable (le GD contient au moins 12 arcs et 7 cycles) la version finale l'est, moyennant un réordonnement des instructions.

(4) la dernière partie de ces transformations consiste à partitionner le corps de boucle et à distribuer la fonction de boucle sur chaque partition. Les différents logiciels procèdent à peu près de la même façon: une partition est une composante fortement connexe du GD (nommée π-block dans PARAFRASE); les différentes partitions sont ordonnées par un tri topologique (P1 > P2 <=> il existe un arc du GD joignant un noeud de P1 à un noeud de P2); puis la fonction de boucle est distribuée sur chaque partition. Ainsi, le corps de boucle précédant donne naissance aux cinq corps de boucle suivants:

```
1: A1(I) = Z(I) + X(I)
2: T(I) = A1(I) * C/2
3: TMP(I) = K(I+1)
4: K(I) = ...
5: R(I) = (K(I-1) + TMP(I))/2
```

II.3.2.4. Analyse et parallélisation du programme

C'est la phase finale du processus de parallélisation. Après que les transformations précédentes ont été effectuées, chaque boucle résiduelle doit être analysée et parallélisée au mieux.

Les boucles dont le GD ne comporte pas de cycle peuvent être traduite en boucle DO-// signifiant que toutes les itérations peuvent être effectuées en parallèle:

```
      DO 10 I = 1,N                          DO-// 10 I = 1,N
10          T(I) = V(I) + W(I)      ⟶   10          T(I) = V(I) + W(I)
```

Pour les autres, différents cas sont envisageables. Certaines macro- instructions peuvent être reconnues:

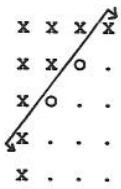
```
      DO 10 I = 1,N                          DOCUMUL I = 1,N
10          SUM = SUM + T(I)        ⟶   10          SUM = SUM + T(I)
```

Lorsque plusieurs boucles sont bien imbriquées et que les indices des tableaux génèrant au moins une dépendance sont des expressions linéaires des indices de boucles, la méthode des vagues est utilisable. Cette méthode fut introduite dans [Kuck 72] puis approfondie dans [Lamp 74] et dans [Bill X]. Elle permet d'effectuer les différentes itérations du corps de boucle en une suite de vagues d'itérations, les différentes itérations de chaque vague pouvant être effectuées en parallèle:

```
        DO 10 I = 2,N
                DO 10 I=2,N
10                      X(I,J) = X(I-1,J) + X(I,j-1) + C
```

```
        DO 101 I = 2,N
                DOALL 101 J = I,N
101                     X(I-J+1,J) = X(I-J,J) + X(I-J+1,J-1)

        K = 3
        DO 102 I = N+1,2N-1
                DOALL 102 J = K,N
102                     X(I-J+1,J) = X(I-J,J) + X(I-J+1,J-1)
```



x éléments déjà calculés
o éléments d'une vague calculables en parallèle.
. éléments non encore calculés.

Dans le cas où la distribution de boucle n'à pas été effectuée mais que le corps de boucle a été partitionné, il est possible de faire chevaucher les exécutions des différentes partitions pour chaque itération; c'est la boucle DOPIPE de PARAFRASE [Davi 81]. L'exemple suivant montre la parallélisation en DOALL et en DOPIPE de la même boucle ainsi que les diagrammes de temps correspondants.

```
                DO 10 I = 1,N
                        T(I) = Z(I) * S(I)      (a)
10                      R = R + T(I)            (b)
```

Version SEQUENTIELLE

```
        DOPIPE 10 I = 1,N
        SEGMENT (10,1)
              T(I) = Z(I) * S(I)
        SEGMENT (10,2)
              R = R + T(I)
10      CONTINUE
```

Version DOPIPE

```
        DOALL 101 I = 1,N
101           T(I) = Z(I) * S(I)

        DO 102 I = 1,N
102           R = R + T(I)
```

Version DOALL

temps processeurs	1	2	3	4	5	...	N	N+1
P1	a1	b1	b2	b3	b4	...	bN-1	bN
P2	a2	-	-	-	-	...	-	-
P3	a3	-	-	-	-	...	-	-
.	.	.	.	.	.	.	.	.
PN	aN	-	-	-	-	...	-	-

temps processeurs	1	2	3	4	5	...	N	N+1
P1	a1	a2	a3	a4	a5	...	aN	-
P2	-	b1	b2	b3	b4	...	bN-1	bN

DOALL => N processeurs & N + 1 unités de temps
DOPIPE => 2 processeurs & N + 1 unités de temps (gain)

II.3.3. Remarques sur les choix à effectuer pendant la parallélisation

Un grand nombre de choix doivent être effectués an cours de la parallélisation. Ceux-ci concernent différents problèmes:

(1) choix et ordonnancement des transformations; vaut-il mieux effectuer la substitution variable-valeur ou l'expansion des scalaires ? Vaut-il mieux systématiquement effectuer le renommage des scalaires avant l'expansion ? Vaut-il mieux remplacer un IF par une affectation conditionnelle ou par un vecteur de contrôle ? etc ...

(2) ordre de traitement des boucles; VESTA analyse les boucles une à une par une procédure récursive appelée pour la boucle la plus externe; ceci implique que les boucles internes sont traitées avant les boucles externes, ce qui a sans doute des conséquences notamment pour l'expansion des scalaires. PARAFRASE traite globalement un ensemble de boucles bien imbriquées [Kuck 81c].

(3) choix du parallélisme exposé; là encore le choix est multiple. Faut-il systématiquement effectuer la distribution de la fonction de boucle en vue d'obtenir des boucles DO-// ou DOCUMUL ? faut-il au contraire rechercher des boucles DOPIPE ? faut-il regrouper plusieurs boucles vectorielles en une seule pour diminuer l'overhead ?

Il semble que répondre à ces questions n'est pas chose simple car de nombreux facteurs doivent être pris en compte concernant la machine cible (iii) (SIMD, MIND, mémoire virtuelle ou non, taille de la mémoire, nombre de registres, d'unités fonctionnelles vectorielles, etc...) et la boucle (nombre d'instructions, complexité des instructions, etc...). Les logiciels existants semblent traiter ces problèmes cas par cas.

(iii) VESTA devait à l'origine effectuer une parallélisation paramétrée par les caractéristiques de la machine cible.

II.3.4. Etat d'avancement des logiciels, autres projets

PARAFRASE semble être actuellement le paralléliseur donnant les meilleurs résultats. Remarquons que D. J. Kuck et son équipe travaillent sur le sujet depuis 1970. Ce logiciel est à même de traiter Fortran dans sa totalité et de générer du code pour différentes machines dont les CRAYs. VESTA est encore à l'état de prototype: il ne traite pas Fortran complet et génére du source agrémenté de constructions vectorielles (DO-// et DOCUMUL).

D'autres projets similaires sont en cours de réalisation; citons notamment VAST, un vectoriseur pour le CYBER 205 [Brod 82] et PFC un programme destiné à convertir Fortran en Fortran 8X [Alle 82].

Citons enfin les compilateurs-vectoriseurs commerciaux et notamment ceux du FACOM VP-200 (Fujistsu) et du Hitachi S-810 qui tirent profit des capacités des machines en reconnaissant des macro-instructions telles que produit scalaire de deux vecteurs, recherche du maximum d'un vecteur, récurrence de premier ordre ($A_i = A_{i-1} * B_i + C_i$) etc.... Citons encore, parmi les plus connus, le vectoriseur du CRAY-1, qui est loin d'être parmi les meilleurs, et parmi ceux tournant sur les "array-processors" - machines vectorielles peu chères conçues pour accélérer une machine traditionnelle telle qu'un VAX 780 - celui du FPS AP164-MAX.

II.4. Conclusion

Nous venons de présenter différentes méthodes de parallélisation automatique. Nous proposons dans cette thèse trois améliorations visant à élargir leur champ d'application (appels de sous-programmes) et les machines cibles utilisables (machines ME*). La présentation de ces améliorations est faite au chapitre III.

CHAPITRE III

III. Propositions d'améliorations des méthodes de parallélisation existantes

Dans ce chapitre, nous rappelons que la structure la plus usuelle d'un paralléliseur se décompose en quatre parties. Les trois premières réalisent respectivement l'analyse, la transformation et la parallélisation du programme. La quatrième à pour but de répondre aux questions d'ordre sémantique qui sont posées par les parties transformation et parallélisation lors de leur exécution.

Les améliorations que nous proposons concernent la partie analyse; elles ont pour but d'augmenter la quantité d'informations mises à la disposition des trois autres parties de façon que les résultats de ces dernières soient de meilleure qualité. Nous proposons trois améliorations.

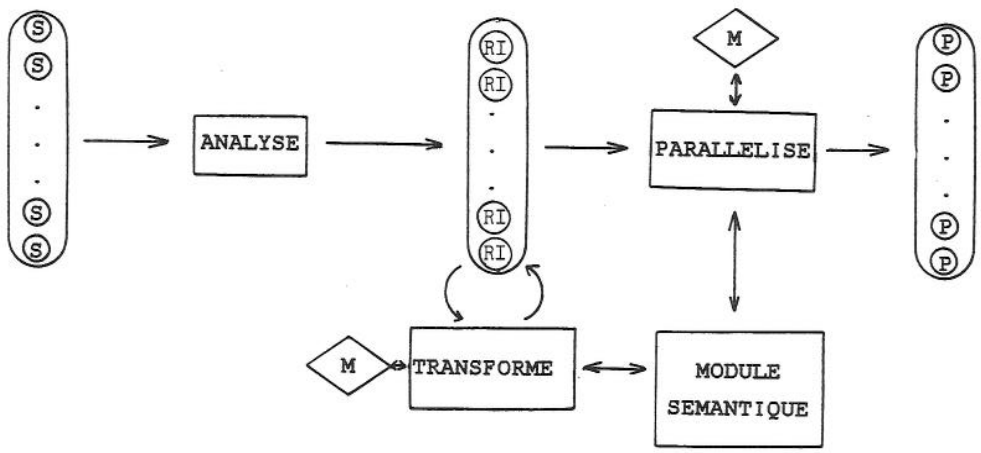
(1) La prise en compte de l'aliasing; nous montrons pourquoi il nous parait important de le calculer et d'en tenir compte dans le processus de parallélisation.

(2) La parallélisation d'instructions comportant des appels de procédure; nous montrons comment ces instructions sont traitées par les paralléliseurs existants, puis nous présentons brièvement notre solution. Nous montrons notamment qu'elle est directement applicable sur une machine d'ores et déjà commercialisée.

(3) L'utilisation d'assertions sur les variables du programme; nous présentons les raisons qui nous conduisent à utiliser des assertions, puis différentes méthodes de calcul qui produisent chacune des assertions de type différent. Nous expliquons pourquoi nous retenons le type "inégalité linéaire entre variables simples entières". Enfin nous montrons que ces assertions, initialement calculées pour améliorer la recherche des parties de tableaux manipulées par un appel de procédure, peuvent être utilisées à d'autres fins.

III.1. Structure générale d'un paralléliseur

La figure III-1 montre la structure usuelle d'un paralléliseur. Le découpage que nous proposons n'est pas nécessairement aussi net dans les paralléliseurs existants.



S : texte source d'une procédure

RI: représentation interne d'une procédure
P : forme parallèle d'une procédure
M : description de la machine cible

Figure III-1: Structure usuelle d'un paralléliseur

III.1.1. La partie ANALYSE

Cette partie à pour but de produire une représentation interne du programme à partir de son texte séquentiel. Par texte séquentiel nous entendons la forme du programme donnée par le programmeur; ce sera généralement un ensemble de modules sources Fortran-77, éventuellement enrichi de directives ou d'informations fournies par le programmeur. Par représentation interne (RI) nous entendons une forme du programme où la décomposition du texte initial à été effectuée; les caractéristiques du programme sont alors directement accessibles par les autres parties du paralléliseur.

En plus du calcul des renseignements usuellement contenus dans une RI destinée à un compilateur classique: table des symboles, description des instructions, graphe de contrôle, etc..., la partie ANALYSE doit effectuer les deux traitements suivants.

(1) L'identification des opérateurs que les parties suivantes vont tenter d'exécuter en paralléle. Ce traitement ne pose pas de problème particulier pour Fortran-77 car les instructions sont simples et généralement associées à un opérateur unique; seule l'instruction DO est associée à plusieurs opérateurs de façon à pouvoir simuler son fonctionnement.

(2) La recherche des variables lues et modifiées par chaque opérateur. Ce traitement est nécessaire pour que la recherche des dépendances (cf. II.3.2.2) - mises sous forme d'un graphe ou d'une relation entre opérateurs - puisse s'effectuer par la suite. Les opérateurs contenant des références à des sous-programmes étant généralement exclus ou traités par une analyse pessimiste (cf. I.4.2.1), ce traitement se ramène à une étude de la syntaxe de Fortran-77, pour recenser les expressions utilisées et les membres gauches définis par chaque instruction, suivi d'une exploration récursive de ces expressions et membres.

III.1.2. La partie TRANSFORME

Son rôle est double: rechercher les dépendances puis modifier le programme de façon à restreindre leur nombre. La recherche des dépendances doit à notre avis s'effectuer ici plutôt que dans la partie ANALYSE; en effet, la structure de données représentant les dépendances doit être entretenue après chaque transformation, il est donc plus cohérent de la faire initialiser par la même partie.

Nous avons présenté la plupart des transformations typiques des paralléliseurs dans le chapitre II. Nous y avons également insisté sur l'importance de ces transformations dans le processus de parallélisation.

III.1.3. La partie PARALLELISE

Cette partie a pour but de générer une version parallèle du programme à partir du couple final (RI, ensemble de dépendances) fourni par la partie TRANSFORME. Le résultat de cette partie peut revêtir différentes formes suivant le paralléliseur. Les vectoriseurs commerciaux produisent généralement du code pour leur machine cible. Cette forme présente

l'inconvénient d'être difficilement utilisable par le programmeur qui désire savoir ce qui à été vectorisé dans son programme. C'est pourquoi les paralléliseurs issus de laboratoires de recherche (PARAFRASE, VESTA) reproduisent du Fortran augmenté de constructions syntaxiques exprimant le parallélisme. Ce Fortran parallèle peut alors, moyennant un "préprocessing" ad-hoc, être compilé par les vectoriseurs commerciaux. Enfin les méthodes globales telles que celle de G. Roucairol produisent directement l'automate de séquencement des opérateurs.

Les parties PARALLELISE et TRANSFORME peuvent être paramétrées par la description de la machine cible, ce qui permet au paralléliseur de générer des versions différentes suivant le type de la machine cible.

III.1.4. La partie MODULE SEMANTIQUE

Les résultats des parties TRANSFORME et PARALLELISE sont améliorés par la connaissance des réponses à un ensemble de questions qu'elles se posent pendant le processus de parallélisation, tel que:

"I et J sont-ils différents en tel endroit ?", "peut-on exprimer K en fonction de I et J ?", "K est-il positif ?" etc...

C'est le rôle du MODULE SEMANTIQUE de répondre à ces questions. Le contenu de cette partie est assez flou. Celui de VESTA est la 'calculette symbolique' réalisée par P. Feautrier [Feau 84]. L'interface entre le MODULE SEMANTIQUE et les autres parties du paralléliseur peut être réalisée par un système "question-réponse".

III.1.5. Améliorations de la partie ANALYSE

Nous proposons d'améliorer la partie ANALYSE sur trois points de façon que les parties suivantes produisent des résultats meilleurs. Les trois points sont:

(1) prise en compte de l'aliasing;

(2) traitement des appels de sous-programme par recherche de leurs effets;

(3) calcul et utilisation d'assertions pour l'affinement du traitement des tableaux.

III.2. Propositions concernant l'aliasing

III.2.1. Définition de l'aliasing

Nous utilisons dans la suite la terminologie de la norme AFNOR [AFNO 83] définissant le langage Fortran-77. Les variables sont nommées entités, le terme variable étant réservé aux variables simples. A un instant donné et à chaque entité est associée une suite de mémoires. Il y a association des suites de mémoires de deux entités lorsque leur intersection est non vide; l'association est totale lorsque les suites de mémoires sont égales, partielle sinon.

L'aliasing est une forme spéciale d'association des suites de mémoires de deux entités d'une procédure, quand une au moins est un paramètre formel (pf). L'autre cause d'association de suites de mémoires est la déclarative EQUIVALENCE, dont l'utilisation est interdite sur un pf.

III.2.2. Cas d'apparition de l'aliasing

L'aliasing apparait en Fortran-77 dans 2 cas; soient P et Q deux procédures telles que P appelle Q et telles que P et Q possèdent la déclaration d'un common C.

(1) Si un des paramètres réels de l'appel à Q dans P est une référence à une entité du common C - ou à un pf de P en aliasing avec cette entité - il y a aliasing dans Q entre certaines entités de C et le pf de Q correspondant à ce paramètre réel. Les descriptions de C dans P et Q peuvent être différentes (cf exemple III-1), la suite de mémoires associée à l'entité de P peut donc intersecter celles de plusieurs entités de Q; c'est pourquoi nous avons utilisé le terme "certaines entités".

(2) Si deux paramètres réels de l'appel à P dans Q sont des références à la même entité ou à des entités différentes dont les suites de mémoires sont en association - par EQUIVALENCE ou par aliasing - il y a aliasing dans Q entre les deux paramètres formels correspondants.

Exemple III-1: soient P et Q les deux procédures suivantes:

```
      SUBROUTINE P
      COMMON /C/ M1,DELTA(12),Z1,K12,MAX(52),EPS,ZINF(36),A
      ...
      CALL Q (MAX(K12),MAX(Z1))
      ...
      END

      SUBBROUTINE Q (X,Y)
      COMMON /C/ PART1(30),PART2(75)
      X = ...
      END
```

Dans cet exemple, les descriptions de C dans P et Q sont différentes; sans connaissance des valeurs de K12 et Z1, nous devons déclarer X et Y en aliasing avec PART1 et PART2. Dans les mêmes conditions, X et Y sont en aliasing par la cause (2).

Ainsi que le montre l'exemple III-1, il est possible de combiner les deux cas d'apparition d'aliasing; nous obtenons dans ce cas aliasing entre plusieurs paramètres formels et plusieurs entités communes de Q.

III.2.3. Les différents traitements possibles de l'aliasing

L'aliasing peut à notre avis être traité de trois façons différentes:

(1) La solution la plus simple consiste à l'ignorer en invoquant le §15.9.3.6 de [AFNO 83] qui stipule que: "Si la référence à un sous-programme entraine qu'un paramètre formel dans le sous-programme référencé devienne associé à un autre paramètre formel, aucun des paramètres formels ne peut devenir défini durant l'exécution de ce sous-programme." La suite de ce paragraphe stipule l'équivalent pour l'aliasing entre entités communes et paramètres formels.
Ceci implique donc qu'une entité en aliasing dans une procédure Q avec une autre entité ne peut pas être modifiée par Q. Or une entité non modifiée ne peut pas créer de dépendances; il en résulte donc que la parallélisation d'un programme correct vis à vis de la norme, sans tenir compte de l'aliasing, sera correcte.
Le principal inconvénient de cette solution est que les compilateurs (paralléliseurs ou

non) ne détectent généralement pas l'aliasing et que le programmeur n'a donc pas la possibilité de savoir si son programme est correct, d'autant plus que la plupart des utilisations d'aliasing, bien qu'illicites, sont sans dommage pour le programme à la différence d'autres erreurs telles que la non initialisation d'une entité.

(2) Cette dernière remarque nous conduit à la deuxième solution: détecter l'aliasing et signaler au programmeur quelles sont les procédures posant problème, mais ne pas tenir compte de cet aliasing dans les algorithmes de parallélisation. Cette solution nous parait raisonnable dans la mesure où le programmeur à ainsi la possibilité de modifier les procédures fautives avant de resoumettre son programme au paralléliseur.
Il est important de noter que nous ne pouvons calculer qu'une approximation de l'aliasing et que certaines procédures seront déclarées à tort comme posant problème (cf exemple III-2).

(3) Enfin, la dernière solution consiste à calculer et conserver l'aliasing de chaque procédure, puis à en tenir compte dans les phases TRANSFORME et PARALLELISE. Cette solution nous parait la plus raisonnable car il n'est pas certain que tous les programmeurs Fortran respectent le §15.9.6.3 de [AFNO 83]. D'autre part, la prise en compte de l'aliasing ne nous parait pas poser de grandes difficultés.
Il faut remarquer que certaines procédures seront à tort non parallélisées puisque nous ne pouvons calculer qu'une approximation de l'aliasing. Il faudra donc laisser la possibilité au programmeur de forcer, par DIRECTIVE, la non prise en compte de l'aliasing. Notons que la méthode de calcul de l'aliasing que nous proposons au chapitre IX et qui utilise des assertions sur les variables des procédures permet de diminuer le nombre de faux aliasing.

Nous choisissons donc la troisième solution. Nous formulons dans le §IV.3.5.1 des conventions concernant la représentation de l'aliasing qui sont utilisées dans le chapite IX entièrement consacré à ce sujet.

III.3. Propositions concernant le traitement des appels de sous-programmes

Nous rappelons tout d'abord comment les méthodes de parallélisation éxistantes traitent les instructions comportant des appels de procédure. Nous présentons ensuite notre solution - traitement des appels de procédure par recherche de leurs effets sur la procédure appelante - sur le cas particulier des sous-routines. Nous rappelons que les programmes parallèles ainsi obtenus sont directement exploitables sur des machines déjà commercialisées. Puis, après avoir montré qu'il est possible de traiter de la même façon les appels de fonctions, nous présentons notre méthode de calcul des effets d'un appel de procédure.

III.3.1. Etat de l'art

Les différentes attitudes adoptées par les méthodes existantes de parallélisation sont les suivantes:

(1) PARAFRASE ignore les boucles comportant des appels de procédure; cette approche est, selon D.J. Kuck, tout à fait justifiée dans la mesure où les appels de procédure peuvent être expansés dans le corps de boucle avant que celle-ci soit soumise au paralléliseur. Nous verrons par la suite quelles sont les critiques que nous formulons sur cette méthode et pourquoi nous lui en préférons une autre.

(2) VESTA ignore de même les boucles comportant des appels de procédure. Les auteurs de ce logiciel font remarquer que ceux-ci ne posent pas de problème particulier si les ensembles de variables modifiées et lues par chaque appel sont connus. Cependant ils ne proposent pas de solution pour calculer ces ensembles.

(3) La notion d'opérateur, dans le cadre théorique où se place G. Roucairol, est suffisamment générale pour englober la notion d'appel de procédure. G. Roucairol suppose cependant connus les ensembles D(a) et R(a) des variables respectivement lues et modifiées par l'opérateur a. Le problème du calcul de ces ensembles lorsque a est un opérateur CALL demeure entier.

(4) Les paralléliseurs commerciaux sont généralement des vectoriseurs qui ont été conçus pour des machines vectorielles (SEA) sur lesquelles l'exécution simultanée de plusieurs procédures est très difficile à réaliser bien que possible dans certains cas: fonction SINUS vectorielle sur le FPS AP-120 par exemple.

Le bilan de ce paragraphe est que les appels de procédure posent un problème qu'à l'évidence personne n'a cherché à résoudre. Ceci est à notre avis dû au nombre restreint de machines existantes sachant exécuter plusieurs procédures en parallèle (HEP [Dong 83], ISIS [Tims 83], etc...).

III.3.2. Traitement d'un appel de procédure par recherche de ses effets

III.3.2.1. Principe

A titre d'exemple, nous exposons le principe de la solution que nous proposons sur des appels de sous-routine.

Soient M un programme principal et Q une sous-routine telle que M contienne un appel à Q. Pendant le processus de parallélisation de M, nous proposons d'associer à l'instruction CALL correspondant à cet appel, un opérateur unique, indivisible au même titre que celui associé à une instruction d'affectation. En supposant que nous soyons capables de calculer les ensembles de variables manipulées en lecture et en écriture par un tel opérateur, nous pouvons effectuer la parallélisation du programme P et aboutir ainsi à une version parallèle; celle-ci autorise l'exécution simultanée de plusieurs opérateurs ou occurences d'opérateurs, dont certains correspondent à l'exécution de la sous-routine Q.

Ce que nous venons de dire sur M peut être appliqué à Q. La parallélisation d'une procédure pose cependant des problèmes spécifiques dûs au fait qu'une procédure doit être parallélisée en fonction du contexte associé à son appel. Ces problèmes sont traités dans le chapitre IV.

Un programme parallèle n'est terminé que lorsque l'exécution de ses différents opérateurs est terminée. De même un opérateur CALL correspondant à une version parallèle d'une procédure ne se termine que lorsque tous les opérateurs de la procédure correspondante ont eux-mêmes terminé leur exécution. Nous entrons là dans le domaine du contrôle des programmes parallèles qui doit être adapté au traitement des appels de procédure. Ce problème n'est pas abordé dans cette thèse. Notons cependant que la méthode de G. Roucairol s'adapte parfaitement en associant une réalisation à files généralisées à chaque occurence de procédure (cf chap IV); l'exécution d'un opérateur CALL est particulière dans la mesure où elle initialise les files de la réalisation associée à l'occurence de procédure appelée; d'autre part, dans chaque réalisation, l'opérateur RETURN est rendu artificiellement dépendant des autres opérateurs de façon que son exécution, qui correspond à la fin de l'opérateur CALL associé, soit correctement séquencée.

L'exemple que nous avons présenté en introduction aboutit à une version paralléle de la sous-routine MM. Cette parallélisation a-t-elle un aspect pratique ? La boucle ml peut-elle effectivement être exécutée en DOALL ? La réponse à ces deux questions est OUI. Ainsi la version parallèle de la sous-routine MM peut être programmée en Fortran-HEP [Dene 82] grâce aux primitives qui y sont inclues et qui permettent de lancer plusieurs tâches en parallèle et de les synchroniser; d'autres exemples de programmes paralléles sont donnés dans [Dong 83] où le parallélisme a lieu entre différentes exécutions de procédure.

III.3.2.2. Généralisation aux appels de fonctions

Ce qui a été dit dans le paragraphe précédent est généralisable aux appels de fonctions. Nous sommes en effet tentés d'imaginer une machine sur laquelle toute instruction qui n'est pas un test pourrait être exécutée soit par le processus courant soit par un nouveau processus. Dans ce dernier cas, le processus courant ne serait pas stoppé, la synchronisation des divers processus restant à la charge du programmeur.

Un tel mécanisme serait particulièrement intéressant pour les instructions au temps d'exécution long (diminution de l'importance de l'overhead) telles que celles contenant des appels de fonctions.

Dans toute la suite, nous ne distinguons plus le type des différentes instructions. A chaque instruction est associé un ensemble d' appels d'externes qui représente l'ensemble des procédures (fonctions ou sous-routines) qui sont activées pendant l'exécution de cette instruction. Pour chaque procédure nous recherchons les effets de chaque appel d'externe individuellement.

III.3.2.3. Principe du calcul des effets d'un appel d'externe

Voici la description informelle de la méthode de calcul que nous proposons d'employer pour calculer les effets de chaque appel d'externe sur les entités de la procédure appelante.

(1) Nous commençons par rechercher quelles sont les entités susceptibles d'être manipulées par chaque procédure. Nous étudions ce problème au §V.1. Cette recherche est moins simple qu'il n'y parait, une procédure pouvant, par le biais des COMMONs et de l'instruction SAVE, manipuler indirectement des entités dont elle ne possède pas la déclaration.

(2) Nous recherchons ensuite quelles sont les entités ou parties d'entités manipulées par chaque instruction de chaque procédure. Durant cette recherche, nous utilisons des assertions sur les variables de la procédure pour améliorer les résultats; c'est l'objet du §V.5. La description d'une partie d'entité est réalisée par ce que nous appelons une région: objet défini au §V.2.

(3) Nous en déduisons l'ensemble des régions manipulées par chaque procédure par union sur l'ensemble de ses instructions suceptibles d'être exécutées (cf. §V.4).

(4) Nous reportons à présent sur chaque instruction d'une procédure appelante comportant un appel de procédure, les ensembles de régions manipulées par la procédure appelée. Ces régions décrivent des parties d'entités de la procédure appelée; nous devons les transformer en des régions décrivant des parties d'entités de la procédure appelante. Dans la suite, nous nommons ce problème traduction d'une région; il est étudié en détails dans le chapitre VI.

Il nous reste ensuite à montrer comment les régions permettent de calculer les

dépendances. Nous proposons une solution au §VIII.7 mais il est clair que ce problème sort du cadre que nous nous sommes fixés pour entrer dans celui de la réalisation du paralléliseur.

III.4. Propositions concernant l'utilisation d'assertions sur les variables d'une procédure pour améliorer le traitement des tableaux.

Dans ce paragraphe nous allons successivement montrer pourquoi nous utilisons des assertions, en présenter différents types automatiquement calculables ainsi que les méthodes de calcul associées. Enfin nous montrons que ces assertions peuvent être utilisées en d'autres occasions.

III.4.1. Pourquoi utiliser des assertions ?

La source principale de parallélisme dans un programme scientifique est le traitement simultané et souvent identique des différents éléments d'un tableau. Les manipulations qu'une procédure effectue sur ses variables doivent être recherchées soigneusement, notamment pour les tableaux, sous peine de voir disparaitre une importante partie de son parallélisme potentiel.

L'exemple III-3 montre deux versions d'une même boucle DO; dans la 2ème version, un calcul d'assertions a été effectué en utilisant les propriétés des boucles DO. Lorsqu'on s'intéresse aux éléments du tableau T manipulés par une occurence des instructions (a) ou (a') - ce que font les paralléliseurs tels que VESTA ou PARAFRASE pour la construction du graphe des dépendances - les assertions calculées sont inutiles. Il n'en est pas de même lorsqu'on s'intéresse aux éléments du tableau T manipulés par toutes les occurences des instructions (a) et (a'):

Exemple III-2:

```
        DO 10 I = 1,10
10              T(I)= ...          (a)

                                   Recherche d'assertions

        DO 10 I = 1,10
              { 1 ≤ I ≤ 10 }
10            T(I) = ...           (a')
```

La Ième occurence des instructions (a) et (a') modifie T(I).
L'instruction (a) modifie T.
L'instruction (a') modifie T([1,10]).

Pendant la recherche des manipulations d'une procédure dans le but de les reporter sur un appel à cette procédure, nous devons rechercher les manipulations de chaque instruction pour toutes ses occurences. Si celle-ci manipule des tableaux et que les lois d'évolution des indices ne sont pas connues, nous devons adopter une attitude pessimiste et considérer que la totalité du tableau est manipulée; ceci conduit généralement à une perte de parallélisme après que la traduction est effectuée.

Nous proposons donc de rechercher des assertions sur les variables de chaque procédure de façon à cerner les lois d'évolution des indices de tableau et d'en déduire des bornes supérieures et inférieures pour chaque indice.

III.4.2. Différents types d'assertions; méthodes de calcul

L'étude bibliographique que nous avons menée nous a permis de retenir trois types d'assertions calculables automatiquement. Les techniques correspondantes ont été développées dans le cadre de l'optimisation de programme, soit pour supprimer les variables de valeur constante [Kild 73], soit pour supprimer des tests dynamiques de débordement de tableau [Cous 78], [Halb 79] et [Jung 83].

(1) G.A Kildall propose dans [Kild 73] une méthode de calcul d'assertions nommée "méthode de propagation des constantes"; ces assertions sont du type "v = c" où v est une variable simple entière et $c \in N$.

(2) P. Cousot a développé dans [Cous 78] une méthode générale permettant de calculer des assertions de différents types. J.P. Jung a implémenté cette méthode sur un sous-ensemble de Pascal [Jung 82] pour calculer des assertions du type "$v \in [a,b]$" avec $a,b \in N$.

(3) N. Halbwacks a implémenté la méthode de P. Cousot sur un sous-ensemble de Pascal [Halb 79] pour calculer des assertions du type plus complexe: "$\sum c_i v_i \leq c$, avec v_i: variable simple entière et $c_i, c \in N$.

Remarquons que les types des assertions (1) et (2) sont des cas particuliers du type des assertions (3).

Nous proposons d'autre part au §VII.3 de calculer des assertions sur les variables de boucles présentant certaines propriétés en se basant sur le mécanisme des boucles DO tel qu'il est décrit dans [AFNO 83] au §11.10.3.

Enfin nous donnons au programmeur la possibilité d'introduire des assertions dans le texte du programme; nous étudions plus précisément cette possibilité au §VII.4.

III.4.3. Nos choix concernant les assertions

Nous proposons de calculer puis d'utiliser des assertions du type

"relation linéaire entre variables simples de type INTEGER"

Dans la suite, nous les désignerons par assertions linéaires. Les raisons de ce choix sont les suivantes:

(1) Nous nous bornons aux variables simples car aucune technique ne permet actuellement de calculer des assertions sur les entités tableaux.

(2) Nous utilisons ces assertions en premier lieu pour affiner le traitement des tableaux; dans cette optique, le type INTEGER est suffisant (à la différence de [Kild 73])

(3) Les assertions linéaires sont les assertions les plus complexes que les techniques étudiées permettent de calculer.

(4) La manipulation des assertions linéaires est particulièrement simple grâce à leur représentation matricielle.

Nous verrons dans le chapitre VII comment adapter les méthodes présentées au paragraphe précédent à notre cas, les principaux problèmes venant des instructions comportant des appels d'externe et des associations de suites de mémoires (aliasing et EQUIVALENCE).

III.4.4. Utilisations secondaires des assertions

Ces assertions peuvent être utilisées en d'autres occasions. Nous les utilisons notamment pour éliminer des cas d'aliasing (cf. chapitre IX), pour éliminer des branches mortes à l'intérieur d'une procédure (cf chapitre V) ou pour améliorer la construction du graphe de dépendances d'une boucle (cf. chapitre VIII). Ces utilisations secondaires, présentées dans les chapitres adéquats, nous ont demandés de savoir évaluer une expression ou d'en comparer deux. Les techniques que nous utilisons pour cela, généralement trouvées dans la littérature et légèrement adaptées, sont décrites dans le chapitre VIII.

C H A P I T R E IV

IV. Problèmes spécifiques au traitement des procédures

Ce chapitre a pour but de présenter les problèmes qui sont posés par l'application de notre méthode sur un programme composé d'un ensemble de procédures, lorsque celles-ci s'appellent sur plusieurs niveaux. Il se décompose en trois parties.

La première partie a pour but de montrer que la méthode de l'expansion, si elle est très utile pour une application locale, ne permet que difficilement de faitre disparaître la totalité des procédures. Cette discussion nous parait importante car elle justifie en partie notre travail.

Dans la deuxième partie nous montrons que les procédures peuvent être étudiées à plusieurs niveaux de précision, le travail à effectuer augmentant avec la précision des résultats. Nous montrons quel niveau nous choisissons, pourquoi, et quelles sont les conséquences de ce choix sur l'organisation d'un paralléliseur.

Enfin, dans la troisième partie, nous introduisons les notations et définitions que nous utilisons dans les chapitres suivants:

IV.1. La méthode de l'expansion de procédure

IV.1.1. Principe

Le principe de l'expansion de procédure est très simple: remplacer chaque appel de procédure par le corps de la procédure appelée. Pour que le résultat soit correct, certaines transformations sont nécéssaires: addition dans l'appelante des déclarations et initialisations (DATA) des variables locales de la procédure appelée, substitution dans le corps de l'appelée des paramètres formels par les paramètres réels, etc...

Cette méthode peut être pratiquée systématiquement - tous les appels de procédure du programme sont expansés - ou sélectivement - seuls sont expansés les appels répondant à certains critères (présence du source, procédure appelée une seule fois, procédure ne contenant pas de retour secondaire, etc...)-.

Dans le cas d'une application systématique, nous pouvons procéder de façon descendante ou ascendante. L'expansion descendante présente l'avantage que la procédure appelante est toujours un programme principal; un paramètre réel de l'appel à expanser ne peut donc pas être un paramètre formel, ce qui supprime les problèmes relatifs aux tableaux ajustables et/ou de taille non spécifiée (cf. [AFNO 83] §5.1.2). L'expansion ascendante présente l'avantage que la procédure appelée ne contient pas elle-même d'appel de procédure, ce qui élimine le problème, peu important, des procédures formelles. Nous préférons donc "l'expansion systématique descendante" à "l'expansion systématique ascendante".

IV.1.2. Avantages de cette méthode

Le principal avantage de cette méthode est de faire disparaître les appels de procédure. Un vectoriseur tel que PARAFRASE, qui traite les appels de procédure de façon pessimiste, vectorisera d'une bien meilleure façon la version d'une procédure où tous les appels ont été expansés que la version originale, ainsi que le montre l'exemple suivant.

Exemple IV-1: Voici quels seraient les résultats d'un vectoriseur tel que PARAFRASE sur la boucle de la subroutine MM dans les deux cas:

Version originale: laissée inchangée !

```
        DO 10 I = 1,N3
10              CALL SMXPY(N2,A(1,I),N1,LDB,C(1,I),B)
```

version expansée avant vectorisation

```
        DO 10 I = 1,N3
                DO 10 J = 1,N1
                        A(J,I) = 0
                        DO 10 K = 1,N2
10                              A(J,I) =  A(J,I) + C(K,I) * M(J,K)
```

version expansséee après vectorisation: le corps de la boucle sur J est coupée en deux; la lère partie est une double boucle DOALL; la 2ème partie aussi une fois que les entêtes de boucle ont été correctement permutés:

```
        DOALL 101 I = 1,N3
                DOALL 101 J = 1,N1
101                     A(J,I) = 0

        DO 102  K =1,N2
                DOALL 102 I = 1,N3
                        DOALL 102 J = 1,N1
102                             A(J,I) = A(J,I) + C(K,I) * M(J,K)
```

De plus, si l'expansion systématique est réalisée et que les procédures disparaissent complètement, toutes les notions "ennuyeuses" qui leurs sont attachées disparaissent à leur tour: paramètres formels et réels, association de paramètres, tableau ajustable et/ou de taille non spécifiée, variables dynamiques et statiques, variables locales et communes, ENTRY etc... Tous les problèmes introduits par ces notions disparaissent.

Un autre avantage de cette méthode est d'être particulièrement adaptée aux machines SEA; nous avons déjà dit au §I.4.2.4 que la boucle de la subroutine MM, transformée en DOALL mais sans expansion de l'appel qu'elle contient, serait exécutée séquentiellement sur certaines machines dites parallèles: CRAY-1, CYBER-205, BSP etc... Au contraire la version expansée et vectorisée de l'exemple IV-1 leurs est tout à fait adaptée.

L'expansion de procédure peut conduire à certaines optimisations. Une procédure appelée à plusieurs endroits du programme sera chaque fois expansée dans un "contexte d'appel" particulier; les paramètres formels associés à des paramètres réels constants seront remplacés par leur valeur. Dans ces conditions, l'utilisation du MODULE SEMANTIQUE par le processus réalisant l'expansion peut éventuellement permettre la suppression de branches mortes dans le corps de l'appelée.

Enfin l'expansion de procédure augmente le parallélisme potentiel entre opérateurs différents du programme. La solution que nous proposons impose de fausses dépendances entre un opérateur de la procédure appelante et ceux de l'appelée dès qu'il existe un opérateur de l'appelée qui est en conflit avec cet opérateur de l'appelante. Au contraire, ces fausses dépendances ne sont pas crées quand l'expansion est réalisée; il en résulte cependant qu'un plus grand nombre de dépendances doit être testé.

IV.1.3. Inconvénients de cette méthode

Cette méthode n'est à notre avis critiquable que dans le cas d'une pratique systématique.

L'expansion n'est pas une méthode orientée vers le multiprocessing. En l'état actuel de la technologie, il n'est pas envisageable de pratiquer le multiprocessing au niveau de l'instruction Fortan-77, en raison de l'overhead qui en résulterait. L'inefficacité de l'exécution d'un programme sur une machine ME* croit avec la granularité de sa parallélisation.

L'expansion, en augmentant cette granularité diminue donc son multiprocessing potentiel. Nous classons cette remarque dans les inconvénients puisque, comme nous l'avons dit dans le chapitre I, la classe des machines à l'avenir prometteur est justement la classe des machines ME*.

Cette méthode, si elle est utilisée systématiquement, peut conduire à la création de programmes principaux très gros. Nous avons effectué des mesures sur deux logiciels scientifiques disponibles au C.A.I.

(1) Un programme d'optimisation de la distribution d'eau dans la région parisienne ouest. Ce programme se compose de 73 procédures totalisant 12225 lignes de Fortran-77.

(2) Un programme d'interprétation d'électro-vectocardiogramme. Ce programme se compose de 85 procédures totalisant 13000 lignes de Fortran IV.(i).

Pour chaque procédure, nous avons calculé les trois nombre suivants: (l'unité est le caractère; tous les commentaires ayant été supprimés).

l1: longueur de la procédure;

l2: somme des longueurs de toutes les procédures appelées directement ou indirectement par la procédure, y compris elle-même.

l3: longueur de la procédure si l'expansion est réalisée systématiquement dans son corps, de façon récursive.

Le chiffre l2 est approximativement proportionnel au volume de code nécessaire pour exprimer les traitements réalisés par cette procédure, lorsque la structure du programme en procédures est conservée; il est donc approximativement proportionnel au travail à effectuer pour paralléliser ou compiler cette procédure.

Le chiffre l3 a les mêmes significations que l2 lorsque la structure du programme n'est pas conservée mais que l'expansion est réalisée.

Les résultats sont données pour toutes les procédures du logiciel (1) dans la table IV-1 et pour le programme principal ("comaib") du logiciel (2) dans la table IV-2. Les procédures pour lesquelles l2=l3="-" sont des procédures ne comportant aucun appel, d'où l1=l2=l3.

(i) Le nombre de lignes est peu significatif à cause des commentaires, nombreux pour le logiciel (1) et presque inexistants pour le logiciel (2).

nom	11	12	13	nom	11	12	13
arbre	2314	-	-	mainco	2324	139752	607313
arbrsy	1874	-	-	lectot	1226	3191	3191
autosp	5755	11167	12034	misapr	564	-	-
brooks	229	-	-	modipr	699	-	-
charge	8511	8927	8927	moncoo	973	74769	169291
coepo	796	1167	1909	monrec	636	38238	82020
comaut	806	1673	2540	openfi	416	-	-
conre	1646	1810	2138	pinit	1292	-	-
coure	951	-	-	placev	840	-	-
courem	888	-	-	pomant	864	-	-
couvai	669	-	-	prinim	3001	6995	10310
dedebl	1713	2285	2285	prodym	7681	18382	29109
dedeb2	1959	2531	2531	prodyf	10173	15435	22847
dedeb4	3664	4236	4236	range	10736	11152	11152
demai	1468	-	-	reinr	371	-	-
destov	1549	-	-	relax	734	-	-
donopt	6348	-	-	tvcp	164	-	-
doptsy	3861	-	-	remod	1401	10322	12208
ecrier	642	-	-	report	661	-	-
ecrito	1228	1644	1644	resk0	1023	1558	1558
eqma2	572	-	-	resol	1203	9706	12906
evalit	2136	5014	5014	resolm	2851	-	-
finrec	3289	45862	209296	simaut	8219	13631	14498
genesy	3660	16627	16627	simulf	10914	18359	22098
iniut	322	-	-	simulh	9703	32303	194491
lecrto	1207	1623	1623	simulm	8514	18176	22584
lectge	10023	45386	46935	simulo	10132	17577	21316
lectlo	5110	10310	10310	somme	255	-	-
liposy	2920	-	-	stokfb	3706	-	-
lisbel	277	3983	7689	stokva	2878	-	-
lispom	4971	-	-	surpre	2789	18314	60780
lesvan	2420	-	-	termin	3819	66788	236158
lisvar	5252	-	-	tri	887	-	-
locasy	3578	6498	6498				

Table IV-1: logiciel 1

nom	11	12	13
comaib	1915	416724	1405354

Table IV-2: logiciel 2

De ces mesures nous pouvons tirer les conclusions suivantes:

(1) Pour la majorité des procédures, l'expansion des appels qu'elles contiennent n'implique pas une augmentation considérable du volume de code; on remarquera en effet que pour 60/67 d'entre-elles le rapport $\frac{13}{12}$ ne dépasse pas 2.

(2) Seul le programme principal du 1er logiciel voit ce rapport dépasser 4; ce qui n'est pas encore catastrophique.

(3) Les deux conclusions précédentes sont en faveur de cette méthode; l'intuition conduisait en effet à penser que ces chiffres devaient être plus importants.

(4) Par contre, alors que la plus grosse procédure du logiciel 1 mesure 11000 octets, le programme principal qui résulterait d'une expansion totale mesure 607313 octets. Le rapport s'élève alors à 55. Pour le logiciel 2, ce même rapport s'élève à 70. Cela signifie que la pratique de l'expansion totale conduirait un compilateur-vectoriseur à traiter des procédures d'un ordre de grandeur sans comparaison avec l'ordre de grandeur habituel. Ceci impliquerait des temps de traitement élevés aussi bien en CPU qu'en échange disque; ceci impliquerait aussi un vectoriseur de taille importante pour pouvoir contenir la représentation interne d'un tel programme.

De la même façon qu'un compilateur peut faire de la compilation séparée, un vectoriseur peut faire de la vectorisation séparée; les avantages sont les mêmes que pour la compilation séparée. La pratique de l'expansion ne permet pas de tirer profit de cette possibilité puisque chaque procèdure contenant un appel à une procédure modifiée devra être revectorisée (et donc modifiée). Si l'expansion est pratiquée totalement, l'ensemble du programme principal devra être retraité à chaque modification, même minime.

Si l'expansion doit être pratiquée systématiquement et que d'autre part tout programme doit être accepté par le paralléliseur, il est nécessaire que celui-ci sache réaliser l'expansion pour Fortan-77 complet. Nous avons étudié cette méthode et dressé un catalogue de problèmes qui se posent pour son application à Fortran-77. Une ou plusieurs de nos solutions accompagnent généralement chacun de ces problèmes. Cette liste se trouve dans [Trio 84]. Il est important de noter que quelques-uns de ces problèmes n'apparaissent que lorsqu'un programme source doit être généré après l'expansion.

IV.1.4. Principaux problèmes liés à la réalisation de l'expansion

Nous classons ces problèmes en 4 catégories (soit P la procédure appelante et Q la procédure à expanser):

(1) problèmes liés aux entités de Q qui ne sont pas des paramètres formels;

(2) problèmes liés aux associations paramètres réels/paramètres formels;

(3) problèmes liés aux initialisations par DATA dans Q;

(4) problèmes liés aux instructions exécutables dans Q;

(1) Toutes les entités de Q exceptées les entitées locales dynamiques posent des problèmes: P ne possède pas nécessairement la déclaration de tous les commons manipulés par Q; P et Q ne possèdent pas nécessairement les mêmes déclarations de leurs commons communs; Q peut manipuler des commons dits dynamiques (cf. §V.1.3) pour dialoguer avec les procédures qu'elle appelle; ces commons, inconnus de P, doivent être traités soigneusement pour ne pas introduire de fausses dépendances quand plusieurs appels à Q sont expansés dans P; etc...

(2) Il faut simuler le mécanisme de passage de paramètres d'un compilateur. Cette catégorie de problèmes est la plus importante. Nous pouvons simuler un passage par valeur ou par valeur-résultat en générant les instructions d'affectation nécessaires; pour les tableaux il est alors important de sauver les valeurs des indices qui peuvent varier par effet de bord. Nous pouvons simuler un passage par adresses grâce aux déclaratives EQUIVALENCE qui ne peuvent hélas être utilisées pour les paramètres réels tableaux qui sont eux-mêmes des paramètres formels de P. Enfin, pour les cas les plus " bizarres" d'associations, il nous faut utiliser la linéarisation des tableaux; cette méthode est assez complexe à implémenter dans certains cas; nous classons l'exemple suivant dans les cas "bizarres": "le paramètre réel de l'association est un paramètre formel de P, tableau ajustable tel que des variables entières globales, modifiées dans Q, sont utilisées dans sa déclaration; même chose pour le paramètre formel de l'association".

(3) Les problèmes liés aux initialisations par DATA sont essentiellement dûs à l'inconsistance du mécanisme de cette déclarative pour les entités locales dynamiques de Q (cf. §V.1); ces problèmes sont peu importants.

(4) Les problèmes liés aux instructions exécutables de Q sont de deux natures différentes. Il faut reporter dans chaque instruction les solutions choisies en (1) et (2) pour chaque entité; ceci implique des substitutions de noms, des réindexages etc... Il faut régler les problèmes spécifiques à chaque instruction: retours secondaires, conflits sur les labels etc ...

IV.1.5. Conclusions sur la méthode de l'expansion

Nous pensons que cette méthode est difficilement applicable systématiquement aussi bien au niveau difficulté de réalisation qu'au niveau intérêt pour la parallélisation.

Cette méthode reste cependant très intéressante pour une application locale sur certaines procédures. Nous devons la considérer comme une transformation de programme, complémentaire de notre solution, que tout paralléliseur devrait proposer dans son catalogue.

IV.2. Approfondissement du traitement des procédures

Puisque la méthode de l'expansion systématique ne semble ni raisonnable ni intéressante, nous approfondissons l'approche que nous avons présentée au §III.3.2. Nous allons voir dans ce paragraphe que les procédures peuvent être analysées à plusieurs niveaux de précision. Nous verrons ensuite quel niveau nous choisissons et quelles sont les conséquences de ce choix sur les parties TRANSFORME, PARALLELISE et MODULE SEMANTIQUE.

IV.2.1. Les différents niveaux d'analyse d'une procédure

L'analyse d'une propriété pour une procédure peut se faire à trois niveaux qui sont:

(1) Le niveau de l'occurence textuelle: la propriété est recherchée sans tenir compte des différents appels à cette procédure,

(2) le niveau de l'occurence statique: la propriété est recherchée pour chaque appel statique qui est fait à cette procédure,

(3) le niveau de l'occurence dynamique: la propriété est recherchée pour chaque appel dynamique qui est fait à cette procédure.

IV.2.1.1. Occurence textuelle d'une procédure

Au niveau de l'occurence textuelle, une procédure est analysée indépendamment de ses appels et à fortiori indépendamment de ses différentes exécutions. L'analyse de la propriété est donc unique et le résultat obtenu doit être vrai pour toutes les exècutions de cette procédure. PP L'avantage de ce niveau est évident: l'analyse de la propriété ne doit être faite qu'une fois, ce qui implique une bonne efficacité. L'inconvénient est tout aussi évident: le résultat est unique pour toutes les exécutions de la procédure, ce qui implique un manque de précision.

L'aliasing associé à une procédure est crée par les appels à cette procédure. Soit Q la procédure de l'exemple IV-2 et la propriété à calculer: "X et Y sont-ils en aliasing ?"; l'analyse au niveau occurence textuelle associera "vrai" à cette priopriété à cause des appels (b) et (c); ce qui pénalisera les exécutions a1, c1, c2, c3, c5, c6, et c7.

La connaissance d'assertions initiales (associées à l'entrée d'une procédure) améliore le résultat d'un calcul automatique d'assertions (cf. chapitre VII). Soit Q la procédure de l'exemple IV-2 et la propriété à calculer: "valeur initiale de L"; l'analyse au niveau occurence textuelle associera "?" (valeur inconnue) à cette propriété.

Exemple IV-2: Soit MAIN un programme principal et Q une procédure:

```
      PROGRAM MAIN
      REAL T1,T2,T(7)
      ...
      CALL Q(T1,T2,3)                    (a)
      ...
      CALL Q(T1,T1,4)                    (b)
      ...
      DO 10 I = 1,7
10         CALL Q(T(4),T(I),I)           (c)
      ...
      END


      SUBROUTINE Q (X,Y,L)
      ...
      END
```

En supposant que les "..." ne contiennent pas d'appel à Q, cette procédure est exécutée 9 fois; ces 9 exécutions sont notées:

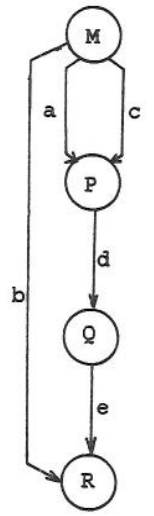
a1	: exécution	associé	à	l'appel	(a)	
b1	:	"	"	"	(b)	
c1 à c7	:	"	"	"	(c)	pour I=1 à 7

IV.2.1.2. Occurence statique d'une procédure

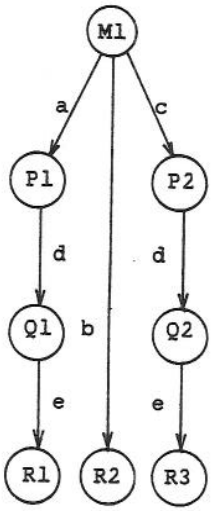
Au niveau de l'occurence statique, une procédure est analysée autant de fois qu'il y a de chaines d'appels conduisant à son exécution. Une chaine d'appels représentant le chemin qu'il faut parcourir dans le graphe des appels du programme pour aboutir à cette procédure. L'ensemble des occurences statiques des procédures d'un programme se représente graphiquement par le graphe des occurences statiques d'appels.

Exemple IV-3: Voici un programme, son graphe des appels et son graphe des occurences statiques d'appels:

```
PROGRAM M          SUBROUTINE P       SUBROUTINE Q       SUBROUTINE R
...                ...                ...                ...
CALL P  (a)        CALL Q  (d)        CALL R  (e)        END
...                ...                ...
CALL R  (b)        END                END
...
CALL P  (c)
...
END
```



Graphe des appels



Graphe des occurences statiques d'appels

P: (a) & (c)

Q: (a,d) & (c,d)

R: (a,d,c), (b) & (c,d,e)

chaines d'appels

L'analyse des procédures au niveau de l'occurence statique produit plusieurs résultats par procédure. Par rapport au niveau précédent, nous notons une perte d'efficacité mais un gain en précision.

Revenons à la procédure Q de l'exemple IV-2; nous notons respectivement Q_a, Q_b, et Q_c les occurences statiques de Q correspondant aux chaines d'appels (a), (b) et (c). La table suivante montre l'amélioration des résultats:

PROPRIETE	Q_a	Q_b	Q_c
"X et Y sont-ils en aliasing ?"	FAUX	VRAI	VRAI
"Valeur initiale de L"	3	4	?

A une occurence statique de procédure correspond généralement plusieurs exécutions de la procédure; le résultat associé à chaque occurence statique doit être vrai pour toutes les exécutions correspondantes.

IV.2.1.3. Occurence dynamique d'une procédure

Au niveau de l'occurence dynamique, une procédure est analysée autant de fois qu'elle est exécutée pendant l'exécution du programme auquel elle appartient. Nous citons ce niveau d'analyse mais n'envisageons pas de l'utiliser. Notons cependant que c'est le niveau d'analyse le plus précis possible. Ainsi l'analyse à ce niveau de la procédure Q de l'exemple IV-2 montre que X et Y sont en aliasing uniquement dans les occurences dynamiques associées aux exécutions b1 et c4; quand à la valeur initiale de L, elle est connue pour chacune de ces occurences dymaniques.

IV.2.2. Résultats d'une analyse au niveau occurence statique de procédure

Nous proposons d'analyser les procédures au niveau de l'occurence statique et de calculer pour chacune les trois fonctions suivantes:

(1) une fonction (ii) dite d'aliasing indiquant quels sont les couples d'entités en aliasing pour cette occurence statique.

(2) une fonction dite d'assertion permettant d'associer à chaque opérateur de cette occurence statique un ensemble d'assertions linéaires.

(3) une fonction dite de manipulation permettant d'associer à chaque appel d'externe de cette occurence statique, les listes de régions lues et modifiées par une exécution quelconque de cet appel. Nous remarquons qu'à un appel d'externe d'une occurence statique de procédure correspond une occurence statique de procédure unique.

Vocabulaire: sans un but de clarté, nous parlons dans la suite de procédure pour occurence textuelle de procédure et d'occurence de procédure pour occurence statique de procédure.

Ce choix a été dicté par les raisons suivantes:

- Les résultats obtenus au niveau occurence de procédure sont plus précis que ceux obtenus au niveau procédure.

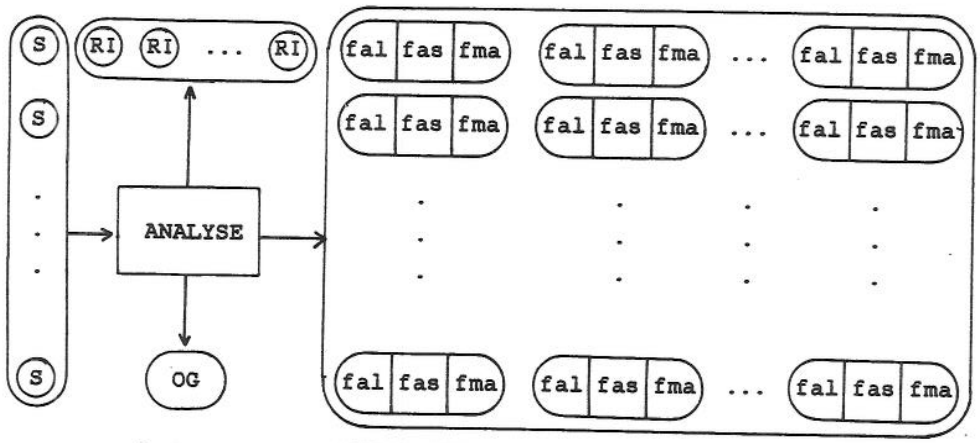
- Les résultats obtenus au niveau occurence de procédure sont transformables en des résultats au niveau procédure; il suffit pour cela de faire l'union des résultas obtenus pour chaque occurence de procédure. L'union d'ensembles de couples de variables ne pose pas de problème. Il n'en est pas de même de l'union d'ensemble d'assertions, qui fait appel à la théorie des polyèdres convexes et est connu sous le nom de "enveloppe convexe de deux polyèdres "; on se reportera à [Halb 79] qui propose une solution pour des polyèdres de R^n. Enfin l'union d'ensemble de régions fait appel à l'union d'ensemble d'assertions et s'opère donc de la même façon.

Le résultat que notre phase ANALYSE associe à chaque procédure est donc formé:

- d'une représentation interne (cf. §III.1.1),

(ii) nous parlons de fonction pour ne pas préjuger de l'implémentation qui en sera faite: tables, liste de couples etc...

- d'un ensemble de triplets de fonctions, chaque triplet étant associé à une occurence de procédure.



OG: Objets Globaux (commons, procédures, graphe des occurences)
(d'appels etc...)

figure IV-I

IV.2.3. Implications de ce résultat sur les autres parties du paralléliseur

Nous supposons que les parties TRANSFORME, PARALLELISE et MODULE SEMANTIQUE savent utiliser le triplet de fonctions associée à une occurence de procédure pour sa parallélisation; nous verrons par exemple comment tester la dépendance entre deux opérateurs en tirant profit de ce triplet au §VIII.7.

Quelles sont alors les possibilités de parallélisation ?

(1) Le paralléliseur calcule une version parallèle pour chaque procédure en pratiquant l'union des différents triplets qui lui sont associés.

(2) Le paralléliseur calcule une version parallèle pour chaque occurence de procédure.

(3) Le paralléliseur effectue un partitionnement des occurences d'une même procédure.

Les critères de partitionnement sont nombreux; étant donné l'importance jouée par le graphe des dépendances dans le processus de parallélisation, le partionnement suivant nous parait intéressant: "deux occurences de la même procédure appartiennent à la même partition si les valeurs initiales de ce graphe, calculées par la phase TRANSFORME, sont égales".

Le paralléliseur pratique ensuite l'union des triplets de fonctions associés à chaque partition et calcule ainsi une version parallèle de la procédure par partition.

Le partitionnement minimum (1 partition par procédure) conduit à la solution (2), le partitionnement maximum (1 partition par occurence de procédure) conduit à la solution (1).

Le domaine des modifications à apporter au reste du paralléliseur pour que celui-ci tire profit des améliorations que nous apportons à la partie ANALYSE n'a pas été étudié. Nous estimons que cela sont du cadre de cette thèse.

IV.3. Définitions, notations et hypothèses

IV.3.1. Décomposition de la partie ANALYSE

Nous décomposons la partie ANALYSE en trois phases:

(1) la phase ASSIA: Analyse Syntaxique et Sémantique IntrAprocédurale,

(2) la phase ASSIE: Analyse Syntaxique et Sémantique IntErprocédurale,

(3) la phase ASI : Analyse Sémantique Itérative

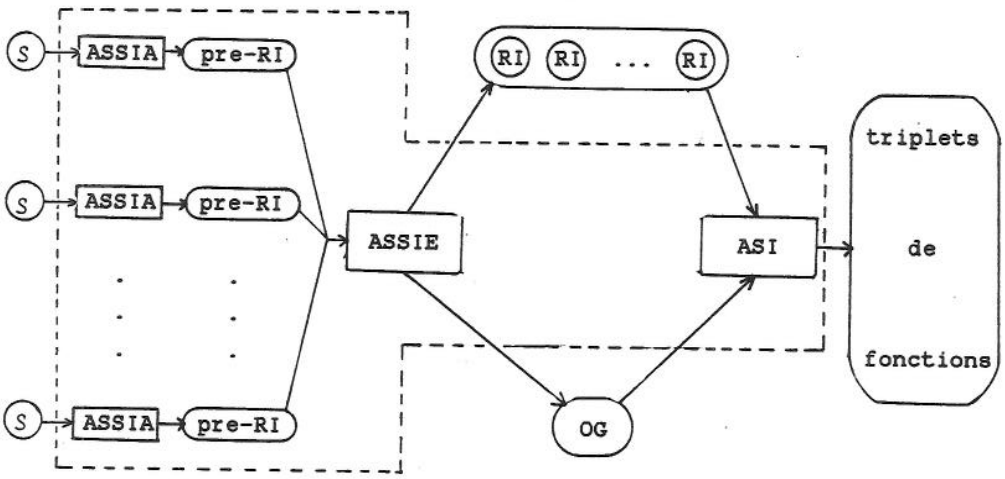


Figure IV-2: Décomposition de la phase ANALYSE

La figure IV-2 montre comment ces différentes phases s'articulent pour former la partie analyse de la figure IV-1. Nous notons que la phase ASSIA est utilisée indépendamment pour chaque procédure source.

IV.3.1.1. La phase ASSIA

Cette phase réalise l'analyse syntaxique et sémantique d'une procédure écrite en Fortran-77. Nous appelons pre-RI le résultat qu'elle produit. Cet objet, que nous ne décrivons pas, contient la plupart des renseignements contenus dans la RI de la même procédure. Seuls sont absents les renseignements uniquement calculables par une analyse interprocédurale:

- toutes les entités manipulables par la procédure ne sont pas décrites dans sa pre-RI (cf. §V.1)
- les références aux objets globaux (COMMONS, procédures etc...) sont faites par le nom de ces objets (chaines de caractères).
- la notion de procédure formelle est encore présente;
- etc ...

IV.3.1.2. La phase ASSIE

Cette phase réalise l'analyse syntaxique et sémantique d'un ensemble de procédures constituant un programme. Le traitement effectué est interprocédural et consiste à combiner les renseignements contenus dans les pre-RIs pour produire:

- une Représentation Interne par procédure du programme,

- une structure de données contenant la description des objets globaux du programme (COMMON, procédure, etc...) ainsi que le graphe des occurences d'appels.

Nous ne décrivons pas le fonctionnement de cette phase, qui fait essentiellement appel aux techniques classiques de compilation; seuls deux points particuliers sont développés:

(1) nous montrons au chapitre X comment éliminer la notion de procédure, formelle

(2) nous montrons au chapitre V comment rechercher l'ensemble des entités manipulables par une procédure, directement ou indirectement, c'est à dire par le biais des appels de procédure qu'elle contient.

Nous décrivons par contre au §IV.3.3 le contenu d'une représentation interne et de la structure de données décrivant les objets globaux. Cette description est indispensable pour la présentation des méthodes de calcul des triplets de fonctions qui est faite dans les chapitres V à IX.

IV.3.1.3. La phase ASI

C'est dans cette phase que sont réalisées effectivement les trois améliorations que nous proposons. Celles-ci ne sont pas calculables individuellement pour les raisons suivantes.

(1) L'aliasing associé à une occurence de procédure est calculable par une analyse descendante du graphe des occurence d'appels, uniquement à partir de l'ensemble des RIs et de la description des objets globaux. Cependant, l'algorithme que nous proposons au chapitre IX donne des résultats meilleurs s'il dispose d'une fonction d'assertion pour le programme.

(2) Les assertions associées aux différents noeuds d'une occurence de procédure sont calculées elles-aussi par une analyse descendante du graphe des occurences d'appels. La connaissance d'une fonction d'aliasing et d'une fonction de manipulation est indispensable pour le calcul des assertions crées et/ou détruites par l'exécution d'un opérateur (cf. chapitre VII).

(3) Le calcul d'une fonction de manipulation utilise une analyse ascendante du graphe des occurences d'appel. La méthode que nous proposons dans le chapitre V nécessite la connaissance d'une fonction d'assertions; ces assertions sont utilisées pour décrire les régions manipulées par chaque opérateur, mais aussi pour tenter d'évaluer certaines expressions et/ou d'en comparer d'autres. L'utilisation d'assertions (cf. chapitre VIII) nécessite d'autre part la connaissance de l'aliasing.

Les dépendances entre les différentes fonctions à calculer - aliasing, assertion et manipulation - sont donc cycliques, ce qui empêche le calcul direct de chaque fonction. nous montrons dans le chapitre X qu'une analyse itérative d'un programme complet conduit à une solution.

Afin de disposer du matériel nécessaire dans les chapitres V à IX, nous formulons dans le §IV.3.5 un ensemble de conventions sur ces fonctions qui seront vérifiées dans ces mêmes chapitres.

IV.3.2. Description des objets globaux

La phase interprocédurale ASSIE construit plusieurs ensembles contenant la description des objets interprocéduraux:

$P = \{P_1, P_2, \ldots\}$: ensemble des procédures du programme,

$G = \{g_1, g_2, \ldots\}$: ensemble des commons du programme, (G comme Global)

$RI = \{ri_1, ri_2, \ldots\}$: ensemble des représentations internes des procédures du programme,

$\Omega = \{\omega_1, \omega_2, \ldots\}$: ensemble des occurences des procédures du programme; cet ensemble nous permet de construire dans le chapitre X le graphe des occurences statiques d'appels du programme.

Ces ensembles sont liés par plusieurs fonctions; nous en décrivons une partie dans ce paragraphe, d'autres seront définies dans les chapitres suivants:

$OP\_P: \Omega \rightarrow P$: associe à une occurence de procédure la procédure dont elle est une occurence

$OP\_RI: \Omega \rightarrow RI$: associe à une occurence de procédure la représentation interne qui la décrit

$RI\_P: RI \rightarrow P$: associe à une représentation interne la procédure dont elle est une représentation. Nous n'imposons pas une bijection entre P et RI; ceci permet à plusieurs occurences de la même procédure d'avoir des représentations internes différentes; nous verrons pourquoi cela est intéressant au chapitre X.

Remarque IV-1: Notons que $OP\_P = OP\_RI \circ RI\_P$.

IV.3.3. Description d'une Représentation Interne

IV.3.3.1. Entités d'une RI

Les entités qui sont des paramètres formels sont nommées entités formelles; une entité qui n'est pas formelle sera dite réelle, à ne pas confondre avec une entité de type REAL (dont nous ne parlerons jamais). Une entité réelle est locale ou commune.

Soit $ri \in RI$, nous notons:

E(ri): l'ensemble des entités de ri,
$V_i(ri)$: l'ensemble des variables de ri de type INTEGER;
$nv(ri) = card(V_i(ri))$.

Les informations relatives à ces entités sont fournies par un ensemble de fonctions; nous en décrivons quelques-unes dans ce paragraphe:

type(ri): E(ri) → { INTEGER,REAL,COMPLEX,DOUBLE,CHARACTER,LOGICAL}
fonction fournissant le type d'une entité.

adr(ri): E(ri) → { LOCALE,COMMUNE,FORMELLE} x (G+{ ?}) x (N+(NxN))
fonction associant une adresse à une entité; les triplets d'adresses corrects sont de la forme:

(LOCALE,?,(bi,bs)): entité locale dynamique, (bi,bs) est le champ d'adresses;

(COMMUNE,g_i,(bi,bs)): entité commune g_i common origine, (bi,bs) champs d'adresses;

(FORMELLE,?,i): entité formelle, i numéro d'ordre dans la liste de paramètres.

Une adresse est donc un objet complexe qui a lui seul permet de résoudre les problèmes liés aux équivalences.

nbdim(ri): E(ri) → [0,7]

fonction associant à une entité son nombre de dimensions, 0 pour une variable (le nombre de dimensions d'un tableau est limité à 7).

Les bornes de déclarations des entités tableaux sont des expressions; pour les tableaux réels, ces expressions sont constantes et évaluables par la fonction "val".

Nous utiliserons les notations suivantes:

bi_j $j \in [1,7]$ borne inférieure de la dimension j.
bs_j $j \in [1,7]$ borne supérieure de la dimension j.

$\forall j \in [1,7]$, bi_j et $bs_j \in X(ri)$ (X(ri) est l'ensemble des expressions de ri).

IV.3.3.2. Membres d'une RI

Les membres d'une représentation interne ri sont, avec les opérateurs, les éléments entrant dans la composition des expressions. Nous notons $M(ri) = \{m_1, m_2, \ldots\}$ l'ensemble de ces membres. Un membre peut être l'un des trois éléments suivants.

(1) Une référence à une entité; ce type de membre sera appelé "lhs" dans la suite pour "left hand side": tout ce qui peut se trouver en partie gauche d'une affectation. Nous parlerons de lhs simple et de lhs tableau; les indices de ces derniers sont des expressions et seront généralement notés z_i, $i \in [1,7]$.

(2) Une référence à une constante.

(3) Un appel d'externe (fonction); les paramètres réels d'un appel d'externe sont des expressions.

Nous notons:

$L(ri) = \{l_1, l_2, \ldots\} \subset M(ri)$ l'ensemble des lhs de ri;

$C(ri) = \{c_1, c_2, \ldots\} \subset M(ri)$ l'ensemble des appels d'externe de ri (C comme CALL bien que C(ri) comprenne les appels de fonction).

hypothèse: Les différentes exécutions d'un appel de procédure ne font pas nécessairement référence à la même procédure, à cause de la notion de procédure formelle. Nous verrons au chapitre X qu'au niveau d'une occurence de procédure, un appel d'externe référence toujours la même procédure. Ceci nous amène à définir la fonction "appelée": soit $ri \in RI$, on note

$$OP\_RI^{-1}(ri) = \{\omega \in \Omega \text{ tq } OP\_RI(\omega) = ri\}$$

$OP\_RI^{-1}(ri)$ est l'ensemble des occurences de la même procédure ayant la même représentation interne. D'où:

$$appelee(ri): OP\_RI^{-1}(ri) \times C(ri) \rightarrow P.$$

IV.3.3.3. Expressions d'une RI

Nous notons $X(ri) = \{x_1, x_2 \ldots\}$ l'ensemble des expressions d'une RI. Nous dirons qu'une expression est simple si elle se compose uniquement d'un lhs: I, T(I+1,Z**F(K)), etc... et complexe: sinon. D'autre part, une expression sera dite constante si sa valeur peut être calculée à la compilation: 2, 3+2*5, etc...

IV.3.3.4. Instructions d'une RI

Nous notons $S(ri) = \{s_1, s_2, \ldots\}$ l'ensemble des instructions de ri (S pour statement). Un élément de S(ri) contient la description exacte d'une instruction de la procédure dont ri est issue. La plupart des méthodes que nous décrivons n'ont pas besoin de la structure d'une instruction: elle est cependant nécessaire, ne serait-ce que pour la régénération du source d'une procédure.

Nous utilisons cette structure dans le §V.4 pour la recherche de l'ensemble des noeuds accessibles d'une occurence de procédure. Nous y montrons quelques-unes des fonctionnalités sur les instructions qui sont nécessaires.

IV.3.3.5. Graphe de contrôle d'une RI

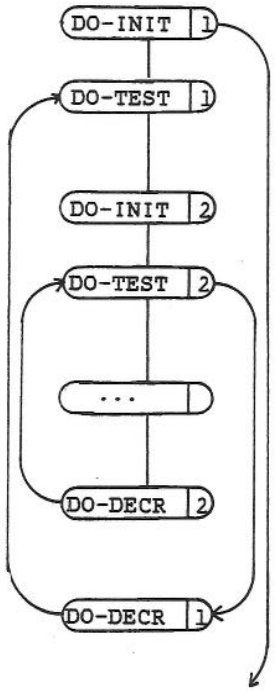
Le graphe de contrôle d'une RI ri est donné par un triplet $(N(ri), \Sigma(ri), rac(ri))$ où:

$N(ri) = \{n_1, n_2, \ldots\}$ est un ensemble de noeuds,
$\Sigma(ri) \subset N(ri) \times N(ri)$ est la relation successeur (ensemble d'arcs),
$rac(ri) \in N(ri)$ est le noeud initial.

La construction du graphe de contrôle est facilement réalisée pour Fortran-77. La plupart des instructions ne génère qu'un noeud (affectation, assignation, test, goto assigné, goto calculé, appel de subroutine, instructions d'entrées-sorties, etc...), certaines ne génèrent aucun noeud (ELSE, ENDIF, GOTO, etc...). Seule l'instruction DO donne naissance à plusieurs noeuds ainsi que le montre l'exemple suivant:

Exemple IV-4:

```
        DO 10 I = 1,N                1
              DO 10 J = 1,N          2
10                    ...
```



Le rôle de ces différents noeuds est de reproduire le mécanisme des boucles DO:

DO-INIT: évalue les expressions bornes; initialise l'indice de boucle (variable-DO); initialise un compteur d'itération;

DO-TEST: teste le compteur d'itération à zéro et finit ou non la boucle;

DO-DECR: décrémente de 1 le compteur d'itération; incrémente la variable-DO de la valeur calculée par DO-INIT.

Les noeuds et les arcs sont etiquettés par un ensemble de fonctions qui seront présentées dans la suite; ainsi l'instruction associée à un noeud est donnée par la fonction suivante: N_S(ri): N(ri) → S(ri).

Nous utilisons la notion d'opérateur définie par G. Roucairol dans [Rouc 78] qui ne doit pas être rapprochée de la notion matérielle d'opérateur. Ainsi, à deux instructions "X = X+1" différentes correspondent deux opérateurs différents.

Nous aurions pu utiliser le terme "tâche", sans doute moins équivoque, mais qui présente l'inconvénient d'être généralement associé à un traitement plus complexe qu'une instruction Fortran-77.

Nous supposons qu'à un noeud est associé un opérateur unique; dans la suite nous parlerons indifféremment de noeud ou d'opérateur.

IV.3.4. Représentation schématique des instructions

Les méthodes que nous allons décrire dans les chapitres suivants n'ont généralement pas besoin de connaître la structure des différentes instructions associées aux opérateurs de la RI. Il leur suffit de savoir associer à un opérateur les trois ensembles suivants.

LDM: Ensemble des lhs directement modifiés par son exécution. Par directement nous signifions que la modification de chaque lhs est due à la nature de l'instruction associée et non pas à un appel d'externe dont l'exécution est impliquée par celle de l'opérateur.

LDU: Ensemble des lhs directement utilisés par son exécution.

AE: Ensemble des appels d'externe dont l'exécution est impliquée par celle de l'opérateur.

Exemple IV-5: Soit F, G deux fonctions; l'instruction:

$$T(I+1) = F(K) + Z(I,J,K) + G(I)$$

donne naissance aux trois ensembles suivants:

LDM = {T(I+1)}, car les lhs que pourraient modifier F et G ne sont pas comptabilisés,

LDU = {I, K, J}; indices de tableau et/ou paramètres réels d'appel,

AE = {F, G}; appel à F et appel à G.

La construction de ces trois ensembles ne présente aucune difficulté. Elle se borne à une analyse détaillée de chaque instruction. La table IV-3 donne, pour chaque type d'instruction de Fortran-77, les lhs directement modifiés et les expressions directement utilisées par l'exécution de cette instruction (Attention à l'instruction DO pour laquelle ces ensembles sont répartis sur les 3 opérateurs conformément à la norme). L'exploration récursive de ces expressions fournit les ensembles (2) et (3).

Les remarques précédentes nous permettent de supposer l'existence des 3 fonctions suivantes:

N_mod(ri): $N(ri) \rightarrow L(ri)^*$ (iii)
fonction associant à un noeud l'ensemble LDM;

N_uti(ri): $N(ri) \rightarrow L(ri)^*$
fonction associant à un noeud l'ensemble LDU;

N_app(ri): $N(ri) \rightarrow C(ri)^*$
fonction associant à un noeud l'ensemble AE.

(iii) Dans la suite nous notons E^* l'ensemble P(E) des parties d'un ensemble E. Cette notation est généralement reservée aux listes d'éléments d'un ensemble E. Ce conflit de notation n'est pas trop grave dans la mesure où les ensembles sont souvent implémentés par des listes.

Instruction	Ens. lhs dir. modifiés	Ens. expr. dir. utilisées
affection	lhs en partie gauche	expr. en partie droite
ASSIGN	lhs situé après le symbole TO	ø

goto incond
ELSE
ENDIF
CONTINUE
STOP
PAUSE
END
	ø	ø

goto calculé	ø	expr. testée
goto assigné	ø	variable testée
if arithm	ø	expr. testée
if logique if bloc if else	ø	expr. testée
DO	variable-DO	expr. borne inf. expr. borne sup. expr. incr.
BACKSPACE REWIND ENDFILE	lhs IOSTAT (opt)	expr. UNIT (obl)
CLOSE	lhs IOSTAT (opt)	expr. UNIT (obl) expr. STATUS (opt)
OPEN	lhs IOSTAT (opt)	expr. UNIT (obl) expr. RECL (opt) expr. FILE (opt) expr. STATUS (opt) expr. ACCES (opt) expr. FORM (opt) expr. BLANK (opt)
INQUIRE	lhs IOSTAT (opt) lhs EXIST (opt) lhs OPENED (opt) lhs NUMBER (opt) lhs NAME (opt) lhs NAMED (opt) lhs ACCES (opt) lhs SEQUENTIAL (opt) lhs DIRECT (opt) lhs FORM (opt) lhs FORMATTED (opt) lhs UNFORMATTED (opt) lhs RECL (opt) lhs NEXTREC (opt) lhs BLANK (opt)	expr. UNIT expr. FILE
	lhs des listes à	expr. FORMAT (obl)

Instruction	Ens. lhs dir. modifiés	Ens. expr. dir. utilisées
PRINT	do-implicite (opt)	expr. liste de sortie expr. bornes do-impl
WRITE Si fic. int. Si fic. ext.	lhs IOSTAT (opt) lhs des listes à do-implicite (opt) lhs servant de fic. int.	expr. REC (opt) expr. FORMAT (obl) expr. liste de sortie expr. bornes do-impl expr. UNIT
READ Si fic. int. Si fic. ext.	lhs IOSTAT (opt) lhs des listes à do-implicite (opt) lhs liste d'entrées	expr. REC (opt) expr. FORMAT (obl) expr. bornes do-impl lhs servant de fic. int. expr. UNIT

Table IV-3

IV.3.5. Conventions sur la structure des triplets de fonctions

L'objet des chapitres suivants est de montrer comment calculer pour chaque occurence de procédure: une fonction d'aliasing, une fonction d'assertion et une fonction de manipulation. Le caractère cyclique de ces calculs, présenté au §IV.3.1.3 nous impose de décrire ces objets dès maintenant et de faire des hypothèses sur leurs utilisations.

IV.3.5.1. Structure d'une fonction d'aliasing

Soit ri ϵ RI une représentation interne; l'ensemble de tous les états d'aliasing possibles associé à ri est l'ensemble de fonctions suivant:

$$FA(ri) = E(ri) \times E(ri) \rightarrow \{ NON, PARTIEL, TOTAL \}$$

Soit fa ϵ FA(ri), e et e' ϵ E(ri); fa décrit l'état d'aliasing suivant:

fa(e,e') = NON => e et e' ne sont pas en aliasing.
fa(e,e') = PARTIEL => e et e' sont en aliasing partiel (cf. chap IX).
fa(e,e') = TOTAL => e et e' sont en aliasing total "

Une fonction d'aliasing est donc un élément de FA(ri); nous notons généralement fa un tel élément.

IV.3.5.2. Structure d'une fonction d'assertion

Nous proposons de calculer et d'utiliser des assertions linéaires de la forme suivante $\sum_i \mu_i \, v_i \leq \mu$ avec $\mu_i \in Z$, $\mu \in Z$ et v_i variable de type INTEGER.

Soit V un ensemble de variable, nous posons $\Lambda(V) = V \rightarrow Z$, $\Lambda(V)$ est l'ensemble des combinaisons linéaires sans terme constant des variables de V. Une assertion linéaire sur les variables de V est donc un élément de l'ensemble $I(V) = \Lambda(V) \times Z$, l'élément de Z donnant le terme constant.

Soit ri ϵ RI une représentation interne, nous simplifions les notations en posant:

$$\Lambda(ri) = \Lambda(V_i(ri)) \text{ et } I(ri) = I(V_i(ri)).$$

Soit n ϵ N(ri) un noeud quelconque, une fonction d'assertion est un objet permettant d'associer à n un ensemble d'assertions $i^* \epsilon I(ri)^*$. Ces assertions sont vraies avant l'exécution de l'opérateur associé à n. Une fonction d'assertion est donc un élément de l'ensemble suivant:

$$FI(ri) = N(Ri) \rightarrow I(ri)^*$$

Un tel élément sera généralement noté fi. Dans la suite, nous utiliserons dans les exemples différentes sortes d'assertions: "I==J, I+J≥4, ..." afin de les rendre plus clairs; il est clair que ces assertions se transforment aisément en des éléments de $I(ri)^*$.

IV.3.5.3. Structure d'une fonction de manipulation

La notion de région est introduite au chapitre suivant. Nous notons R(ri) l'ensemble des régions d'une occurence de procédure.

Soit $\omega \epsilon \Omega$ une occurence de procédure, ri = OP_RI(ω) sa RI associée et c ϵ C(ri) un appel d'externe quelconque.

Une fonction de manipulation est un objet permettant d'associer à c deux ensembles de régions r_m^* et r_u^* représentant respectivement les régions modifées et utilisées par l'exécution de cet appel d'externe. Une fonction de manipulation est donc un élément de l'ensemble suivant:

$$FM(ri) = C(ri) \rightarrow R(ri)^* \times R(ri)^*$$

un tel élément sera généralement noté fm.

IV.3.6. Hypothèses sur l'utilisation des assertions

Nous proposons au chapitre VIII différents moyens d'utiliser les assertions associées à un noeud pour:

(1) évaluer une expression entière ou booléenne,
(2) comparer une expression par rapport à 0 ou
(3) comparer deux expressions.

Ceci est réalisé par les fonctions suivantes:

$X\_ifovali(ri) : FA(ri) \times I(ri)^* \times X(ri) \rightarrow N \cup \{?\}$

$X\_ifovalb(ri) : FA(ri) \times I(ri)^* \times X(ri) \rightarrow BOOL \cup \{?\}$

$X\_ifocmp0(ri) : FA(ri) \times I(ri)^* \times X(ri) \rightarrow \{POS, NEG, NUL, ?\}$

$X\_ifoequ(ri) : FA(ri) \times I(ri)^* \times X(ri) \times X(ri) \rightarrow BOOL$

$X\_ifodif(ri) : FA(ri) \times I(ri)^* \times X(ri) \times X(ri) \rightarrow BOOL$

Les arguments de ces fonctions sont:

- une fonction d'aliasing associée à l'occurence de procédure en cours d'analyse (indispensable pour l'utilisation des assertions (cf. chap VIII),
- Les assertions associées au noeud en cours d'analyse,
- la ou les expressions à traiter.

C H A P I T R E V

V. Calcul des effets de l'exécution d'une occurence de procédure sur ses
- V.1. Univers des entités d'une procédure
  - V.1.1. Classe des entités locales dynamiques
  - V.1.2. Classe des entités locales statiques
  - V.1.3. Classes des entités communes
    - V.1.3.1. Rappel des propriétés
    - V.1.3.2. Etude de la prise en compte des entités communes
    - V.1.3.3. Conséquences sur la représentation interne associée à une occurence de procédure
  - V.1.4. Classe des entités formelles
  - V.1.5. Conclusion
- V.2. Définition d'une région
  - V.2.1. Cas des entités simples
  - V.2.2. Cas des entités tableaux
    - V.2.2.1. Contraintes sur les régions
    - V.2.2.2. Structure proposée pour une région
    - V.2.2.3. Variables de description des régions
    - V.2.2.4. Avantage de cette structure
    - V.2.2.5. Ensembles des régions d'une représentation interne
- V.3. Eléments nécessaires à la recherche des effets de l'exécution d'une occurence de procédure sur ses entités.
- V.4. Recherche de l'ensemble des noeuds accessibles
  - V.4.1. Solution proposée
  - V.4.2. Algorithme proposé
- V.5. Recherche des régions manipulées par l'exécution d'un noeud
  - V.5.1. Régions directement manipulées; traduction d'un lhs en région
  - V.5.2. régions indirectement manipulées
  - V.5.3. Utilisation du contexte local au noeud analysé
- V.6. Recherche des régions manipulées par l'exécution d'une occurence de procédure
  - V.6.1. Principe de l'élargissement d'une région
  - V.6.2. Insuffisance du type d'assertions calculées
  - V.6.3. Phase d'analyse sémantique supplémentaire
    - V.6.3.1. Principe
    - V.6.3.2. Recherche des variables entières formelles ou communes non modifiées
  - V.6.4. Rétrécissement d'une région; utilisation des bornes de déclaration d'un tableau
- V.7. Conclusion

V. Calcul des effets de l'exécution d'une occurence de procédure sur ses entités

Ce chapitre et le chapitre suivant sont complémentaires. Leur objectif est de montrer comment calculer les effets de l'exécution d'un appel d'externe sur les entités de l'occurence de procédure qui le contient. Ce calcul s'effectue en deux temps.

(1) La recherche des effets de l'exécution de l'occurence de procédure appelée sur ses propres variables. Cette recherche est l'objet du chapitre V; elle conduit à construire les deux ensembles de "parties d'entités", de la procédure appelée, qui sont modifiées et/ou utilisées par cette exécution. Une partie d'entité est décrite par un objet nommé "région".

(2) Le report des effets calculés en (1) sur l'appel d'externe analysé. Cette phase consiste à "traduire" chaque région précédemment calculée, en une région décrivant une partie d'entité de l'occurence de procédure appelante. Ce problème, nommé dans la suite "traduction d'une région" est l'objet du chapitre VI.

Nous nous intéressons donc dans ce chapitre à la recherche des régions lues et modifiées par l'exécution d'une occurence de procédure. Nous parlerons dans la suite de la recherche des régions manipulées.

Ce chapitre s'organise de la façon suivante:

(1) Nous montrons que toutes les classes d'entités de Fortran-77 ne doivent pas être prises en compte lorsque le but de cette recherche est de reporter les régions trouvées sur l'appel d'externe correspondant.

(2) Nous définissons la notion de région: objet nous permettant de décrire une partie d'entité.

(3) Nous énonçons les éléments nécessaires à cette recherche.

(4) Nous montrons comment calculer l'ensemble des noeuds accessibles de l'occurence de procédure; c'est-à-dire ceux qui sont suceptibles d'être exécutés une ou plusieurs fois. Nous montrons notamment que l'utilisation d'assertions permet d'affiner ce calcul.

(5) Nous montrons comment rechercher les régions manipulées par chaque opérateur, chaque région devant décrire une partie constante d'entité. Nous montrons comment utiliser le contexte local de l'opérateur pour affiner chaque région. Enfin nous montrons que le type d'assertions que nous calculons ne nous permet pas de décrire précisément le comportement de la procédure; ceci nous conduit à définir une phase d'analyse sémantique supplémentaire pour résoudre ce problème.

(6) Nous en déduisons simplement les ensembles de régions modifiées et/ou utilisées par une exécution de cette occurence de procédure.

V.1. Univers des entités d'une procédure

Une procédure manipule cinq classes d'entités différentes:

- les entités locales dynamiques,
- les entités locales statiques,
- les entités communes dynamiques,
- les entités communes statiques,
- les entités formelles.

Nous allons rappeler les propriétés de chacune de ces classes et en déduire s'il faut ou non en tenir compte lors de notre recherche.

V.1.1. Classe des entités locales dynamiques

Les entités locales d'une procédure Q qui ne sont pas l'objet d'une déclarative SAVE (cf. [AFNO 83] §8.9) sont dites dynamiques. Ces entités ne sont connues que de Q; elles ont une durée de vie égale à la durée d'une exécution de Q et ne conservent donc pas leur valeur entre deux exécutions de Q.

Seules les manipulations d'entités réelles qui restent vivantes à la fin de l'exécution d'une occurence de procédure peuvent créer des conflits entre les occurences des procédures appelantes.

Les manipulations d'entités locales dynamiques sont donc sans importance et doivent être ignorées. Cela signifie que ces entités doivent être dupliquées lorsque deux exécutions de la même occurence de procédure ont lieu simultanément.

V.1.2. Classe des entités locales statiques

Les entités locales d'une procédure Q qui sont l'objet d'une déclarative SAVE sont dites statiques. Ces entités ne sont connues que de Q; elles ont une durée de vie égale à la durée de l'exécution du programme et conservent donc leur valeur entre deux exécutions de Q.

Dans un but de simplification, nous éliminons le cas des entités locales statiques en remarquant que les entités d'un common statique déclaré dans une seule procédure ont les mêmes propriétés que les entités locales statiques de cette procédure. Nous pouvons donc supposer que la phase ASSIA élinime ces entités en les regroupant dans un nouveau common statique.

V.1.3. Classes des entités communes

V.1.3.1. Rappel des propriétés

A Classe des entités communes dynamiques

Une entité commune est dynamique si le common auquel elle appartient est dynamique. Un common est dynamique si, dans aucune procédure où il est déclaré, il est l'objet d'une instruction SAVE; d'autre part le common blanc n'est pas dynamique. Un common dynamique est vivant - ainsi que toutes ses entités - tant que s'exécute au moins une procédure où il est déclaré.

B Classe des entités communes statiques

Une entité commune est statique si le common auquel elle appartient est statique. Un common est statique s'il est l'objet d'une instruction SAVE dans au moins une procédure où il est déclaré; d'autre part le common blanc est statique.

Un common statique est vivant - ainsi que toutes ses entités - pendant toute la durée de l'exécution du programme.

V.1.3.2. Etude de la prise en compte des entités communes

A Entités communes statiques

Etant donné que ces entités restent vivantes pendant toute la durée du programme, les manipulations qui en sont faites par une occurence quelconque d'une procédure p peuvent créer des conflits entre les exécutions des occurences des procédures ascendantes de p, et doivent donc être reportées sur l'appel d'externe correspondant de l'occurence de procédure appelante.

B Entités communes dynamiques

Les entités d'un common dynamique CD manipulées par une occurence d'une procédure p restent vivantes à la fin de son exécution s'il existe une occurence d'une procédure o en cours d'exécution qui possède la déclaration de CD. Dans le cas contraire, les valeurs des entités de CD sont perdues; nous disons alors que l'occurence de la procédure p correspond à une instantiation du common CD.

V.1.3.3. Conséquences sur la représentation interne associée à une occurence de procédure

Pour que le report des manipulations d'entités faites indirectement par une occurence de procédure ω - indirectement signifiant par l'intermédiaire des appels d'externe qu'elle contient - puisse s'effectuer, il est nécessaire que ces entités soient connues dans la représentation interne ri associée à ω (OP_RI(ω)=ri). L'exemple V-1 montre le problème.

Exemple V-1:

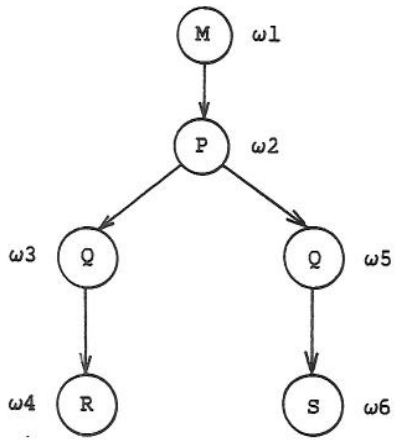
```
PROGRAM M
...
CALL P
...
END
```

```
SUBROUTINE P
COMMON /CS1/ ...
COMMON /CS2/ ...
SAVE CS1, CS2
...
CALL Q(R)
...
CALL Q(S)
...
END
```

```
SUBROUTINE Q(PROC)
COMMON /CD/ ...
...
CALL PROC
...
END
```

```
SUBROUTINE R
COMMON /CS1/ ...
COMMON /CD/  ...
...
END
```

```
SUBROUTINE S
COMMON /CS2/ ...
COMMON /CD/  ...
...
END
```



Graphe des occurences d'appels

La table suivante montre pour chaque occurence de procédure les commons qu'elle peut manipuler, directement (DM) ou indirectement (IM).

OP/P Common	ω1/M	ω2/P	ω3/Q	ω4/R	ω5/Q	ω6/S
CS1	IM	DM	IM	DM	/	/
CS2	IM	DM	/	/	IM	DM
CD	/	/	DM	DM	DM	DM

Remarque V-1:

- les deux occurences de la procédure Q manipulent des commons différents;
- les occurences de procédure ω1,ω3 et ω4 manipulent les commons CS1 et/ou CS2 alors que M et Q n'en possèdent pas la déclaration;
- les occurences de la procédure Q correspondent à des instantiations du common CD; les manipulations de CD qu'elles effectuent ne doivent pas être reportées sur les appels correspondants dans l'occurence de la procédure P.

C'est la phase ASSIE qui construit les représentations internes. Nous n'avons encore fait aucune supposition sur le nombre de RIs qu'elle associe à chaque procédure; rien empêche en effet à cette phase d'associer une RI par occurence de procédure.

Dans le cas simple où la phase ASSIE associe une RI à chaque procédure, celle-ci doit contenir la description de tous les commons manipulables par toutes les occurences de cette procédure. Dans le cas de l'exemple V-1, la RI associé à Q doit contenir la description des commons CS1, CS2 et CD.

L'ensemble GM(p) des commons manipulables par toutes les occurences d'une procédure p se déduit des ensembles suivants:

CDD(p): ensemble des commoms dynamiques déclarés dans le pre-RI de p,
CSD(p): ensemble des commons statiques déclarés dans le pre-RI de p,
DESC(p) et ASC(p)

définis comme suit.

Définitions

Soit desc la relation de descendance définie sur les éléments de P de la façon suivante: $\forall p_1 , p_2 \in P$

$$(p_1,p_2) \in desc \iff \exists\ ri_1 \in RI,\ c \in C(ri_1) \text{ et } \omega_1 \in \Omega \text{ tels que}$$
$$OP\_RI(\omega_1) = ri_1 \wedge$$
$$OP\_P(\omega_1) = p_1 \wedge$$
$$appelee(ri_1)(\omega_1,c) = P_2$$

ceci signifie qu'il existe une occurence de la procédure p_1 qui possède un appel à une occurence quelconque de p_2. Nous notons desc* la fermeture transitive de desc.

Nous en déduisons les ensembles des procédures descendantes et ascendantes d'une procédure $p \in P$:

$$DESC(p) = \{q \in P \text{ tq } (p,q) \in desc^*\}$$
$$ASC(p) = \{o \in P \text{ tq } (o,p) \in desc^*\}$$

Nous proposons donc d'insérer dans la représentation interne ri d'une procédure p, une description de chaque common de GM(p), qui n'y est pas déjà. GM(p) est donné par l'équation suivante:

$$GM(p) = I \cup II \cup III \cup IV$$

$$I: \quad CDD(p)$$
$$II: \quad CSD(p)$$
$$III: \quad \bigcup_{q \in DESC(p)} CSD(q)$$
$$IV: \left[\bigcup_{q \in DESC(p)} CDD(q)\right] \cap \left[\bigcup_{o \in ASC(p)} CDD(o)\right]$$

Les manipulations d'entités communes doivent donc être reportées sur l'appel d'externe correspondant sauf si l'occurence de procédure en cours d'analyse correspond à une instantiation de ce common. Ce qui se traduit par:

$$c \in CDD(p) \wedge \nexists\ o \in p \text{ tq } o \in ASC(p) \wedge c \notin CDD(o)$$

V.1.4. Classe des entités formelles

Une entité est formelle si elle apparaît dans la liste des paramètres formels de la procédure analysée. La notion de durée de vie n'a pas de sens pour ces entités.

Les associations de paramètres réels et formels sont bien évidemment un moyen, pour la procédure appelée, d'utiliser et/ou de modifier les entités de la procédure appelante. Les manipulations d'une procédure sur ses entités formelles peuvent créer des conflits; il est donc nécessaire de recenser ces manipulations et de les reporter sur l'appel d'externe correspondant de la procédure appelante.

V.1.5. Conclusion

Nous venons de voir que toutes les manipulations d'entités exercées par l'exécution d'une occurence de procédure ne doivent pas être recensées quand il s'agit de rechercher les conflits provoqués par ces manipulations dans les occurences de procédure appelante. Nous n'évoquerons plus ce problème dans la suite du chapitre, le supposant résolu par une structure de données ad-hoc, fournie par la phase ASSIE, indiquant quelles entités doivent être prises en compte.

Nous avons vu d'autre part qu'une occurence de procédure peut manipuler indirectement des entités qui ne sont pas déclarées dans la pre-RI de la procédure dont elle est une occurence. Ceci implique que les déclarations de ces entités doivent être intégrées dans la RI correspondante. Ceci peut être fait systématiquement par la phase ASSIE, ou bien plus simplement au moment du report des manipulations de ces entités (cf. chapitre VI).

V.2. Définition d'une région

Une région est un objet qui permet de décrire une partie d'entité. Il est intéressant de savoir décrire une partie d'entité car cela permet de ne pas déclarer dépendants deux opérateurs manipulant des parties différentes d'une même entité.

V.2.1. Cas des entités simples

Tout accès à une variable modifie ou utilise totalement cette variable. La notion de partie de variable n'a donc pas de sens. Une région d'entité simple décrira donc toujours la totalité de cette entité.

V.2.2. Cas des entités tableaux

Il s'agit de savoir décrire des parties de tableau de façon suffisamment fine. La solution la plus fine consiste à énumérer la liste des éléments du tableau qui forment la partie à décrire. Cette solution est peu envisageable dans le cas de tableaux de grandes dimensions, elle est de toutes façons inutilisable dans le cas de tableaux ajustables comme le montre l'exemple V-2:

Exemple V-2:

```
      DIM T(10000)                    DIM S(N)
      DO 10 I = 1,N                   DO 10 I = 1,N
10          T(I) = 0. (a)         10          S(I) = 0. (b)
```

Sans connaissance de la valeur de N, nous pouvons donner une approximation des effets de (a) sur T en listant les 10000 éléments du tableau T. Cette solution n'est pas envisageable pour les effets de (b) sur S.

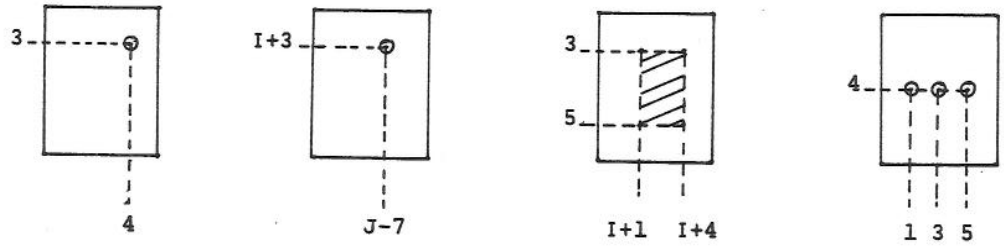
V.2.2.1. Contraintes sur les régions

Nous voulons cependant avoir la possibilité de décrire les parties de tableaux usuellement manipulées telles que:

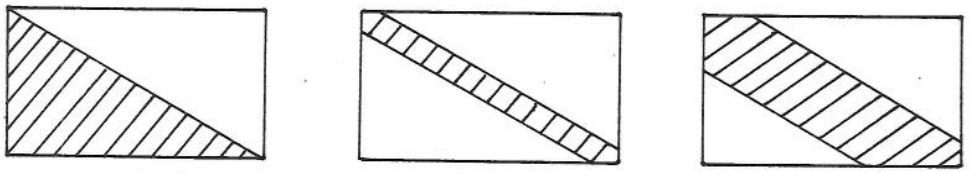
(a) élément de tableau: T(3,4), T(I+3,J-7), ...;

(b) ensemble régulier d'élements de tableau: T([3,5,1],[I+1,I+4,1]), T(4,[1,6,2]), ... [,,] désignant la progression arithmétique;

(c) parties plus générales: sous-matrice triangulaire inférieure, diagonale principale d'une matrice, ...



(a) (a) (b) (b)



sous-matrice triangulaire inférieure

diagonale principale

bande autour de la diagonale principale

(c) (c) (c)

Il ne suffit pas de savoir décrire des parties de tableaux complexes. Il faut aussi être capable d'effectuer certaines opérations sur ces parties de tableau, telle que la recherche d'un conflit entre deux parties d'un même tableau. De même, il faut savoir utiliser une éventuelle connaissance d'assertions sur les variables pour affiner ces parties de tableau.

V.2.2.2. Structure proposée pour une région

Soit ri ϵ RI une représentation interne et e ϵ E(ri) une entité de dimension d=nbdim(ri)(e). Nous proposons de décrire une partie de e grâce à un ensemble d'assertions linéaires fournissant le domaine des valeurs que peuvent prendre les indices des différentes dimensions.

Afin de matérialiser ces indices, nous introduisons dans E(ri) 7 nouvelles entités simples de type INTEGER notées $(\varphi_i)_{i=1..7}$, φ_i représentant l'indice de la ième dimension d'une région. Une région de l'entité e est donc un doublet $r=(e,i^*)$ avec $i^* \in I(ri)^*$. L'exemple V-3 montre les régions décrites dans l'exemple V-2:

Exemple V-3:

(a) $(T,\{\rho_1=3,\ \rho_2=4\})$
(a) $(T,\{\rho_1-I=3,\ \rho_2-J=-7\})$
(b) $(T,\{-\rho_1\leq-3,\ \rho_1\leq5,\ I-\rho_2\leq-1,\ \rho_2-I\leq4\})$
(b) $(T,\{\rho_1=4,\ -\rho_2\leq-1,\ \rho_2\leq6,\ \rho_2-2*k=1\})$
(c) $(T,\{\rho_2\leq\rho_1-1\})$
(c) $(T,\{\rho_1=\rho_2\})$
(c) $(T,\{\rho_1-k\leq\rho_2,\ \rho_2\leq\rho_1+k\})$

Une région de variable est de la forme (e,Ø).

V.2.2.3. Variables de description des régions

Les variables $(\rho_i)_{i=1..7}$ sont appelées variables de description des régions. Toutes les régions sont décrites avec les mêmes variables ρ_i. Ces variables ont un champ d'adresses tel qu'elles ne sont en équivalence avec aucune autre variable. Nous disposons des deux fonctions suivantes:

is_vdr(ri): E(ri) → BOOL
fonction indiquant si une entité est ou non une des variables de description de région,

vdr(ri): [1,7] → E(ri)
fonction permettant d'accéder aux différentes variables de description de région.

L'unicité des variables de description des régions pose quelques problèmes lors d'opération sur les régions mettant en oeuvre deux ou plusieurs régions (comparaison, intersection, etc...). Il est alors nécessaire de les renommer.

V.2.2.4. Avantage de cette structure

Notre notion de région présente de nombreux avantages. Nous les exposons dans la suite de ce chapitre puis dans les chapitres VI et VIII. Nous pouvons d'ores et déjà citer:

- la finesse de description autorisée par les régions,
- l'utilisation facile du contexte associé à un noeud pour l'affinement d'une région,
- la recherche des dépendances par l'existence d'une solution pour un système d'assertions.

V.2.2.5. Ensembles des régions d'une représentation interne

Nous notons R(ri) cet ensemble, $R(ri) = E(ri) \times I(ri)^*$.

V.3. Eléments nécessaires à la recherche des effets de l'exécution d'une occurence de procédure sur ses entités.

Soit $\omega \in \Omega$ l'occurence de procédure dont nous recherchons les effets et $ri \in RI$ la représentation interne qui la décrit: ri=OP_RI(ω). Nous supposons que nous disposons des trois éléments suivants.

(1) Une fonction d'aliasing $fa \in FA(ri) = E(ri) \times E(ri) \rightarrow$ {NON,PARTIEL,TOTAL}, qui représente l'état d'aliasing associé à ω.

(2) Une fonction d'assertion fi ϵ FI(ri) = N(ri) $\rightarrow$ I(ri)* qui associe à chaque noeud du graphe de contrôle de ri un ensemble d'assertions, vraies avant l'exécution de ce noeud.

(3) Une fonction de manipulation fm ϵ FM(ri) = C(ri) $\rightarrow$ R(ri)* x R(ri)* qui associe à chaque appel d'externe de ri l'ensemble des régions que son exécution modifie et/ou utilise. Ces régions sont des éléments de R(ri), ce qui signifie qu'elles décrivent des parties d'entités de la procédure associée à ri et non pas des parties d'entités de la procédure appelée par cet appel.

Cette hypothèse sera justifiée dans le chapitre X où nous montrons comment organiser les traitements pour un programme complet.

Remarque V-2: Nous allons voir dans la suite de ce chapitre que l'analyse que nous effectuons ne dépend pas de ω mais du quadruplet (ri,fa,fi,fm). Ceci signifie que les résultats sont valables pour toutes les occurences de procédure ω' de même représentation interne, vérifiant les mêmes conditions pour l'aliasing, les assertions et les manipulations.

V.4. Recherche de l'ensemble des noeuds accessibles

V.4.1. Solution proposée

L'exécution d'une procédure implique au maximum l'exécution de tous les noeuds de son graphe de contrôle. Nous pouvons effectuer un premier affinement en calculant l'ensemble des descendants du sommet du graphe: rac (ri), ce qui permet de détecter les noeuds inacessibles et de les éliminer.

Nous proposons un deuxième affinement utilisant la fonction d'assertion fi. Il n'est pas rare qu'un programmeur regroupe plusieurs traitements différents dans une même procédure, parce qu'ils utilisent une structure de données commune ou font appel à un traitement commun. L'exemple V-4 montre une telle procédure:

Exemple V-4: Soient I1, I2, ..., IN un ensemble de variables à partir desquelles un indice I est calculé par une formule F complexe. La procédure MTB range une valeur VAL dans TAB(I) si ISENS=1 ou définit VAL avec TAB(I) si ISENS=2:

```
      SUBROUTINE MTB(TAB,VAL,ISENS,I1,I2,...,IN)
      INTEGER ISENS, I1, I2, ..., IN, I
      REAL VAL, TAB(*)
      I = F(I1,I2,...,IN)                          M1
      GOTO (10,20) ISENS                           M2
      STOP 'VALEUR DE ISENS INCORRECTE'            M3
10    TAB(I) = VAL                                 M4
      GOTO 30                                      M5
20    VAL = TAB(I)                                 M6
30    RETURN                                       M7
      END
```

Soient les deux appels à MTB suivants (que nous supposons contenus dans un programme principal):

```
      CALL MTB(T,V,1,...)   (a)
      CALL MTB(T,V,2,...)   (b)
```

L'adaptation des méthodes de calcul d'assertions que nous proposons au chapitre VII permet de propager des assertions entre les occurences de procédure. Ainsi, ces méthodes permettent d'associer à chaque noeud de l'occurence de la procédure MTB de l'exemple V-4 correspondant à l'apppel (a) (resp (b)) l'assertion {ISENS=1} (resp {ISENS=2}).

Si l'algorithme de recherche des noeuds accessibles utilise ces assertions il lui est alors possible d'éliminer certains noeuds, inaccessibles pour la ou les occurences de procédure concernées. Ainsi les noeuds accessibles de l'occurence de la procédure MTB correspondant à l'appel (a) (resp. (b)) sont {M1,M2,M4,M5,M7} (resp. {M1,M2,M6,M7}).

V.4.2. Algorithme proposé

L'algorithme classique de recherche des noeuds accessibles d'un graphe utilise un ensemble de noeuds W;

1) marquer chaque noeud de N(ri) "nonvisité";
2) initialiser W avec rac(ri);
3) si W=Ø aller en 8;
4) extraire n de W;
5) si n est marqué "visité" aller en 3;
6) marquer n "visité";
7) W = W U {n' ϵ N(ri) tq (n,n') ϵ Σ(ri)};
8) tous les noeuds marqués "visité" sont accessibles.

Nous proposons d'utiliser cet algorithme en affinant le traitement effectué en 7. Il nous faut pour cela analyser l'instruction associée au noeud. La plupart des instructions de Fortran-77 ont plusieurs successeurs:

IF logique, IF et ELSEIF bloc, IF arithmétique, GOTO calculé, GOTO assigné, instructions d'entrée-sorties (clause erreur (ERR) et fin de fichier (END)), DO, CALL (retours secondaires).

Nous éliminons les instructions de test suivantes:

GOTO assigné: car nous ne calculons pas d'assertion sur les variables contenant des valeurs d'étiquette d'instruction;

CALL: car l'expression testée n'est pas disponible, elle se trouve dans la représentation interne de l'occurence de procédure appelée;

Entrées-sorties: car l'expression testée n'est pas disponible.

Pour les autres instructions, la structure de données d'une RI doit nous permettre de connaitre:

- l'expression à tester (soit x),
- le type du test à effectuer: test booléen (IF logique et bloc), test numérique (GOTO calculé) ou test à zéro (IF arithmétique et boucle DO).

Ces renseignements sont associés à chaque noeud qui est un noeud de test.

Cette structure de données doit aussi nous permettre de savoir associer à un noeud de test le successeur privilégié par chaque valeur de x. Ceci est réalisé par une fonction d'étiquettage des arcs de Σ(ri):

typearc: $\Sigma(ri) \rightarrow \{ VRAI, FAUX, POS, NUL, NEG, NUMERIQUE\} \times (NU\{ ?\})$

les doublets corrects étant de la forme (VRAI,?), (FAUX,?), ..., (NEG,?) et (NUMERIQUE,i) pour le goto calculé.

Avec ces divers renseignements, nous sommes en mesure de modifier l'algorithme. Rappelons que nous avons introduit au §IV.3.6 les trois fonctions d'évaluation suivantes:

X_ifovali(ri) : $FA(ri) \times I(ri)^* \times X(ri) \rightarrow N \cup \{?\}$
fonction a utiliser pour un test numérique,

X_ifovalb(ri) : $FA(ri) \times I(ri)^* \times X(ri) \rightarrow BOOL \cup \{?\}$
fonction à utiliser pour un test booléen

X_ifocmp0(ri) : $FA(ri) \times I(ri)^* \times X(ri) \rightarrow \{POS, NEG, NUL, ?\}$
fonction à utiliser pour un test à zéro

La phase 7 de l'algorithme précédent devient:

7.1) si n n'est pas un noeud de test aller en 7.8;
7.2) soit x l'expression à tester;
7.3) évaluer x avec la fonction X_ifovaln, X_ifovalb ou X_ifocmp0 suivant le type du test, les arguments étant fa et fi(n);
7.4) si la valeur trouvée en 7.3 est "?" aller en 7.8;
7.5) soit n' tq $(n,n') \in \Sigma(ri)$, désigné par la valeur trouvée en 7.3 et typarc(n,n');
7.6) $W = W \cup \{n'\}$;
7.7) aller en 8;
7.8) $W = W \cup \{n' \in N(ri) \text{ tq } (n'n') \in \Sigma(ri)\}$;

Dans la suite, l'ensemble des noeuds accessibles sera notée NA.

V.5. Recherche des régions manipulées par l'exécution d'un noeud

L'exécution d'un noeud entraine la modification et l'utilisation d'un ensemble de régions. Certaines de ces régions sont directement manipulées - celles qui correspondent aux lhs modifiées et utilisés par ce noeud - d'autres sont indirectement manipulées - celles qui correspondent aux effets des appels d'externe dont l'exécution est impliquée par celle du noeud -.

Ces régions sont manipulées alors que les variables entières de l'occurence de procédure analysée vérifient un ensemble d'assertions qui sont données par la fonction d'assertion $fi \in FI(ri)$, que nous avons supposée connue au §V.3. Nous montrons comment profiter de ce "contexte local".

V.5.1. Régions directement manipulées; traduction d'un lhs en région

Nous rappelons que l'analyse des diverses instructions de I(ri) nous permet de construire les deux fonctions suivantes (cf. §IV.3.4), associant à un noeud l'ensemble des lhs qu'il manipule:

$$N\_mod(ri): N(ri) \rightarrow L(ri)^*$$
$$N\_uti(ri): N(ri) \rightarrow L(ri)^*$$

Le passage d'un lhs à une région est extrêment simple. Un lhs est soit une variable soit un élément de tableau. La structure d'un lhs, que nous ne détaillons pas, permet de lui associer:

l'entité qu'il décrit: e

les expressions d'indices $z_i \in X(ri)$, $i \in [1,nbdim(ri)(e)]$

Cas des lhs variable

La région associée est $(e,\emptyset)$.

Cas des lhs élément de tableau

A chaque expression z_i, $i \in [1,nbdim(ri)(e)]$ qui est une combinaison linéaire des variables de $V_i(ri)$ nous associons les assertions "$\rho_i - z_i \leq 0$" et "$z_i - \rho_i \leq 0$"; les expressions d'indice d'une autre forme sont perdues.

Pour mener à bien cette traduction, nous avons besoin de quelques fonctions de manipulation des expressions dont la réalisation, qui ne pose aucun problème, est dépendante de la représentation choisie pour les expressions:

$$isxcl(ri): X(ri) \rightarrow BOOL$$

fonction indiquant si une expression est une combinaison linéaire des variables de $V_i(ri)$ plus un éventuel terme constant.

Soit XCL(ri) l'ensemble des "expressions combinaisons linéaires":

$$XCL(ri) = \{x \in X(ri) \text{ tq } isxcl(ri)(x) = VRAI\}$$

Nous avons besoin de savoir passer d'une telle expression à un élément de $\Lambda(ri)$, sans oublier le terme constant. C'est une simple conversion de représentation:

$$Xtocl(ri): XCL(ri) \rightarrow \Lambda(ri)$$
$$Xtotc(ri): XCL(ri) \rightarrow Z$$

Nous avons d'autre part besoin de quelques fonctions de manipulation et de construction sur les combinaisons linéaires de variables:

$Vtocl(ri): V_i(ri) \rightarrow \Lambda(ri)$

$Vtocl(ri)(v)(v') =$ si $v=v'$ alors 1 sinon 0 $\forall v' \in V_i(ri)$

Vtocl est une fonction qui construit une combinaison linéaire de une variable;

$somcl(ri): \Lambda(ri) \times \Lambda(ri) \rightarrow \Lambda(ri)$

$somcl(ri)(l_1,l_2)(v) = l_1(v)+l_2(v) \quad \forall v \in V_i(ri)$

somcl fait la somme de deux combinaisons linéaires;

negcl(ri): $\Lambda(ri) \rightarrow \Lambda(ri)$

$negcl(ri)(l)(v) = -l(v)\ \forall v \in V_i(ri)$

negcl renvoie l'opposée d'une combinaison linéaire.

La région associée à un lhs tableau est donc (e,i^*) avec:

$$i^* = \bigcup_{k=1}^{ndbim(e)} (\text{si } isxcl(z_k) \text{ alors } \{(somcl(Xtocl(z_k),negcl(Vtocl(\rho_k))),-Xtotc(z_k)),\ (somcl(negcl(Xtocl(z_k)),Vtocl(\rho_k)),Xtotc(z_k))\} \text{ sinon } \emptyset \quad (i)$$

Exemple V-5: Voici quelques lhs et leur région associée:

$V \rightarrow (V,\emptyset)$

$T(I+2j+1) \rightarrow (T,\{\rho_1-I-2J \leq 1,\ I+2J-\rho_1 \leq -1\})$

$T(I+2,I*K) \rightarrow (T,\{\rho_1-I \leq 2,\ I-\rho_1 \leq -2\})$

I*K est non linéaire, donc toute information sur ρ_2 est perdue.

Nous obtenons donc la liste des régions directement modifiées (resp. utilisées) par un noeud n en traduisant chaque lhs de N_mod(ri)(n) (resp. N_uti(ri)(n)) en une région, par la méthode que nous venons d'indiquer.

V.5.2. régions indirectement manipulées

Nous avons supposé la connaissance d'une fonction de manipulation fm pour le ou les occurences de procédure analysées:

$$fm: C(ri) \rightarrow R(ri)^* \times R(ri)^*$$

cette fonction associe à chaque appel d'externe de ri les régions que son exécution modifie (1er champ) et utilise (2ème champ).

Nous rappelons d'autre part que l'analyse des diverses instructions de I(ri) nous permet de construire la fonction suivante (cf. §IV.3.4), associant à chaque noeud l'ensemble des appels d'externe qu'il contient:

$$N\_app(ri): N(ri) \rightarrow C(ri)$$

De ces deux fonctions, nous déduisons les ensembles de régions indirectement manipulées par un noeud par simple union.

V.5.3. Utilisation du contexte local au noeud analysé

Soit r une des régions manipulées par l'exécution d'un noeud n, directement ou indirectement. La fonction d'assertion $fi \in FI(ri)$, que nous supposons connue, associe à n un ensemble d'assertions fi(n) qui constitue le contexte d'exécution de n. Ceci signifie que les valeurs des variables de $V_i(ri)$ respectent les assertions de fi(n) au moment de

(i) toutes les fonctions de cette formule sont dépendantes de ri.

l'exécution de n.

Nous proposons de tenir compte de ces assertions dans la description de la région $r=(e,i^*)$ en construisant $r_1 = (e,i^* \cup fi(n)) = (e,i_1^*)$. à noter que ceci est inutile dans le cas d'une région décrivant une variable.

Exemple V-6:

```
      DO 10 I = 1,N
            DO 10 J = I+1,N
10                T(I,J) = ...     (a)
```

Nous obtenons (cf. chapitre VII):

$$fi(a) = \{1 \leq I \leq N,\ I+1 \leq J \leq N\}$$

la région $r=(T,\{\rho_1=I,\rho_2=J\})$ devient donc:

$$r_1 = (T,\{\rho_1=I,\ \rho_2=J,\ 1\leq I\leq N,\ I+1\leq J\leq N\})$$

En conclusion, nous sommes donc en mesure d'associer à chaque noeud $n \in NA$, les deux ensembles RM(n) et RU(n) des régions modifiées et utilisées par l'exécution de ce noeud.

V.6. Recherche des régions manipulées par l'exécution d'une occurence de procédure

Nous allons voir qu'il est indispensable que les effets de l'exécution d'une occurence de procédure soient exprimés avec des régions décrivant des parties constantes d'entité. Le passage d'une région quelconque à une région "constante" conduit dans notre cadre, a de mauvais résultats, à cause de l'insuffisance du type d'assertions calculées.

Nous proposons donc d'améliorer ces résultats grâce à une phase d'analyse sémantique supplémentaire, consistant à rechercher l'ensemble des variables non modifiées par l'occurence de procédure. Nous proposons d'autre part d'améliorer ces résultats en considérant que le programme que nous analysons est correct et que les accès aux tableaux sont faits en respectant les bornes de déclaration.

Note: tout ce que nous allons dire dans les 2 paragraphes suivants ne concerne que les régions décrivant des parties de tableau.

V.6.1. Principe de l'élargissement d'une région

Les régions associées à un noeud n, calculées au §V.5, sont décrites à l'aide de variables entières. Soit $r = (e,i^*)$ une telle région; la partie de l'entité e, réellement décrite par r, n'est connue que s'il est possible d'associer à chaque variable, utilisée dans les assertions i^*, l'ensemble des valeurs qu'elle prend pour toutes les exécutions de n pendant l'exécution de l'occurence de la procédure analysée.

Le principe de l'élargissement est donc fort simple: il suffit d'éliminer toutes les variables des assertions de i^*, après y avoir ajouté la description de l'ensemble des valeurs prises par chaque variable.

Nous ne connaissons pas de façon précise ces ensembles de valeurs; nous en possédons cependant une approximation: les assertions sur les variables données par la

fonction d'assertion fi au noeud n. Ces assertions sont déjà inclues dans l'ensemble i^* (cf. §V.5.3). L'élargissement de la région r se résume donc à l'élimination de toutes les variables dans les assertions i^* (ii).

L'élimination d'une variable dans un système d'assertions linéaires, sans perte d'information sur les autres variables, fait appel à la théorie des polyèdres convexes, que nous présentons dans le chapitre VIII.

Exemple V-7: voici quelques exemples d'élargissement:

$(V,\emptyset)$	$\rightarrow$	$(V,\emptyset)$
$(T,\{\rho_1=0\})$	$\rightarrow$	$(T,\{\rho_1=0\})$
$(T,\{\rho_1=I, I=5\})$	$\rightarrow$	$(T,\{\rho_1=5\})$
$(T,\{\rho_1=I, 1 \leq I \leq N\})$	$\rightarrow$	$(T,\{1 \leq \rho_1\})$
$(S,\{\rho_1=J,\rho_2=K, I \leq J \leq K \leq L\})$	$\rightarrow$	$(S,\{\rho_1 \leq \rho_2\})$

V.6.2. Insuffisance du type d'assertions calculées

L'application de ce principe de base conduit hélas à de bien mauvais résultats. Nous n'avons en effet pas introduit la notion de constante symbolique dans nos assertions. Nous n'avons donc pas la possibilité de représenter la valeur initiale d'une variable entière commune lorsque la valeur numérique de la variable entière commune correspondante dans l'occurence de procédure appelante est inconnue. Le problème est identique pour les variables entières formelles associées à des paramètres réels de valeur numérique inconnue.

Exemple V-8:

```
        DO 10 I = 1,N                  SUBROUTINE P(IT,IX,IV)
10              CALL P(ITAB,I,3)       DIM IT(*)
                                       IT(IX) = IV            (a)
                                       END
```

Sans notion de constante symbolique, le contexte local à (a) est: {IV=3}. La région modifiée par (a) est $(IT,\{\rho_1=IX, IV=3\})$, elle est élargie en $(IT,\emptyset)$.

Avec notion de constante symbolique, le contexte local à (a) serait $\{IV=3, IX=IX_o\}$. La région modifiée par (a) serait toujours $(IT,\{\rho_1=IX, IV=3, IX=IX_o\})$, elle serait élargie en $(IT,\{\rho_1=IX_o\})$.

Nous présentons plusieurs méthodes de calcul d'assertions au chapitre VII. Il n'est pas simple à notre avis, de les adapter pour qu'elles calculent des assertions comportant des constantes symboliques, notion qui devrait d'ailleurs être définie. C'est de toutes façons hors du cadre de cette thèse.

V.6.3. Phase d'analyse sémantique supplémentaire

(ii) à l'exception, bien évidemment, des variables de description de région.

V.6.3.1. Principe

Nous proposons d'élargir une région $r=(e,i^*)$ en ne conservant dans les assertions de i^* que les variables de description de région et les variables communes ou formelles qui ne sont modifées par aucun des noeuds de NA.

Cette modification revient à rechercher les variables v initialement définies (donc communes ou formelles) pour lesquelles une recherche d'assertions avec notion de constante symbolique aurait associé à chaque noeud n de N(ri) l'assertion $\{v=v_o\}$. Cette modification ne doit donc être vue que comme un palliatif à l'insuffisance des assertions effectivement calculées.

V.6.3.2. Recherche des variables entières formelles ou communes non modifiées

Nous connaissons l'ensemble NA des noeuds accesibles. Nous connaissons d'autre part, l'ensemble RM(n) des régions modifiées par chaque noeud n de NA (cf. §V.5). Nous examinons donc pour chaque variable $v \in V_i(ri)$ telle que v est une entité formelle ou commune, s'il existe $n \in NA$ et $r \in RM(n)$ tels que r et v sont en conflit; en conflit signifiant que la modification de r entraîne la modification de v (et vice versa).

Un conflit entre une région $r=(e,i^*)$ et une variable v a lieu quand:

- e = v,
- e et v sont en aliasing: $fa(e;v) \neq NON$,
- e et v sont en équivalence.

Il est possible d'éliminer certaines collisions dues aux équivalences. L'étude du conflit entre une région et une variable est l'objet du §VII.1.2.2.2. Ceci nous permet de calculer l'ensemble VNM des variables non modifiées par l'exécution de la ou des occurences de procédure analysées.

L'élargissement est alors réalisé en n'éliminant du système d'assertions que les variables qui ne sont pas des variables de description de région ou des variables de l'ensemble VNM.

V.6.4. Rétrécissement d'une région; utilisation des bornes de déclaration d'un tableau

Notation préliminaire

soit d=nbdim(ri)(e): nombre de dimensions d'une entité;

soit $(bi_k)_{k=1..d}$ les expressions bornes inférieures de la déclaration de e;

soit $(bs_k)_{k=1..d}$ les expressions bornes supérieures de la déclaration de e;

$\forall$ k=1..d $bi_k \in X(ri)$ et $bs_k \in X(ri)$.

Il est possible d'affiner la partie de tableau décrite par une région en faisant l'hypothèse que le programme ne fait aucun débordement de tableau. Dans ces conditions nous pouvons générer les assertions $bi_k \leq \rho_k \leq bs_k$ $\forall k \in [1,d]$, pour matérialiser cette hypothèse.

Nous justifions cette attitude grâce au §5.4.2 de [AFNO 83] qui interdit les débordements de tableau, ce qui implique qu'un programme qui en effectue est incorrect. Si la parallélisation que nous réalisons est incorrecte pour un programme effectuant des

débordements de tableau, cela revient à paralléliser incorrectement un programme incorrect, ce qui n'est pas trop grave.

Avant de montrer comment générer ces assertions, rappelons qu'un tableau peut être:

- constant: ses bornes de déclaration sont des expressions constantes,
- de longueur non spécifiée: la borne supérieure de la dernière dimension est une étoile,
- ajustable: ses bornes de déclaration contiennent des références à des variables entières communes ou formelles; le §5.5.1 de [AFNOR 83] stipule alors que la valeur utilisée pour dimensionner le tableau est la valeur initiale de la variable.

La génération de l'assertion $\rho_k \leq bs_k$ est possible dans trois cas.

(1) L'expression borne bs_k est une expression constante (cf. §IV.3.3.1). Ceci est obligatoire si e est une entité réelle. C'est de plus très souvent le cas des expressions bornes inférieures des tableaux formels (valeur 1). Soit vbs_k la valeur de bs_k, l'assertion à générer est: $(Vtocl(ri)(\rho_k), vbs_k)$.

(2) L'expression borne bs_k est une expression évaluable avec le contexte associé à la racine du graphe de contrôle rac(ri). En effet, le §5.5.1 de [AFNO 83] stipule que les tableaux ajustables sont dimensionnés en évaluant les expressions bornes avant toute exécution d'une instruction de la procédure. Soit donc vbs_k=X_ifoval(fa,fi(rac(ri)),bs_k), si vbs_k est différent de "?", nous sommes ramenés au cas (1).

(3) L'expression borne bs_k est une combinaison linéaire de variables entières formelles ou communes. Cette condition se traduit par:

$$isxcl(ri)(bs_k) = VRAI \text{ et}$$

$$\forall v \in Vi(ri),\ Xtocl(ri)(bs_k)(v) \neq 0 \Rightarrow$$
$$v \in \{\rho_1, \ldots, \rho_k\}V \text{ ou } v \in VNM$$

Nous pouvons dans ce cas générer l'assertion:

$$(somcl(Vtocl(phi_k), negcl(Xtocl(bs_k))), Xtotc(bs_k)) \quad (ii)$$

La génération de l'assertion $-\rho_k \leq -bi_k$ est réalisable dans les trois mêmes conditions. Les assertions ainsi générées sont ajoutées aux assertions de la région r.

Exemple V-9: Soit Q la subroutine suivante:

```
      SUBROUTINE Q(R,T,L,M,N,K)
      REAL R, T(2*N,L*K:M), Z(-5:+7)
      ...                                <- (a)
      END
```

Si nous supposons que fi(a) = {L = 6, K = 2}, nous pouvons générer les assertions suivantes:

(ii) toutes les fonctions de cette formule sont dépendantes de ri.

pour les régions décrivant une partie de T:

$\rho_1 - 2N \leq 0$, $\rho_2 - M \leq 0$ (cas (3))
$-\rho_1 \leq -1$ (cas (1))
$-\rho_2 \leq -12$ (cas (2))

pour les régions décrivant une partie de Z:

$-5 \leq \rho_1 \leq 7$ (cas (1))

Remarque V-3: Si les assertions ainsi ajoutées créent une impossibilité dans le système d'assertions associé à la région, cela implique que le programme effectue des débordements de tableau. Notre méthode permet donc, en outre, de détecter certaines anomalies du programme.

V.7. Conclusion

Nous avons déterminé l'ensemble NA des noeuds accessibles. Pour chaque noeud n de NA nous avons déterminé les ensembles RM(n) et RU(n) des régions modifiées et utilisées par l'exécution de ce noeud. L'application de l'élargissement puis du rétrécissement à chaque région de ces ensembles, puis l'union de toutes les régions obtenues, sur l'ensembles NA, conduit à la construction des deux ensembles suivants:

RCM(ω): ensemble des régions constantes modifiées par l'exécution de ω,
RCU(ω): ensemble des régions constantes utilisées par l'exécution de ω.

Les propriétés d'une région constante sont très importantes pour la suite:

Soit $r=(e,i^*) \in RCM(\omega) \cup RCU(\omega)$; $\forall i \in i^*$ $i=(\lambda,\mu)$ ou λ est une combinaison linéaire de $\Lambda(ri)$ et $\mu \in Z$.

La propriété importante est la suivante:

$\forall v \in V_i(ri)$ tq $\lambda(v) \neq 0$, v est une variable globale ou formelle et la valeur qu'elle représente dans la description de la région est sa valeur initiale, au début de l'exécution de ω.

Nous rappelons que ceci est vrai parce que les variables qui ne sont pas éliminées durant l'élargissement sont les variables non modifiées de VNM et que celles qui sont introduites pendant le rétrécissement représentent justement cette valeur initiale, utilisée pour le dimensionnement des tableaux ajustables.

C H A P I T R E VI

VI. Calcul des effets de l'exécution d'un appel d'externe

Ce chapitre est la suite logique du précédent. Nous venons de voir comment calculer les ensembles de régions modifiées et utilisées par l'exécution d'une occurence de procédure. Nous allons voir à présent comment reporter ces deux ensembles sur l'appel d'externe de l'occurence de procédure appelante responsable de cette exécution. Dans la suite nous parlerons de traduction de région.

Ce problème est important: nous avons vu dans le chapitre V que les régions recensées décrivent de façon précise les manipulations d'entités qui sont faites. Il faut donc éviter que cette précision soit perdue par la faute d'une traduction trop grossière. Il faut d'autre part garantir la correction de cette traduction, nous nous basons pour cela sur la norme [AFNO 83].

Ce chapitre est découpé en trois parties.

- Nous listons les éléments dont nous avons besoin pour mener à bien cette traduction.
- Nous montrons comment effectuer la traduction lorsque la région à traduire décrit une partie d'entité commune. Nous introduisons le problème qui se présente lorsque les occurences des procédures appelantes et appelées ne possèdent pas la même description du common. Ce problème, bien que peu important, est la cause d'une discussion de fond sur certains choix qui ont été effectués.
- Nous montrons comment effectuer la traduction lorsque la région à traduire décrit une partie d'entité formelle. Le cas des entités simples est rapidement traité car ne posant pas de problème; pour les entités tableaux, nous proposons une méthode grossière pour le cas général et une méthode plus précise pour un cas particulier d'association paramètre réel/paramètre formel qui est très souvent utilisé en Fortran-77 ou fortran-IV.

VI.1. Eléments nécessaires à la réalisation de la traduction d'une région

Dans la suite nous indiçons par p tout ce qui concerne l'occurence de procédure appelante et par q tout ce qui concerne l'occurence de procédure appelée.

Soit ω_p (resp. ω_q) $\in \Omega$ l'occurence de procédure appelante (resp. appellée).

Soit ri_p (resp. ri_q) $\in$ RI sa représentation interne: $OP\_RI(\omega_p)=ri_p$ (resp. $ri_q = OP\_RI(\omega_q)$)

VI.1.1. Eléments relatifs à l'occurence de procédure appelée

Ce sont ceux qui sont nécessaires à la recherche des effets de son exécution, c'est-à-dire:

- une fonction d'aliasing:

$$fa_q \in FA(ri_q) = E(ri_q) \times E(ri_q) \rightarrow \{ NON, PARTIEL, TOTAL\}$$

  indiquant quels couples d'entités de ω_q sont en aliasing, et le type de cet aliasing;

- une fonction d'assertion:

$$fi_q \in FI(ri_q) = N(ri_q) \rightarrow I(ri_q)^*$$

  associant à chaque noeud de ω_q un ensemble d'assertions vraies avant l'exécution de ce noeud;

- une fonction de manipulation:

$$fm_q \in FM(ri_q) = C(ri_q) \rightarrow R(ri_q)^* \times R(ri_q)^*$$

associant à chaque appel d'externe de ω_q les ensembles de régions modifiées et utilisées par son exécution.

VI.1.2. Eléments relatifs à l'occurence de procédure appelante

Nous supposons que nous disposons pour ω_p des deux éléments suivants:

- une fonction d'aliasing:

$$fa_p \in FA(ri_p) = E(ri_p) \times E(ri_p) \rightarrow \{ NON, PARTIEL, TOTAL\}$$

indiquant quels couples d'entités de ω_p sont en aliasing, et le type de cet aliasing;

- une fonction d'assertion:

$$fi_p \in FI(ri_p) = N(ri_p) \rightarrow I(ri_p)^*$$

associant à chaque noeud de ω_p un ensemble d'assertions qui sont vraies avant l'exécution de ce noeud.

Remarque VI-1: Nous ne disposons bien évidemment pas d'une fonction de manipulation pour ω_p puisque c'est justement le but des chapitres V et VI de montrer comment en calculer une.

Remarque VI-2: Cette remarque complète la remarque V-1: nous avons vu dans le chapitre V que la recherche des effets de l'exécution d'une occurence de procédure ω_q ne dépend pas de ω_q mais uniquement des objets ri_q, fa_q, fi_q et fm_q. De même la traduction d'une région manipulée par l'exécution de ω_q en une région manipulée par l'exécution de ω_p ne dépend ni de ω_p ni de ω_q mais uniquement des objets ri_p, ri_q, fa_p, fa_q, fi_p, fi_q et fm_q. Cela implique que le calcul peut-être mené pour différentes occurences de procédure ω_p^1, ω_p^2, ... et ω_q^1, ω_q^2, ... à la condition que

$$OP\_RI(\omega_q^k) = ri_q, \quad k = 1,2,\ldots \text{ et } OP\_RI(\omega_p^k) = ri_p \quad k = 1,2,\ldots$$

et que l'aliasing, les assertions et les manipulations soient les mêmes pour toutes les occurences de procédure ω_q^i et ω_p^i.

VI.1.3. Notations; définitions

Définition VI-1: une région formelle est une région décrivant une partie d'entité formelle, elle est simple si l'entité est simple, tableau sinon.

Définition VI-2: une région commune est une région décrivant une partie d'entité commune; elle est simple si l'entité est simple, tableau sinon.

Dans la suite nous notons:

- $r_q = (e_q, i_q^*)$ la région à traduire avec $e_q \in E(ri_q)$ et $i_q^* \in I(ri_q)^*$;
- $d_q = nbdim(ri_q)(e_q)$;

- $\forall k \in [1,d_q]$ $bi_q^k \in X(ri_q)$ désigne la borne inférieure de la kième dimension de e_q;
- $\forall k \in [1,d_q]$ $bs_q^k \in X(ri_q)$ désigne la borne supérieure de la kième dimension de e_q;
- $c_p \in C(ri_p)$ l'appel d'externe de ω_p qui correspond à ω_q; à noter que ceci n'est pas incompatible avec la remarque VI-2 car plusieurs occurences de la même procédure peuvent correspondre au même appel d'externe dès que les procédures s'appellent sur plus d'un niveau (cf. chapitre X);
- $n_p \in N(ri_p)$ est le noeud du graphe de contrôle de ri_p qui contient l'appel c_p.
- pr_p^1, pr_p^2, ... les paramètres réels de c_p; ce sont des expressions de $X(ri_p)$;
- $r_p = (e_p, i_p^*)$ la région à traduire avec $e_p \in E(ri_p)$ et $i_p^* \in I(ri_p)$;
- d_p=nbdim(ri_p)(e_p)
- $\forall k \in [1,d_p]$ $bi_p^k \in X(ri_p)$ désigne la borne inférieure de la kième dimension de e_p;
- $\forall k \in [1,d_p]$ $bs_p^k \in X(ri_p)$ désigne la borne supérieure de la kième dimension de e_p;
- (ρ_i^p) i=1..7 sont les variables de description de région de ri_p;
- (ρ_i^q) i=1..7 sont les variables de description de région de ri_q.

VI.2. Traduction d'une région commune

Conformément à la définition VI-1 d'une région commune, e_q est une entité commune. Nous allons être confrontés au problème des représentations différentes d'un common dans ri_p et ri_q. Par représentation nous entendons la déclaration d'un ensemble d'entités de ce common, dont les adresses en mémoire se suivent. Si ri_p et ri_q possèdent chacun une représentation du common, ce qui n'est pas obligatoire (cf. §V.1.3.3), elles ne sont pas nécessairement identiques, ainsi que le montre l'exemple VI-1.

Exemple VI-1:

```
PROGRAM M
COMMON /G1/ MAT(10,10)
COMMON /G2/ I,J,VEC(12),M2(4,3),Z1,Z2
...
END

SUBROUTINE P.
COMMON /G1/ VEC(100)
COMMON /G2/ BUF1(20),M232,BUF2(7)
...
END
```

La différence des descriptions de G1 entre M et P peut être justifiée, par contre les deux descriptions de G2 n'ont rien de commun.

VI.2.1. Recherche dans l'occurence de procédure appelante d'une entité de même déclaration que l'entité de la région à traduire

Soit $g \in G$ le common auquel l'entité e_q appartient. Les représentations de g dans ri_p et ri_q n'ont pas besoin d'être totalement identiques. Il suffit de pouvoir trouver dans

$E(ri_p)$ une entité e_p vérifiant quelques propriétés que nous allons énoncer.

L'adresse de e_q est $adr(ri_q)(e_q)$ = (COMMUNE,g,[d,f]) (cf. §IV.3.3.1) [d,f] désignant le champ d'adresses associé à e_q. L'entité $e_p \in E(ri_p)$ que nous cherchons doit vérifier:

$$adr(ri_p)(e_p) = adr(ri_q)(e_q) \wedge d_p = d_q \wedge$$
$$\text{les types de } e_p \text{ et } e_q \text{ sont de mêmes longueurs} \wedge$$
$$\forall k \in [1,dp],\quad vbi_p^k = vbi_q^k \wedge vbs_p^k = vbs_q^k$$

où vbs_p^k, vbs_q^k, vbi_p^k et vbi_q^k désignent respectivement les valeurs numériques des expressions bs_p^k, bs_q^k, bi_p^k et bi_q^k, valeurs qui existent nécessairement puisque, e_p et e_q étant des entités réelles, ces expressions sont constantes (cf. §IV.3.3.3).

Trois cas peuvent se produire lors de la recherche d'une telle entité: e_p existe et est unique, e_p n'existe pas et il existe plusieurs entités e_p vérifiant les conditions. Nous allons expliquer ce que nous faisons dans les deux derniers cas.

A Cas où e_p n'existe pas

Une première solution consiste à rechercher la partie du common g correspondant à la région r_q, et à l'exprimer en fonction des entités de $E(ri_p)$, du common g, qui ont des adresses en commun avec elle. Cette solution conduit généralement à une perte de précision dans la description de la partie de g décrite par r_q, à moins d'envisager tous les cas d'associations possibles, qui sont nombreux surtout que les types peuvent être différents entre les deux représentations et éventuellement se chevaucher (COMPLEX et LOGICAL par exemple). Cette solution ne présente donc aucun intérêt.

Une deuxième solution consiste à créer une nouvelle entité e_p de même déclaration que e_q et à l'ajouter aux entités de $E(ri_p)$. Ceci n'est pas gênant dans la mesure où une telle entité peut exister, du fait des équivalences. Nous choisissons donc cette solution.

B Cas où il existe plusieurs entités e_p de même déclaration que e_q

Ce cas peut se présenter à cause des équivalences, et a pour conséquences qu'une région peut être traduite de différentes façons.

Nous ne traduisons cependant chaque région que par une seule région dans l'occurence de procédure appelante, ce qui nous impose par la suite la prise en compte des équivalences pour chaque traitement sur les régions, recherche de conflit région/région ou région/variable, calcul des dépendances etc... Ceci n'est pas très grave puisqu'il faut de toutes façons tester l'aliasing pour ces mêmes traitements.

C Conclusion

Nous supposons donc dans la suite que l'entité $e_p \in E(ri_p)$, de même déclaration que e_q, existe.

VI.2.2. Discussion sur la structure des régions

Le paragraphe précédent fait apparaître un problème de fond sur la structure des régions. En effet une région décrit une partie d'entité, ne serait-il pas préférable de lui faire décrire partie de la mémoire ?

Une région serait alors définie par un offset par rapport à une base (début de la zone associée à un common, début de la zone associée à la procédure, début de la zone

associée au ième paramètre formel, etc...) et par une structuration, similaire à la déclaration d'un tableau. Cette solution a pour avantage d'éliminer les problèmes qui sont liés aux équivalences et aux représentations différentes d'un même common.

Notons tout d'abord que notre solution n'est pas tellement éloignée de celle-ci, et que toutes les propositions que nous formulons seraient sans problème adaptées à ce nouveau type de région.

La raison principale qui a guidé les différents choix que nous avons eus à faire pour l'ensemble de ce travail, a été au contraire de rester aussi proche que possible du programme initial. Ceci nous parait indispensable, dans l'optique où notre méthode serait intégrée dans un paralléliseur dialoguant avec les utilisateurs, éventuellement de façon interactive. Que ce soit pour donner des résultats ou pour poser des questions, il est clair qu'il vaut mieux le faire en termes d'entités:

"Quel est le signe de K à l'entrée de la subroutine Q ?"

"La boucle de label 30 n'est pas vectorisée à cause du conflit sur la région $(T,\{\rho_1 = I,\ \rho_2 = J,\ \rho_1 \leq \rho_2\})$"

qu'en termes de bases, d'offsets et de descripteurs de dimensions.

De même, si un programmeur utilise les équivalences ou l'aliasing, il a sans doute de bonnes raisons pour le faire; nous pensons qu'il faut respecter ces raisons et désigner, dans la mesure du possible, les objets par les mêmes noms que le programmeur.

VI.2.3. Réalisation de la traduction d'une région

VI.2.3.1. Cas d'une région commune simple

Lorsque e_q est une variable, la région à traduire est de la forme

$$r_q = (e_q, \emptyset)$$

la traduction en est immédiate et fournit:

$$r_p = (e_p, \emptyset)$$

VI.2.3.2. Cas d'une région commune tableau; traduction d'une assertion

VI.2.3.2.1. Principe

La région r_q représente l'ensemble des éléments de tableau

$$e_q(\rho_1^q,\ \rho_2^q,\ \ldots,\ \rho_d^q)$$

tels que les indices, matérialisés par les pseudo-variables $(\rho_i^q)_{i=1..7}$, vérifient un ensemble de contraintes donné par les assertions de i_q^*.

Conformément au §8.3.4 de [AFNO 83] et à la condition que e_p et e_q aient la même déclaration, les éléments de tableau $e_q(z_1, \ldots, z_d)$ et $e_p(z_1, \ldots, z_d)$ ont la même adresse à la condition que les indices $(z_i)_{i=1..d}$ soient compris entre les bornes inférieures et supérieures de la dimension i correspondante.

Nous pouvons donc traduire r_q en $r_p = (e_p, i_p^*)$ à la condition que les pseudo-

variables $(\rho_i^p)_{i=1..7}$ vérifient un ensemble de contraintes inférieur ou égal à celui vérifié par les pseudo-variables $(\rho_i^q)_{i=1..7}$, cette condition étant nécessaire pour que toutes les adresses représentées par r_q le soient par r_p.

Nous sommes donc ramenés au problème de la construction de i_p^* à partir de i_q^*. Les contraintes sur les (ρ_i^q) sont exprimées en fonction de constantes numériques et de variables de $E(ri_q)$, communes ou formelles, chaque variable représentant sa valeur initiale (cf. §V.7). La valeur initiale de ces variables de ω_q ont-elles une signification vis-à-vis de ω_p ? C'est encore une fois la norme [AFNO 83] qui nous renseigne; nous ne citons pas le texte exact de la norme, mais un résumé.

Soient P et Q deux procédures partageant un common C telles que P appelle Q.

(1) à l'exécution de l'appel à Q, les entités de C déclarées dans Q sont définies avec la valeur des entités correspondantes de C déclarées dans P; cette définition résultant du mécanisme d'association des mémoires;

(2) à l'exécution de l'appel à Q les paramètres formels de Q sont définis avec la valeur des paramétres réels correspondant de l'appel; cette définition ne résulte pas nécessairement d'une association de mémoires, sauf pour les paramètres formels tableaux.

De (1) et (2) nous concluons que les contraintes de i_q^* peuvent être exprimées en fonction d'entités de ω_p, de la façon suivante.

(a) Soit v_q une variable commune de $E(ri_q)$ utilisée dans une assertion de i_q^*. Sa valeur initiale, utilisée pour décrire r_q, est la valeur de l'entité commune v_p de $E(ri_p)$ qui lui est associée. Nous pouvons donc faire la substitution dans l'assertion de v_q par v_p.

(b) Soit v_q une variable formelle de $E(ri_q)$ utilisée dans une assertion de i_q^*. Sa valeur initiale, utilisée pour décrire r_q, est la valeur du paramètre réel pr_p^k qui lui est associé. Nous pouvons donc faire la substitution dans l'assertion de v_q par pr_p^k.

Attention, de telles substitutions peuvent conduire à des contraintes qui ne sont plus des combinaisons linéaires de variables entières de $V_i(ri_p)$. En effet, à cause du problème des représentations différentes d'un common, l'entité v_p de même adresse que v_q, dans le cas (a), n'est pas nécessairement une variable entière. D'autre part un paramètre réel peut être une expression quelconque. L'exemple suivant montre de tels cas:

Exemple VI-2:

```
      PROGRAM M
      COMMON /G1/ IBUF(10)
      ...
      CALL P(T(I),I*J)
      ...
      END

      SUBROUTINE P(L,M)
      COMMON /G1/ N,IBUF (9)
      ...
      END
```

Les substitutions suivantes ne peuvent avoir lieu:

- N car l'entité correspondante est l'élément de tableau IBUF(1),
- L car le paramètre réel associé est un élément de tableau T(I),
- M car le paramètre réel associé est une expression non linéaire.

Remarque VI-3: Il est envisageable d'utiliser les assertions pour rendre linéaires certaines expressions non linéaires. Par exemple, avec le contexte {I==1, ...} l'expression "2I*J+7" est linéaire. Ceci pourrait être inclus dans la fonction "isxcl" sans aucun problème.

VI.2.3.2.2. Réalisation de la traduction d'un ensemble d'assertions linéaires

La réalisation s'effectue selon les 3 phases suivantes.

(1) Nous recensons l'ensemble des variables $V_i(ri_q)$ telles que les valeurs à leur substituer s'expriment linéairement en fonction des variables de $V_i(ri_p)$, plus un éventuel terme constant. A chacune de ces variables, soit v_q, nous associons:

- une combinaison linéaire de $\Lambda(ri_p)$, notée $cl(v_q)$,
- un terme constant de Z, noté $tc(v_q)$.

(2) Nous éliminons dans i_q^* la totalité des variables de $V_i(ri_q)$ qui sont déclarées non-substituables par la phase précédente, en utilisant l'algorithme décrit au §VIII.3. Cette phase construit un ensemble d'assertions il_q^*.

(3) Chaque assertion $i_q=(\lambda_q,\mu_q)$ de il_q^* est traduite en une assertion $i_p=(\lambda_p,\mu_p) \in I(ri_p)$ par les équations suivantes:

$$\forall v_p \in V_i(ri_p), \quad \lambda_p(v_p) = \sum_{v_q \in V(i_q)} \lambda_q(v_q) * cl(v_q)(v_p)$$

et

$$\mu_p = \mu_q - \sum_{v_q \in V(i_q)} \lambda_q(v_q) * tc(v_q)$$

Où $V(i_q) \subset V_i(ri_q)$ est l'ensemble des variables utilisées dans i_q:

$$V(i_q) = \{v \in V_i(ri_q) \quad tq \quad \lambda_q(v) \neq 0\}$$

L'exemple suivant montre une telle traduction.

Exemple VI-3:

```
PROGRAM P                       SUBROUTINE Q (IX,IY)
COMMON /G1/ L                   COMMON /G1/ M
...                             ...
CALL Q(I+2J,K+I-1)              END
...
END
```

soit à traduire $i_Q = \rho_1^Q - 2.IX + IY - 3.M \leq 7$.

$V(i_Q) = \{\rho_1^Q, IX, IY, M\}$. D'où:

v_q	$cB(v_q)$	$tc(v_q)$
ρ^Q_1	ρ^P_1	0
IX	I+2J	0
IY	K+1	-1
M	L	0

Les formules fournissent alors: $i_p = \rho^P_1$-I-4J+K-3L ≤ 8

Il reste à montrer comment caractériser les variables substituables et comment calculer la combinaison linéaire $cl(v_q)$ et le terme constant $tc(v_q)$ à leur substituer.

a) v_q est une variable commune

La substitution est réalisable si:

$$\exists v_p \in V_i(ri_p) \quad tq \quad adr(ri_p)(v_p) = adr(ri_q)(v_q)$$

Ce qui signifie que les représentations du common de v_q sont identiques localement à v_q (formule tirée de celle du §VI.2.1 quand $d_q = 0$). Dans ce cas, nous avons:

$$cl(v_q) = Vtocl(ri_p)(v_p) \text{ et } tc(v_q) = 0.$$

Dans le cas contraire, la variable v_q n'est pas substituable et doit être éliminée de i^*_q.

b) v_q est une variable formelle

Soit $pr_p \in X(ri_p)$ son paramètre réel associé; la substitution est réalisable si

$$isxcl(ri_p)(pr_p) = VRAI$$

ce qui signifie que ce paramètre réel est une expression combinaison linéaire des variables de $V_i(ri_p)$ (cf. §V.5.1). Dans ce cas, nous avons:

$$cl(v_q) = Xtocl(ri_p)(pr_p) \text{ et } tc(v_q) = Xtotc(ri_p)(pr_p)$$

Dans le cas contraire, la variable v_q n'est pas substituable et doit être éliminée de i^*_q.

c) v_q est une variable de description de région

$\exists k \in 1..7$ tq $v_q = \rho^q_k$; la substitution est toujours réalisable (par définition de ces pseudo-variables), et nous avons:

$$cl(v_q) = Vtocl(ri_p)(\rho^p_k) \text{ et } tc(v_q) = 0$$

VI.2.3.2.3. Conclusion

La traduction d'une région commune tableau $r_q=(e_q, i^*_q)$, dans le cas où les représentations du common $g \in G$ sont identiques localement à e_q, conduit à une région $r_p=(e_p, i^*_p)$ où

- e_p est l'entité de $E(ri_p)$ vérifiant les conditions du §VI.2.1,
- i^*_p est construit en traduisant l'ensemble d'assertions i^*_q.

Exemple VI-4:

```
PROGRAM P
COMMON /G/ MAT(10,20),VEC(50),IND
...
CALL Q(I*J,2*I-J+7,VEC(I))
...
END

SUBROUTINE Q(L,M,M)
COMMON /G/ MAT(10,20),VEC(50),IND
...
END
```

Voici quelques exemples de traduction de région:

$(MAT, \{\rho_1^Q - L = 0,\ \rho_2^Q - M = 0,\ \rho_1^Q - \rho_2^Q \leq 0,\ 1 \leq \rho_1^Q \leq 10,\ 1 \leq \rho_1^Q \leq 20\})$

$\rightarrow \quad (MAT, \{\rho_2^P - 2I + J = 7,\ \rho_1^P - \rho_2^P \leq 0,\ 1 \leq \rho_1^P \leq 10,\ 1 \leq \rho_2^P \leq 20\})$

l'assertion $\rho_1^Q - L$ est perdue puisque le paramètre réel associé à L est non linéaire.

$(VEC, \{\rho_1^Q - N = 0,\ 1 \leq \rho_{1_Q} \leq 50\}) \quad \rightarrow \quad (VEC, \{1 \leq \rho_1^P \leq 50\})$

l'assertion "$\rho_1^Q - N = 0$" est perdue puisque le paramètre réel associé à N n'est pas construit sur les variables entières de P.

$(VEC, \{\rho_1^Q - IND = 1,\ 1 \leq \rho_1^Q \leq 50\}) \quad \rightarrow \quad (VEC, \{\rho_1^P - IND = 1,\ 1 \leq \rho_1^P \leq 50\})$

VI.3. Traduction d'une région formelle

Conformément à la définition VI-2 d'une région formelle, e_q est une entité formelle. Nous allons tout d'abord examiner le cas où e_q est une variable, qui est simple à traiter bien qu'offrant plus de possibilités d'associations paramètre réel/paramètre formel. Nous verrons notamment que certaines associations nous conduisent à ignorer la région à traduire.

Nous nous intéressons ensuite au cas où e_q est un tableau. Nous sommes alors confrontés à un problème semblable à celui des représentations différentes d'un common bien que plus complexe. Ce problème nous conduit à proposer deux solutions: une solution brutale pour le cas général et une solution fine, adaptée à un type d'association particulier, très souvent utilisé en Fortran-77.

Dans la suite, nous notons $pr_p \in X(ri_p)$ le paramètre réel associé à l'entité formelle e_q (i).

(i) La phase ASSIA intègre au graphe de contrôle les associations étiquette-d'instruction/*; d'autre part les associations nom-de-procédure/procédure-formelle sont intégrées au graphe des occurences d'appels par le procédé que nous proposons au chapitre X.

VI.3.1. Cas d'une région formelle simple

La norme [AFNO 83] nous indique que le paramètre réel associé à un paramètre formel variable doit être de l'un des deux types suivants:

(1) un lhs: dans ce cas tout se passe comme si l'association entre la suite de mémoires du lhs et celle de la variable formelle était effective (ii);

(2) une expression complexe (cf. §IV.3.3.3): dans ce cas, l'expression est évaluée et sa valeur est donnée au paramètre formel au moment de l'appel; celui-ci ne doit pas être modifié par l'exécution de l'occurence de procédure appelée.

Dans le cas (1), la manipulation de la région r_q à traduire correspond à la manipulation du lhs associé; la région traduite r_p doit donc décrire ce lhs. Nous sommes donc ramenés au passage d'un lhs de $L(ri_p)$ à une région r_p de $R(ri_p)$. Ce passage a été étudié en détails au §V.5.1.

Dans le cas (2), la manipulation de la région r_q à traduire correspond à la manipulation de certaines structures de données temporaires nécessaires pour réaliser l'évaluation de l'expression: registres ou pile etc... Nous ne comptabilisons pas ces manipulations, la région r_q doit donc être ignorée. A noter que nous détectons une erreur si r_q appartient à la liste des régions modifiées par l'exécution de ω_q.

Il est important de remarquer que la gestion de ces structures de données temporaires n'est pas simple à effectuer, dans le cas d'appels effectués en parallèle. C'est la même chose pour le résultat d'une fonction.

Exemple VI-5:

```
PROGRAM P                          SUBROUTINE Q(I,J,K,L)
...                                ...
CALL Q(IT(7,K+L),3,K+7,IZ)         END
...
END
```

Voici quelques régions formelles simples de Q et leur traduction:

(I,Ø) → (IT,{ρ_1=7, ρ_2=K+L})
(J,Ø) → ignorée
(K,Ø) → ignorée
(L,Ø) → (IZ,Ø)

VI.3.2. Cas d'une région formelle tableau

La norme [AFNO 83] nous indique que le paramètre réel doit nécessairement être soit un nom de tableau, soit un nom d'élément de tableau; un nom de tableau est une forme raccourcie pour indiquer le premier élément de ce tableau.

La norme nous indique d'autre part qu'il y a association entre les suites de mémoires des deux paramètres, le ler élément du tableau formel étant associé à l'élément de

(ii) Ceci signifie que le choix d'implémentation est laissé entre un passage de paramètres par "adresse" ou par "copy in/out" (à ne pas confondre avec "valeur résultat" [Tai 82]), techniques équivalentes pour ce type d'association.

tableau paramètre réel et ainsi de suite. Soit $pr_{p_k} \in X(ri_p)$ le paramètre réel; c'est un lhs élément du tableau e_p, dont les indices sont $(z_p^k)_{k=1..dp}$, avec $z_p^k \in X(ri_p)$ $\forall k \in [1,dp]$.

Rien n'empêche en Fortran d'associer, par le biais des associations de paramètres, deux tableaux dont les dimensionnements sont différents. Dans le cas général, il est très difficile de traduire les variations des indices du tableau e_q en des variations d'indices du tableau e_p, autrement que par la technique bien connue de la linéarisation des tableaux.

Cependant, les utilisations courantes de l'association de tableaux formels et réels respectent des conditions qui nous permettent de traduire les variations des indices du tableau e_q en des variations d'indices du tableau e_p.

Nous sommes donc conduit à proposer deux solutions: une solution adaptée au cas courant d'associations et une solution générale.

VI.3.2.1. Solution adaptée au cas où les dimensionnements des deux tableaux formels et réels ne sont pas quelconques

VI.3.2.1.1. Présentation du cas particulier

Un tableau à 1 dimension peut-être considéré comme une succession de variables (tableau à 0 dimension), un tableau à 2 dimensions peut-être considéré comme une succession de "sous-tableaux" à 1 dimension, etc... Par exemple, un vecteur est une succession de scalaires, une matrice est une succession de vecteur colonne etc... (iii).

Le cas particulier que nous étudions se présente lorsque le tableau formel est utilisé pour décrire un sous-tableau du tableau réel. La différence entre le nombre de dimension des deux tableaux n'étant pas nécessairement égale à 1. Dans ce cas, les manipulations du tableau formel se traduisent facilement en des manipulations du tableau réel.

L'exemple présenté au §I.4.2 montre une telle association: l'appel m2 de MM associe au paramètre formel X de SMXPY la Ième colonne de la matrice C.

VI.3.2.1.2. Réalisation de la traduction

Dans le cas où le tableau formel est exactement dimensionné comme un sous-tableau du tableau réel, les deux éléments de tableaux suivants ont la même adresse:

$$e_q(y^1, y^2, \ldots, y^{d_q}) \text{ et } e_p(y^1, y^2, \ldots, y^{d_q}, z_p^{d_q+1}, \ldots, z_p^{d_p})$$

à la condition que $\forall k \in [1,d_q]$, $vbi_p^k \leq y^k \leq vbs_p^k$.

Si nous faisons abstraction des $d_p - d_q$ dernières dimensions, nous sommes ramenés au problème de deux tableaux en association totale; la discussion du §VI.2.3.2. est donc valable. Cependant, ce dont nous avons besoin pour décrire précisément la partie de e_p qui correspond à la région r_q est un objet hybride entre la région et le lhs, où les variations des d_q premiers indices seraient décrits par des assertions et où les autres indices seraient fixés et donnés par des expressions.

Nous ne créons pas cet objet et passons directement de la région r_q à une région $r_p=(e_p, i_p^*)$ où i_p^* est l'union de:

(iii) à noter que ceci n'est pas vrai dans un langage tel qu'algol-W.

- un ensemble il_p^* construit en traduisant les assertions de i_q^* de la façon présentée au §VI.2.3.2.2; cet ensemble décrit les variations des d_q premiers indices de la région;
- un ensemble $i2_p^*$ construit à partir des expressions d'indice z_p^k, $k \in [d_q+1,d_p]$ de la façon utilisée au §V.5.1 pour passer d'un lhs à une région; cet ensemble décrit les variations des d_p-d_q derniers indices de la région.

VI.3.2.1.3. Caractérisation de ce cas particulier

Nous utilisons les notations du §VI.1.3 auxquelles nous ajoutons les suivantes.

Soit vbs_q^k (resp. vbi_q^k), $k=1..d_q$ la valeur de l'expression bs_q^k (resp. bi_q^k) avant le démarrage de l'exécution de l'occurence de procédure ω_q. Cette valeur détermine la valeur de la borne supérieure (resp. inférieure) du tableau e_q. Ces valeurs ne sont pas nécessairement connues à cause des tableaux ajustables, nous discutons de ce problème au §VI.3.2.1.4.

Nous définissons de façon identique vbs_p^k et vbi_p^k, $k=1..d_p$ pour le tableau e_p.

Soit vz_p^k, $k=1..d_p$ la valeur de l'expression z_p^k avant l'exécution de l'appel c_p (qui est l'appel d'externe de ω_p responsable de l'exécution de ω_q). Cette valeur est utilisée pour déterminer l'adresse de l'élément de tableau paramètre réel.

Le cas particulier qui nous intéresse a lieu lorsque les cinq conditions suivantes sont respectées:

(1) $d_q \leq d_p$

(2) $\forall k \in 1..dq \qquad vbi_q^k = vbi_p^k$

(3) $\forall k \in 1..dq-1 \qquad vbs_q^k = vbs_p^k$

(4) $\forall k \in 1..dq \qquad vbi_p^k = vz_p^k$

(5) $\rho_{d_q^{d_p}} \leq vbs_p^{d_q}$

La justification de ces conditions ainsi que la justification du principe de traduction que nous avons exposé au §VI.3.2.1.2 est basée sur la comparaison du calcul de l'offset d'un élément du tableau e_p et d'un élément du tableau e_q; ce calcul est long et ennuyeux, c'est pourquoi nous ne l'exposons pas.

Exemple VI-6:

```
SUBROUTINE P(TAB,L,M,N)
DIM TAB(L,M,N),TC(10,10)
...
CALL Q (TAB(1,1,I),L,TC(1,I),IND)
...
END

SUBROUTINE Q (MAT,LDM,VEC,IX)
DIM MAT(LDM,*),VEC(*)
...
END
```

Soit à traduire la région (VEC,$\{\rho_1^Q = IX,\ \rho_1^Q \geq 1\}$) les conditions (1),(2),(3) et (4) sont vérifiées:

(1) → 1 ≤ 2
(2) → 1=1
(3) → rien à vérifier
(4) → 1=1

d'où $il_p^* = \{\rho_1^P = IND,\ \rho_1^P \geq 1\}$ et $i2_p^* = \{\rho_2^P = I\}$.

La région traduite sera: (TC,$\{\rho_1^P = IND,\ \rho_1^P \geq 1,\ \rho_2^P = I\}$ si le contexte associé à l'appel à Q dans P permet de garantir la condition (5), c'est à dire: $\rho_1^P \leq 10$ ou encore IND ≤ 10.

VI.3.2.1.4. Vérification des conditions

Cette solution n'est réalisable qu'à la condition de pouvoir vérifier automatiquement les conditions exposées au §VI.3.2.1.2. Cette vérification pose un problème car les expressions concernées ne sont pas nécessairement des expressions constantes. C'est vrai des expressions d'indice (z_p^k) qui peuvent être quelconques mais aussi des expressions bornes à cause des tableaux ajustables (iv).

Nous proposons donc d'utiliser les fonctions d'assertions fi_p et fi_q que nous avons supposé connues, pour augmenter le nombre d'utilisations de cette méthode. Les expressions bornes doivent être évaluées dans le contexte associé au noeud racine de la représentation interne (ri_p pour e_p et ri_q pour e_q), la norme indiquant en effet que les tableaux ajustables sont dimensionnés avec les valeurs initiales des expressions bornes, c'est à dire avant toute exécution d'une instruction de l'occurence de procédure. Les expressions d'indices doivent être évaluées avec le contexte associé au noeud n_p de $N(ri_p)$ qui contient l'appel analysé: $c_p \in C(ri_p)$.

Bien que les tableaux ajustables soient souvent utilisés, les raisons suivantes nous font penser que les conditions (1) à (4) seront très souvent vérifiées:

(1) les expressions bornes supérieures, très souvent ajustables, ne sont utilisé dans les conditions que si $d_q \geq 2$; toutes les associations matrice/vecteur sont donc sans problème;

(iv) A noter que les tableaux de taille non spécifiée ne sont pas gênants puisque la borne supérieure de la dernière dimension (d_q) ne doit vérifier aucune condition.

(2) la très grande majorité des bornes inférieures sont constantes et égales à 1, pour la raison historique que Fortran IV ne permet pas de choisir les bornes inférieures d'un tableau;

(3) il en est de même pour les expressions d'indices; notons que l'utilisation en paramètre réel d'un nom de tableau nous dispense de la vérification de la condition (4);

(4) la dernière raison est plus profonde: nous pensons qu'une phase de propagation des constantes interprocédurale comme celle que nous proposons aux chapitres VII et X permet de découvrir la valeur d'un grand nombre de variables dans chaque occurence de procédure; en effet Fortran ne permet pas de dimensionner dynamiquement un tableau, contrairement à un langage comme algol-W:

```
debut
   ecrire("Entrez la dimension du système: ");
   lire(N);
   si N > 1 alors
   debut
      entier tableau MAT(1..N,1..N);
      ...
```

le programmeur Fortran dimensionne donc tous ses tableaux et toutes ses boucles en fonction d'un ensemble de constantes symboliques (en Fortran-77) ou de variables initialisées par DATA (en Fortran-IV) qui sont passées en paramètres et ne sont jamais modifiées; ce sont ces constantes là que nous pensons récupérer; notons que l'artifice qui consiste à stocker une matrice dans un tableau plus grand que cette matrice, et à propager dans les sous-programmes la dimension "statique" du tableau et la dimension "dynamique" de la matrice (cet artifice est utilisé dans l'exemple 1) ne nous gêne pas puisque le tableau reste statiquement dimensionné; nous pensons donc que beaucoup de cas seront traités grâce au calcul et à l'utilisation des assertions;

VI.3.2.2. Solution générale

Le paragraphe 15.9.3.3 de [AFNO 83] stipule que la taille d'un tableau formel ne doit pas dépasser la taille du tableau réel associé. Ceci implique que la manipulation d'un élément du tableau e_q n'a d'influence que sur l'entité e_p et sur les entités qui lui sont associées, par aliasing ou par équivalence; la remarque VI-3 explique pourquoi nous ne nous occupons pas de ces dernières.

En conséquence, la traduction de la région formelle tableau $r_q=(e_q,i_q^*)$ par la région $r_p=(e_p,\emptyset)$, où e_p est le tableau réel associé à e_q, est correcte. C'est la solution générale que nous proposons.

Il existe une solution plus élégante utilisant la linéarisation des tableaux. Cette solution pose les mêmes problèmes que notre première solution en ce qui concerne les tableaux ajustables. Nous ne la présentons pas vu le peu d'intérêt pratique qu'elle présente.

VI.4. Conclusion

Nous venons d'examiner comment effectuer le report des effets de l'exécution d'une occurence de procédure sur l'appel d'externe correspondant de l'occurence de procédure appelante. Ce report revient à traduire une région décrivant une partie d'entité de l'occurence de procédure appelée en une région décrivant une partie d'entité de l'occurence

de procédure appelante.

Cette étude a été divisée en trois cas principaux suivant la nature de la région à traduire et le type de l'association entre les entités des deux occurences de procédure. Elle a été menée en supposant connus:

-une fonction d'aliasing fa_p pour ω_p,

-une fonction d'assertion fi_p pour ω_p,

-la représentation interne ri_p associée à ω_p,

-le noeud $n_p \in N(ri_p)$ et l'appel $c_p \in C(ri_p)$ reliant ω_q et ω_p,

-une fonction d'aliasing fa_q pour ω_q,

-une fonction d'assertion fi_q pour ω_q,

-une fonction de manipulation fm_q pour ω_q.

Cette étude nous conduit à la construction des deux ensembles $RM(c_p)$ et $RU(c_p) \in R(ri_p)^*$ des régions modifiées et utilisées par l'exécution de l'appel C_p.

C H A P I T R E VII

VII. Calcul d'assertions sur les variables d'une occurence de procédure

Une des trois propositions d'amélioration des méthodes de parallélisation existantes est l'utilisation d'assertions sur les variables de chaque occurence de procédure, pour améliorer le traitement des manipulations de tableaux. Nous avons vu dans les chapitres V et VI comment nous réalisons cette amélioration. Les assertions sont utilisées pour améliorer la recherche de l'approximation des parties de tableaux réellement manipulées par l'exécution d'un opérateur, mais aussi pour les décrire, par les régions.

Les assertions peuvent être utilisées à d'autres fins. Nous avons vu au §V.4 qu'elles nous permettent d'éliminer certaines branches du graphe de contrôle, entrainant ainsi une réduction du nombre d'opérateurs des occurences de procédure. Nous verrons d'autre part comment elles nous permettent d'affiner sensiblement le calcul de l'aliasing dans le chapitre IX.

La connaissance d'assertions n'est jamais nécessaire. Elle est toujours introduite, dans les méthodes que nous avons décrites ou que nous allons décrire comme un moyen d'améliorer les résultats.

Le but de ce chapitre est de présenter différentes méthodes de calcul automatique d'assertions sur les variables d'un programme. Deux de ses méthodes ont déjà fait l'objet de publications, mais nous montrons comment les adapter à notre cas particulier. Il est organisé de la façon suivante.

(1) Nous présentons la méthode de propagation des constantes [Kild 73]. L'aspect théorique est rapidement traité, puis nous montrons quels sont les problèmes posés par sa réalisation dans le cadre de Fortran-77 complet, les principaux étant liés aux procédures.

(2) Nous présentons la méthode générale développée par P. Cousot dans [Cous 78]. Puis nous décrivons les résultats fournis par deux implémentations qui en ont été faites par N. Halbwacks [Halb 79] et J.P Jung [Jung 83]. Les problèmes posés par l'adaptation de cette méthode à Fortran-77 sont identiques à ceux de (1).

(3) Nous proposons une méthode simple de détection d'assertions, basée sur l'étude des boucles-DO d'une occurence de procédure.

(4) Nous discutons de l'intérêt présenté par l'introduction d'assertions dans le texte source des procédures, par le programmeur ou par le paralléliseur lui-même.

(5) Nous montrons comment calculer un ensemble d'assertions initiales, c'est à dire associées au noeud racine de l'occurence de procédure analysée, à partir des assertions qui sont associées, dans l'occurence de procédure appelante, au noeud contenant l'appel d'externe associé.

VII.1. Etude de la méthode de propagation des constantes

Cette méthode à été proposée en 1973 par G. Kildall [Kild 73]. Elle permet d'associer à chaque noeud du graphe de contrôle d'un programme l'ensemble des variables "constantes" pour ce noeud, c'est à dire telles que tous les chemins d'exécution conduisant à ce noeud, lui affectent la même valeur.

VII.1.1. Cadre théorique de cette méthode

VII.1.1.1. Treillis des assertions

La méthode de propagation des constantes à été développée dans le modèle des treillis complets.

Un treillis complet est un couple $(T,\sqcap)$, où T est un ensemble non vide et $\sqcap$ une opération binaire sur T, idempotente, commutative et associative. Les éléments du treillis complet représentent l'information attachée à un noeud du graphe de contrôle; l'opération binaire permet de calculer l'information en un noeud où convergent plusieurs chemins d'exécution. Dans la suite, nous supposons que l'ensemble T est de longueur finie: toute chaine strictement décroissante d'éléments de T est de longueur finie.

L'effet de l'opérateur associé à un noeud du graphe de contrôle sur un élément du treillis complet T est modélisé par une opération sur T. G. Kildall introduit pour cela la notion d'espace d'opérateurs monotones associé à T. C'est un ensemble F d'opérateurs sur T vérifiant quatre conditions. Dans la suite T est partiellement ordonné par la relation $\sqsubseteq$ réflexive, antisymétrique et transitive.

(C1) Tout opérateur $f \in F$ est monotone:
$x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$ ou encore $f(x \sqcap y) \sqsubseteq f(x) \sqcap f(y)$

(C2) F contient un opérateur identité, noté e: $\exists e \in F$ tq $\forall x \in T$ $e(x) = x$

(C3) F est stable pour la composition: $\forall f, g \in F$, $f \circ g \in F$ avec $f \circ g(x) = f(g(x))$, $\forall x \in T$

(C4) $\forall x \in T \exists f \in F$ tq $x = f(0)$ où 0 est l'élément zéro du treillis T défini par $\nexists x \in T$ tq $x \sqsubseteq 0$.

M. S. Hecht donne une signification intuitive de ces quatre conditions dans [Hech 77]:

* la condition (C1) est observée dans tous les problèmes qui relèvent de l'analyse de flot de données,

* la condition (C2) est imposée par le fait qu'un noeud peut ne pas modifier l'information quand le contrôle lui est donné,

* la condition (C3) reflète le passage d'information à travers deux noeuds successifs.

* La condition (C4) empêche l'addition dans T d'information inconsistante.

Ces définitions permettent à G. Kildall de définir un "modèle monotone d'analyse de flot de données" et une instantiation d'un tel modèle:

définition: un modèle monotone d'analyse de flot de données (ou modèle monotone) est un triplet $(T,\sqcap,F)$ où:

$(T,\sqcap)$ est un treillis complet avec 0 comme élément zéro;
F est un espace d'opérateurs monotone associé à T.

définition: une instantiation d'un modèle monotone est un doublet $I = (G,M)$ où:

$G = (N,A,r)$ est un graphe: N est un ensemble de noeuds, $A \subset N \times N$ est la relation prédécesseur et r est la racine du graphe;

M: N → F est une fonction qui associe à tout noeud de N un opérateur dans F.

VII.1.1.2. La solution "Meet Over all Paths" (MOP)

La solution recherchée est celle qui permet de connaitre pour chaque noeud le maximum d'information disponible au début de ce noeud, en tenant compte de tous les chemins d'exécution qui y conduisent.

La méthode intuitive permettant de calculer cette information consiste à calculer pour chaque chemin C du noeud r (racine) au noeud i, l'élément de T disponible en i si l'exécution suit le chemin C. Le résultat final s'obtient en calculant "l'intersection" de ces éléments.

Notations

Soit I=(G,M) une instantiation d'un modèle monotone (T,⊓,F); on note $f_i = M(i)$ l'opérateur associé au noeud i.

Soit $Q = x_1, x_2, .., x_m, x_{m+1}$ un chemin de G. On note

$$f_q = f_m \circ f_{m-1} \circ .. \circ f_1 \quad (f_{m+1} \notin f_q).$$

Enfin on note A[i] l'élément de T représentant l'information associée au noeud i.

Définition

La solution MOP est donnée par les équations:

$$A[r] = 0$$
$$A[x] = \bigsqcap_{Q \in PATH(x)} f_Q(0)$$

où PATH(x)= {Q tq Q est un chemin de G menant de r à x}.

M.S. Hecht prouve dans [Hech 77] qu'il n'existe pas d'algorithme permettant de calculer pour toutes les instantiations d'un modèle monotone la solution MOP [Hech 77]. Nous devons donc nous contenter d'une solution approchée appelée la solution "Maximum Fixed Point" calculable par plusieurs algorithmes.

VII.1.1.3. La solution "Maximum Fixed Point"

Les noeuds de N sont supposés ordonnés selon un ordre nommé reverse post order (i) noté rpo dans la suite. On définit d'autre part les deux ensembles:

$$PRED(x) = \{y \in N \text{ tq } (y,x) \in A\} \quad \forall x \in N$$
$$PRED^*(x) = \{y \in N \text{ tq } y \in PRED(x) \text{ et } y < i \text{ (suivant rpo)}\}$$

La solution MFP est donnée par les équations suivantes:

$$X[1] = 0$$
$$X[i] = \bigsqcap_{j \in PRED(i)} f_j\,(X[j]), \ \forall i = 2..n$$

(i) Le reverse post order est particulièrement utile en analyse de flot de données pour le parcours des graphes. On trouvera un algorithme dans [hech 77].

L'avantage de cette solution est d'être automatiquement calculable. Il existe différents algorithmes permettant de résoudre les équations citées plus haut; nous présentons l'algorithme original de Kildall [Kild 73] car il est particulièrement simple à implémenter. Il est cependant moins efficace que d'autres variantes telles que celles proposés par Kam et Ulmann dans [Kam 76].

VII.1.1.4. L'algorithme itératif général; version "round-robin"

L'algorithme suivant permet de résoudre les équations précédentes:

```
procedure GIA
        soit TEMP ∈ T;
        soit i ∈ N;
        soit CHANGE ∈ {VRAI, FAUX};

        A [1] := 0,
        pour i := 2 jusqua n faire
                A[i] := ⌐⌐ f_g(A[g])
                        g∈PRED*(i)

        CHANGE := VRAI;
        tantque CHANGE faire
                CHANGE := FAUX;
                pour i := 2 jusqua n faire
                        TEMP := ⌐⌐ f_q(A[q])
                                q∈PRED(i)
                        si TEMP ≠ A[i] alors
                                CHANGE := VRAI;
                                A[i] := TEMP;
                        finsi;
                finpour;
        fintantque;
finproc;
```

VII.1.2. Cadre de réalisation de la méthode de propagation des constantes

Nous allons à présent étudier les problèmes qui sont posés par l'adaptation de cette méthode au langage Fortran-77. La construction du graphe de contrôle du programme est réalisée par la phase ASSIA et n'est donc pas abordée dans ce chapitre.

Nous nous intéressons tout d'abord au choix du treillis complet. Ceci nous amème à aborder les problèmes relatifs aux associations d'entités, par équivalence ou par aliasing. Nous montrons ensuite comment associer à chaque noeud du graphe de contrôle, un opérateur sur ce treillis, exprimant l'effet de l'exécution de ce noeud sur l'information qui lui est associée. Ceci nous conduit à évoquer les problèmes relatifs aux appels de procédure. Nous expliquons pourquoi nous ne conservons que l'aspect destructeur d'un appel de procédure.

Dans la suite nous travaillons sur une occurence de procédure $\omega \in \Omega$, de représentation interne ri $\in$ RI. Nous supposons disposer des éléments suivants:

- une fonction fa ϵ FA(ri) = E(ri) x E(ri $\rightarrow$ {NON,PARTIEL,TOTAL} indiquant quels couples d'entités de ω sont en aliasing, et le type de cet aliasing;

- une fonction fm ϵ FM(ri) = C(ri) $\rightarrow$ R(ri)* x R(ri)* associant à chaque appel d'externe de ω les ensembles de régions modifées et utilisées par son exécution.

VII.1.2.1. Choix du treillis des assertions

A Différents treillis envisageables

M.S. Hecht propose d'utiliser le treillis complet $(T_v(ri),\sqcap)$ où:

$T_v(ri)=(V_i(ri)xN)^*$ et $\sqcap$ est l'intersection d'ensembles.

T_v peut être vu comme l'ensemble des fonctions allant d'un sous-ensemble fini de $V_i(ri)$ dans N. Un élément de $T_v(ri)$ associe une valeur à certaines variables de l'occurence de procédure. L'élément zéro de $T_v(ri)$ est $\emptyset$.

Ce treillis ne permet pas de modéliser le fait que certaines variables sont en association. Son utilisation pose donc des problèmes pour expliciter le fait que la valeur associée à une variable peut être détruite lorsque la valeur associée à une autre variable est modifiée.

Il vient alors à l'esprit d'associer les valeurs à la mémoire plutôt qu'aux variables. Il est tentant d'introduire un ensemble MEM(ri) représentant la mémoire, et une fonction MAP(ri): $V_i(ri) \rightarrow$ MEM(ri) associant une mémoire à chaque variable. Le treillis associé étant $(T_m(ri),\sqcap)$ où:

$T_m(ri) = (MEM(ri) \times N)^*$ et $\sqcap$ est l'intersection d'ensembles (ii).

Le problème cité plus haut disparaît. En effet, la modification de la valeur associée à une mémoire implique la destruction d'une éventuelle valeur transmise par une autre variable à cette même mémoire. Cependant ce treillis est inapplicable à Fortran à cause des variables formelles. Nous verrons en effet, au chapitre IX, que le champ d'adresses associé à une variable formelle, pour une occurence de procédure donnée, n'est généralement pas réduit à une seule mémoire. Il en résulte que la fonction MAP n'est pas définissable.

B Choix du treillis; conséquences

Nous utilisons donc le treillis $(T_v(ri),\cap)$ malgré les problèmes qu"il pose pour les associations. Puisque ceux-ci ne peuvent être réglés par le choix du treillis, nous les reportons à la construction de l'opérateur sur $T_v(ri)$, associé à chaque noeud.

Le résultat de la procédure GIA, présentée au §VII.1.1.4, est une fonction A: N(ri) $\rightarrow T_v(ri)$ associant à chaque noeud un ensemble de couples de $V_i(ri)xN$. Chaque couple (v,c), associé à un noeud n, indique que v a la valeur c avant toute exécution de n.

Nous avons choisi de représenter les assertions associées à un noeud par un élément de I(ri)*. Nous devons donc montrer comment passer d'un élément t ϵ $T_v(ri)$ à l'élément de I(ri)* correspondant. Ceci est réalisé en associant à chaque couple (v,c) de t les deux

(ii) dans le cadre où se place M.S. Hecht et G. Kildall, T_v et T_m sont équivalents puisqu'il existe une bijection de V dans M.

assertions {v ≤ c, -v ≤ -c} par la formule suivante:

$$\bigcup_{(v,c)\ \epsilon\ t} \{(Vtocl(ri)(v),c)\ ,\ (negcl(ri)(Vtocl(ri)(v)),-c)\}$$

VII.1.2.2. Construction de l'opérateur monotone associé à chaque noeud

Un problème de vocabulaire se pose: nous appelons déjà opérateur l'objet, associé à chaque noeud, qui réalise partiellement ou totalement le traitement décrit par l'instruction dont est issu le noeud. Nous renommons temporairement "tâche" cet objet là. Le but de ce paragraphe est de montrer comment construire, pour chaque tâche, l'opérateur sur $T_v(ri)$ qui décrit l'effet de son exécution sur les informations associées au noeud.

L'exécution d'une tâche peut créer et/ou détruire de l'information. Nous devons donc analyser l'ensemble des instructions de Fortran-77, et rechercher celles qui sont suceptibles de créer de l'information et celles qui sont susceptibles d'en détruire.

VII.1.2.2.1. Instructions de Fortran-77 créatrices d'information

L'analyse de Fortran-77 montre que la seule instruction susceptible de créer de l'information, par sa sémantique propre, est l'instruction d'affectation. Cependant toutes les instructions peuvent en créer par l'intermédiaire des appels d'externes qu'elles contiennent. Nous examinons séparémment ces deux possibilités.

A Cas de l'instruction d'affectation

Soit s ϵ S(ri) l'instruction associée à un noeud n. Si s est une instruction d'affectation, la représentation interne ri nous fournit l'expression x ϵ X(ri) affectée à l ϵ L(ri), le lhs destination.

Les instructions d'affectation créatrices d'information sont celles qui vérifient:

(1) le lhs l est une référence à une variable entière v ϵ $V_i(ri)$;

(2) l'expression x est constante ou évaluable par la fonction X_ifovali, avec le contexte associé à n par la fonction A qui est en train d'être calculée.

La vérification du point (1) est sans problème. Pour vérifier le point (2), il suffit de transformer les assertions contenues dans A(n) en des assertions assimilables par la fonction X_ifovali (cf. §VIII.6.2), grâce à la formule proposée au §VII.1.2.1 (B), et d'appeler cette fonction d'évaluation sur x.

Si ces deux points sont vérifiés, nous créons une assertion représentée par le couple (v,c), où c est la valeur renvoyée par l'appel à X_ifovali. Devons-nous générer des assertions équivalentes pour toutes les variables v' ϵ $V_i(ri)$ qui sont en association totale avec v soit par équivalence (adr(ri)(v) = adr(ri)(v')) soit par aliasing (fa(v,v') = TOTAL) ? La réponse est non, pour la raison suivante.

Nous proposons de "prétraiter" le système d'assertions associé à un noeud, avant de l'utiliser pour évaluer une expression ou prouver une assertion. Ce prétraitement, que nous présentons au §VIII.1, consiste essentiellement à introduire dans le système un ensemble d'assertions traduisant l'égalité des valeurs de deux variables en association totale. Les expressions à évaluer et les assertions à prouver subissent le même prétraitement.

Ce prétraitement pourrait être effectué au moment de la création d'assertions.

Nous pensons qu'il est préférable de le reporter pour deux raisons.

(1) Il faudrait introduire ce prétraitement dans toutes les méthodes de calcul d'assertions à implémenter, ce qui conduirait à un manque de modularité.

(2) La deuxième raison rejoint ce que nous avons dit au §VI.2.2. Si une mémoire m est désignée par le nom s1 au noeud n1 et par le nom s2 au noeud n2, nous pensons que les assertions connues sur m au noeud n1 (resp. n2) doivent être exprimées par l'intermédiaire de s1 (resp. s2). Les dialogues avec l'utilisateur n'en seront que meilleurs.

L'effet destructeur d'une instruction d'affectation est étudié au §VII.1.2.2.2.

B Cas des instructions comportant des appels d'externe

L'exécution d'un appel d'externe implique l'exécution d'une occurence de procédure complète, il est donc évident qu'elle peut conduire à la création de certaines informations. Une méthode applicable pour les calculer est la suivante:

(1) Une phase de calcul d'assertions sur les variables de l'occurence de procédure appelée est effectuée en traduisant les assertions connues au moment de l'appel sur les variables communes et en évaluant les paramètres réels associés à des variables formelles entières avec ces mêmes assertions. Les assertions ainsi calculées sont associées au noeud racine de l'occurence de procédure appelante.

(2) La méthode de propagation des constantes est effectuée sur l'occurence de procédure appelée.

(3) La transformation inverse de (1) est effectuée sur les assertions associées au noeud "return" de l'occurence de procédure appelée. La comparaison de ce résultat et des assertions initiales fournit l'information générée par l'appel.

Cette solution ne nous paraît pas raisonnable car elle implique un grand nombre d'itérations. Elle est en effet équivalente à l'application de la propagation des constantes sur le programme qui résulterait de l'expansion systématique des appels de procédure.

La solution que nous préconisons ne permet pas de tirer profit de l'information générée par un appel d'externe. Nous proposons en effet de ne conserver d'un appel que son effet destructeur (cf. §VII.1.2.2.2). Entre ces deux extrêmes, nous pouvons imaginer une méthode de calcul symbolique qui analyserait une procédure et lui associerait une fonction, permettant de calculer ses effets sur les paramètres réels et variables globales. C'est l'approche développée par P. Fautrier, dans sa calculette symbolique, pour un ensemble d'instructions (corps de boucle). Cette méthode n'a pas, à notre connaissance, été utilisée pour condenser les effets d'un appel de procédure.

En conclusion, nous ne cherchons pas à calculer les assertions créées par l'exécution d'un appel d'externe.

Remarque VII-1: La méthode de propagation des constantes associe un opérateur unique à un noeud. Il n'est donc pas possible d'utiliser la terminaison d'un opérateur pour en déduire de l'information.

```
IF (I. EQ.2) THEN
        ...             (a)
ENDIF
```

L'information {I = 2} n'est pas récupérable en (a).

VII.1.2.2.2. Instructions de Fortran-77 destructrices d'informations

L'exécution d'une tâche détruit l'information qui est associée aux variables qu'elle modifie. Ces modifications peuvent être dues à la nature de l'instruction associée à la tâche - une instruction READ modifie les entités de sa liste d'entrées - soit aux exécutions d'appels d'externe impliquées par l'exécution de la tâche.

Nous avons déjà mené cette discussion, aux §V.5.1 & §V.5.2, pour montrer qu'il est possible de construire, pour chaque noeud d'une occurence de procédure, l'ensemble des régions directement et indirectement modifiées par l'exécution de la tâche associée à ce noeud. Nous ne sommes pas placés dans les mêmes conditions, puisque nous ne disposons pas d'une fonction d'assertion f_i, supposée connue au chapitre V. Cependant, étant donnée que cette fonction n'est pas utilisée dans ces paragraphes, nous sommes capables de calculer RM(n): ensemble des régions modifiées par l'exécution de n.

Soit t = A(n) les assertions associées au noeud n. La recherche des destructions engendrées par n se ramène à vérifier pour chaque couple (v,c) ϵ t s'il existe r ϵ RM(n) telle que r et v sont en conflit. Dans ce cas, le couple (v,c) doit être détruit. Nous étudions donc le conflit variable/région.

Etude du conflit variable/région

Nous posons $r=(e,i^*)$. Le conflit a lieu dans trois cas:

1) e = v;
2) e et v sont en aliasing;
3) e et v sont équivalence.

Etudions ces trois cas.

1) Ce cas est trivial. Une région d'une variable v décrit la totalité de cette variable. Le conflit est donc sûr. D'où:

e = v => r et v sont en conflit

2) Si e est une variable, nous sommes ramenés au cas 1), que l'aliasing soit total - signifiant que les suites de mémoires associées à e et v sont égales - ou partiel - signifiant que les suites de mémoires associées à e et v ont une intersection non vide. Si e est un tableau, l'aliasing entre e et v est nécessessairement partiel. Cependant, la représentation de l'aliasing que nous avons choisie ne permet pas de décrire précisemment l'élément du tableau e en aliasing avec v. En conséquence, même si r décrit un élément précis du tableau e (e.g. $(T,\{\rho_1 = 3\})$), nous ne sommes pas en mesure d'affiner le calcul du conflit. A noter que ce choix est justifié dans le chapitre IX. D'où:

fa(e,v) ≠ NON => r et v sont en conflit

3) Notons que e et v sont nécessairement des entités réelles. Si e est une variable, nous sommes ramenés au cas 1). Si e est un tableau, nous pouvons éliminer certains conflits en utilisant les assertions A(n) associés au noeud n. Voici l'algorithme à suivre:

(1) Nous transformons l'élément A(n) $\in T_v(ri)$ en un élément $i_1^* \in I(ri)^*$ selon la formule du §VII.1.2.1 (B). Cette phase ne doit être effectuée que dans le cadre de la propagation des constantes. Nous rappelons en effet que la recherche des conflits variable/région est utilisée au §V.6.3.2, avec des régions telles que le contexte local associé au noeud est déjà inclus dans les assertions de ces régions; dans ce cas $i_1^* = \emptyset$.

(2) Nous formons i_2^* en utilisant la phase de rétrécissement d'une région présentée au §V.6.4. Cette phase a théoriquement besoin d'une fonction d'assertion f_i, que nous ne connaissons pas. Cependant elle n'est pas utilisée dans le cas présent puisque e est une entité réelle de déclaration faite avec des expressions constantes. Nous rappelons que cette phase de rétrécissement a pour but d'introduire les assertions du type $\rho_k - vbs_k \leq 0$ et $vbi_k - \rho_k \leq 0$ où vbi_k (resp. vbs_k) désigne la valeur de la borne inférieure (resp. supérieure) de la dimension k de e.

(3) Nous formons $i_3^* = i^* \cup i_1^* \cup i_2^*$.

(4) Nous éliminons dans i_3^* la totalité des variables qui ne sont pas des variables de description de région. L'algorithme à utiliser est décrit au §VIII.3.

(5) Nous recherchons grâce au système résultant une borne inférieure (resp. supérieure) pour chaque variable de description de région présente dans les assertions. Notons que cette borne existe nécessairement grâce à la phase 2. L'algorithme de cette recherche est donnée au VIII.3.

(6) A partir de ces bornes inférieures (resp. supérieures), nous calculons la valeur minimum (resp. maximum) de l'offset d'un élément du tableau e décrit par la région $r = (e, i_3^*)$. Nous en déduisons un champ d'adresses restreint pour e, <u>inclus</u> dans celui donné par adr(ri)(e).

(7) Le conflit entre v et e est calculé par intersection du champ d'adresses associé à v et du nouveau champ d'adresses associé à e. Les risques de conflit sont donc diminués.

<u>Exemple VII-1</u>: Soit T un tableau déclaré par DIM T(-5:+5,100). Nous appliquons l'algorithme ci-dessus à la région $(T,\{\rho_1 = I, \rho_2 = J+1\})$ en un noeud où le contexte est $\{I \geq 0, J \geq I+1\}$. Nous ne nous plaçons pas dans le cadre de la propagation des constantes et démarrons donc en (2).

(2) $i_2^* = \{-5 \leq \rho_1 \leq 5,\ 1 \leq \rho_2 \leq 100\}$

(3) $i_3^* = \{\rho_1 = I,\ \rho_2 = J+1,\ I \geq 0,\ J \geq I+1,\ -5 \leq \rho_1 \leq 5,\ 1 \leq \rho_2 \leq 1000\}$

(4) $i_3^* = \{\rho_1 \geq 0,\ \rho_2 \geq 2,\ -5 \leq \rho_1 \leq 5,\ 1 \leq \rho_2 \leq 100\}$

(5) $\rho_1 min = 0,\ \rho_2 min = 2,\ \rho_1 max = 5,\ \rho_2 max = 100$

(6) offsetmin = (0-(-5)) + (2-1) x (5-(-5)+1) = 16
offsetmax = (5-(-5)) + (100-1) x (5-(-5)+1) = 1099

Si a est l'adresse du 1er élément de T, le champ d'adresses associé à T par la fonction adr est [a,a+1099], nous l'avons réduit à [a+16,a+1099].

VII.1.2.2.3. Conclusions sur la construction de l'opérateur monotone associé à un noeud

Soit $t \in T_v(ri)$; l'algorithme de l'opérateur associé à un noeud n est donc:

(1) les couples (v,c) $\in$ t, vérifiant $\exists r \in RM(n)$ tq r et v sont en conflit, sont détruits dans t;

(2) soit $s \in S(ri)$ l'instruction associée à n; si s est une instruction d'affectation vérifiant les conditions du §VII.1.2.2.1 (A), ajouter à t le couple (v,c) où c est la valeur affectée à v par s.

VII.1.3. Conclusion sur la méthode de propagation des constantes

Notre étude de cette méthode n'est pas tout à fait achevée. Nous proposons en fait au chapitre X de réaliser une phase de calcul d'assertions interprocédurale par une analyse descendante du graphe des occurences d'appels. Le calcul intraprocédural utilise des assertions initiales, associées au noeud racine, issues des assertions associées au noeud contenant l'appel à l'occurence de procédure concernée.

Remarquons que ceci est tout à fait compatible avec l'algorithme général itératif présenté au §VII.1.1.4. Il suffit de remplacer l'affectation:

```
A[1] = 0;
```

par

```
A[1] = t_i; /* t_i représente les assertions initiales */
```

Le calcul des assertions initiales est l'objet du §VII.5.

VII.2. Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis; analyse sémantique des programmes

VII.2.1. Méthode générale

P. Cousot présente dans [Cous 78] une méthode générale d'analyse sémantique de programmes. Il est impossible ici de décrire cette méthode de façon détaillée, vu sa complexité. Nous souhaitons cependant la présenter car elle permet de calculer différentes propriétés sémantiques, et notamment des relations linéaires entre les variables d'un programme.

Pour une classe de propriétés sémantiques recherchées, la définition d'un algorithme d'analyse approchée des programmes se décompose en 4 étapes:

E1 Choisir le sous-espace des propriétés approchées que l'on souhaite calculer. Bien évidemment ce sous-espace doit vérifier certaines propriétés exposées dans [Cous 78].

E2 A partir de ce sous-espace et des définitions des règles de construction des systèmes d'équations sémantiques en avant ou en arrière [Cous 78], établir, à la main, les règles de construction des systèmes d'équations approchées.

E3 Ecrire l'algorithme permettant d'associer à tout programme les systèmes d'équations approchées (en avant et/ou en arrière).

E4 Ecrire l'algorithme de résolution de ces systèmes, en utilisant éventuellement les techniques d'accélération de la convergence si celle-ci n'est pas garantie en un nombre fini de pas.

VII.2.2. Quelques exemples de propriétés sémantiques

Toujours dans [Cous 78], P. Cousot explicite les étapes E1 et E2 pour différents problèmes. Ainsi, pour un programme quelconque, il est possible avec cette méthode d'associer à chaque noeud du programme:
(1) le signe des variables numériques;
(2) la parité des variables entières;
(3) les variables vivantes;
(4) les expressions disponibles;
(5) les variables de valeur constante (propagation des constantes);
(6) le type des variables (programme écrit dans un langage sans déclaration).
(7) les pointeurs nuls et non nuls;
(8) les pointeurs repérant des enregistrements distincts.

Deux autres classes de propriétés sémantiques nous intéressent tout particulièrement:
- la recherche d'un intervalle de valeurs pour les variables numériques d'un programme;
  la recherche de relations linéaires entre les variables d'un programme.

Ces deux classes de propriétés ont fait l'objet d'une implémentation sur un sous-ensemble de PASCAL. L'implémentation de la recherche d'intervalles de valeurs a été effectuée par J.P. Jung [Jung 83], celle de la recherche de relations linéaires a été effectuée par N. Halbwacks [Halb 79].

VII.2.3. Application de cette méthode à Fortran-77

P. Cousot, N. Halbwachs et J.P. Jung justifient en partie leurs travaux en montrant l'intérêt qu'ils présentent pour prouver les accès aux tableaux d'un programme et supprimer ainsi une partie des tests dynamiques de contrôle des débordements de tableaux. Cette préoccupation n'est pas vraiment éloignée de la notre qui est de rechercher les parties de tableaux manipulées par chaque opérateur du programme. Ceci explique pourquoi les types d'assertions ainsi calculés nous conviennent tout particulièrement.

Bien que la méthode de P. Cousot soit beaucoup plus puissante que la méthode de propagation des constantes, la technique que nous utilisons pour adapter cette dernière à un programme comportant des appels de procédure convient parfaitement à la méthode de P. Cousot. C'est pourquoi nous ne nous étendons pas plus sur cette méthode.

Voici un exemple extrait de [Halb 79]. Les assertions trouvées automatiquement ont été reportées manuellement entre accolades.

```
spécification d'entrée: 1≤F≤N
D := N ; G := 1 ;   {1≤F≤N, G=1, D=N}
tantque G<D faire   {1≤G≤F≤D≤N, G+1≤D}
   R:= B⟦F⟧;I := G ; J := D ;
   tantque I≤J faire   {1≤G≤F≤D≤N, G≤I≤J≤D, G+1≤D, G+1≤I+J}
      tantque B⟦I⟧<R faire
         I := I+1 ;  {1≤G≤F≤D≤N, G+1≤I≤N+1, J≤D, G+1≤D}
      fait ;   {1≤G≤F≤D≤N, G+1≤D, G≤I, J≤D, I≤N}
      tantque R<B[J]  faire
         J := J-1 ;   {1≤G≤F≤D≤N, G+1≤D, G≤I, 0≤J≤D-1}
      fait ;
      si I≤J alors  {1≤G≤F≤D≤N, G≤I≤J≤D, G+1≤D}
         W := B⟦I⟧  ;  B⟦I⟧  := B⟦J⟧  ;  B⟦J⟧ := W ;
         I := I+1 ; J := J-1 ;
      finsi ;
   fait ;
   si F≤J alors D := J   {1≤G≤F≤J = D≤N, G+1≤N, G+1≤I+J, J+1≤I}
   sinon
   si I≤F alors G := I   {1≤J+1≤I = G≤F≤D≤N, 2≤D, 2≤I+J}
   sinon stop
   finsi ;
fait ;
```

VII.3. Recherche d'assertions par l'analyse des propriétés des boucles DO

Les méthodes développées à partir de la théorie de P. Cousot sont très puissantes, mais en contrepartie très longues à implémenter. Chaque implémentation qui en a été faite, sur un sous-ensemble de Pascal, a demandé environ 2 ans de travail.

C'est pourquoi nous proposons la méthode suivante, qui tire partie des propriétés des boucles-DO pour générer des assertions. Les boucles-DO sont beaucoup utilisées par les programmeurs Fortran, sans doute parce que c'est la seule structure de contrôle itérative de ce langage. Ainsi une boucle-DO remplace assez souvent une boucle while:

```
      DO 10 I = 1,N
              ...
              IF (T(I) ...) GOTO 20
              ...
10    CONTINUE
20    ...
```

VII.3.1. Caractérisation des "bonnes-boucles"

La grande majorité des boucles-DO présentent les propriétés suivantes.

C1: La variable de la boucle, nommée variable-DO, est de type INTEGER.

C2: La variable-DO n'est pas modifiée par le corps de la boucle; à noter que le contraire est interdit par la norme [AFNO 83] §11.10.5.

C3: Le contrôle n'est donné au corps de la boucle que par l'intermédiaire de l'instruction-DO; cela signifie qu'aucun branchement n'est effectué depuis l'extérieur du corps de boucle vers celui-ci. A noter que le contraire est interdit par la norme [AFNO 83] §11.10.8, même par l'intermédiaire d'un GOTO assigné.

C4: Les expressions bornes de l'instruction-DO sont des combinaisons linéaires de variables entières, non modifiées par les instructions du corps de boucle.

C5: Le signe de l'expression incrément est connu.

Pour de telles boucles, conformément au processus d'exécution des boucles-DO décrit aux §11.10.3 & §11.10.7 de [AFNO 83], la variable-DO reste comprise entre les deux bornes de la boucle-DO. Nous pouvons donc associer à chaque noeud du corps de la boucle-DO les deux assertions linéaires correspondantes. De telles boucles seront nommées dans la suite "bonnes boucles-DO".

Exemple VII-2:

```
        DO 10 I = 1,N
                DO 20 J = I+1,N
                        ...             (a)
20              CONTINUE
10      CONTINUE
```

Au noeud (a) nous asssocions les assertions:

$$\{-I \leq -1,\ I-N \leq 0,\ I+1-J \leq 0,\ J-N \leq 0\}$$

à la condition que (a) ne modifie pas N.

VII.3.2. Recherche des "bonnes-boucles"

Dans ce paragraphe, nous montrons informellement comment vérifier les conditions C1 à C5. Nous formulons deux remarques préliminaires.

Remarque VII-2: Le corps d'une boucle-DO se compose des instructions se trouvant entre l'instruction-DO et l'instruction terminale de la boucle DO, repérable par son étiquette. Sa construction, par la phase ASSIA, ne pose donc aucun problème.

Remarque VII-3: Nous supposons disposer d'une fonction d'assertion $fi \in FI(ri)$ calculée, par exemple, par la méthode de propagation des constantes. Nous remarquons qu'il est toujours possible d'utiliser la fonction d'assertion nulle suivante: finulle: $N(ri) \rightarrow I(ri)^*$ définie par $\forall n \in N(ri)$ finulle(n) = $\emptyset$.

La vérification de C1 correspond à la recherche d'un renseignement inclus dans ri.

La vérification de C2 n'est pas obligatoire. Cependant, la même vérification doit être effectuée sur les variables entrant dans la composition des expressions bornes. Se reporter à la vérification de C4.

La vérification de la condition C3 fait appel à des techniques d'analyse syntaxique. Nous la reportons donc sur la phase ASSIA. A noter que cette vérification n'est pas obligatoire.

La vérification de C4 est plus complexe; ri fournit les deux expressions bornes de la boucle: xbi, xbs $\in X(ri)$. Ce sont des combinaisons linéaires si

$$isxcl(ri)(xbi) = isxcl(ri)(xbs) = VRAI.$$

Dans ce cas, l'ensemble V des variables entrant dans ces expressions est:

$$V=\{v \in V_i(ri) \text{ tq } Xtocl(ri)(xbi)(v) \neq 0 \text{ ou } Xtocl(ri)(xbs)(v) \neq 0\}$$

Nous ajoutons à V la variable-DO (vérification de C2).

Soit n ϵ N(ri) un noeud correspondant à une instruction du corps de boucle. Il faut vérifier que l'exécution de n ne modifie aucune des variables de V. C'est le même problème que celui rencontré au §VII.1.2.2.2, avec comme contexte local les assertions fournies par $f_i(ri)$, f_i étant la fonction d'assertion dont nous avons supposé l'existence dans la remarque VII-2.

La vérification de la condition C5 est immédiate si cette expression notée xinc ϵ X(ri) est constante. Sinon nous essayons de l'évaluer grâce aux assertions fournies par f_i et à la fonction X_ifocmp0, que nous avons présentée au §IV.3.6 et qui sera explicitée dans le chapitre VIII.

A chaque instruction-DO sont associés trois noeuds (cf. §IV.3.3.5). C'est le noeud DO-INIT qui évalue les expressions; c'est donc avec les assertions qui lui sont associées qu'il faut tenter l'évaluation.

VII.3.3. Avantages de cette solution

Le principal avantage de cette méthode est sa simplicité. Notons que la plupart des traitements à effectuer pour sa réalisation sont de toutes façons effectués par tout paralléliseur: vérification de C2, C3, recherche du corps d'une boucle, etc...

Les assertions qu'elle fournit sont à notre avis très intéressantes. Dans les exemples que nous donnons, les assertions sont généralement obtenues, manuellement, en suivant cette méthode.

VII.4. Commentaires sur l'introduction d'assertions dans le texte source d'un programme

Le principe est simple: la syntaxe du langage étudié est augmentée de façon à pouvoir introduire un ensemble d'assertions dans le texte du programme. Ces assertions peuvent être introduites par le programmeur c'est un moyen de donner des renseignements au paralléliseur. Elles peuvent être introduites par le paralléliseur, lors de la régénération du source du programme, pour montrer au programmeur les assertions qui ont été calculées automatiquement; ceci peut en effet aider à la mise au point du programme.

La syntaxe de Fortran-77 permet facilement d'ajouter des assertions. Nous avons modifié l'analyseur lexical Fortran-77 d'UNIX en utilisant les caractères {& }, respectant ainsi la tradition.

Exemple VII-3:

```
      READ (S,10) , PAS
             { PAS.GT.0 }
      DO 20 I = 1,N,PAS
             ...
20    CONTINUE

      DO 30 I = 1,N,PAS
             ...
30    CONTINUE
```

Le programmeur peut introduire n'importe quel type d'assertion:

{I > J si K > 0, I+J premier, K div 13 = 0, ...}

ce qui ne sert à rien si nous ne savons pas les utiliser. Nous n'avons exploré que le domaine des assertions linéaires (cf. chapitre VIII).

Ces assertions peuvent être fausses (erreur de frappe, de logique, ...). Nous pouvons cependant vérifier qu'elles n'entraînent pas d'incohérences entre-elles et avec celles calculées automatiquement. Nous présentons au chapitre VIII une méthode permettant de prouver qu'un ensemble d'assertions est cohérent [Shos 81].

Ces assertions doivent être vraies pour toutes les occurences de la procédure.

Ces assertions doivent être propagées sur l'ensemble de la procédure. Il est difficilement concevable d'imposer au progammeur de les propager lui-même. Ainsi le test de "bonne-boucle" pour la 2ème boucle n'est réalisable que si l'assertion concernant PAS a été propagée. La nécessité d'implémenter les techniques de P. Cousot demeure.

Cette méthode permet cependant d'obtenir facilement des assertions. Ceci peut être intéressant dans le cadre d'une réalisation visant à valider l'utilisation d'assertions.

VII.5. Recherche des assertions initiales

Nous venons de voir quatre méthodes permettant de calculer une fonction d'assertion pour une ou plusieurs occurences de la même procédure. Pour les deux premières méthodes, qui propagent sur les successeurs d'un noeud les assertions calculées en ce noeud, les assertions initiales sont importantes. Par assertions initiales, nous entendons celles qui sont associées au noeud racine et donc vraies avant que l'exécution de l'occurence de procédure concernée démarre. Nous allons examiner différents moyens d'associer à une occurence de procédure un ensemble d'assertions initiales.

VII.5.1. Assertions initiales issues des DATAs

Les entités qui sont l'objet d'une déclarative DATA sont initialement définies. Cela signifie qu'elles ont la valeur spécifiée par cette déclarative avant que le programme principal ne démarre (cf. [AFNO 83] §17.2). Il résulte de cette définition que les initialisations par DATA des entités locales d'une procédure ne concernent que la première exécution dynamique de cette procédure.

Nous ne cherchons pas les procédures qui ne sont exécutées qu'une fois. Nous proposons donc de ne transformer les initialisations par DATA en assertions initiales que pour le programme principal. Ces assertions sont très importantes, nous en avons donné les

raisons au §VI.3.2.1.4).

VII.5.2. Assertions initiales issues d'un contexte d'appel

Nous reprenons les notations du §VI.1.3. A l'exécution de l'appel d'externe c_p, les entités de l'occurence de procédure appelante (ω_p) sont dans un certain contexte, qui est transmis sur les entités de l'occurence de procédure appelée (ω_q) par le mécanisme d'association des commons et par le mécanisme de passage de paramètres. Si nous disposons d'une fonction fi_p pour ω_p, une partie de ce contexte est donné par les assertions $i_p^* = fi_p(n_p)$ où $n_p \in N(ri_p)$ est le noeud contenant l'appel c_p.

Nous allons montrer comment transmettre une partie de ce contexte sur les variables de $V_i(ri_q)$.

VII.5.2.1. Association des variables communes

Nous rappelons que la norme [AFNO 83] indique qu'à l'exécution d'un appel d'externe, les entités communes de la procédure appelée sont définies avec la valeur des entités communes de la procédure appelante qui leur sont associées.

Soit $vc_p \in V_i(ri_p)$ une variable commune. Nous recherchons $vc_q \in V_i(ri_q)$ telle que $adr(ri_p)(vc_p) = adr(ri_q)(vc_q)$. Nous rappelons que vc_q n'existe pas nécessairement à cause du problème des représentations différentes d'un même common. D'autre part, vc_q n'est pas nécessairement unique à cause des équivalences sur les variables de ri_q. Ceci ne pose pas de problème grâce au prétraitement des assertions. Après le début de l'exécution de n_p et avant l'exécution du noeud racine de ri_q nous avons l'assertion $vc_p = vc_q$. Cette assertion met en jeu des variables de $V_i(ri_p)$ et de $V_i(ri_q)$. D'où:

$V_i = V_i(ri_p) \cup V_i(ri_q)$,

$\Gamma(V_i) = V_i \to Z$ ensemble des combinaisons linéaires des variables de V_i,

$I = I(V_i) = \Gamma(V_i) \times Z$ ensemble des assertions linéaires sur les variables de V_i.

Nous formons $i_1^* \in I^*$ en créant pour chaque couple (vc_p, vc_q) les deux assertions "$vc_p - vc_q \leq 0$" et "$vc_q - vc_p \leq 0$".

Nous avons besoin pour cela de la fonction:

$$catcl(ri_p, ri_q) : \Gamma(ri_p) \times \Gamma(ri_q) \to \Gamma(V_i)$$
$$catcl(ri_p, ri_q)(\lambda_p, \lambda_q)(v) = \text{si } v \in V_i(ri_p) \text{ alors } \lambda_p(v) \text{ sinon } \lambda_q(v)$$

Les deux assertions associées au couple (vc_p, vc_q) sont:

$$(catcl(ri_p, ri_q)(Vtocl(ri_p)(vc_p), negcl(ri_q)(Vtocl(ri_q)(vc_q))), 0) \text{ et}$$
$$(catcl(ri_p, ri_q)(negcl(ri_p)(Vtocl(ri_p)(vc_p)), Vtocl(ri_q)(vc_q)), 0)$$

VII.5.2.2. Association paramètre formel/paramètre réel

Nous nous intéressons aux associations telles que le paramètre formel est une variable entière, notée $pf_q \in V_i(ri_q)$, et telles que le paramètre réel, expression de $X(ri_p)$ notée pr_p, est une combinaison linéaire des variables de $V_i(ri_p)$.

Conformément à la norme, un paramètre formel est défini avec la valeur de son paramètre réel associé. Cela signifie, qu'après le début de l'exécution de n_p et avant

l'exécution du noeud racine de ri_q, nous avons l'assertion $pf_q = pr_p$.

Nous formons $i_2^* \in I_*$ en créant pour chacune de ces associations, les deux assertions "$pf_q - pr_p \leq 0$" et "$pr_p - pf_q \leq 0$". Ce qui se traduit par

$$(catcl(ri_p,ri_q)(negcl(ri_p)(XLP),XLQ),\ -Xtotc(ri_p)(pr_p)) \text{ et}$$
$$(catcl(ri_p,ri_q)(XLP,negcl(ri_q)(XLQ)),\ Xtotc(ri_p)(pr_p))$$

avec $XLQ = Vtocl(ri_q)(pf_q)$ et $XLP = Xtocl(ri_p)(pr_p)$

VII.5.2.3. Contexte d'appel

Les assertions produites dans les deux paragraphes précédents traduisent les échanges de valeurs au moment de l'appel. Nous leur ajoutons les assertions contenues dans $i^* = fi_p(n_p)$ qui sont les assertions de $I^*(ri_p)$ représentant le contexte de l'appel, après les avoir traduites en des assertions de I^* de la façon suivante.

Soit $i \in i^*$, $i = (\lambda,\mu)$ avec $\lambda \in \Gamma(ri_p)$ et $\mu \in Z$.

Soit $\lambda_q^o \in \Gamma(ri_p)$ telle que $\forall v_q \in V_i(ri_q)$ $\quad \lambda_q^o(v_q) = 0$.

Nous formons $i_3^* \in I^*$ de la façon suivante:

$$i_3^* = \bigcup_{i \in i^*} catcl(ri_p,ri_q)(\lambda,\lambda_q^o)$$

VII.5.2.4. Création des assertions initiales

Nous formons alors $I_4^* = i_1^* \cup i_2^* \cup i_3^*$.

i_4^* contient les assertions sur les variables de $V_i(ri_p)$, qui représentent l'approximation du contexte d'appel que nous connaissons. Pour répercuter ce contexte sur les variables de $V_i(ri_q)$, il suffit d'éliminer dans i_4^* les variables de $V_i(ri_p)$. Nous utilisons pour cela l'algorithme présenté au §VIII.1.

Nous obtenons ensuite les assertions initiales de ω_q en retraduisant chaque assertion de i_4^* en une assertion de $I(ri_q)^*$ par la fonction:

$$extractclq(ri_p,ri_q): \Gamma(V_i) \rightarrow \Gamma(ri_q)$$
$$extractclq(ri_p,ri_q)(\lambda)(v_q) = \lambda(v_q) \quad \forall v_q \in V_i(ri_q)$$

cette méthode est très puissante, comme en témoigne l'exemple suivant:

Exemple VII-4: Soit Q la subroutine suivante:

```
SUBROUTINE Q(IX,IY)
COMMON /G/ IZ
...
END

PROGRAM P
COMMON /G/ J
...
{I = 0, J > I}
CALL Q(I*J+1,I+1)
...
END
```

Associations de common valides: (J,IZ)

$\Rightarrow i_1^* = \{J = IZ\}$

Association de paramètres valides: (I+1,IY)

$\Rightarrow i_2^* = \{IY = I+1\}$

traduction du contexte

$\Rightarrow i_3^* = \{I = 0, J > I\}$

$\Rightarrow i_4^* = \{I = 0, J > I, J = IZ, IY = I+1\}$

élimination de I

$\Rightarrow i_4^* = \{J > 0, IY = 1, J > IY-1, J = IZ\}$

élimination de J (assertions initiales de Q)

$\Rightarrow i_4^* = \{IZ > 0, IZ > IY-1, IY = 1\}$

Remarque VII-4: Nous remarquons dans l'exemple précédent la perte de l'information {IX = I*J+1} qui conduit à {IX = 0}. Nous évitons ce genre de situation en recherchant les couples (pf_q, pr_p) où pr_p n'est pas une combinaison linéaire des variables de $V_i(ri_p)$ mais où pr_p est évaluable par la fonction X_ifovali, appelée avec le contexte donné par $fi_p(n_p)$.

VII.6. Conclusion

Nous avons vu dans ce chapitre comment calculer une fonction d'assertion pour une occurence de procédure $\omega \in \Omega$. Nous avons détaillé deux méthodes:
- la méthode de propagation des constantes,
- la méthode des 'bonnes boucles-DO' (méthode originale),

Nous en avons présenté deux autres:
- la méthode de P. Cousot,
- l'introduction d'assertions dans le texte source.

La fonction d'assertion peut-être obtenue par l'application successive de plusieurs de ces méthodes. Les éléments nécessaires à cette recherche sont:
- une fonction d'aliasing fa,

- une fonction de manipulation fm,
- un ensemble i^* d'assertions initiales associées au noeud racine.

Nous avons vu ensuite comment calculer un ensemble d'assertions initiales pour une occurence de procédure ω_q, à partir des assertions associées au noeud contenant l'appel correspondant dans l'occurence de procédure appelante ω_p. Les éléments nécessaires à cette recherche sont:

- un ensemble d'assertions "contexte d'appel" $i_p^* \in I(ri_p)^*$;
- une fonction d'aliasing fa_p;
- une fonction d'aliasing fa_q.

Les algorithmes d'ordonnancement, permettant de calculer des fonctions d'assertions pour toutes les occurences de procédure d'un programme, sont l'objet du chapitre X.

C H A P I T R E VIII

VIII. Utilisation des assertions

Les propositions d'améliorations des méthodes existantes de parallélisation que nous formulons dans cette thèse reposent en grande partie sur notre capacité à savoir utiliser les assertions que nous calculons. Nous avons par exemple supposé savoir effectuer les traitements suivants:

- évaluation d'une expression booléenne,
- évaluation d'une expression numérique,
- élimination d'une variable dans un système d'assertions.

D'autres traitements seraient utiles, dans le cadre d'une réalisation:

- élimination de la redondance dans un système d'assertions,
- réalisation de l'union de deux régions.

Le but de ce chapitre est de montrer comment réaliser ces divers traitements. Nous présentons un ensemble de méthodes, plus ou moins connues, de manipulation d'assertions, puis nous montrons comment les utiliser pour notre cas particulier. Bien que contenant peu d'idées originales, ce chapitre nous parait important dans la mesure où il prouve que nos hypothèses sur l'utilisation des assertions sont justifiées. Il s'organise de la façon suivante.

(1) Nous décrivons la phase de prétraitement des assertions, qui a le double rôle de normaliser les assertions associées à un noeud et de résoudre le problème posé par les associations de variables.

(2) Nous nous intéressons au test de faisabilité d'un système d'assertions linéaires. Nous présentons la méthode de R. Shostak.

(3) Nous montrons comment réaliser l'élimination d'une variable dans un système d'assertions linéaires.

(4) Nous présentons une deuxième méthode de test de faisabilité d'un système d'assertions linéaires, par éliminations successives de variables.

(5) Nous présentons la méthode de Smith de résolution d'un système d'équations linéaires en nombres entiers.

(6) Nous montrons comment appliquer les résultats précédents pour évaluer certaines expressions, dans un contexte donné.

(7) Nous montrons comment appliquer les résultats précédents au calcul du graphe des dépendances, dans un contexte donné.

VIII.1. Prétraitement des assertions

Dans toute la suite de ce chapitre, nous allons manipuler des systèmes d'inéquations (resp. équations) linéaires de la forme:

$$A.V \leq B \quad \text{(resp. } A.V = B\text{) avec}$$

$A = (a_{ij})$ $i \in [1,m]$, $j \in [1,n]$ est une m.n-matrice à coefficients dans Z,

$B = (b_i)$ $i \in [1,m]$ est un m-vecteur à coefficients dans Z,

$V = (v_i)$ $i \in [1,n]$ est un n-vecteur de variables prenant des valeurs quelconques dans Z.

L'ensemble des assertions associées à un noeud d'une occurence de procédure par les méthodes décrites dans le chapitre précédent ne constituent pas nécessairement un tel système, et ce pour deux raisons.

(1) Les différentes assertions peuvent ne pas avoir le même format à cause de leurs origines différentes: couple variable/valeur, couple variable/intervalle, format quelconque (assertions données par le programmeur) ou assertions linéaires.

(2) Les variables sur lesquelles ces assertions sont construites sont les variables entières d'une procédure Fortran. Certaines peuvent être en association par aliasing ou par équivalence, ce qui implique que les valeurs respectives qu'elles peuvent prendre ne sont pas quelconques.

VIII.1.1. Standardisation des différents types d'assertions

Nous proposons d'exprimer toutes les assertions à l'aide d'inéquations de la forme: $a_1v_1 + a_2v_2 + \ldots + a_nv_n \leq b$. D'une part, c'est la forme d'assertion la plus générale que nous savons actuellement calculer automatiquement, par la méthode de P. Cousot et par la méthode des bonnes boucles-DO. D'autre part, un système d'inéquations linéaires, de la forme $A.V \leq B$, définit un polyèdre convexe de Z^n; la théorie des polyèdres convexes de R^n est un domaine bien connu, auquel nous pouvons emprunter différentes méthodes.

Cette standardisation se fait de façon classique. Une équation est transformée en une double inéquation, quand aux inéquations strictes elles sont transformées en retranchant 1 au second membre, puisque nous travaillons en nombres entiers.

Remarque VIII-1: Lorsque le prétraitement est effectué en vue de tester la faisabilité du système (cf.§VIII.2 & §VIII.4), il est plus efficace d'utiliser chaque équation pour éliminer une variable du système, puis de l'abandonner.

Les coefficients de chaque inéquation sont ensuite normalisés de la façon suivante: chaque coefficient a_i est remplacé par son quotient par d, pgcd des valeurs absolues des coefficients a_i (i=1..n), quand à b il est remplacé par le plus grand entier inférieur au quotient de b par d. Cette normalisation est importante dans la mesure où elle permet de détecter des impossibilités dans les systèmes d'assertions.

Exemple VIII-1:

$$\{3.I = 4\} \rightarrow \{ 3.I \leq 4,\ -3.I \leq -4\}$$
$$\rightarrow \{ I \leq 1,\ -I \leq -2\} \rightarrow \text{système infaisable}$$

VIII.1.2. Prise en compte des associations de variables

Nous proposons de tirer profit de certaines associations de variables pendant l'utilisation des assertions. L'association par équivalence de deux variables (entités simples) entières est nécessairement totale (égalité des suites de mémoires associées). Nous verrons d'autre part dans le chapitre IX que certaines associations de variables entières par aliasing sont totales.

Nous rappelons que les associations de variables ne sont pas prises en compte, durant le calcul d'assertions, par la phase de création d'assertions. Ce choix est discuté au §VII.1.2.2.1. Cependant, les valeurs de deux variables en association totale sont

nécessairement égales puisqu'elles correspondent aux mêmes mémoires. Nous proposons donc de traduire cette égalité des valeurs en ajoutant au système d'assertions le couple d'inéquations linéaires suivant:

$$\{v_i - v_k \leq 0 \ , \ v_k - v_i \leq 0\}$$

pour tout couple (v_i, v_k) de variables de V tel que v_i et v_k sont en association totale.

VIII.1.3. Conclusion sur le prétraitement des assertions

Les arguments de cette phase sont:

- une ou deux représentations internes (i),
- une ou deux fonctions d'aliasing.

Son résultat est un système d'inéquations linéaires de la forme $A.V \leq B$.

VIII.2. Etude de la faisabilité d'un système d'inéquations linéaires par la méthode de R. Shostak

Un système d'inéquations linéaires est faisable si les variables de V peuvent prendre des valeurs telles que toutes les inéquations sont satisfaites. Dans le cas contraire, il est infaisable. Nous verrons dans la suite de ce chapitre que le test de la faisabilité d'un système nous permet d'évaluer certaines expressions booléennes, mais aussi de calculer le graphe des dépendances d'une occurence de procédure.

La première méthode que nous exposons est celle que R. Shostak propose dans [Shos 81]. Elle est particulièrement adaptée aux systèmes dont les inéquations ne manipulent que peu de variables. Elles nous intéresse donc, dans la mesure où les expressions d'une procédure ne font que rarement intervenir un grand nombre de variables. R. Shostak propose une solution dans le cas où chaque inéquation manipule au plus deux variables, et une extension de cette solution aux systèmes quelconques, que nous présentons brièvement. Nous supposons dans la suite que chaque variable apparait dans au moins une inéquation.

VIII.2.1. Définitions

VIII.2.1.1. Variables primaires

Les variables de V sont ordonnées de façon arbitraire: $v_1 < v_2 < \ldots < v_n$. Une variable de V est primaire si elle est une des deux variables de plus bas rang dans chaque équation où elle apparait. Ceci se traduit par:

$\forall i \in [1,n]$, v_i est une variable primaire (VP) si et seulement si $\forall k \in [1,m]$ tel que $a_{ki} \neq 0$, il existe au plus 1 a_{kj} non nul tq $j \in [1,i-1]$.

Soit p le nombre de VP du système. Nous réordonnons les variables de façon que $v_1, \ldots, v_p$ soient les VP (ii).

Exemple VIII-2: Soient I, J, K, L ($I < J < K < L$) les 4 variables du système suivant:

(i) Le calcul d'assertion initiales implique la création et l'utilisation d'assertions construites sur l'union des variables entières de 2 occurences de procédure.
(ii) A noter que v_1 et v_2 sont nécessairement primaires.

$$\begin{aligned} I + 2J - K &\leq 0 \\ I + 4L &\leq 18 \\ K + L &\leq 0 \end{aligned}$$

Les variables primaires sont I, J et L.

VIII.2.1.2. Variable spéciale v_o

Afin que chaque inéquation manipule exactement deux VP, il est nécessaire d'en introduire une spéciale, notée v_o, qui apparait dans les inéquations à 1 VP avec un coefficient nul.

VIII.2.1.3. Graphe associé au système d'inéquations

Nous associons au système d'inéquations le graphe G = (N,E) avec:

$$N = \{n_o, n_1, \ldots, n_p\},$$
$$E = \{e_1, e_2, \ldots, e_m\},$$

construits de la façon suivante:

à chaque VP v_i, $i \in [0,p]$ nous associons le noeud n_i,

à chaque inéquation d'indice k, nous associons l'arête e_k qui relie les deux noeuds associés aux deux VP qu'elle manipule.

VIII.2.1.4. Liste de i-quadruplets associée à une chaine de G

Un i-quadruplet est un quadruplet de la forme $q = \alpha, \beta, \gamma, \delta)$ avec $\alpha \in Z$, $\beta \in Z$, $\gamma \in Z^{n-p}$ et $\delta \in Z$.

Soit $n_{k_1} e_{j_1} n_{k_2} e_{j_2} \cdots e_{j_c} n_{k_{c+1}}$ une chaine de G; nous lui associons la liste de i-quadruplets $q_1, q_2, \ldots, q_n$ avec $q_i = (\alpha_i, \beta_i, \gamma_i, \delta_i)$ où:

$$\alpha_i = a_{j_i k_i}, \quad \beta_i = a_{j_i k_{i+1}}, \quad \delta_i = b_{j_i} \text{ et}$$

$$\gamma_i = (a_{j_i p+1}, a_{j_i p+2}, \ldots, a_{j_i n}),$$

(nous posons $a_{io} = 0$, $\forall i \in [1,m]$)

VIII.2.1.5. Cycle élémentaire admissible

Une chaine est dite admissible si $\forall i \in [1,c-1]$, le i-quadruplet q_i associé à l'arête e_{j_i} vérifie : α_{i+1} et β_i sont de signes opposés. D'autre part, elle est élémentaire si ses noeuds sont 2 à 2 distincts. Un cycle élémentaire admissible (CEA) est donc une chaine élémentaire admissible telle que $n_{k_1} = n_{k_{c+1}}$.

VIII.2.1.6. Résidu d'un cycle élémentaire admissible

Le résidu d'un CEA de longueur c+1 noeuds, identifié par sa liste de i-quadruplets associée $q_1, q_2, \ldots, q_c$ est le i-quadruplet $q_r = q_1 * q_2 * \ldots * q_c$ avec l'opération * définie par:

$q_i * q_{i+1} = (\alpha, \beta, \gamma, \delta)$ où

$$\alpha = \epsilon\alpha_i\alpha_{i+1},\ \beta = -\epsilon\beta_i\beta_{i+1},\ \delta = \epsilon(\delta_i\alpha_{i+1} - \delta_{i+1}\beta_i),$$

$$\gamma = (\epsilon(a_{i,k}\alpha_{i+1} - a_{i+1,k}\beta_i))_{k=p+1..n} \text{ et } \epsilon = \text{signe}(\alpha_{i+1}).$$

Remarque VIII-2: L'opération * consiste à éliminer entre les deux équations i et i+1 la VP qu'elles ont en commun. Ainsi

$$\{x-2y \leq 5\} * \{4y-3z \leq 0\} = \{2x-3z \leq 10\}.$$

VIII.2.1.7. Cycle élémentaire admissible irréalisable

Un CEA est irréalisable si son résidu q_r vérifie les deux conditions suivantes:

$$\alpha_r + \beta_r = 0 \quad \text{et} \quad \gamma_r.(v_i)_{i=p+1..n} > \delta_r$$

ceci arrive notamment si $\delta_r < 0$ et ($\gamma_r = 0(Z^{n-p})$ ou $p = n$). Nous dirons alors que le CEA est immédiatement irréalisable.

VIII.2.1.8. Fermeture du système, fermeture du graphe

Soit s le nombre de CEAs du graphe G tels que leur résidu vérifie $\alpha_r+\beta_r \neq 0$. Soient $n_{f_1}, n_{f_2}, \ldots, n_{f_s}$ les noeuds initiaux de ces CEAs. La fermeture du système initial est formée en lui ajoutant les s inéquations suivantes:

$$(\alpha_{f_k}+\beta_{f_k})v_{f_k} + \gamma_{f_k}.(v_i)_{i=p+1,n} \leq \delta_{f_k} \qquad \forall k \in [1,s]$$

où $(\alpha_{f_k}, \beta_{f_k}, \gamma_{f_k}, \delta_{f_k})$ est le résidu associé au CEA de noeud initial n_{f_k}.

La fermeture du graphe G est obtenue en ajoutant une arête pour chacune de ces inéquations, entre n_{f_k} et n_o (associé à v_o).

VIII.2.2. Théorème de faisabilité du système

R. Shostak prouve dans [Shos 81] le théorème suivant:

la fermeture du graphe G contient un CEA irréalisable <=> le système initial est infaisable.

VIII.2.3. Algorithme testant la faisabilité d'un système

La méthode proposée par R. shostak consiste à étudier une suite de systèmes d'inéquations $S_1, S_2, \ldots$ (S_1 est le système initial) de la façon suivante.

A l'étape i, nous recherchons si le le système S_i contient un CEA immédiatement irréalisable. Si oui, les systèmes $S_1, \ldots, S_i$ sont infaisables.

Si non, le système S_{i+1} est formé en traduisant le fait que tous les résidus du système S_i doivent être réalisables. Ce nouveau système ne s'exprime qu'en fonction des variables non

primaires de S_i. L'algorithme s'arrête quand S_i ne comporte que des variables primaires, ce qui arrive nécessairement puisque le nombre de variables diminue à chaque étape. D'où:

E1: Rechercher les VP de S_i; réordonner les variables.

E2: Construire le graphe G associé à S_i; rechercher les CEAs de G et calculer les résidus.

E3: Si un CEA de G au moins est immédiatement irréalisable, S_1, ..., S_i sont infaisables (arrêt de l'algorithme).

E4: Construire G', fermeture de G associée à la fermeture de S_i; rechercher les CEAs et calculer les résidus.

E5: Si un CEA de G' au moins est immédiatement irréalisable, S_1, ..., S_i sont infaisables (arrêt de l'algorithme).

E6: Si S_i n'a pas de variable non primaire, S_1, ..., S_i sont faisables (arrêt de l'algorithme).

E7: Former le système S_{i+1}; il contient une inéquation par CEA de G' tel que son résidu $q_r = (\alpha_r, \beta_r, \gamma_r, \delta_r)$ vérifie $\alpha_r + \beta_r = 0$; cette inéquation est:

$$\gamma_r.(v_i)_{i=p+1..n} \leq \delta_r.$$

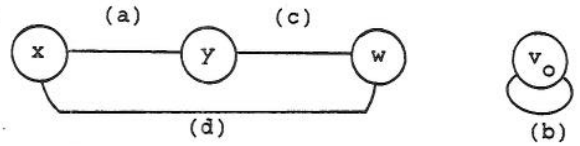
Exemple VIII-3: Soit à prouver le système S_o suivant:

$$\begin{array}{rrrrrrrl} x & - & 2y & & + z & \leq & c & (a) \\ & & & & - z + 5t & \leq & -1 & (b) \\ & & y & - w & - 2t & \leq & 0 & (c) \\ -x & & & + 2w & - t & \leq & 7 & (d) \end{array}$$

Les variables sont ordonnées de la façon suivante: $x < y < w < z < t$. Voici les résultats des différentes phases de l'algorithme:

E1: VP : x, y, w;

E2:



2 CEAs : ((a),(c),(d)) de résidu (1,-1,(1,-5),c+7),
((b)) " " (0, 0,(-1,5), -1);

E3: pas de CEA immédiatement irréalisable;

E4: la fermeture du système est identique au système car aucun CEA vérifie $\alpha_r+\beta_r = 0$;

E5: idem E3;

E6: S_1 a des variables non primaires: z, t;

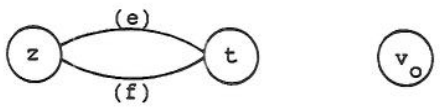
E7: le nouveau système S_2 est donc:

$z - 5t \leq c+7$ (e)

$-z + 5t \leq -1$ (f)

E1: VP: z, t;

E2:



1 CEA: ((e),(f)) de résidu (1,-1,(),c+6);

E3: si c < -6 l'unique CEA est irréalisable, le système est infaisable => arrêt de l'algorithme;

E3: si c ≥ -6 aucun CEA n'est immédiatement irréalisable; l'algorithme continue en E4;

E4: fermeture identique au sytème initial;

E5: idem E3;

E6: le système n'a pas de variable non primaires; S_1 et S_2 sont donc faisables; Arrêt de l'algorithme;

Remarque VIII-3: L'exemple précédent ne met pas en évidence la nécessité de construire la fermeture du système. R. Shostak propose dans son article un exemple tel que la fermeture du système contient un CEA immédiatement irréalisable, alors que le système initial n'en contient pas.

VIII.2.4. Remarques sur les performances de cette méthode

L'étude complète des performances de cette méthode n'est pas simple car elles dépendent de deux facteurs inconnus à priori: le nombre c de CEAs et le nombre s de systèmes à étudier. Plaçons nous dans le pire des cas: s est égal à n/2; c varie de façon

exponentielle en fonction du nombre e d'équations; un CEA a une longueur de l'ordre du nombre v de variables; le travail à effectuer pour chaque CEA est en O(n+v), pour la recherche de ce CEA dans le graphe (par l'algorithme de R. Tarjan [Tarj 73] par exemple), et en O(v) pour le calcul du résidu.

Nous voyons donc que dans ce cas, cette méthode a un temps d'exécution variant exponentiellement avec la taille du système. En pratique, nous ne pensons pas avoir de tels systèmes à prouver. Les relations entre variables d'un programme ont souvent la forme de chaînes transitives ($1 \leq I \leq J+1 \leq N$) et ne présentent que rarement des boucles. D'autre part chaque inéquation ne manipule que peu de variables, de l'ordre de deux ou trois. Dans de telles conditions la méthode de R. Shostak est particulièrement efficace puisque s diminue (s = 2 si aucune inéquation ne manipule plus de 3 variables) ainsi que c.

Il nous parait important de remarquer que cette méthode impose de savoir rechercher les cycles d'un graphe non-orienté, alors que les algorithmes usuellement proposés dans la littérature [Targ 73], [Mate 76], [John 75] permettent de rechercher les circuits d'un graphe orienté (K. Paton propose dans [Pato 69] un algoritme de recherche de cycles, mais il n'est pas très performant). La solution consistant à remplacer chaque arête par deux arcs de sens opposés présente les inconvénients d'introduire e circuits parasites (de longueur 2) et de générer deux circuits par cycle. D'autre part, ces algorithmes ne prennent généralement pas en compte les arêtes multiples. Ces trois problèmes, bien que facilement résolus, diminuent les performances de cette méthode.

VIII.2.5. Recherche des équations générées par un système d'inèquations

Théorème

Les inéquations associées à un CEA tel que son résidu $q_r = (\alpha_r, \beta_r, \gamma_r, \delta_r)$ vérifie $\alpha_r = \beta_r$ et $\delta_r = 0$ et $\gamma_r = 0(Z^{n-p})$ sont des équations.

démonstration

La démonstration formelle implique de longs calculs, aussi nous lui préférons une démonstration intuitive. L'opération * présentée au §VIII.2.1.6 consiste à former une équation en éliminant la variable commune à deux inéquations grâce à la transitivté de la relation d'ordre.

$$a_1u - b_1v \leq c_1 \text{ et } a_2v - b_2w \leq c_2 \Rightarrow$$
$$a_1a_2u \leq b_1a_2v + c_1a_2 \leq b_1b_2w + c_1a_2 + c_2b_1$$

Cette opération, appliquée sur la longueur d'un CEA conduit à une chaîne d'inéquations de la forme:

$$K_1v_{k_1} \leq K_2v_{k_2}+r_2 \leq K_3v_{k_3}+r_2+r_3 \leq \ldots \leq K_{c+1}v_{k_{c+1}}+r_2+r_3+\ldots+r_{c+1}$$

où r_j est une combinaison linéaire des variables non primaires et où chaque inéquation $K_iv_{k_i} \leq K_{i+1}v_{k_{i+1}} + r_{i+1}$ est obtenue à partir de l'inéquation $a_{j_ik_i}v_{k_i} + a_{j_ik_{i+1}}v_{k_{i+1}} \leq b_{j_i}$ en multipliant chaque membre par un facteur ad-hoc. D'autre part, $v_{k_1} = v_{k_{c+1}}$.

Nous admettrons que si le résidu de ce CEA vérifie les hypothèses du théorème, nous avons $K_1 = K_{c+1}$ et $r_2+r_3+\ldots+r_{c+1} = 0$. Ceci implique que la chaîne d'inéquations ci-dessus est en fait une chaîne d'équations, et donc que:

$$\forall i \in [1,c], \quad a_{j_i k_i} v_{k_i} + a_{j_i k_{i+1}} = b_{j_i}$$

Application

La méthode de R. Shostak est donc facilement modifiée pour rechercher les équations impliquées par un système d'inéquations, cette recherche s'effectuant en parallèle avec la preuve de faisabilité.

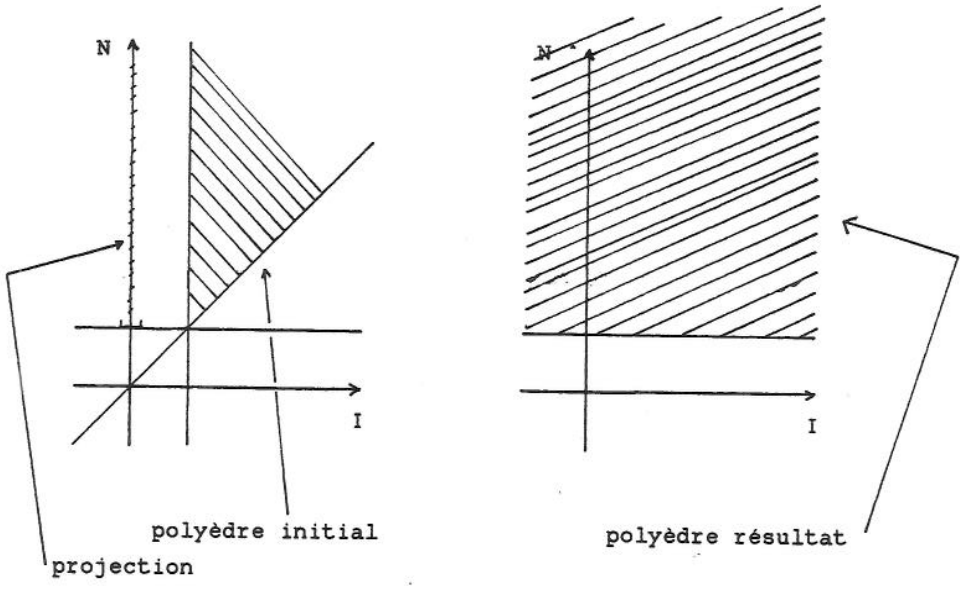
VIII.3. Elimination d'une variable dans un système d'inéquations linéaires

VIII.3.1. Principe

L'élimination d'une variable dans un système d'inéquations linéaires est une opération que nous avons utilisée plusieurs fois: calcul d'assertions initiales, traduction d'une assertion linéaire, élargissement des régions etc... L'algorithme réalisant cette opération est très simple; avant de le présenter, montrons en quoi elle consiste en terme de polyèdres convexes.

Il s'agit de supprimer toutes les contraintes sur la variable à éliminer (soit v_k, $1 \leq k \leq n$) sans perdre de contraintes sur les autres variables. Le polyèdre de Z^n résultant est donc un cylindre parallèle à l'axe correspondant à la variable v_k, et son intersection avec tout hyperplan parallèle à l'hyperplan $v_k = 0$ est égale à la projection du polyèdre initial sur cet hyperplan.

Exemple VIII-4: Soit le système d'inéquations $\{1 \leq I, I \leq N\}$, dans lequel nous éliminons la variable I:



La conservation des contraintes sur les autres variables implique que le polyèdre initial est inclus dans le polyèdre résultat. Ceci est important pour la suite du chapitre.

VIII.3.2. Algorithme

En pratique, l'élimination d'une variable se ramène au calcul de la projection d'un polyèdre convexe. Cet algorithme, bien connu, s'effectue en deux étapes. Soit à éliminer la variable v_k, $1 \leq k \leq n$.

(1) La conservation des contraintes sur les autres variables est obtenue en calculant toutes les assertions qu'il est possible de calculer par transitivité sur v_k. Tout couple d'inéquations d'indices i et j tel que

$$a_{ik} \neq 0 \quad \text{et} \quad a_{jk} \neq 0 \quad \text{et} \quad \text{signe}(a_{ik}) \neq \text{signe}(a_{jk})$$

donne naissance à une nouvelle inéquation, par l'opération "*" présentée au §VIII.2.1.6.

(2) L'élimination des contraintes sur v_k est obtenue en supprimant du système initial toute inéquation telle que $a_{ik} \neq 0$.

VIII.3.3. Exemple

Soit à éliminer I et J dans le système suivant :

$1 \leq I \leq N$
$1 \leq J \leq I-1$
$F = J+2$

Il se met sous la forme matricielle suivante:

I	J	N	F			
-1	0	0	0		-1	(1)
1	0	-1	0		0	(2)
0	-1	0	0	$\leqslant$	-1	(3)
-1	1	0	0		-1	(4)
0	-1	0	1		2	(5)
0	1	0	-1		-2	(6)

L'élimination de I conduit au système:

J	N	F				
-1	0	0		-1	(3)	→ (7)
-1	0	1		2	(5)	→ (8)
1	0	-1	$\leqslant$	-2	(6)	→ (9)
0	-1	0		-1	(1)&(2)	→ (10)
1	-1	0		-1	(2)&(4)	→ (11)

L'élimination de J conduit au système

N	F			
-1	0		-1	(10)
0	-1		-3	(7)&(9)
-1	0	$\leqslant$	-2	(7)&(11)
0	0		0	(8)&(9)
-1	1		1	(8)&(11)

Soit, "retraduit"

$N \geq 1$
$F \geq 3$
$N \geq 2$
$F \leq N+2$

VIII.4. Etude de la faisabilité d'un système d'inéquations linéaires par éliminations successives

VIII.4.1. Principe

Durant la phase 1 de l'algorithme précédent (cf. §VIII.3.2), deux sortes d'inéquations sont générées: celles qui contiennent des variables et celles qui n'en contiennent pas. Ces dernières sont de la forme $0 \leq c$, où $c \in Z$. Si c est strictement négatif, l'élimination conduit à une impossibilité. Le principe de cette 2ème méthode de test de faisabilité s'énonce comme suit.

Un système d'inéquations linéaires est faisable si et seulement si l'élimination successive de la totalité des variables de ce système ne conduit pas à une impossibilité.

Le polyèdre convexe associé au système correspondant à une ou plusieurs éliminations contient le polyèdre associé au système initial (cf. §VIII.3.1). Si une série d'éliminations conduit à une impossibilité, le polyèdre associé est vide; il en est donc de même pour le polyèdre initial, prouvant ainsi que le système initial est infaisable.

Supposons au contraire que toutes les éliminations "se passent bien". Nous construisons ainsi une suite de systèmes S_1, S_2, ... où le nombre de variables manipulées par chaque système est décroissant. Il est alors possible de "construire" une solution, initialisée au dernier système, en choisissant des valeurs arbitraires autorisées par S_{i+1}, et en les reportant dans S_i; à chaque étape, le choix fait "respecte" le système S_i puisque toutes les contraintes imposées par la variable éliminée sont inclues dans S_{i+1}.

VIII.4.2. Comparaison avec la méthode de R.Shostak

Cette méthode a pour avantage d'être très simple à implémenter. Par contre, ses performances sont à son désavantage. Nous pouvons facilement montrer qu'elle est souvent moins performante que celle de R. Shostak.

Nous allons pour cela analyser l'effet de l'élimination des variables primaires sur le graphe que la méthode de R. Shostak associe au système. L'élimination d'une variable primaire v_k, entre deux inéquations d'indices l et m, manipulant respectivement les variables primaires v_i, v_k et v_k,v_j, revient à remplacer la chaine admissible $n_i e_l n_k e_m n_j$ par la chaine $n_i e n_j$ où e est l'arête associée à la nouvelle inéquation. Nous en déduisons que l'élimination de toutes les variables primaires conduit au calcul des résidus le long de toutes les chaines élémentaires admissibles alors que la méthode de Shostak conduit au même calcul, uniquement le long des cycles élémentaires admissibles. La méthode par élimination successives ne peut donc être meilleure que si le temps passé à la construction du graphe et à la recherche des CEAs est supérieur à celui passé au calcul des résidus le long des châines que ne sont pas des cycles.

Dans le §VIII.2.4, nous avons vu que les systèmes réels se présentent sous la forme de chaînes transitives et ne comportent que peu de cycles; dans ce cas, la méthode de R. Stostak est sans doute meilleure.

VIII.4.3. Recherche des équations générées par un système d'inéquations

La méthode par éliminations successives permet, elle aussi, de détecter les équations induites par le système. En effet, si la somme de plusieurs inéquations, éventuellement pondérée par des coefficients positifs, conduit à une équation, alors toutes les inéquations entrant dans cette somme sont des équations.

Nous proposons donc d'associer à chaque inéquation la liste des indices des inéquations qui sont entrées dans sa construction -l'élimination d'une variable entre deux inéquations implique la concaténation de leurs listes associées-. Lorsqu'une suite d'éliminations conduit à l'inéquation $\{0 \leq 0\}$ alors toutes les inéquations de la liste qui lui est associée sont des équations.

Exemple VIII-5: Soit à éliminer I, J et K dans le système suivant:

$$\begin{aligned} I - 2J &\leq 3 \quad (0) \\ 2J + K &\leq 1 \quad (1) \\ -K - I &\leq -4 \quad (2) \end{aligned}$$

La liste associée à chaque inéquation est entre parenthèses.

Elimination de I:

$$\begin{aligned} 2J + K &\leq 1 \quad (1) \\ -2J - K &\leq -1 \quad (0,2) \end{aligned}$$

Elimination de J:

$0 \leq 0$ (0,2,1) → les inéquations 0, 2 et 1 sont des équations.

VIII.5. Résolution d'un système linéaire en nombres entiers

Nous décrivons rapidement la méthode de résolution d'un système linéaire en nombres entiers qui consiste à rechercher une forme réduite de Smith de la matrice du système. Cette méthode n'est pas unique, il est également possible d'utiliser les formes réduites de Hermite [Mino 82].

Le système à résoudre est le suivant:

$$A.V = B \quad , \quad V \in Z^n \qquad (I)$$

VIII.5.1. définitions

a) Une matrice unimodulaire est une matrice carrée à coefficients dans Z de déterminant +1 ou -1.

b) Une matrice de permutation U(i,j) est une matrice carrée d'ordre n telle que: $\forall k,l \in [1,n]$ tq $k \neq i$, $k \neq j$, $l \neq i$, $l \neq j$ et $k \neq l$ on ait:

c) Une matrice élémentaire $V(i,j,\alpha)$ est une matrice carrée d'ordre n telle que (soient k et l vérifiant les conditions de b):

$$v_{kk} = 1, \; v_{kl} = 0, \; v_{ij} = -\alpha \qquad \alpha \in Z.$$

VIII.5.2. Propriétés

a) Soient y et z deux n-vecteurs et Q une n.n-matrice unimodulaire liés par la relation y = Q.z; on a

y entier <=> z entier.

b) La prémultiplication (resp. postmultiplication) de A par une matrice de permutation U(i,j) a pour effet de permuter les lignes (resp. colonnes) i et j de A.

c) La prémultiplication (resp. postmultiplication) de A par une matrice élémentaire V(i,j,α) a pour effet de retrancher à la i-ème ligne (resp. j-ième colonne) α fois la j-ième ligne (resp. i-ème colonne).

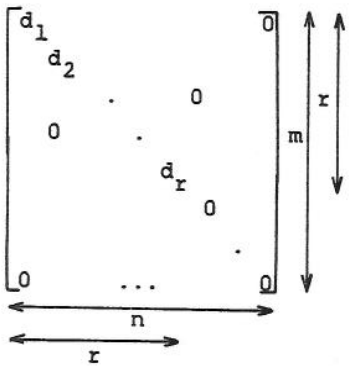
d) Les matrices élémentaires et les matrices de permutation sont des matrices unimodulaires.

e) Le produit de deux matrices unimodulaires est une matrice unimodulaire.

Forme réduite de Smith

Pour toute m.n-matrice de rang r à coefficients dans Z, il existe deux matrices unimodulaires P et Q telles que:

- P est une m.m-matrice,
- Q est une n.n-matrice,
- D = PAQ est une m.n-matrice du type:



VIII.5.3. Obtention de la forme réduite de Smith

La matrice D est construite par l'algorithme suivant:

E1: Par permutations des lignes et des colonnes (pré ou postmultiplications par des matrices de permutation) nous amenons le plus petit terme (en valeur absolue) des termes non nuls de la matrice en position (1,1); soit a_{11} ce terme.

E2: Nous remplaçons chaque terme de la 1ère ligne (resp. colonne) autre que a_{11} par le reste de sa division par a_{11} (post (resp. pré) multiplications par des matrices élémentaires).

E3: Si tous les termes autres que a_{11} de la lère ligne et de la lère colonne sont nuls, aller en E4 sinon aller en E1.

E4: Nous formons une nouvelle matrice en supprimant la lère ligne et la lère colonne de la matrice; puis le processus est itéré.

VIII.5.4. Application à la résolution des systèmes linéaires

Appliquons ces résultats à la résolution de (I). Nous recherchons tout d'abord P et Q telles que PAQ = D, D forme réduite de Smith de A. Puis un calcul très simple (cf. [Mino 82]) conduit aux résultats suivants.

notations. Soit J un ensemble d'indices et M une matrice. M_J (resp. M^J) désigne la matrice composée des lignes (resp. colonnes) d'indice dans J. Cette notation s'applique à un vecteur.

$$\text{I à une solution entière} <=> \begin{cases} (D_{1..r}^{1..r})^{-1}.P_{1..r}.B \text{ est entier et} \\ P_{r+1..m}.B = 0 \end{cases}$$

Dans ce cas, la solution est $V = V_o + S.X$ où

$$V_o = Q^{1..r}.(D_{1..r}^{1..r}).P_{1..r}.B,$$

$$S = Q^{r+1..n} \text{ et } X \text{ est un } (n-r)\text{-vecteur de } Z$$

VIII.6. Evaluation des expresssions

Nous avons supposé dans le §IV.3.6 savoir évaluer certaines expressions numériques ou booléennes, en utilisant un ensemble d'assertions linéaires associé au noeud contenant l'expression à évaluer.

D'autre part, la connaissance de bornes inférieures et supérieures pour une expression permet d'améliorer la recherche de conflits région/variable (cf. §VII.1.2.2.2) en restreignant le champ d'adresses associé à chaque région pour diminuer les risques de conflit par équivalence.

Nous allons brièvement montrer comment réaliser ces traitements. Dans la suite, nous supposons que le système d'assertions "contexte local" est prétraité c'est-à-dire mis sous la forme $A.V \leq B$.

VIII.6.1. Evaluation des expressions booléennes

En Fortran-77, les expressions booléennes sont construites à partir d'entités logiques, d'appels de fonctions logiques et d'expressions relationnelles. Très souvent, les deux expressions arithmétiques formant une expression relationnelle sont des combinaisons linéaires des variables entières du programme (K.GT.0, I+2J.NE.N+1, ...). Dans ce cas l'expression relationnelle peut se mettre dans l'une des trois formes suivantes: i, i.AND.i' et i.OR.i' où i et i' sont des assertions linéaires.

Exemple VIII-6:

K .GT. 0	devient	$-K \leq -1$
I+2J .NE. N+1	"	$I+2J-N \leq 0$ .OR. $N-I-2J \leq -2$
K+1 .EQ. J+4	"	$K-J \leq 3$ .AND. $J-K \leq -3$

Une assertion linéaire, donnée par une l.n-matrice C à coefficients dans Z et un élément c de Z ($C.V \leq c$), ne peut prendre pour valeur que VRAI ou FAUX. Nous proposons de tester si elle est égale à FAUX (resp. VRAI) en testant la faisabilité du système $A'.V \leq B'$ où A' est une m+l.n-matrice obtenue en ajoutant à A la ligne donnée par C (resp. -C) et où B' est un m+l-vecteur obtenu en ajoutant à B la coordonnée c (resp. -c-l). Si le système est faisable, nous ne pouvons rien en conclure. Sinon cela signifie que compte tenu du contexte $A.V \leq B$, nous ne pouvons pas avoir $C.V \leq c$ (resp. $-C.V \leq -c-1$) ce qui implique que l'assertion initiale est égale à FAUX (resp. VRAI).

L'évaluation d'une expression logique est donc effectuée en reportant, pour chacun des termes qui la composent, la valeur donnée par la méthode ci-dessus: VRAI, FAUX ou ?. Cette connaissance partielle peut permettre l'évaluation puisque

"FAUX .AND. ? = FAUX" et "VRAI .OR. ? = VRAI".

VIII.6.2. Evaluation des expressions numériques

Le même principe ne peut pas être utilisé car l'ensemble des valeurs susceptibles d'être prises par une expression numérique est infini. C'est pourquoi nous proposons la solution suivante, applicable uniquement pour les expressions construites sur les variables entières (entités simples de type INTEGER).

(1) Nous extrayons du système d'inéquations $A.V \leq B$, le système d'équations $A'V = B'$ qu'il induit (cf. §VIII.2.5 ou §VIII.4.3); puis nous résolvons ce système (cf. §VIII.5), ce qui conduit à la solution $V = V_o + S.X$, où $X = (x_i)_{i=1..n-r}$ sont les inconnues.

(2) Nous substituons dans l'expression à évaluer chaque variable par sa valeur trouvée en (1). Puis nous évaluons symboliquement l'expression résultante. Si toutes les variables inconnues (x_i) s'éliminent, la valeur numérique obtenue est la valeur de l'expression, sinon l'expression n'est pas évaluable avec le contexte donné par $A.V \leq B$.

Remarque VIII-4: Nous pouvons élargir le champ d'application de cette méthode en remplaçant chaque terme de l'expression, qui n'est pas une variable entière, par la valeur "?", et en remarquant que 0 x ? = 0.

Remarque VIII-5: Si l'expression y à évaluer est une combinaison linéaire des variables entières du programme, donnée par $y = C.V + c$, la phase 2 se réduit à:

y est évaluable si $C.S = 0$, sa valeur est alors $C.V_o + c$.

VIII.6.3. Recherche d'une borne inférieure et d'une borne supérieure pour une expression

De telles bornes sont facilement obtenues lorsque l'expression -soit y- est une combinaison linéaire des variables entières de la procédure. Nous introduisons pour cela une variable spéciale, soit v_s, que nous rendons égale à y en ajoutant au système d'assertions l'équation "$v_s = y$". Puis, nous éliminons du système résultant toutes les variables à l'exception de v_s. A la fin de cette opération, les inéquations restantes sont

de la forme:

$$v_s \leq bs \quad ou \quad -v_s \leq -bi$$

Nous en déduisons des bornes pour y en choisissant les inéquations telles que bs (resp. bi) soit minimum (resp. maximum).

VIII.7. Calcul de dépendance entre deux régions par test de faisabilité d'un système d'assertions linéaires

VIII.7.1. Introduction

A notre connaissance, la boucle suivante n'est pas reconnue comme vectorielle par les méthodes décrites au chapitre II, quelles que soient les assertions connues au noeud (a):

```
        DO 10 I = 1,N
10      T(I+K) = T(I) + ...      (a)
```

Il en est de même pour la boucle de subroutine MM de l'exemple présentée au §I.4.2:

```
        DO 10 I = 1,N3
10              CALL SMXPY(N2,A(1,I),N1,LDB,C(1,I),B)      (m2)
```

Nous allons montrer que ces deux boucles peuvent être vectorisées grâce aux améliorations que nous apportons.

Remarque VIII-6: Dans la suite, nous manipulons simultanément deux régions, c'est pourquoi nous nommons différemment leurs variables de description de régions.

Remarque VIII-7: Dans la suite, nous manipulons deux régions de la même entité. Cependant, tout ce que nous disons reste vrai pour deux régions de deux entités en aliasing parfait (cf. chapitre IX).

VIII.7.2. Système d'assertions associé à un calcul de dépendance

Soit B une boucle d'indice I, et S,T deux de ses instructions, non nécessairement distinctes. Il y a dépendance de S vers T s'il existe une valeur I1 de l'indice I - à laquelle correspond l'occurence S_{I1} de l'instruction S - et une valeur I2 - à laquelle correspond l'occurence T_{I2} de l'instruction T - telles que:

(C1) S_{I1} s'exécute avant T_{I2} selon l'ordre séquentiel d'exécution;

(C2) les suites de mémoires manipulées par S_{I1} et T_{I2} sont en conflit (iii).

Soient r_S et r_T deux régions d'un même tableau à d dimensions telles que r_S (resp. r_T) soit manipulée par S (resp. T). Nous proposons de traduire l'éventuelle dépendance de S vers T, induite par r_S et r_T, par le système d'assertions linéaires construit selon la méthode suivante.

(iii) le type de dépendance (PP,PC,CP) est donné par la nature de ce conflit (écriture/écriture, écriture/lecture, lecture/écriture).

E1: La condition (C1) se traduit par l'assertion:

{I1 = I2} si nous recherchons une dépendance directe,
{I1 < I2} si nous recherchons une dépendance de fermeture.

E2: La suite de mémoires manipulée par r_S (resp. r_T) est représentée par les assertions de cette région, auxquelles nous ajoutons les assertions associées au noeud contenant S (resp. T), données par une fonction d'assertions.

E3: La condition (C2) se traduit par le système suivant:

$$\{\rho_1^S = \rho_1^T,\ \rho_2^S = \rho_2^T, \ldots, \rho_d^S = \rho_d^T\}.$$

L'existence d'une dépendance a lieu si les variables entières du programme peuvent prendre des valeurs telles que les assertions construites par les étapes E1, E2 et E3 soient vérifiables simultanément. Nous avons donc ramené le calcul d'une dépendance au test de la faisabilité d'un système d'inéquations linéaires.

Attention, la création du système à tester par l'union des systèmes construits aux étapes E1, E2 et E3 n'a de sens que si les variables autres que l'indice de boucle désignent les mêmes valeurs dans chaque système. Une condition suffisante est que ces variables ne soient pas modifiées dans la boucle, ce qui est facilement vérifiable. A noter que les conditions sur la boucle imposées par la méthode des "bonnes boucles DO" font que les assertions ainsi calculées respectent cette condition.

VIII.7.3. Exemples

Nous avons implémenté la méthode d'étude de faisabilité par éliminations successives. Nous sommmes donc en mesure de traiter les deux boucles présentées au §VIII.7.1. Bien évidemment le système d'assertions à tester est construit manuellement.

Boucle 1

Nous supposons que la méthode des bonnes boucles DO associe au noeud (a) l'assertion {1 ≤ I ≤ N}. Nous étudions la dépendance PC* de (a) (T(I+K)) sur (a) (T(I)), en faisant varier l'information sur K: {ρ}, {K > 0}, {K = 0}, {K < 0} et {K > N}. Voici les résultats, ainsi que les temps d'exécution obtenus sur un Perkin Elmer 3220 tournant sous UNIX-V7. (FIS désigne ρ_1^S, FIT désigne ρ_1^T):

```
VAR I1, I2, N, K, FIS, FIT
{
        -I1 <= -1,
        -I2 <= -1,
         I1-N <= 0,
        +I2-N <= 0,
        -I1-K+FIS <= 0,
         I1+K-FIS <= 0,
        -I2+FIT <= 0,
        +I2-FIT <= 0,
        +FIS-FIT <= 0,
        -FIS+FIT <= 0,
         I1-I2 <= -1
}
Ce systeme est FAISABLE
      (.0597 sec.)
```

```
VAR I1, I2, N, K, FIS, FIT
{
        -K <= -1,
        -I1 <= -1,
        -I2 <= -1,
         I1-N <= 0,
        +I2-N <= 0,
        -I1-K+FIS <= 0,
         I1+K-FIS <= 0,
        -I2+FIT <= 0,
        +I2-FIT <= 0,
        +FIS-FIT <= 0,
        -FIS+FIT <= 0,
         I1-I2 <= -1
}
Ce systeme est FAISABLE
      (.0696 sec.)
```

```
VAR I1, I2, N, K, FIS, FIT
{
        +K <= -1,
        -I1 <= -1,
        -I2 <= -1,
         I1-N <= 0,
        +I2-N <= 0,
        -I1-K+FIS <= 0,
         I1+K-FIS <= 0,
        -I2+FIT <= 0,
        +I2-FIT <= 0,
        +FIS-FIT <= 0,
        -FIS+FIT <= 0,
         I1-I2 <= -1
}
Ce systeme est INFAISABLE
      (.0687 sec.)
```

```
VAR I1, I2, N, K, FIS, FIT
{
        +N-K <= -1,
        -I1 <= -1,
        -I2 <= -1,
         I1-N <= 0,
        +I2-N <= 0,
        -I1-K+FIS <= 0,
         I1+K-FIS <= 0,
        -I2+FIT <= 0,
        +I2-FIT <= 0,
        +FIS-FIT <= 0,
        -FIS+FIT <= 0,
         I1-I2 <= -1
}
Ce systeme est INFAISABLE
      (.0749 sec.)
```

```
VAR I1, I2, N, K, FIS, FIT
{
        +K <= 0,
        -K <= 0,
        -I1 <= -1,
        -I2 <= -1,
         I1-N <= 0,
        +I2-N <= 0,
        -I1-K+FIS <= 0,
         I1+K-FIS <= 0,
        -I2+FIT <= 0,
        +I2-FIT <= 0,
        +FIS-FIT <= 0,
        -FIS+FIT <= 0,
         I1-I2 <= -1
}
Ce systeme est INFAISABLE
      (.0766 sec.)
```

Boucle 2

Nous verrons dans le chapitre X que la subroutine MM lit et écrit la région suivante: $(A,\{1 \leq \rho_1 \leq N1,\ \rho_2 = I\})$. D'autre part, la méthode des bonnes boucles DO associe au noeud m2 la double assertion $\{1 \leq I \leq N3\}$. D'où le résultat:

```
VAR I1, I2, N1, N3, FIS1, FIT1, FIS2, FIT2
{
        -I1 <= -1,
        -I2 <= -1,                    <-  Bonne Boucle-DO
         I1-N3 <= 0,
        +I2-N3 <= 0,
        -I1+FIS2 <= 0,
         I1-FIS2 <= 0,                <-  Assertions de la région r_s
        -FIS1 <= -1,
        -N1+FIS1 <= 0,
        -I2+FIT2 <= 0,
        +I2-FIT2 <= 0,                <-  Assertions de la région r_r
        -FIT1 <= -1,
        -N1+FIT1 <= 0,
        +FIS1-FIT1 <= 0,
        -FIS1+FIT1 <= 0,              <-  Intersection des suites de mémoire ≠ ∅
        +FIS2-FIT2 <= 0,
        -FIS2+FIT2 <= 0,
         I1-I2 <= -1                  <-  Execution de S avant T
}
Ce systeme est INFAISABLE
      (.1358 sec.)
```

VIII.8. Conclusion

Dans ce chapitre, plusieurs méthodes ont été présentées permettant de vérifier les hypothèses concernant l'utilisation des assertions formulées dans les autres chapitres.

Signalons pour conclure que les deux traitements suivants seraient utiles dans le cadre d'une réalisation.

(1) L'élimination de la redondance dans un système d'assertions; l'intérêt d'un tel traitement est évident, surtout à cause de l'algorithme d'élimination de variable, générateur d'inéquations redondantes. A noter cependant que la phase d de standardisation proposée au §VIII.1.1 permet d'éliminer une partie de la redondance.

(2) L'union de deux régions; ce traitement nous permettrait d'exprimer les effets de l'exécution d'un appel d'externe sur une entité par une région unique. La recherche de l'enveloppe convexe de deux polyèdres convexes permet de réaliser cette union:

$$(T,\{\rho_1 = 1\}) \cup (T,\{\rho_1 = 10\}) \doteq (T,\{1 \leq \rho_1 \leq 10\})$$

A noter que la simplification obtenue se solde par une perte de précision.

Des algorithmes permettant de réaliser ces deux traitements sont donnés dans [Halb 79]. On remarquera notamment l'algorithme de recherche d'un système générateur -autre représentation d'un polyèdre convexe, formée d'un ensemble de points, vecteurs et droites- par intersections successives qui est intéressant dans la mesure où il permet de travailler en nombres entiers contrairement aux algorithmes classiques issus de la programmation linéaire.

C H A P I T R E IX

IX. Calcul de l'aliasing

Nous proposons dans ce chapitre une solution permettant de calculer l'aliasing associé à chaque occurence de procédure. Une partie du chapitre III est consacrée à ce problème (cf. §III.2). Nous y définissons l'aliasing, décrivons les principales causes de son apparition et expliquons pourquoi il est nécessaire de le calculer et d'en tenir compte dans le processus de parallélisation, bien que la norme Fortran-77 ([AFNO 83]) puisse être interprétée de façon à l'ignorer. Ce chapitre se décompose en deux parties.

(1) Nous présentons tout d'abord la méthode généralement utilisée pour calculer l'aliasing, et nous montrons pourquoi elle est mal adaptée à Fortran.

Nous présentons alors notre solution, qui calcule l'aliasing associé à une occurence de procédure en trois phases successives:

- phase de propagation d'adresses .sur les paramètres formels de chaque occurence de procédure;

- phase de détection des couples d'entités en aliasing;

- phase d'affinement des résultats.

(2) Nous donnons pour chaque phase des indications concernant sa réalisation. Nous montrons notamment comment l'utilisation des assertions permet d'améliorer les résultats.

IX.1. Présentation de la méthode adoptée pour le calcul de l'aliasing

IX.1.1. Présentation de la solution usuelle

La méthode proposée dans [Hech 77] calcule l'aliasing au niveau des entités, c'est à dire que les tableaux sont traités globalement. Elle consiste à associer à chaque paramètre formel d'une occurence de procédure la liste des entités dont il partage les suites de mémoires. Ceci est fait en analysant la liste de paramètres réels de l'appel d'externe correspondant de l'occcurence de procédure appelante, pour laquelle ces listes sont à jour.

Les listes associées au même paramètre formel, pour plusieurs occurences de procédure, peuvent être fusionnées, éventuellement en même temps que leur construction, ce qui permet d'associer l'aliasing à la procédure plutôt qu'à l'occurence de procédure.

Cette méthode présente plusieurs inconvénients.

(1) Le traitement global des tableaux crée de faux couples d'aliasing qu'un traitement plus fin permet d'éliminer. Il ne permet pas d'autre part de détecter l'aliasing total, cas où les suites de mémoires des deux entités sont égales. Enfin le traitement global des tableaux compromet l'utilisation d'assertions pour l'amélioration des résultats.

(2) Cette méthode n'est pas adaptée au problème des représentation différentes d'un même common, ainsi que le montre l'exemple IX-1.

Exemple IX-1:

```
PROGRAM P            SUBROUTINE Q(X)      SUBROUTINE R(Y)
COMMON /G/ A,B,C     COMMON /G/ BUF(3)    COMMON/G/ A,B,C
...                  ...                  ...
CALL Q(A)            CALL R(X)            END
...                  ...
END                  END
```

L'aliasing dans Q est: {(X,BUF)}; après propagation dans R, nous obtenons: {(Y,A),(Y,B),(Y,C)}. Les couples (Y,B) et (Y,C) sont de faux couples d'aliasing. Il aurait fallu mémoriser qu'à X n'est associée que la 1ère mémoire du common G.

IX.1.2. Présentation de notre solution

IX.1.2.1. Phase de propagation d'adresses sur les paramètres formels

Nous proposons de calculer l'aliasing associé à chaque occurence de procédure d'un programme par une analyse descendante du graphe des occurences d'appels. Cette analyse construit, pour chaque occurence de procédure $\omega \in \Omega$, une représentation de l'aliasing permettant d'associer à chaque paramètre formel les deux informations suivantes:

(1) la plage d'adresses dans laquelle il évolue,

(2) un booléen dont la signification est donnée plus loin.

Nous rappelons que les entités statiques locales d'une procédure sont éliminées par la phase ASSIA, qui les transforme en des entités communes (cf. §V.1.2). Les adresses qui sont suceptibles d'être prises par un paramètre formel appartiennent donc soit à un common $g \in G$, soit à la mémoire locale d'une procédure $p \in P$. La première information associée à un paramètre formel est donc un élément de l'ensemble

$$AD = (G + P) \times N \times N.$$

Un tel élément sera noté (s,ai,as) avec s = (GLOB,g) ou s = (PROC,p). Lorsque la plage d'adresses est réduite à un élément, nous notons ai = as = a.

La deuxième information associée à un paramètre formel est un booléen qui indique par la valeur VRAI que la première information est exacte -c'est à dire que la plage d'adresses est réellement occupée par le paramètre formel- et par la valeur FAUX que cette information est incertaine -c'est à dire que la plage d'adresses réellement occupée par le paramètre formel est inclue dans la plage indiquée-. les couples d'informations correctes sont donc de la forme:

paramètre formel variable:

((s,a ,a),VRAI): le pf est à l'adresse a;
((s,ai,as),FAUX): le pf est à l'adresse a, $ai \leq a \leq as$;

paramètre formel tableau:

((s,ai,as),VRAI): le premier élément du paramètre formel est à l'adresse ai et le dernier à l'adresse as;

((s,ai,as),FAUX): le premier élément du paramètre formel est à l'adresse ai' et le dernier à l'adresse as' avec: ai ≤ ai' ≤ as' ≤ as;

ce booléen sera noté "exact" dans la suite.

Avant de donner un exemple, notons que le §15.9.3.3 de [AFNO 83] stipule que la longueur d'un paramètre formel tableau ne peut pas dépasser la longueur du paramètre réel tableau correspondant, éventuellement diminuée de l'offset de l'élément de tableau paramètre réel. L'exemple IX.2 montre une association incorrecte.

Exemple IX-2:

```
        PROGRAM P                       SUBROUTINE Q(T1,T2,IS)
        INTEGER TAB(100)                DIM T1(100),T2(100)
        ...                             ...
        DO 10 I = 1,100                 END
10          CALL Q(TAB,TAB(I),N-I)
        ...
        END
```

L'association TAB/T1 est correcte contrairement à l'association TAB(I)/T2. Il faudrait soit DIM T2(*), soit DIM T2(IS) dans Q.

Bien que le respect de cette règle soit actuellement invérifiable, nous l'utilisons pour propager les plages d'adresses: la plage d'adresses associée à un paramètre formel est au maximum égale à celle du paramètre réel associé.

Exemple IX-3:

```
        PROGRAM M                       SUBROUTINE P(TX,TY)
        DIM T1(100),T2(10,10),T3(4)     DIM TX(*),TY(10)
        COMMON /G1/ T2                  ...
        ...                             END
        CALL P(T1(I),T2(7,4))
        ...
        CALL Q (T1(I),I)                SUBROUTINE Q(S1,S2)
        ...                             ...
        END                             END
```

Les adresses des entités réelles de M sont données par la fonction adr(M) (cf. §IV.3.3.1):

```
T1 : (LOCALE, ? ,  0)
T2 : (COMMUNE,G1,  0)
T3 : (LOCALE, ? ,100)
I  : (LOCALE, ? ,104)
```

On en déduit les adresses des paramètres formels de P et Q:

```
TX : ((PROC, M),  0, 99) & exact = FAUX
TY : ((GLOB,G1), 36, 45) & exact = VRAI

S1 : ((PROC, M),  0, 99) & exact = FAUX
S2 : ((PROC  M),104,104) & exact = VRAI
```

Nous remarquons que sans connaissance de la valeur de I, les plages d'adresses associées aux paramètres formels TX et S1 sont incertaines et de longueur égale à celle de T1. La plage associée à TY débute à l'adresse 36 -l'offset de l'élément de tableau T2(7,4) est connu et égal à 36- et finit à l'adresse 45 -la longueur du paramètre formel est connue et égal à 10-.

IX.1.2.2. Phase de détection des couples d'entités en aliasing

Nous ne pensons pas qu'il soit utile de conserver les deux informations associées à chaque paramètre formel car l'aliasing est assez rarement utilisé. C'est pourquoi nous suggérons de transformer le résultat de la première phase en une forme plus concise, en utilisant la représentation présentée au §III.2.1 et formalisée au §IV.3.5.1, dans laquelle l'aliasing est donnée par une fonction permettant de savoir si les suites de mémoires de deux entités sont égales -aliasing total-, différentes et d'intersection non vide -aliasing partiel- ou différentes et d'intersection vide -pas d'aliasing-.

Le coût de cette phase est non négligeable puisqu'elle se ramène à la comparaison deux à deux des variables simples de type entier. Comme d'autre part, la forme définitive de l'aliasing est moins précise que sa forme initiale, il est raisonnable de se demander si cette phase est intéressante. Le profit tiré de sa réalisation doit être évalué à partir des éléments suivants:

- gain sur le stockage des couples, dont le nombre décroit avec l'état réel de l'aliasing associé à chaque occurence de procédure, par rapport au stockage des adresses, de nombre constant quel que soit l'état d'aliasing et quelle que soit l'occurence de procédure;
- gain sur la recherche des conflits par parcours d'un ensemble de couples, temps décroissant avec l'état réel d'aliasing, ou par comparaison d'adresses, de temps constant.

Seule l'expérience permettrait de réellement évaluer l'intérêt de cette phase. A noter que si elle devait disparaitre, les modifications à apporter au reste de nos propositions seraient malgré tout en nombre limité.

IX.1.2.3. Phase d'affinement des résultats

Les deux premières phases sont tout à fait efficaces pour la détection de l'aliasing entre paramètres formels et entités globales. Elle sont notamment adaptées aux problèmes posés par les représentations différentes d'un common ou par les équivalences. Cependant certains cas de non-aliasing ou d'aliasing total leur échappent quand ils concernent deux paramètres formels, ainsi que le montre l'exemple IX.4.

Exemple IX-4: Soit M, MULSCA et MULVEC les trois procédures suivantes:

```
      PROGRAM M
      REAL MAT(100,100),VEC(100)
      ...
      DO 10 I = 1,100
10          CALL MULVEC(MAT(1,I),MAT(1,I),VEC,100)
      J = 0
      DO 20 I = 1,49
            J = I + 2
20          MAT(I,I) = MULSCA(VEC(I),VEC(J))                    (a)
      ...
      END


      SUBROUTINE MULVEC(R,V1,V2,N)
      REAL R(*),V1(*),V2(*).
      DO 10  I = 1,N
10          R(I) = V1(I) * V2(I)
      END


      FUNCTION MULSCA(S1,S2)
      MULSCA = S1 * S2
      END
```

Après les deux premières phases, l'aliasing est:

MULVEC: {(R,V1,PARTIEL)}, signifiant que tout élément du tableau R est en aliasing incertain avec tout élément du tableau V1;

MULSCA: {(S1,S2,PARTIEL)} signifiant que S1 et S2 sont peut-être en aliasing.

Ce résultat empêche la vectorisation de MULVEC.

Nous proposons donc une phase d'affinement des résultats, qui sélectionne certains couples d'entités en aliasing partiel -seul cas où nous ne soyons pas sûr du résultat- suivant des critères à préciser, et qui tente de les supprimer, ou de changer le type d'aliasing de partiel en total.

Nous proposons les trois améliorations suivantes:

A1: élimination des couples de variables simples formelles en aliasing partiel telles que les paramètres réels correspondant sont des références à des éléments différents d'un même tableau:

```
S = MULSCA(VEC(I),VEC(I+1))
MAT(I,I) = MULSCA(VEC(I),VEC(J)) sous le contexte {J = 2*I+1}
...
```

A2: changement du type d'aliasing de partiel à total pour les couples de variables formelles telles que les paramètres réels correspondant dont des références à un même élément de tableau:

```
S = MULSCA (VEC(I),VEC(I))
S = MULSCA (VEC(I),VEC(J-K)) sous le contexte {I-J+K = 0}
...
```

A3: changement du type d'aliasing de partiel à total pour les couples de tableaux formels de même déclaration tels que les paramètres réels correspondant sont des références au même élément de tableau:

```
CALL MULVEC(MAT(1,I),MAT(1,I),VEC,100) (i)
CALL MULVEC(VEC,VEC,VEC,100)
...
```

Cette dernière phase est évolutive et devra être programmée de façon que de nouveaux critères puissent être traités facilement.

IX.2. Réalisation de cette solution

L'objet de ce paragraphe est de donner des indications concernant la réalisation des différentes phases de cette solution.

IX.2.1. Notations; définitions

soit $\omega_p \in \Omega$ une occurence de procécdure, de représentation interne $ri_p \in RI$, appelant une occurence de procédure $\omega_q \in \Omega$, de représentation interne $ri_q \in RI$, par un appel d'externe $c_p \in C(ri_p)$ contenu dans un noeud $n_p \in N(ri_p)$.

Les paramètres réels de c_p sont des expressions de $X(ri_p)$; ils sont notés: $(pr_k)_{k=1,...}$.

Les paramètres formels de ω_q sont des expressions de $E(ri_q)$; ils sont notés: $(pf_k)_{k=1,...}$.

Lorsque nous nous intéressons à un couple paramètre réel/paramètre formel k particulier nous notons pr pour pr_k et pf pour pf_k.

Lorsque pr est un lhs nous notons $epr \in E(ri_p)$ l'entité de ce lhs, et $(z_k)_{k=1,...}$ ses expressions d'indices.

Nous définissons les ensembles suivants, associés à une représentation interne $ri \in RI$:

EF(ri) = {e ∈ E(ri) tq adr(ri)(e) = (FORMELLE,?,i)}
ensemble des entités formelles;

FAD(ri) = EF(ri) → AD x BOOL
ensemble des fonctions associant une plage d'adresses aux entités formelles de ri; le booléen indique si cette plage d'adresses est exacte ou non.

IX.2.2. Eléments nécessaires à cette réalisation

Le calcul de l'aliasing est réalisé par une analyse descendante du graphe des occurences d'appels. L'algorithme que nous proposons dans le chapitre X (cf. §X.2.1.2) suppose connus les éléments suivants:

(i) cette amélioration conduit à la vectorisation de MULVEC

- une fonction d'aliasing $fad_p \in FAD(ri_p)$ associée à ω_p,
- une fonction d'aliasing $fa_p \in FA(ri_p)$ associée à ω_p,
- une fonction d'assertion $fi_p \in FI(ri_p)$ associée à ω_q.

Cela signifie que nous disposons des représentations temporaires et définitives de l'aliasing associé à ω_p et que nous savons associer à tout noeud de $N(ri_p)$ les assertions qui sont vraies avant son exécution.

IX.2.3. Réalisation de la phase de calcul d'adresses

Le tableau IX-1 montre les différents types d'association pr/pf possibles. Il indique, pour chaque type, la plage d'adresses à donner au paramètre formel en fonction de celle du paramètre réel. Les remarques suivantes aident à la compréhension de ce tableau.

R1: Si pr est une expression complexe, il est évalué dans un temporaire local à la procédure. Nous disposons de temporaires en nombre suffisant, d'adresses: ((TMP,a,a),VRAI).

R2: Si pr est un lhs et que epr est une entité réelle, son adresse est calculée à partir de la fonction $adr(ri_p)$ et est notée ((X,ai,as),VRAI) (X signifiant (PROC,p) ou (GLOB,g)).

R3: Si pr est un lhs et que epr est une entité formelle, son adresse est donnée par la fonction fad_p et est notée ((X,ai,as),b) avec b = VRAI ou FAUX.

R4: Nous rappelons que l'offset d'un élément de tableau peut être calculé si ce tableau n'est pas ajustable. Si les expressions d'indices ne sont pas constantes, nous tentons de les évaluer par la fonction X_ifovali avec $fi_p(n_p)$ comme contexte d'assertions. Nous rappelons d'autre part que nous tentons de rendre constant un tableau ajustable en évaluant ses expression bornes avec $fi_p(rac(ri_p))$ comme contexte d'assertions.

R5: La longueur de pf, si celui-ci est un tableau, n'est connue que si ce tableau est constant.

Ce tableau, ainsi que les remarques R1 à R5 permettent de réaliser la phase de calcul d'adresses sans problème. Son résultat est une fonction $fad_q \in FAD(ri_q)$, associée à ω_q.

IX.2.4. Réalisation de la phase de calcul des couples d'entités en aliasing

La réalisation de cette phase ne pose pas de problème particulier. Les couples d'entités en aliasing se décomposent en deux catégories:
(1) les couples (e_q,pf) où e_q est une entité globale et pf une entité formelle,
(2) les couples (pf,pf') où pf et pf' sont deux entités formelles.

A Recherche des couples (e_q,pf) et du type d'aliasing

Soit (((G,g),ai,as),b) la plage d'adresses associée à pf par la fonction fad_q, résultat de la phase 1.

Soit (((G,g'),ai',as'),VRAI) celle associée à e_q par la fonction $adr(ri_q)$.

Il y a aliasing entre e_q et pf si

PARAMETRE REEL (pr)			PARAMETRE FORMEL (pf)		
type par. réel	ofs	adresse	variable	tableau lg. = 1	tableau lg. = ?
expr. complexe	ss	(TMP,a,a),V	(TMP,a,a),V	ERROR	ERROR
lhs var. réel	ss	(X,a,a),V	(X,a,a),V	ERROR	ERROR
lhs var. form.	ss	(X,a,a),V	(X,a,a),V	ERROR	ERROR
		(X,ai,as),F	(X,ai,as),F	ERROR	ERROR
lhs élément de tableau réel	?	(X,ai,as),V	(X,ai,as),F	idem →	(X,ai,as),F
	v	(X,ai,as),V	(X,ai+v,ai+v),V	(X,ai+v,ai+v+1),V	(X,ai+v,as),V
lhs élément de tableau formel	?	(X,ai,as),V	(X,ai,as),F	idem →	(X,ai,as),F
		(X,ai,as),F	(X,ai,as),F	idem →	(X,ai,as),F
	v	(X,ai,as),V	(X,ai+v,ai+v),V	(X,ai+v,ai+v+1),V	(X,ai+v,as),V
		(X,ai,as),F	(X,ai+v,as),F	(X,ai+v,ai+v+1),F	(X,ai+v,as),F

ss = sans signification, V = VRAI, F = FAUX, ofs = offset

tableau IX-1

$$g = g' \text{ et } [ai,as] \cap [ai',as,] \neq \emptyset.$$

Cet aliasing est total si ai = ai', as = as', b = VRAI et si e_q et pf ont des déclarations identiques: deux variables ou deux tableaux pareillement dimensionnés. Sinon l'aliasing est partiel.

A noter que si e_q et pf sont deux tableaux, la comparaison de leur dimensionnement n'est possible que si pf est un tableau constant.

B Recherche des couples (pf,pf') et du type d'aliasing

Soit (((P,p),ai,as),b) (resp. (((P,p'),ai',as'),b')) la plage d'adresses associée à pf (resp. pf') par la fonction fad_q, résultat de la phase 1.

Il y a aliasing entre pf et pf' si

$$p = p' \text{ et } [ai,as] \cap [ai',as'] \neq \emptyset.$$

Cet aliasing est total si ai = ai', as = as', b = b' = VRAI et que pf et pf' ont des déclarations identiques.

A noter que si pf et pf' sont deux tableaux, la comparaison de leur dimensionnement peut être réalisée numériquement (tableaux constants) ou symboliquement (tableaux ajustables).

Le résultat de cette phase est une fonction d'aliasing $fa_q \in FA(ri_q)$, associée à ω_q.

IX.2.5. Réalisation de la phase d'affinement

Les traitements des trois améliorations proposées au §IX.1.2.3 sont similaires. Nous traitons le cas de l'amélioration (A1).

Soit (pf,pf') ϵ $E(ri)^2$ un couple de variables formelles tel que $fa_q(pf,pf')$ = PARTIEL, où fa_q est la fonction calculée par la phase 2.

Soit pr (resp. pr') ϵ $X(ri_p)$ le paramètre réel associé à pf (resp. pf'). Le couple (pf,pf') rentre dans le cadre de l'amélioration A1 si

- pr est une référence à un élément de tableau, donné par e_p et $(z_k)_{k=1,...}$
- pr' est une référence à un élément de tableau, donné par e'_p et $(z'_k)_{k=1,...}$ et que $e_p = e'_p$ (ii).

Dans ce cas, il faut comparer chaque couple d'expressions $(z_k,z'_k)_{k=1,...}$. Cette comparaison peut être effectuée de différentes façons:

(1) par un système de calcul formel, qui forme l'expression $z_k - z'_k$ et tente de montrer qu'elle est constante et différente de 0: "I" et "I+1" peuvent être ainsi comparées;

(2) en évaluant numériquement chaque expression z_k et z'_k par la fonction X_ifovali (cf. chapitre VIII) avec le contexte d'assertions $fi_p(n_p)$;

(3) si z_k et z'_k sont deux combinaisons linéaires, nous formons la combinaison linéaire $z_k - z'_k$ -en utilisant les fonctions Xtocl, Xtotc, negcl et somcl de ri_p- et tentons d'évaluer son signe par la fonction X_ifocmp0 (cf. chapitre VIII);

(4) etc...

Le résultat de cette dernière phase est du même type que celui de la phase 2: une fonction fa_q ϵ $FA(ri_q)$ associée à ω_q.

IX.3. Conclusion

Dans ce chapitre nous avons proposé une méthode de calcul de l'aliasing particulièrement adaptée au langage Fortran-77. La description de cette méthode nous a amené à définir deux nouveaux ensembles.

(1) L'ensemble AD = (G + P) x N x N; un élément de AD représente une plage d'adresses dans laquelle évolue un paramètre formel.

(2) L'ensemble FAD(ri) = EF(ri) → AD x BOOL; un élément de FAD(ri) est une fonction décrivant l'aliasing associé à une occurence de procédure, sous forme de plages d'adresses. Cette représentation de l'aliasing sera nommée dans la suite <u>représentation temporaire</u>;

Les éléments nécessaires au calcul de l'aliasing associé à une occurence de procédure ω_q sont:

(ii) Cette condition peut être affaiblie en utilisant l'aliasing associé à ω_p; en effet e_p et e'_p désignent les mêmes suites de mémoires avec le même dimensionnement si $fa_p(e_p,e'_p) \neq$ TOTAL.

- la représentation temporaire de l'aliasing dans l'occurence de procédure appelante ω_p, donnée par $fad_p \in FAD(ri_p)$,
- la représentation définitive de cet aliasing donnée par $fa_p \in FA(ri_p)$,
- une fonction d'assertions $fi_p \in FI(ri_p)$ associée à l'occurence de procédure appelante ω_p.

C H A P I T R E X

X.3.3.4.4. Phase de recherche des variables entières non modifiées par l'exécution de ω_q (MA4)

X.3.3.4.5. Phase d'élargissement des régions (MA5)

X.3.3.4.6. Phase de rétrécissement des régions (MA6)

X.3.3.4.7. Phase de traduction des régions (MA7)

X.3.3.4.8. Conclusion sur la croissance de la fonction calcul_FM

X.3.4. Croissance des fonctions calcul_EP_A, calcul_EP_M et calcul_EP_I

X.3.5. Convergence de l'algorithme itératif

X.3.5.1. Convergence des suites de fonctions ep_a, ep_i et ep_m

X.3.5.2. Limite de ces suites de fonctions

X.3.6. Conclusion sur la convergence de l'algorithme iteratif

X. Traitement d'un programme complet

Nous allons voir dans ce chapitre comment calculer une structure de données permetttant d'associer à chaque occurence de procédure d'un programme un triplet de fonction: d'aliasing, de manipulation et d'information. Cette structure de donnée est nommée environnement de parallélisation. Ce chapitre se décompose en 3 parties.

(1) Nous commençons par définir la notion de programme complet, objet permettant de formaliser certaines propriétés, vérifiées par tout programme Fortran-77 correct. Nous montrons ensuite quels sont les principaux problèmes posés par le calcul d'un programme complet initial à partir d'un ensemble de pre-RIs. Enfin nous donnons la définition d'un environnement de parallélisation.

(2) Nous reprenons les résultats des chapitres V, VI, VII & IX et montrons comment organiser les calculs que nous y avons exposés pour calculer itérativement un environnement de parallélisation à partir d'un programme complet.

(3) Nous prouvons la convergence de l'algorithme itératif proposé en (2). Cette preuve se décompose en deux étapes: preuve de la croissance des suites de fonctions d'aliasing, d'assertion et de manipulation calculées par les différentes itérations, puis preuve de l'existence d'une limite atteinte en un nombre fini d'itérations.

X.1. Programme complet et environnement de parallélisation

X.1.1. Définition d'un programme complet

Un programme complet est un 8-uplet:

$$PC = (P, \Omega, \omega_m, RI, G, OP\_RI, RI\_P, C\_dw) \quad \text{ou}$$

- $P = \{P_1, P_2, \ldots\}$ est un ensemble fini de procédures;
- $\Omega = \{w1, w2, \ldots\}$ est un ensemble fini d'occurences de procédures;
- $\omega_m \in \Omega$ est une occurence de procédure particulière, associée au programme principal;
- $RI = \{ri_1, ri_2, \ldots\}$ est un ensemble fini de représentations internes;
- $G = \{g_1, g_2, \ldots\}$ est un ensemble fini de commons;
- OP_RI: $\Omega \rightarrow RI$ est une fonction associant à chaque occurence de procédure la représentation interne qui la décrit;
- RI_P: $RI \rightarrow P$ est une fonction associant à chaque représentation interne la procédure dont elle est une représentation interne; nous n'imposons pas une bijection entre les procédures et les représentations internes;
- C_dw est une fonction définissant le graphe des occurences d'appels; cf. paragraphe suivant.

Dans la suite nous notons $OP\_P = OP\_RI \circ RI\_P$; cette fonction associe à chaque occurence de procédure, la procédure dont elle est une occurence.

X.1.1.1. Graphe des occurences statiques d'appels

Nous rappelons qu'à une procédure sont associées autant d'occurences de procédure que de suites d'appels statiques d'externes conduisant à son exécution (cf. §IV.2.1.2). Le graphe des occurences statiques d'appels est donc un graphe où les noeuds représentent les occurences de procédure et les arcs les appels qui existent entre elles.

Soit $C = \bigcup_{ri \in RI} C(ri)$ l'ensemble de tous les appels du programme.

Le graphe des occurences statiques d'appels est le triplet $G_a = (\Omega, C\_dw, \omega_m)$ où C_dw est la fonction suivante:

$$C\_dw: \Omega \times C \rightarrow \Omega \cup \{?\}$$
$$(\omega_p, c_p) \qquad \omega_q$$

cette fonction associe à une occurence de procédure ω_p et à un appel d'externe c_p l'occurence de procédure ω_q appelée; elle n'est définie que pour les couples (ω,c) tels que c est un appel de ω, ce qui se traduit par

$$\forall \omega \in \Omega, \forall c \in C, c \notin C(OP\_RI(\omega)) \Rightarrow C\_dw(\omega,c) = ?$$

X.1.1.2. Propriétés du graphe des occurences statiques d'appels

Par définition, une occurence de procédure a au plus un prédécesseur, sinon cela signifierait que deux suites d'appels conduisent à la même occurence de procédure. D'autre part, une occurence de procédure qui n'a pas de prédécesseur est l'occurence d'un programme principal.

Nous en déduisons que le graphe des occurences statiques d'appels est un arbre. En effet, tous les noeuds ont exactement un prédécesseur, sauf celui associé à ω_m qui n'en a pas; d'autre part $\forall \omega \in \Omega$, il existe par définition un chemin de ω_m à ω, donné par la suite d'appels conduisant à ω.

Un programme correct vis à vis de [AFNO 83] ne contient pas d'appel récursif. Ceci implique que le graphe des appels est sans cycle et que l'arbre des occurences statiques d'appels n'a pas de branche de longueur infinie.

Nous pouvons donc définir la fonction "père" suivante:

$C\_up: \Omega \rightarrow \Omega \cup \{?\}$

$$\forall \omega_p, \omega_q \in \Omega, C\_up(\omega_q) = \omega_p \Leftrightarrow \exists c_p \in C(OP\_RI(\omega_p)) \text{ tq } C\_dw(\omega_p, c_p) = \omega_q$$

X.1.2. Obtention d'un programme complet initial

C'est le rôle de la phase d'analyse syntaxique interprocédurale (ASSIE) de construire un programme complet initial à partir d'un ensemble de pre-RIs (cf. figure IV-2). Une pre-RI est un objet analogue à une représentation interne, avec comme principale différence que les références aux objets globaux sont faits par les noms (chaine de caractères). Cette construction pose deux difficultés.

(1) L'élaboration du graphe des occurences d'appels, à cause de la notion de procédure formelle;

(2) la recherche des commons manipulables, directement ou indirectement, par chaque occurence de procédure.

Nous proposons de créer un programme complet initial tel qu'à chaque procédure soit associée une représentation interne unique, directement issue de la pre-RI correspondante. La phase ASSIE se décompose alors en trois étapes.

Etape 1

A partir de l'ensemble des pre-RIs, nous construisons l'ensemble RI des représentations internes. Le recensement des procédures réelles utilisées par le programme fournit l'ensemble P, le recensement des commons manipulés fournit l'ensemble G. Les références aux noms de commons sont remplacées par des références aux éléments de G. La procédure appelée par chaque appel d'externe est indiquée par une référence à un élément de P si elle est réelle ou par son rang dans la liste des paramètres si elle est formelle. D'autre part, la fonction RI_P est construite.

Etape 2

Le problème posé par les procédures formelles dans la construction du graphe des occurences d'appels est résolu de la façon suivante. L'association d'une procédure réelle à une procédure formelle ne peut être fait que par le mécanisme de passage de paramètres. C'est pourquoi nous proposons de créer ce graphe par une analyse récursive des représentations internes, en propageant, pour chaque occurence de procédure ω_q créée par un appel d'externe c_p d'une occurence de procédure ω_p déjà créée, une fonction W_q associant à chaque procédure formelle de ω_q la procédure réelle qui lui est associée. Cette fonction W_q est créée à partir de la fonction W_p, propagée pour ω_p, et de l'analyse des paramètres réels de c_p. D'où:

```
construire_GOA(ω_p,W_p)
        ri_p = OP_RI(ω_p); p = RI_P(ri_p);
        pourtout c_p ∈ C(ri_p) faire
                soit q la procédure appelée par c_p;
                créer une occurence de procédure ω_q;
                si formelle(q) alors q = W_p(q);
                Ω = Ω U {ω_q};
                soit ri_q ∈ RI tq RI_P(ri_q) = q;
                OP_RI(ω_q) = ri_q;
                C_up(ω_q) = ω_p;
                C_dw(ω_p,c_p) = ω_q;
                calculer W_q à partir de W_p et
                        des paramètres réels de c_p;
                construire_GOA(ω_q,W_q);
        finfaire
finproc
```

L'appel initial étant:

créeer une occurence de procédure ω_m;
$\Omega = \{\omega_m\}$;
construire_GOA(ω_m,$\emptyset$);
C_up(ω_m) = ?

Pendant cette phase sont construits l'ensemble Ω et les fonctions OP_RI, C_up, C_dw.

Etape 3

La connaissance du graphe des occurences statiques d'appels permet de construire pour chaque $p \in P$ les ensembles DESC(p) et ASC(p) définis au §V.1.3.3. Ces deux ensembles nous permettent d'insérer dans chaque $ri \in RI$ une description des commons qui sont indirectement manipulés par au moins une occurence de la procédure dont ri est la représentation interne, en utilisant les équations proposées au §V.1.3.3. (C). A noter que cette phase n'est pas obligatoire.

Nous ne rentrons pas plus en détails dans la réalisation de chaque étape, il faudrait pour cela formaliser le contenu d'une pre-RI. A noter que toutes les procédures doivent être représentées par une pre-RI, y compris celles usuellement contenues dans des bibliothèques.

X.1.3. Définition d'un environnement de parallélisation

Un environnement de parallélisation est un quadruplet

$$EP = (PC, ep\_i, ep\_a, ep\_m) \quad \text{où}$$

PC est un programme complet et ep_i, ep_a et ep_m sont des fonctions permettant d'associer à chaque occurence de procédure $\omega \in \Omega$, un triplet de fonctions: assertion, aliasing et manipulation.

La définition des trois fonctions ep_i, ep_a et ep_m nous demande quelques définitions préliminaires:

$$FA = \bigcup_{ri \in RI} FA(ri) \quad \text{avec } FA(ri) = E(ri) \times E(ri) \rightarrow \{ NON, PARTIEL:, TOTAL\}$$

$$FAD = \bigcup_{ri \in RI} FAD(ri) \quad \text{avec } FAD(ri) = EF(ri) \rightarrow AD \times BOOL$$

$$FI = \bigcup_{ri \in RI} FI(ri) \quad \text{avec } FI(ri) = N(ri) \rightarrow I(ri)^*$$

$$FM = \bigcup_{ri \in RI} FM(ri) \quad \text{avec } FM(ri) = C(ri) \rightarrow (R(ri)^* \times R(ri)^*$$

d'où:

$$ep\_i: \Omega \rightarrow FI$$
$$ep\_i(\omega) = fi \quad \wedge \quad fi \in FI(OP\_RI(\omega))$$

$$ep\_a: \Omega \rightarrow FA$$
$$ep\_a(\omega) = fa \quad \wedge \quad fa \in FA(OP\_RI(\omega))$$

$$ep\_m: \Omega \rightarrow FM$$
$$ep\_m(\omega) = fm \quad \wedge \quad fm \in FM(OP\_RI(\omega))$$

Nous aurons besoin d'une fonction temporaire ep_ad: $\Omega \rightarrow FAD$, pour le calcul de l'aliasing.

Rappelons que l'utilisation d'un environnement de parallélisation par un paralléliseur a été brièvement discutée au §IV.2.3. Ce problème sort du cadre de cette thèse, c'est pourquoi nous ne l'approfondissons pas.

Au niveau d'une occurence de procédure, l'apport du triplet de fonctions est important: prise en compte possible de l'aliasing, connaissance des effets de chaque appel d'externe, possibilité de calculer le graphe des dépendances de boucles DO contenant des appels, connaissance d'assertions associées à chaque noeud des occurences de procédure permettant de faciliter la tâche du MODULE SEMANTIQUE, etc...

Au niveau du programme complet, il conviendrait d'étudier les critères qui permettraient au paralléliseur de partitionner les occurences d'une même procédure de façon à ne pas générer, pour chacune, une version parallèle différente. Nous n'avons pas fait cette étude.

X.2. Algorithme itératif de calcul d'un environnement de parallélisation

L'objet de ce paragraphe est de proposer un algorithme de calcul d'un environnement de parallélisation à partir d'un programme complet. Les calculs des différentes fonctions ep_a, ep_i et ep_m ne sont pas indépendants, aussi, après avoir montré comment calculer chaque fonction en supposant les autres connues, nous proposons un algorithme itératif. Les dépendances entre les calculs de ep_a, ep_i et ep_m sont matérialisées par la figure X-1.

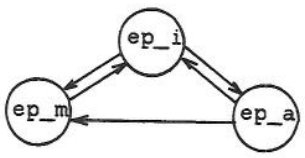


figure X-1

Notations

Nous reprenons nos notations habituelles (cf. §VI.1), auxquelles nous ajoutons:

$fa_p \in FA(ri_p)$, fonction d'aliasing définitive associée à ω_p,
$fad_p \in FAD(ri_p)$, fonction d'aliasing temporaire associée à ω_p,
$fi_p \in FI(ri_p)$, fonction d'information associée à ω_p,
$fm_p \in FM(ri_p)$, fonction de manipulation associée à ω_p.

De même pour fa_q, fad_q, fi_q et fm_q. D'autre part $rm = OP\_RI(\omega_m)$.

Remarque X-1: Les algorithmes que nous proposons dans les paragraphes suivants sont basés

sur le parcours de l'arbre des occurences d'appels, en ordre descendant ou ascendant. Ces types de parcours sont bien connus.

L'ordre descendant garantit que chaque occurence de procédure ω_p est "visitée" avant les occurences de procédure ω_q qu'elle appelle; le cas de ω_m doit être traité dans l'initialisation.

L'ordre ascendant garantit que chaque occurence de procédure ω_q est "visitée" avant l'occurence de procédure ω_p qui l'appelle; le cas des occurences de procédures terminales -ne contenant pas d'appel- doit être traité dans l'initialisation.

X.2.1. Calcul de l'aliasing pour un programme complet

X.2.1.1. Rappel des résultats obtenus

Nous proposons dans le chapitre IX un algorithme de calcul de l'aliasing, sous forme temporaire et définitive, pour l'occurence de procédure ω_q à partir des éléments suivants:

- deux fonctions d'aliasing fa_p et fad_p,
- une fonction d'information fi_p.

Nous définissons donc la fonction calcul_FA, qui associe à un quintuplet $(\omega_p,\omega_q,fa_p,fad_p,fi_p)$ le doublet (fa_q,fad_q). D'où:

$$\text{calcul\_FA: } \Omega \times \Omega \times FA \times FAD \times FI \rightarrow FA \times FAD$$

X.2.1.2. Calcul de la fonction ep_a

Le calcul de ep_a est réalisé par une analyse descendante du graphe des occurences d'appels. Les fonctions ep_a et ep_ad sont construites en parallèle, la fonction ep_ad n'étant utilisée que pendant le calcul de ep_a et pouvant ensuite être détruite. Ce calcul nécessite la connaissance de ep_i.

```
calcul_EP_A(ω_p)
        pourtout c_p ∈ C(ri_p) faire
                soit ω_q = C_dw(ω_p,c_p);
                (fa_q,fad_q) = calcul_FA(ω_p,ω_q,ep_a(ω_p),ep_ad(ω_p),ep_i(ω_p))
                ep_a(ω_q) = fa_q;
                ep_ad(ω_q) = fad_q;
                calcul_EP_A (ω_q);
        finfaire
finproc
```

Il n'y a pas d'aliasing dans le programme principal. Nous posons donc:

$fa_m \in FA(ri_m)$, $\forall (e_1,e_2) \in E(ri_m)^2$ on a $fa_m(e_1,e_2)$ = NON.

fad_m = ? , puisque fad_m est une fonction d'ensemble de définition vide.

D'où la séquence initiale du calcul de ep_a et ep_ad:

$ep\_a(\omega_m) = fa_m$;
$ep\_ad(\omega_m) = fad_m$;
$calcul\_EP\_A(\omega_m)$;

X.2.2. Calcul d'assertions pour un programme complet

X.2.2.1. Rappel des résultats obtenus

Nous exposons dans le chapitre VII plusieurs algorithmes de calcul d'assertions linéaires pour l'occurence de procédure ω_p, à partir des éléments suivants:

- une fonction d'aliasing fa_p,
- une fonction de manipulation fm_p,
- un ensemble d'assertions initiales $i_p^* \in I(ri_p)^*$, associé au noeud racine de ri_p, facultatif mais dont la connaissance améliore les résultats.

Nous définissons donc la fonction calcul_FI qui associe à un quadruplet $(\omega_p, fa_p, fm_p, i_p^*)$ une fonction fi_p. D'où

$$calcul\_FI: \Omega \times FA \times FM \times I^* \rightarrow FI$$

Nous proposons d'autre part un algorithme de calcul d'assertions initiales pour l'occurence de procédure ω_q, à partir des éléments suivants:

- deux fonctions d'aliasing fa_p et fa_q,
- un ensemble d'assertions contexte d'appel $i_p^* \in I(ri_p)^*$ associé à n_p, noeud de ri_p contenant l'appel c_p tel que $C\_dw(\omega_p, c_p) = \omega_q$.

Nous définissons donc la fonction calcul_I_init, qui associe à un quintuplet $(\omega_p, \omega_q, fa_p, fa_q, i_p^*)$ l'élément $i_q^* \in I(ri_q)^*$. D'où

$$calcul\_I\_init: \Omega \times \Omega \times FA \times FA \times I^* \rightarrow I^*$$

X.2.2.2. Calcul de la fonction ep_i

Le calcul de ep_i est réalisé par une analyse descendante du graphe des occurences d'appels; il nécessite la connaissance des fonctions ep_a et ep_m.

```
calcul_EP_I(ω_p, i_p^*)
        fi_p = calcul_FI(ω_p, ep_a(ω_p), ep_m(ω_p), i_p^*);
        ep_i(ω_p) = fi_p;
        pourtout c_p ∈ C(ri_p) faire
                soit ω_q = C_dw(ω_p, c_p);
                i_q^* = calcul_I_init(ω_p, ω_q, ep_a(ω_p), ep_a(ω_q), fi_p(n_p));
                calcul_EP_I(ω_q, i_q^*);
        finfaire
finproc
```

Les assertions initiales du programme principal sont soit $\emptyset$, soit un ensemble VIDATA $\in I(rm)^*$ extrait des DATAs par la phase ASSIA lors de l'analyse syntaxique du

programme principal (cf. §VII.5.1).

La séquence initiale est donc:

calcul_EP_I(ω_m,VIDATA ou $\emptyset$);

X.2.3. Calcul des manipulations pour un programme complet

X.2.3.1. Rappel des résultats obtenus

Nous proposons dans les chapitres VI et VII un algorithme de calcul des ensembles $RM(c_p)$ et $RU(c_p) \in R(ri_p)^*$, des régions modifiées et utilisées par l'exécution de l'appel d'externe c_p d'une occurence de procédure ω_p, à partir des éléments suivants:

- deux fonctions d'aliasing fa_p et fa_q,
- deux fonctions d'assertions fi_p et fi_q,
- une fonction de manipulation fm_q.

Nous définissons donc la fonction calcul_FM qui associe à un 8-uplet $(\omega_p,\omega_q,c_p,fa_p,fa_q,fi_p,fi_q,fm_q)$ un doublet (rm_p^*,ru_p^*).

$$\text{calcul\_FM: } \Omega \times \Omega \times C \times FA \times FA \times FI \times FI \times FM \rightarrow R^* \times R^*$$

X.2.3.2. Calcul de la fonction ep_m

Le calcul de ep_m est réalisé par une analyse ascendante du graphe des occurences d'appels; il nécessite la connaissance des fonctions ep_a et ep_i.

```
calcul_EP_M(ω_p)
        pourtout c_p ∈ C(ri_p) faire
                soit ω_q = C_dw(ω_p,c_p);
                calcul_EP_M(ω_q);
                (rm*_p,ru*_p) = calcul_FM(ω_p,ω_q,c_p,ep_a(ω_p),
                        ep_a(ω_q),ep_i(ω_p),ep_i(ω_q),ep_m(ω_q));
                ep_m(ω_p)(c_p) = (rm*_p,ru*_p);
        finfaire
finproc
```

Pour une occurence de procédure terminale ω -ne contenant pas d'appel d'externe- la fonction ep_m peut-être initialisée à "?" puisque $ep\_m(\omega)$ a alors un domaine de définition vide. D'où la séquence initiale:

```
pourtout ω ∈ Ω tq C(OP_RI(ω)) = ∅ faire ep_m(ω) = ? finfaire
calcul_EP_M(ω_m);
```

X.2.4. Algorithme itératif

Nous avons montré dans les paragraphes précédents que nous savons associer à un programme complet:

- une fonction ep_a connaissant une fonction ep_i,
- une fonction ep_m connaissant une fonction ep_i et une fonction ep_a,
- une fonction ep_i connaissant une fonction ep_m et une fonction ep_a.

Nous allons calculer, de façon itérative, trois listes de fonctions $(ep\_a)_k$, $(ep\_i)_k$ et $(ep\_m)_k$ $k = 0,1,\ldots$ grâce à la fonction ep_i_o, définie par:

$$\forall \omega \in \Omega \quad fi = ep\_i_o(\omega) \quad \text{verifie} \quad \forall n \in N(OP\_RI(\omega)), \quad fi(n) = \emptyset$$

ep_i_o n'associe en fait aucune assertion.

L'algorithme itératif est alors:

```
k = 0;
NONFINI = VRAI;
tantque NONFINI faire
        calculer ep_a_k   par calcul_EP_A en fonction de ep_i_k;
        calculer ep_m_k   par calcul_EP_M en fonction de ep_i_k et ep_a_k;
        calculer ep_i_k+1 par calcul_EP_I en fonction de ep_a_k et ep_m_k;

        mettre-à-jour NONFINI par décrémentation d'un compteur
        d'itérations ou par test de stationnarité en fonction de
        ep_i_k et ep_i_k+1;
finfaire
```

X.3. Preuve de convergence

Ce paragraphe est consacré à la preuve de la convergence de l'algorithme itératif proposé au paragraphe précédent.

Nous montrons tout d'abord que les suites de fonctions $(ep\_a)_k$, $(ep\_i)_k$ et $(ep\_m)_k$ sont croissantes. Pour cela nous définissons des relations d'ordre sur les ensembles FI, FA et FM, puis nous prouvons que les fonctions calcul_FA, calcul_FI et calcul_FM sont croissantes par rapport à tous leurs arguments. Nous en déduisons alors simplement que les fonctions calcul_EP_A, calcul_EP_I et calcul_EP_M sont croissantes ainsi que les suites $(ep\_a)_k$, $(ep\_i)_k$ et $(ep\_m)_k$.

Nous montrons ensuite l'existence d'une limite que nos algorithmes permettent d'obtenir en un nombre fini d'itérations.

Il n'est pas question d'adopter ici une démarche formelle, étant donné la complexité du problème et l'absence d'une sémantique précise pour Fortran-77. La preuve que nous proposons montre cependant que tout se passe bien, ce qui n'est pas évident à priori étant donné le nombre de fonctions et d'ensembles mis en jeu.

X.3.1. Définition de relation d'ordre sur FA, FAD, FI et FM

Les relations d'ordre que nous définissons dans la suite sont des relations d'ordre partiel, que nous notons toutes par le symbole "$\succ$". Nous n'aurons besoin de comparer que des fonctions relatives à la même occurence de procédure, c'est pourquoi nos ordres ne permettent de comparer que des fonctions appartenant à un même ensemble FA(ri), FI(ri) ou FM(ri).

X.3.1.1. Relation d'ordre sur FA

Soit fa une fonction d'aliasing et (e,e') un couple d'entités. Nous rappelons que fa(e,e') peut prendre trois valeurs:

- NON: e et e' ne sont pas en aliasing,
- TOTAL: e et e' sont en aliasing,
- PARTIEL: e et e' sont peut-être en aliasing.

Pour les valeurs NON et TOTAL, le résultat est sûr, pour PARTIEL il est incertain. La croissance d'une fonction d'aliasing doit traduire une augmentation de la connaissance de l'aliasing, c'est à dire que la valeur reste inchangée pour les couples en aliasing NON ou TOTAL, et qu'il existe au moins un couple tel que la valeur passe de PARTIEL à NON ou à TOTAL. Ceci se traduit par:

Soit fa',fa ∈ FA, fa' > fa =>

$\exists ri \in RI$ tq fa',fa ∈ FA(ri) ∧
$\forall e,e' \in E(ri)$, fa(e,e') = TOTAL → fa'(e,e') = TOTAL ∧
$\forall e,e' \in E(ri)$, fa(e,e') = NON → fa'(e,e') = NON ∧
$\exists e,e' \in E(ri)$ tq fa(e,e') = PARTIEL ∧ fa(e,e') = NON ou TOTAL

Nous notons fa' ≥ fa pour fa' > fa ou fa' = fa.

X.3.1.2. Relation d'ordre sur FAD

Soit fad une fonction d'aliasing temporaire et e une entité formelle, nous rappelons que fad(e) = ((gp,bi,bs),b) est l'adresse de e où:

- gp indique le type de mémoire associée à e (common ou procédure),
- [bi,bs] est un intervalle d'adresses mémoires,
- b indique si l'entité e recouvre l'intervalle [bi,bs] de façon partielle (FAUX) ou totale (VRAI).

La croissance d'une fonction d'aliasing temporaire doit traduire une augmentation de la connaissance de l'aliasing temporaire. Celle-ci peut prendre deux formes: réduction de l'intervalle [bi,bs] et passage du booléen b de FAUX à VRAI. Ceci se traduit par:

Soit fad',fad ∈ FAD, fad' ≥ fad =>

$\exists ri \in RI$ tq fad',fad ∈ FAD(ri) ∧
$\forall e \in EF(ri)$ tq fad(e) = ((gp,bi,bs),VRAI) on a fad'(e) = ((gp',bi',bs'),VRAI) ∧
$\forall e \in EF(ri)$ tq fad(e) = ((gp,bi,bs),FAUX) on a fad'(e) = ((gp',bi',bs'),b')

avec gp = gp' ∧ [bi',bs'] ∩ [bi,bs] ∧ (b' = FAUX ou b' = VRAI).

X.3.1.3. Relation d'ordre sur FI

Soit fi une fonction d'assertion et n un noeud. Nous rappelons que fi(n) est un ensemble d'assertions, soit i^*, qui définit un polyèdre convexe de Z^{nv} où nv est le nombre de variables entières de l'occurence de procédure. Ce polyèdre définit un ensemble de points de Z^{nv} que nous supposons obtenir par la fonction "sémantique" S:

$$\forall ri \in RI \quad S(ri): I(ri)^* \rightarrow (Z^{nv(ri)})^*$$

Cette fonction sémantique et l'inclusion dans $Z^{nv(ri)}$ définissent un ordre sur $I(ri)^*$ de la façon suivante:

$$\forall i^*, i'^* \in I(ri)^*, \quad i'^* > i^* \iff S(ri)(i'^*) \subsetneq S(ri)(i^*)$$

Cela signifie que certains nv(ri)-uplets de valeurs autorisés par i^* pour les variables de $V_i(ri)$ ne le sont plus par i'_*.

La croissance d'une fonction d'assertion doit traduire une augmentation de la connaissance des valeurs relatives des différentes variables en au moins un point du programme. Ceci se traduit par:

Soit $fi', fi \in FI$, $fi' > fi$ =>

$$\exists ri \in RI \text{ tq } fi', fi \in FI(ri) \wedge$$
$$\forall n \in N(ri), \quad S(ri)(fi'(n)) \subseteq S(ri)(fi(n)) \wedge$$
$$\exists n \in N(ri) \text{ tq } S(ri)(fi'(n)) \subsetneq S(ri)(fi(n))$$

Nous notons $fi' \geq fi$ pour $fi' > fi$ ou $fi' = fi$.

X.3.1.4. Relation d'ordre sur FM

Une région de r de R(ri) est un couple (e, i^*) où $i^* \in I(ri)^*$ définit les valeurs que peuvent prendre les indices des différentes dimensions, matérialisés par les variables de description de région. L'ordre sur $I(ri)^*$ induit un ordre sur R(ri):

$$\forall r, r' \in R(ri) \quad r = (e, i^*),\ r' = (e', i'^*), \text{ on a } r' > r \Rightarrow e = e' \wedge i'^* > i^*$$

Nous ne comparons que des régions de la même entité. Nous notons $r' \geq r$ pour $r' > r$ ou $r' = r$.

Nous en déduisons un ordre sur les ensembles de régions, de la manière suivante:

$$\forall r'^*, r^* \in R(ri)^*, \quad r'^* \geq r^* \Rightarrow \forall r' \in r'^*, \exists r \in r^* \text{ tq } r' \geq r$$

Cette définition peut paraître curieuse. Elle signifie qu'il n'existe pas d'entité $e \in E(ri)$, pour laquelle les parties décrites par les régions de r'^* sont plus grandes que celles décrites par les régions de r^*. Nous en déduisons un ordre sur FM par:

Soit $fm', fm \in FM$, $fm' \geq fm$ =>

$$\exists ri \in RI \text{ tq } fm', fm \in FM(ri) \wedge$$

$$\forall n \in RI, \ fm'(n) = (r'^*_m, r'^*_u) \text{ on a}$$

$$r'^*_m \geq r^*_m \text{ et } r'^*_u \geq r^*_u$$

X.3.2. Croissance d'opérations élémentaires

Les fonctions calcul_FA, calcul_FI et calcul_FM utilisent les 3 opérateurs de base suivants, dont nous allons montrer la croissance:

(1) l'évaluation d'une expression grâce à un ensemble d'assertions,

(2) l'élimination d'une variable dans un ensemble d'assertions,

(3) la recherche de conflit région/variable.

Dans la suite, nous omettons de signaler l'appartenance à ri -représentation interne de l'occurence de procédure analysée- des différents éléments manipulés.

X.3.2.1. Croissance des fonctions d'évaluation d'expressions

Les fonctions d'évaluation présentées au §VIII.6 ne permettent pas d'évaluer toutes les expressions d'une occurence de procédure. Nous allons montrer que plus d'expressions sont évaluables quand les arguments s'améliorent. L'évaluation d'une expression x nécessite deux arguments (cf. §IV.3.6): une fonction d'aliasing fa et un ensemble d'assertions i^*; elle est réalisée en deux phases:

(1) le prétraitement des assertions i^*, à partir de fa, pour former un nouvel ensemble d'assertions i_1^*,

(2) l'évaluation proprement dite de x à partir de i_1^*.

A Croissance du prétraitement des assertions

Nous l'avons présenté au §VIII.1. La standardisation des assertions n'a aucune influence sur leur sémantique. La prise en compte des associations nous conduit à former i_1^* en ajoutant à i^* un ensemble i_a^* construit à partir des couples de variables en association totale, par équivalence -dont le nombre est constant pour un programme donné- ou par aliasing TOTAL -dont le nombre dépend de fa.

Croissance du prétraitement par rapport à i^*

Elle se ramène à la preuve de croissance de l'union d'ensembles d'assertions. En effet, soit $i'^* \geq i^*$, nous devons montrer que:

$$i'^*_1 = i'^* \cup i_a^* \geq i_1^* = i^* \cup i_a^*$$

C'est évident car la fonction sémantique S, sur laquelle est basée notre ordre, vérifie la propriété suivante:

$$\forall i_r^*, i_s^* \in I^*, \quad S(i_r^* \cup i_s^*) = S(i_r^*) \cap S(i_s^*)$$

En effet, tout élément de Z^{nv} vérifiant l'union de deux ensembles de contraintes vérifie chacun séparément, et réciproquement.

Croissance du prétraitement par rapport à fa

L'amélioration de fa conduit à une amélioration de i_a^* et donc à une amélioration de i^*. En effet, la définition de l'ordre sur FA nous indique que le nombre de couples d'entités en aliasing TOTAL ne peut pas diminuer quand fa s'améliore.

B Croissance de l'évaluation

Une expression x est évaluable avec un ensemble d'assertions i_1^* si $\forall z \in S(i_1^*)$ les valeurs des variables de V_i impliquées par z déterminent la même valeur pour x, différente de ?. Par définition de l'ordre sur i^*, l'amélioration de i_1^* se traduit par une diminution de $S(i_1^*)$. Il est donc évident que toute expression évaluable avec i_1^* le sera encore avec i'^*_1 tq $i'^*_1 \geq i_1^*$.

X.3.2.2. Croissance de la fonction d'élimination d'une variable

L'élimination d'une variable dans un ensemble d'assertions i^* est généralement précédée d'une phase de prétraitement sur laquelle nous ne revenons pas. L'opération d'élimination (cf. §VIII.3) construit un ensemble d'assertions i_1^* où toutes les contraintes sur la variable à éliminer, soit v_k, ont été éliminées sans que celles sur les autres variables le soient. Nous en déduisons $S(i_1^*)$ à partir de $S(i^*)$:

$$S(i_1^*) = \bigcup_{z \in S(i^*)} \{(z_1,\ldots,z_{k-1},y,z_{k+1},\ldots,z_{nv}) \text{ tq } y \in Z \text{ et } z = (z_1,\ldots,z_k,\ldots,z_{nv})\}$$

De cette formule, nous déduisons simplement la croissance de l'élimination d'une variable par rapport à i^*: une amélioration de i^* implique une diminution de $S(i^*)$ et donc de $S(i_1^*)$. (propriété de U), une diminution de $S(i_1^*)$ traduit une amélioration de i_1^*.

X.3.2.3. Croissance de la fonction de recherche de conflit région/variable

Cette recherche, que nous avons étudiée au §VII.1.2.2.2 nécessite trois arguments: une fonction d'aliasing fa, un ensemble d'assertions "contexte local" i_c^* et la région $r = (e,i^{*})$. Nous allons montrer que tout conflit existant avec des arguments améliorés avaient lieu avec les arguments primitifs.

A Croissance de la recherche de conflit direct

Soit $r' = (e',i'^{*})$ une région telle que $r' \geq r$ et telle que r' est en conflit direct avec une variable v; ceci implique que e' = v. Deux régions comparables décrivent des parties de la même entité, d'où e = e' = v. Donc r et v sont en conflit direct. Les autres arguments ne sont pas utilisés.

B Croissance de la recherche de conflit par aliasing

La croissance de cette recherche par rapport à r se montre comme en A car les assertions i^* ne sont pas utilisées. Montrons la croissance par rapport à fa. Soit fa' telle que fa' $\geq$ fa. Supposons que r soit en conflit avec v pour fa'; ceci implique que fa'(e,v) $\neq$ NON. Par définition de l'ordre sur FA, nous avons:

$$fa' \geq fa \wedge fa'(e,v) \neq \text{NON} \Rightarrow fa(e,v) \neq \text{NON}$$

Ceci implique que r et v sont en conflit pour fa. D'autre part, l'argument i_c^* n'est pas utilisé.

C Croissance de la recherche de conflit par équivalence

La recherche de conflit par équivalence utilise les arguments r, fa et i_c^*. Elle se décompose en 6 phases, visant à réduire la plage des adresses couverte par la région r. Montrons intuitivement que cette réduction sera d'autant meilleure quand les arguments s'améliorent, diminuant ainsi les risques de conflits. Une amélioration de i_c^* ou de i^* (croissance par rapport à r) implique une diminution de $S(i_c^* \cup i^*)$; l'ensemble des valeurs possibles des variables -et donc des expressions d'indices- est donc réduit. De même, une amélioration de fa implique une amélioration des évaluations d'expressions (cf. paragraphes précédents).

Remarque X-2: Dans nos algorithmes, la recherche de conflit a généralement lieu entre une variable et un ensemble de régions. la croissance de cette recherche est simplement déduite.

Soit r'^* et r^* deux listes de régions telles que $r'^* \succeq r^*$ et telles que r'^* est en conflit avec une variable v.

r' sup * et v sont en conflit => $\exists r' \in r'^*$ tq r' et v sont en conflit.

D'autre part $r'^* \succeq r^*$ => $\forall r' \in r'^*, \exists r \in r^*$ tq $r' \succeq r$.

r' et v sont en conflit $\wedge$ r' >= r => r et v sont en conflit.
=> r^* et v sont en conflit.

X.3.3. Croissance des fonctions calcul FI, calcul I init, calcul FA et calcul FM

La preuve formelle de la croissance de ces quatres fonctions ne poserait pas de problème technique, elle serait cependant longue et ennuyeuse. Aussi nous préférons une preuve informelle; pour cela, les quatres fonctions sont segmentées en phases dont nous prouvons la croissance séparémment. D'autre part, les noms de ces phases sont utilisées comme repère dans l'exemple de l'annexe A.

X.3.3.1. Croissance de la fonction calcul FI

Les arguments de calcul_FI sont ω_p, fa_p et i_p^* (assertions initiales); son résultat est fi_p. Nous supposons que cette fonction est réalisée par la méthode de propagation des constantes.

Nous ne souhaitons pas rentrer dans le détail du fonctionnement de cette méthode, aussi nous formulons les deux hypothèses suivantes:

(H1) fi_p s'améliore quand i_p^* s'améliore,

(H2) fi_p s'améliore quand les assertions calculées par l'opérateur associé à un noeud s'améliorent.

L'hypothèse H1 implique la croissance de calcul_FI par rapport à i_p^*.

L'opérateur de calcul d'assertions associé à un noeud se décompose en deux phases (cf. §VII.1.2.2): création d'assertions et destruction d'assertions.

A Croissance de la phase de création d'assertions (IF1)

Seuls sont concernés les noeuds associés à une instruction d'affectation. Une assertion est créée pour chaque noeud tel que l'expression associée est évaluable, en utilisant fa_p et les assertions associées au noeud par le calcul en cours.

La croissance de cette phase est déduite de la croissance des fonctions d'évaluation d'expressions: plus de noeuds sont créateurs d'assertions quand les arguments s'améliorent.

B Croissance de la phase de destruction d'assertions (IF2)

Cette phase construit, pour chaque noeud, l'ensemble des régions modifiées par son exécution. Une partie de cet ensemble ne dépend que de la nature de l'instruction et reste donc constante au cours des itérations. Une autre partie est déduite de fm_p; la croissance de fm_p se traduit donc par une amélioration de l'ensemble des régions modifiées.

Les assertions portant sur des variables en conflit avec cet ensemble sont détruites. De la croissance de la fonction de recherche de conflit nous déduisons la croissance de cette phase par rapport à fm_p et fa_p: moins de variables seront en conflit, et donc moins d'assertions seront détruites en chaque noeud.

C Conclusion

La croissance des arguments fa_p, fm_p et i_p^* se traduit par une croissance des assertions créées en chaque noeud (IF1), une décroissance des assertions détruites en chaque noeud (IF2), et une croissance des assertions initiales (i_p^*). Ces résultats, combinés aux hypothèses H1 et H2 implique la croissance de la fonction calcul_FI.

X.3.3.2. Croissance de la fonction calcul I init

Les arguments de calcul_I_init sont ω_p, ω_q, fa_p, fa_q et i_p^* (contexte d'appel); son résultat est i_q^*.

Cette fonction calcule un ensemble d'assertions i^* en ajoutant à i_p^* un ensemble constant d'assertions, déduites des associations de paramètres entre ω_p et ω_q. La croissance de i_p^* implique donc une croissance de i^*. Puis i_q^* est formé en éliminant dans i^* les variables de $V_i(ri_p)$, qui sont en en nombre constant.

La croissance de calcul_I_init par rapport à fa_p, fa_q et i_p^* est donc directement déduite de la croissance de la fonction d'élimination d'une variable dans un système d'assertions.

X.3.3.3. Croissance de la fonction calcul FA

Les arguments de calcul_FA sont ω_p,ω_q,fa_p fad_p et fi_p; son résultat est fa_q et fad_q. Elle se décompose en trois phases.

X.3.3.3.1. Croissance de la phase de propagation d'adresses (AL1)

Cette phase calcule la fonction fad_q à partir de fad_p et de fi_p. Le champ d'adresses d'une entité formelle de ω_q est calculé à partir de

- $adr(ri_p)$ si le paramètre réel associé est un lhs d'une entité réelle de ω_p;

- fad_p si le paramètre réel associé est un lhs d'une entité formelle de ω_p.

Il est ensuite éventuellement restreint par un calcul d'offset.

Enfin le booléen, indiquant si le champ d'adresses est exact ou non, est calculé à partir de sa valeur initiale dans ω_p et du fait que l'offset est connu ou non.

A Croissance de AL1 par rapport à fa_p et fi_p

La croissance de la fonction d'évaluation se traduit par une augmentation du nombre d'expressions évaluables, du nombre de tableaux ajustables transformés en tableaux constants, et donc du nombre d'offset connus.

L'analyse du tableau IX-1 montre que le passage de la valeur de l'offset de "?" à une valeur v connue améliore l'adresse du paramètre formel correspondant.

Exemple X-1: Prenons le cas où le paramètre réel est un lhs élément de tableau formel, d'adresse ((X,ai,as),FAUX), et le paramètre formel est une variable. L'adresse propagée est:

((X,ai,as),FAUX) si la valeur de l'offset est ?
((X,ai+v,as),FAUX) si la valeur de l'offset est v, ce qui est meilleur

B Croissance par rapport à fad_p

L'amélioration de fad_p n'a de répercussion que sur les paramètres formels de ω_q qui sont associés à des paramètres réels lhs d'entités formelles de ω_p. Seule une analyse de tous les cas d'associations possibles, grâce au tableau IX-1, peut prouver qu'une amélioration de fad_p implique une amélioration de fad_q. Nous nous contentons d'examiner le cas de l'exemple précédent:

Exemple X-2: Nous supposons que l'offset est connu et vaut v. L'amélioration de l'adresse du paramètre réel peut prendre deux formes suivant que le booléen passe à VRAI ou non. Le tableau suivant montre la croissance:

adresse initiale:	((X,ai ,as),FAUX)		→ Cas 1
amélioration 1:	((X,ai',as'),FAUX)	[ai',as'] ⊂ [ai,as]	→ Cas 2
amélioration 2:	((X,ai ,as'),VRAI)	[ai',as'] ⊂ [ai,as]	→ Cas 3

adresse propagée cas 1: ((X,ai+v,as),FAUX)
adresse propagée cas 2: ((X,ai+v,as'),FAUX) ≥ ((X,ai+v,as),FAUX)
adresse propagée cas 3: ((X,ai+v,as'),VRAI) ≥ ((X,ai+v,as),FAUX)

X.3.3.3.2. Croissance de la phase de calcul des couples d'entités en aliasing (AL2)

Cette phase calcule la fonction fa_q à partir de la fonction fad_q calculée en AL1 ; elle est présentée au §IX.2.4.

La croissance de fad_q peut se traduire par une diminution du champ d'adresses associé à certaines entités formelles. Dans ce cas, les risques de collisions entre entités décroissent, impliquant que l'aliasing associé par fa_q à certains couples d'entités va

passer de PARTIEL à NON.

La croissance de fad_q peut aussi se traduire par le passage du booléen de FAUX à VRAI pour certaines entités formelles. Ceci a pour conséquence que l'aliasing associé par fa_q à certains couples d'entités va passer de PARTIEL à TOTAL.

Dans les deux cas, la fonction résultante fa_q est d'autant meilleure que fad_q est meilleure.

X.3.3.3.3. Croissance de la phase d'affinement (AL3)

L'affinement de la fonction fa_q calculée en AL2 est réalisé en deux temps (cf. §IX.2.5). Les couples d'entités de ω_q pour lesquels fa_q est suceptible d'être affinée sont recherchés. Cette recherche est invariable au cours des itérations car elle ne dépend que du type des associations de paramètres.

L'affinement proprement dit consiste en des évaluations (comparaisons) d'expressions. La croissance de cette phase par rapport à fi_p et fa_p est donc directement déduite de la croissance de la fonction d'évaluation d'expressions: plus d'expressions seront évaluables et donc comparables, impliquant ainsi un plus grand nombre d'affinements.

X.3.3.3.4. Conclusion sur la croissance de calcul FA

Les trois phases de calcul_FA sont croissantes par rapport à ses arguments et par rapport aux résultats des phases précédentes. Nous en déduisons que calcul_FA est croissante puisque la composition de fonctions croissantes est croissante.

X.3.3.4. Croissance de la fonction calcul FM

Cette fonction est de loin la plus complexe. Ses arguments sont ω_p, ω_q, c_p, fa_p, fa_q, fi_p, fi_q et fm_q; son résultat est un doublet (rm^*_p, ru^*_p). Elle se décompose en sept phases.

X.3.3.4.1. Phase de calcul des noeuds accessibles (MA1)

La recherche de l'ensemble NA des noeuds accessibles de ω_q fait appel aux fonctions fi_q et fa_q, pour l'évaluation des expressions testées dans les noeuds de test du graphe de contrôle de ω_q. L'amélioration des fonctions fi_q et fa_q implique que plus d'expressions testées vont être évaluées et donc que moins de noeuds seront déclarés accessibles. Cette phase est donc croissante par rapport à fi_q et fa_q.

X.3.3.4.2. Phase de recherche des régions manipulées par chaque noeud (MA2)

L'ensemble de régions manipulées par l'exécution d'un noeud est formé à partir de l'ensemble des régions directement manipulées -donné par les fonctions $N\_mod(ri_q)$ et $N\_uti(ri_q)$ et donc constant pour toutes les itérations- et de l'ensemble des régions indirectement manipulées, donné par la fonction fm_q. La croissance de fm_q se traduit par une croissance de cet ensemble de régions; cette phase est donc croissante par rapport à fm_q (propriété de U).

X.3.3.4.3. Phase d'utilisation du contexte local associé à chaque noeud (MA3)

Cette phase ajoute aux assertions des régions des ensembles précédemment calculés les assertions associées à chaque noeud de ω_q par fi_q. L'opération utilisée pour cela est l'union d'ensembles d'assertions, qui est une opération croissante. Nous en déduisons donc

que cette phase est croissante par rapport à fi_q et par rapport au ensembles calculés en MA2.

X.3.3.4.4. Phase de recherche des variables entières non modifiées par l'exécution de ω_q (MA4)

Une variable $v \in V_i(ri)$ est dite modifiée s'il existe un noeud de NA tel que v est en conflit avec une des régions modifiées par ce noeud.

La croissance de cette phase se traduit par le fait que moins de variables vont être déclarées modifiées, ce qui améliore les phases suivantes.

La croissance par rapport à NA est dûe au fait que la diminution du nombre de noeuds accessibles limite les risques de conflit, puisque moins de régions sont modifiées. La croissance par rapport à fa_q et aux ensembles de régions calculés en MA3 est dûe à la croissance de la fonction de recherche de conflit ensemble-de-régions/variable.

Cette phase calcule l'ensemble VNM_q des variables de $V_i(ri_q)$ non modifiées par ω_q.

X.3.3.4.5. Phase d'élargissement des régions (MA5)

Cette phase élimine dans les assertions des régions des ensembles calculés en (MA3), les variables de $V_i(ri_q)$ qui n'appartiennent pas à VNM_q. L'élimination d'une variable dans un ensemble d'assertions est une opération croissante.

L'ordre sur les régions étant déduit de l'ordre sur les assertions, nous en déduisons que l'élimination d'une variable dans la description d'une région $r = (e,i^*)$ est une opération croissante par rapport à r et à fa_q.

Nous n'éliminons pas une variable unique, mais toutes celles de $V_i(ri_q) - VNM_q$. L'élimination d'une variable est une opération dégradante pour la région car le polyèdre de $Z^{nv(ri)}$ associé aux assertions après élimination est plus grand que celui associé aux assertions primitives. Nous en déduisons que la région finale est d'autant meilleure que le nombre de variables contenues dans VNM_q est plus grand. Ceci montre la croissance de cette phase par rapport à VNM_q.

Enfin les deux ensembles $RCM(\omega_q)$ et $RCU(\omega_q)$, des régions modifiées et utilisées par ω_q, sont contruits par union sur NA des ensembles de régions élargies. Il est évident que ces deux ensembles seront d'autant meilleurs que NA est petit.

Cette phase est donc croissante par rapport à NA, VNM_q, fa_q et par rapport aux ensembles de régions calculées en (MA3).

X.3.3.4.6. Phase de rétrécissement des régions (MA6)

Cette phase ajoute à la description de chaque région des ensembles calculés par la phase précédente, un ensemble d'assertions résultant de l'analyse de de la déclaration de l'entité de la région, et de l'évaluation de certaines expressions bornes.

Les opérations utilisées pour cela sont l'union d'ensembles d'assertions et la fonction d'évaluation d'expressions, opérations croissantes.

Nous en déduisons que cette phase est croissante par rapport à fa_q, fi_q et par rapport aux ensembles de régions calculées en MA5.

X.3.3.4.7. Phase de traduction des régions (MA7)

Cette phase traduit les régions de ω_q, appartenant aux ensembles calculés en MA6, en des régions de ω_p. La méthode diffère sensiblement suivant que la région à traduire est une région formelle ou commune.

La traduction d'une région commune se ramène à la traduction d'un ensemble d'assertions linéaires, qui se réalise en trois phases.

(1) La première phase consiste à recenser l'ensemble des variables de $V_i(ri_q)$ de valeur substituable dans ω_p. Seules des considérations d'ordre syntaxique sur les associations des variables de $V_i(ri_p)$ et $V_i(ri_q)$ sont prises en compte pour ce recensement. Cette phase est donc constante.

(2) Les variables non substituables sont éliminées des assertions à traduire. L'élimination de variables est un opérateur croissant, cette phase est donc croissante par rapport à fa_q et par rapport aux ensembles de régions calculées en MA6.

(3) La substitution de la valeur de chaque variable substituable est effectuée dans chaque assertion à traduire, calulée en (2).

Cette substitution repose sur les mêmes mécanismes que le calcul des assertions initiales (phase IF3, fonction calcul_I_init).

Nous ajoutons aux assertions de description des régions obtenues en (2) un ensemble d'assertions déduites des associations de variables par common ou par passage de paramètres. Puis nous éliminons dans l'ensemble d'assertions résultant les variables de $V_i(ri_q)$. Cette substitution ne fait donc appel qu'à des opérateurs croissants, elle est donc croissante.

La traduction d'une région formelle utilise deux méthodes distinctes suivant les caractéristiques de l'association paramètre réel/paramètre formel. La méthode générale est constante par rapport aux ensembles de régions calculés en (MA6), puisque la traduction est identique quelles que soient les assertions de la région à traduire (cf. §VI.3.2.2). Quand à la méthode fine, elle est semblable à la méthode de traduction d'une région commune, et donc croissante par rapport à tous ses arguments.

Notons que fi_p,fa_p,fi_q et fa_q sont utilisées pour augmenter le nombre de cas où la traduction fine est utilisée. Cette utilisation consiste en des évaluations et comparaisons d'expressions bornes et d'expressions d'indices, opérations croissantes. Comme d'autre part la méthode fine ne peut être que meilleure que la méthode générale (cf. chapitre VI), nous en déduisons que la traduction d'une région formelle est une opération croissante par rapport à tous ses arguments.

X.3.3.4.8. Conclusion sur la croissance de la fonction calcul FM

Les sept phases de cette fonction sont croissantes par rapport à ses arguments et par rapport aux résultats des phases précédentes. Nous en déduisons donc que la fonction calcul_FM est croissante (composition de fonctions).

X.3.4. Croissance des fonctions calcul EP A, calcul EP M et calcul EP I

La croissance de ces fonctions est facilement démontrée. Chacune de ces fonctions peut être vue comme la composition d'un ensemble de fonctions croissantes -les fonctions calcul_FA, calcul_FI, calcul_I_init et calcul_FM- l'ordre de ces compositions étant

constant car donné par le graphe des occurences d'appels. La composition de fonctions croissantes étant croissante nous en déduisons que:

calcul_EP_I	est croissante	par	rapport	à	ep_a et ep_m,
calcul_EP_A	"	"	"	"	ep_i,
calcul_EP_M	"	"	"	"	ep_a et ep_i.

X.3.5. Convergence de l'algorithme itératif

X.3.5.1. Convergence des suites de fonctions ep a, ep i et ep m

L'algorithme itératif, que nous proposons au §X.2.4, calcule les trois suites de fonctions $(ep\_i)_k$, $(ep\_m)_k$ et $(ep\_a)_k$ k = 0, ... à partir de la fonction ep_i_o définie par

$$\forall \omega \in \Omega,\ \forall n \in N(OP\_RI(\omega)),\quad ep\_i_o(\omega)(n) = \emptyset$$

et des équations:

$$ep\_a_k = calcul\_EP\_A(ep\_i_k) \qquad (A)$$
$$ep\_m_k = calcul\_EP\_M(ep\_i_k, ep\_a_k) \qquad (M)$$
$$ep\_i_k = calcul\_EP\_I(ep\_m_{k-1}, ep\_a_{k-1}) \qquad (I)$$

Nous montrons par récurrence que ces suites sont croissantes. Supposons que $ep\_i_k \geq ep\_i_{k-1}$.

La croissance des fonctions calcul_EP_A, calcul_EP_M et calcul_EP_I garantit:

par (A) que	ep_a_k	$\geq ep\_a_{k-1}$
par (M) que	ep_m_k	$\geq ep\_m_{k-1}$
par (I) que	ep_i_{k+1}	$\geq ep\_i_k$

Il nous faut vérifier notre hypothèse de récurrence pour k = 1. La fonction ep_i_o est la plus mauvaise qui soit pour notre ordre puisque

$$\forall \omega \in \Omega,\quad S(ri)(\emptyset) = Z^{nv(ri)} \qquad \text{où}\quad ri = OP\_RI(\omega)$$

notre hypothèse de récurrence est donc vérifiée pour k = 1.

X.3.5.2. Limite de ces suites de fonctions

La croissance des trois suites de fonctions ep_a_k, ep_m_k et ep_i_k nous permet d'ores et déjà d'utiliser notre algorithme itératif en remplaçant le test du booléen NON-FINI par le comptage d'un nombre d'itérations, fixé à l'avance, éventuellement par l'utilisateur. Certains compilateurs proposaient autrefois une phase d'optimisation, dont le nombre d'itérations pouvait être choisi en fonction du degré d'optimisation souhaité et du temps CPU disponible.

Nous pouvons cependant faire mieux, et prouver que notre algorithme conduit, en un nombre fini d'itérations, à une limite. Dans la suite nous parlons de convergence pour convergence en un nombre fini d'itérations.

(1) La convergence de la suite $(ep\_a)_k$ est garantie car les fonctions ep_a_k sont construites sur des ensembles finis.

(2) La convergence de la suite $(ep\_i)_k$ se démontre par l'absurde. Etant donné que Ω est un ensemble fini et que le nombre de noeuds de chaque occurence de procédure est fini, une amélioration infinie de la suite $(ep\_i)_k$ impliquerait une amélioration infinie des assertions en au moins un noeud. Cette amélioration infinie serait due à l'amélioration infinie de la suite $(ep\_m)_k$ puisque la suite $(ep\_a)_k$ devient stationnaire en un nombre fini d'itérations.

Nous avons vu au §VII.1.2.2 que seul l'aspect destructeur des appels d'externe est pris en compte pour le calcul de la fonction d'assertion associée à une occurence de procédure. L'amélioration minimale d'une fonction de manipulation qui implique l'amélioration de la fonction d'assertion est donc la suppression d'un conflit avec une variable entière de l'occurence de procédure concernée. Le nombre de ces variables étant fini, les améliorations successives des fonctions de manipulation ne vont permettre d'améliorer le calcul des fonctions d'assertion qu'un nombre fini de fois.

Ceci implique qu'une amélioration infinie de la suite $(ep\_i)_k$ est impossible. Cette suite devient donc stationnaire en un nombre fini d'itérations.

(3) La convergence de la suite $(ep\_m)_k$ est garantie par la convergence des deux suites $(ep\_a)_k$ et $(ep\_i)_k$ en un nombre fini d'itérations.

L'algorithme itératif que nous proposons au §X.2.4, tout inefficace qu'il soit, permet d'atteindre une limite pour les trois suites de fonctions en un nombre fini d'itérations.

X.3.6. Conclusion sur la convergence de l'algorithme iteratif

Il nous faudrait maintenant montrer que le résultat ainsi obtenu est le meilleur qu'on puisse obtenir, sans modification des fonctions calcul_FI, calcul_FM et calcul_FA, et donc que l'ordonnancement simple des calculs que nous avons proposé dans ce chapitre n'a de conséquences qu'au niveau des performances. Nous n'avons malheureusement aucun résultat sur ce point.

Remarquons que la preuve de convergence n'a pu être obtenue que grâce au choix qui a été fait pour le calcul des assertions, de ne pas "remonter" d'information des occurences de procédure appelées vers l'occurence de procédure appelante. Revenir sur ce choix reviendrait sans doute à réintroduire le problème que P. Cousot résoud par son "élargissement supérieur" [Cous 78].

La réalisation que nous avons entreprise n'ayant pu être terminée, les besoins en temps CPU et en espace mémoire de nos algorithmes sont inconnus. Malgré la preuve de convergence, il faudra peut-être laissé le soin à l'utilisateur de décider du nombre d'itérations ...

CHAPITRE XI

XI. Conclusion

XI. Conclusion

Les trois propositions d'améliorations que nous avons formulées au chapitre III ont été effectivement explorées. La méthode que nous décrivons dans cette thèse permet de calculer pour chaque occurrence de procédure les éléments suivants:

(1) une fonction de manipulation associant à chaque appel d'externe les effets de son exécution sur les entités de l'occurrence de procédure;

(2) une fonction d'assertion associant à chaque noeud un ensemble d'assertions linéaires vérifiées par les variables simples entières du programme avant l'exécution de ce noeud;

(3) une fonction d'aliasing associant l'ensemble des couples d'entités qui sont en aliasing, et le type de cet aliasing;

Toute l'étude développée dans cette thèse est basée sur le langage Fortran-77 et lui est directement applicable. Nous avons en effet pris soin de traiter toutes les constuctions de ce langage, le document utilisé pour cela étant la norme AFNOR Fortran-77 [AFNO83].

Nous présentons des idées et algorithmes originaux: notion de région, d'occurrence de procédure, recherche d'assertions par étude des propriétés des boucles DO, calcul de l'aliasing, etc... D'autre part, nous présentons un grand nombre de méthodes existantes, généralement développées dans un cadre théorique, que nous avons pu adapter à notre cas particulier.

Un certain nombre de choix ont été effectués alors que nous envisagions les possibilités d'intégrer cette méthode dans un "outil de parallélisation intéractive de programme Fortran". Le dialogue "outil <-> homme" était alors favorisé: retour au programme source "commenté" par l'outil, introduction d'assertions par le programmeur dans le source, etc... Cet objectif était trop ambitieux, nous avons dû en changer, tout en conservant certains des choix qui avaient été faits. Ceci explique notamment que nous décrivons la mémoire en terme d'entités plutôt qu'en adresses.

Les informations que nous calculons peuvent, moyennant quelques modifications mineures, aider à la mise au point de programme. Elles peuvent aider à effectuer des vérifications généralement non faites par les compilateurs car interprocédurales: vérification des associations paramètre formel/paramètre réel en nombre, type, longueur (tableau formel de longueur supérieure à celle du tableau réel) et mode de passage (modification d'une constante ou d'une expression complexe, ...), vérification des longueurs des commons et des associations par common, vérification de l'absence d'aliasing, détection de certains débordements de tableau etc...

Ces mêmes informations peuvent renseigner le programmeur sur le fonctionnement de son programme, en lui donnant par exemple: l'arbre des appels, l'utilisation des commons par chaque procédure, la liste des procédures midifiant une entité (directement ou indirectement) etc...

L'état actuel de la réalisation est peu avancé. Elle fut au départ commencée sur la

totalité de Fortran-77; nous nous sommes ensuite restreints à un sous-ensemble de Fortran, augmenté de quelques constructions spécifiques: boucle DO parallèle, affectation conditionnelle, etc... Bien que travaillant sous Unix et ayant utilisé YACC, nous n'avons pas eu le temps de dépasser les deux phases d'analyse syntaxique, ASSIA et ASSIE.

Quelles suites pourrait recevoir notre travail? mis à part la poursuite de la réalisation du prototype qui permettrait de valider nos idées (temps d'exécution, encombrement mémoire, vitesse de convergence, ...), deux idées viennent à l'esprit.

En amont il faudrait étudier les modifications à apporter à un paralléliseur pour qu'il profite de nos résultats. Nous avons montré au chapitre VIII comment utiliser les assertions pour le calcul des dépendances. Le module sémantique (cf §III.1.4) gagnerait sans doute à utiliser la fonction d'assertion. Enfin rappelons que chaque triplet est associé à une occurrence de procédure, et qu'il faut regrouper les différentes occurrences de la même procédure en un ensemble de partitions, si on ne veut pas générer une version parallèle différente par occurrence. Une étude des différents critères de partitionnement est donc nécessaire (cf §IV.2.3).

En aval, il faudrait à nouveau se poser la question: "Qu'est ce qu'une information?"

Nous nous sommes restreints au domaine bien connu des assertions linéaires. Peut-on faire mieux? Nous nous sommes restreints aux variables simples. Peut-on faire mieux et arriver ainsi à traiter les accès aux tableaux dont les indices sont des accès aux tableaux dont les ... ?

Avant d'explorer de nouveaux types d'assertions, il nous parait nécessaire de proposer une méthode de calcul symbolique interprocédurale (extension des travaux de P. FEAUTRIER). Ceci nous permettrait d'epurer légèrement notre méthode, notamment en ce qui concerne l'élargissement d'une région.

A N N E X E

Présentation d'un exemple

A. Annexe: Traitement de l'exemple présenté dans l'introduction

L'implémentation que nous avons commencée n'étant pas suffisamment avancée pour permettre des démonstrations, nous montrons dans cette annexe le déroulement de la méthode que nous proposons sur l'exemple présenté dans l'introduction.

A.1. Corps de l'exemple

```
      PROGRAM TEST
      PARAMETER (LDA=100,LDB=100)
      PARAMETER (L=10,M=20,N=30)
      REAL MATA(LDA,LDA),MATB(LDB,LDB)
      ...
                                                <------ nt1
      CALL MM(MATA,LDA,L,N,MATB,LDB,M,MATB,LDB)          (ct1)
      ...
      END


      SUBROUTINE MM(A,LDA,N1,N3,B,LDB,N2,C,LDC)
      DIMENSION A(LDA,*),B(LDB,*),C(LDC,*)
                                                <------ nm1
      DO 10 I = 1,N3
                                                <------ nm2
10       CALL SMXPY(N2,A(1,I),N1,LDB,C(1,I),B)           (cm1)
      RETURN
      END
```

```
      SUBROUTINE SMXPY(N2,Y,N1,LDM,X,M)
      DIMENSION X(*),Y(*),M(LDM,*)
                                                <------ ns1
      DO 10 J=1,N1
                                                <------ ns2
         Y(J)=0
                                                <------ ns3
         DO 10 K = 1,N2
                                                <------ ns4
10          Y(J) = Y(J)+X(K)*M(J,K)
      RETURN
      END
```

A.2. Programme complet associé

Dans la suite, tous les noms d'objets relatifs a TEST (resp. MM,SMXPY) comportent un t (resp. m,s).

Ce programme se compose de trois procédures: P = { pt , pm , ps }

L'ensemble des commons est vide: G = ∅

Nous avons donc trois représentations internes: RI = { rit, rim, ris } avec

N(rit) = { nt1, ... } (les autres noeuds ne m'intéressent pas)
N(rim) = { nm1, nm2, ... } idem
N(ris) = { ns1, ns2, ns3, ns4, ... } idem

et

C(rit) = { ct1 } , C(rim) = { cm1 }

et

E(rit) = { MATA, MATB }
E(rim) = { A, B, C, LDA, LDB, LDC, N1, N2, N3, I }
E(ris) = { LDM, M, N1, N2, X, Y, J, K }

et

adr(rit)	adr(rim)	adr(ris)
MATA : (LOC,?,0..9999)	A : (FOR,?,1)	M : (FOR,?,6)
	B : (FOR,?,5)	
MATB : (LOC,?,10000..19999)	C : (FOR,?,8)	LDM : (FOR,?,4)
	LDA : (FOR,?,2)	X : (FOR,?,5)
	LDB : (FOR,?,6)	
	LDC : (FOR,?,9)	Y : (FOR,?,2)
	N1 : (FOR,?,3)	N1 : (FOR,?,3)
	N2 : (FOR,?,7)	N2 : (FOR,?,1)
	N3 : (FOR,?,4)	
		J : (LOC,?,0)
	I : (LOC,?,0)	K : (LOC,?,1)

La construction du graphe des occurrences d'appels conduit à Ω = { ωt, ωm, ωs } avec

$$C\_up(\omega t) = ? \ , \ C\_up(\omega m) = \omega t \ , \ C\_up(\omega s) = \omega m$$
$$\text{et} \quad C\_dw(\omega t, ct1) = \omega m \quad \text{et} \quad C\_dw(\omega m, cm1) = \omega s$$

Nous avons d'autre part:

$$OP\_RI(\omega t) = rit,\ OP\_RI(\omega m) = rim \quad et \quad OP\_RI(\omega s) = ris$$

A.3. Calcul de la liste de fonction d'aliasing

Le calcul de l'aliasing donne les mêmes résultats pour toutes les itérations. Nous détaillons ce calcul une seule fois.

aliasing associé à ωt (séquence d'initialisation)

$$ep\_ad(\omega t) = ? \quad et \quad \forall\ e,e' \in E(rit) \quad on\ a \quad ep\_a(\omega t)(e,e') = NON$$

aliasing associé à ωm

Nous commençons par calculer ep_ad(ωm). L'appel ctl a deux sortes de paramètres réels:

- des lhs d'entités réelles: MATA, MATB équivalents à MATA(1,1) et MATB(1,1)
- des expressions complexes: LDA, LDB, L, M, N qui sont évaluées dans des temporaires d'adresses (TMP,1), (TMP,2), ...

MATA et MATB sont des tableaux réels, les offsets de MATA(1,1) et MATB(1,1) sont connus. Ceci nous permet de calculer $ep\_ad(\omega_m)$;

```
A    :  (((PROC,pt),    0   ,  9999), VRAI)
B    :  (((PROC,pt), 10000  , 19999), VRAI)
C    :  (((PROC,pt), 10000  , 19999), VRAI)
LDA  :  (((TMP ,pt),     0  ,     0), VRAI)
LDB  :  (((TMP ,pt),     1  ,     1), VRAI)
LDC  :  (((TMP ,pt),     2  ,     2), VRAI)
N1   :  (((TMP ,pt),     3  ,     3), VRAI)
N2   :  (((TMP ,pt),     4  ,     4), VRAI)
N3   :  (((TMP ,pt),     5  ,     5), VRAI)
```

Nous calculons ensuite ep_a(ωm). Un seul conflit apparait, entre B et C; B et C n'ayant pas des déclarations identiques, nous en déduisons:

$$ep\_a(\omega m)(B,C) = PARTIEL$$

Nous voyons apparaître un défaut de notre solution: rappelons que pour cette phase, ainsi que la précédente, nous ne pouvons pas utiliser les assertions disponibles pour comparer les déclarations des deux tableaux formels puisque toute utilisation des assertions implique l'utilisation de l'aliasing, que nous sommes en train de calculer. Il faudrait pouvoir utiliser les assertions calculées à l'itération précédente ce que nous ne proposons pas.

La phase d'affinement ne donne rien pour ep_a(ωm).

aliasing associé à ωs

Le calcul de ep_ad(ωs) et ep_a(ωs) est fait de façon similaire. Nous verrons qu'aucune des fonctions d'assertions associées à ωm ne nous permet d'évaluer l'offset du paramètre réel C(1,I). Nous aboutissons donc à:

$$ep\_a(\omega s)(X,M) = PARTIEL$$

Récapitulatif

Pour toutes les itérations nous avons:

ep_a(ωt)	associe	NON	à tous les couples d'entités			
ep_a(ωm)	associe	PARTIEL	à (B,C)	et	NON	aux autres couples
ep_a(ωs)	associe	PARTIEL	à (M,X)	et	NON	aux autres couples

A.4. Itération 0

A.4.1. Calcul d'assertions

La fonction ep_i_o vérifie:

$$\forall\ \omega \in \Omega\ ,\ \forall\ n\ N(OP\_RI(\omega))\ \text{ on a }\ ep\_i_o(\omega)(n) = \emptyset$$

A.4.2. Calcul de l'aliasing

cf. §A.3.

A.4.3. Calcul des manipulations

A.4.3.1. Manipulations associées à ωs (séquence initiale)

$$ep\_m_o(\omega_s) = ?$$

A.4.3.2. Manipulations associées à ωm

Le calcul de ces manipulations implique l'analyse de ω_s:

MA1: L'ensemble des noeuds accessibles sera le même pour toutes les itérations, car les fonctions d'assertions ne permettront pas d'évaluer les expressions test. D'où

$$NAs = \{\ ns1\ ,\ ns2\ ,\ ns3\ ,\ ns4\ \}$$

MA2: Les fonctions N_mod(ris) et N_uti(ris), que nous n'avons pas explicitées, nous permettent de construire le tableau suivant:

	utilise	définit
ns1	(J,Ø), (N1,Ø)	(J,Ø)
ns2	Ø	(Y,{ ρ_1 = J }
ns3	(K,Ø), (N2,Ø)	(K,Ø)
ns4	(Y,{ ρ_1 = J }), (X,{ ρ_1 = K }), (M,{ ρ_1 = J, ρ_2 = K })	(Y,{ ρ_1 = J })

MA3: Cette phase ne change pas les résultats obtenus en MA2 puisque $ep\_i_o(\omega s)$ est la fonction d'assertion nulle,

MA4: Nous recherchons l'ensemble des variables initialement définies (donc communes en formelles) qui ne sont pas modifiées par ωs. Les seules variables "candidates" sont N1, N2 et LDM, aucune n'est modifiée, ni directement, ni par aliasing ou par équivalence. D'où

$$VNMs = \{ N1, N2, LDM \}$$

A noter que ce résultat est invariable au cours des itérations.

MA5: Les régions sont élargies, ce qui signifie dans ce cas que J et K doivent être éliminées des assertions de description. Nous obtenons, après union sur tous les noeuds:

$$RCU(\omega s) = \{ (N1,\emptyset), (N2,\emptyset), (Y,\emptyset), (X,\emptyset), (M,\emptyset) \}$$
$$RCM(\omega s) = \{ (Y,\emptyset) \}$$

A noter que les régions décrivant des parties d'entités non visibles dans l'occurrence de procédure appelante ωm (à savoir J et K) sont éliminées.

MA6: Nous introduisons dans les régions calculées en MA5, les assertions que nous obtenons grâce à l'étude de la déclaration des tableaux:

Pour X et Y nous obtenons $\{ 1 \leq \rho_1 \}$
Pour M nous obtenons $\{ 1 \leq \rho_1 \leq LDM, 1 \leq \rho_2 \}$

d'où

$RCU(\omega s) = \{ (N1,\emptyset), (N2,\emptyset), (Y,\{ 1 \leq \rho_1 \}), (X,\{ 1 \leq \rho_1 \}), (M,\{ 1 \leq \rho_1 \leq LDM, 1 \leq \rho_2 \}) \}$
$RCM(\omega s) = \{ (Y,\{ 1 \leq \rho_1 \}) \}$

MA7: La traduction ne pose pas de problème pour N1 et N2. Etudions en détails la traduction de la région (X,{ $1 \leq \rho_1$ }). Pour pouvoir utiliser la méthode fine de traduction d'une région formelle, il nous faut vérifier que:

(1) le nombre de dimensions de X est inférieur à celui de c: $(1 \leq 2)$

(2) les bornes inférieures de la 1ère dimension de X et C sont égales $(1 = 1)$

(3) pas de vérifications sur les bornes supérieures

(4) l'expression du 1er indice a la même valeur que la borne inférieure de la 1ère dimension de C (1=1)

(5) La 5ème condition concerne la valeur de la borne supérieure de la 1ère dimension de C, que nous ne connaissons pas. La condition "$\rho_1 \leq$ valeur(LDC)" dans la région (X,{ $1 \leq \rho_1$ }) n'est donc pas vérifiable.

La méthode fine n'est donc pas utilisable. Nous en déduisons que

$$ep\_m_o\ (wm)(cm1) = (\{(A,O)\}, \{(N1,\emptyset),(N2,\emptyset),(A,\emptyset),(B,\emptyset),(C,\emptyset)\})$$

manipulation de ωt

Nous n'en avons pas besoin pour la suite de l'énoncé. Nous donnons malgré tout le résultat:

$$ep\_m_o(\omega m)(ctl) = (\{\ (MATA,\emptyset)\ \},\ \{\ (L,\emptyset),\ (M,\emptyset),\ (N,\emptyset),\ (MATA,\emptyset),\ (MATB,\emptyset)\ \})$$

Remarquons que nous ne savons pas nous rendre compte que MATA n'est pas utilisé. Il faudrait pour cela étendre les techniques de "Data-Flow analysis" aux tableaux, ce qui n'est pas si simple!

A.5. Itération 1

A.5.1. Calcul d'assertions

A.5.1.1. Assertions associées à ωt

Nous ne connaissons pas la totalité du corps de la procédure TEST, ce qui n'est pas important car nous supposons que le calcul d'assertions dans ωt conduit à:

$$ep\_i_1(\omega t)(ntl) = \emptyset$$

A.5.1.2. Assertions initiales de ωm

L'analyse des associations de paramètres nous permet les assertions suivantes, contruites sur $V_i(rim) \cup V_i(rit)$:

$$\{\ LDA_m = 100,\ N1_m = 10,\ N3_m = 30,\ LDB_m = 100,\ N2_m = 20,\ LDC_m = 100\ \}$$

Nous leur ajoutons le contexte associé au noeud ntl, soit ∅ puis nous éliminons dans l'ensemble résultat les variables de $V_i(rit)$. Nous obtenons donc:

$$ep\_i_1(\omega m)(nml) = \{\ LDA = 100,\ N1= 1\ 0,\ N3 = 30,\ LDB = 100,\ N2 = 20,\ LDC = 100\ \}$$

A.5.1.3. Assertions associées à ωm

La méthode de propagation des constantes et la méthode des "bonnes boucles-DO" sont particulièrement efficaces dans ce cas puisque l'appel cml ne modifie aucune variable simple. D'où

$ep\_i_1(\omega_m)(nm2)$ = { LDA = 100, N1 = 10, N3 = 30, LDB = 100, N2 = 20, LDC = 100, $1 \leq I \leq N3$ }

A.5.1.4. Assertions initiales de ωs

L'analyse des associations de paramètres nous donnent les assertions suivantes construites sur $V_i(rim)$ U $V_i(ris)$;

$$\{ N2_m = N2_s, N1_m = N1_s, LDN_s = LDB_m \}$$

Après addition des assertions de $ep\_i_1(\omega_m)(nm2)$ et élimination des variables de $V_i(rim)$, soit { $N2_m$, $N1_m$, LDB_m }, nous obtenons:

$$ep\_i_1(\omega s)(ns1) = \{ N2 = 20, N1 = 10, LDM = 100 \}$$

A.5.1.5. Assertions associées à ωs

Comme pour ωm, la méthode de propagation des constantes et celle des "bonnes boucles-DO" donnent d'excellents résultats:

$ep\_i_1(\omega s)(ns2)$ = { N2 = 20, N1 = 10, LDM = 100, $1 \leq J \leq N1$ }
$ep\_i_1(\omega s)(ns3)$ = "
$ep\_i_1(\omega s)(ns4)$ = { N2 = 20, N1 = 10, LDM = 100, $1 \leq J \leq N1$, $1 \leq K \leq N2$ }

Etant donné qu'il n'y a aucun conflit région/variable dans les occurrences de procédure ωt, ωm et ωs, nous pouvons d'ores et déjà affirmer que ep_i_2 ne sera pas supérieure à ep_i_1 et que donc l'itération 2 est inutile.

A.5.2. Calcul de l'aliasing

cf. §A.3.

A.5.3. Calcul des manipulations

A.5.3.1. manipulations associées à ωs (séquence initiale)

$$ep\_m_1(\omega s) = ?$$

A.5.3.2. Manipulations associées à ωm

L'analyse de ωs diffère sensiblement par rapport à l'itération 0, puisque la fonction ep_i_1 est supérieure à la fonction ep_i_o.

MA1: rien de changé

MA2: rien de changé

MA3: La fonction d'assertion $ep\_i_1(\omega s)$ n'est pas nulle. Nous profitons donc de ce contexte local. Nous ne l'incluons que dans les régions décrivant des parties de tableaux:

	utilise
ns1	(J,Ø), (N1,Ø)
ns2	Ø
ns3	(K,Ø), (N2,Ø)
ns4	(Y,{ ρ_1 = J, N2 = 20, N1 = 10, LDM = 100, 1 ≤ J ≤ N1, 1 ≤ K ≤ N2 }), (X,{ ρ_1 = K, N2 = 20, N1 = 10, LDM = 100, 1 ≤ J ≤ N1, 1 ≤ K ≤ N2 }), (M,{ ρ_1 = J, ρ_2 = K, N2 = 20, N1 = 10, LDM = 100, 1 ≤ J ≤ N1, 1 ≤ K ≤ N2 }),

	définit
ns1	(J,Ø)
ns2	(Y,{ ρ_1 = J, N2 = 20, N1 = 10, LDM = 100, 1 ≤ J ≤ N1, 1 ≤ K ≤ N2 }),
ns3	(K,Ø)
ns4	idem ns2

MA4: rien de changé

MA5: l'élargissement simplifie les régions obtenues en MA3. Nous ne montrons le résultat que sur la région définit par ns2:

ns2 définit (Y, { N2 = 20, N1 = 10, LDM = 100, 1 ≤ ρ_1 ≤ N1 })

MA6: Le rétrécissement des régions ne donne pas de meilleurs résultats qu'à l'itération 0, car les trois tableaux X,Y,M sont de longueur non spécifiée. D'où

RCU(ωs) = { (N1,Ø), (N2,Ø), (Y{ N2 = 20, N1 = 10, LDM = 100,1 ≤ ρ_1 ≤ N1 })
(X{ N2 = 20, N1 = 10, LDM = 100, 1 ≤ ρ_1 ≤ N2 }),
(M{ ρ_1 ≤ LDM, N2 = 20, N1 = 10, LDM = 100, 1 ≤ ρ_1 ≤ N1, 1 ≤ ρ_2 ≤ N2 }) }

RCM(ωs) = { (Y{ N2 = 20, N1 = 10, LDM = 100, 1 ≤ ρ_1 ≤ N1 }),

MA7: Nous reprenons en détails la traduction de la région concernant X dans RCU(ωs). Les 4 premières conditions sont encore vérifiées. La vérification de la condition (5) était infaisable à l'itération 0, car la valeur de la borne supérieure de la 1ère dimension de C était inconnue. Son évaluation par la fonction X_ifovali avec le contexte $ep\_i_1(\omega m)(nml)$ fournit sans problème { LDC = 100 }.

Il nous suffit de prouver que ρ_1 reste inférieur à 100 dans la région concernée. Rappelons que ceci est fait en testant la faisabilité du système d'assertions associé à la région, auquel on ajoute l'assertion $\rho_1 > 100$.

Dans notre cas, le système associé à la région concernant le tableau X, contient les assertions { ρ_1 ≤ N2, N2 = 100 }. Le test de faisabilité répond donc INFAISABLE, prouvant ainsi que $\rho_1 \leq 100$.

La méthode fine de traduction s'applique donnant le résultat souhaité:

$$(C,\{\ 1 \leq \rho_1,\ N2 = 20,\ N1 = 10,\ LDB = 100,\ 1 \leq \rho_1 \leq N2,\ \rho_2 = I\ \})$$

d'où $ep\_m_1(\omega m)(cm1)$: (nous omettons les assertions N2 = 20, N1 = 10, LDB = 100)

région modifiée: $(A,\{\ 1 \leq \rho_1 \leq N1,\ \rho_2 = I\ \})$

régions utilisées: $(A,\{\ 1 \leq \rho_1 \leq N1,\ \rho_2 = I\ \})$,
$(B,\{\ \rho_1 \leq LDB,\ 1 \leq \rho_1 \leq N1,\ 1 \leq \rho_2 \leq N2\ \})$,
$(C,\{\ 1 \leq \rho_1 \leq N2,\ \rho_2 = I\ \})$

Nous arrêtons là l'analyse de cet exemple. Nous formulons en conclusion les remarques suivantes:

(1) les résultats obtenus pour le noeud nm2 au niveau aliasing, assertions et manipulations permettent de paralléliser la boucle de MM, en calculant le graphe des dépendances comme indiqué au §VIII.7. C'est bien le résultat anoncé dans l'introduction.

(2) 2 itérations sont nécessaires pour aboutir à ce résultat, la 3ème ne servant qu'à se rendre compte qu'il faut s'arrêter.

(3) De nombreux calculs étant identiques pour toutes les itérations, les algorithmes que nous proposons devraient être sérieusement optimisés, dans le cadre d'une implémentation.

(4) Toujours dans le cadre d'une implémentation, il faudrait sans doute introduire la notion d'assertions globales -Nous l'avions envisagé à une époque- C'est à dire vraies pour tous les noeuds d'une occurrence de procédure (par exemple dans ωm { N1 = 10, N2 = 20, LDM = 100 }).

BIBLIOGRAPHIE

References (i)

[Alle 74] F. E. Allen, "Interprocedural Data Flow Analysis," in Proceedings of the IFIP Congress, North-Holland publishing company. (1974).

[Alle 82] J. R. Allen and K. Kennedy, "PFC: A Program to Convert Fortran to Parallel Form," in Elaboratori paralleli calcolo scientifico, Roma (3-5 Mar 1982).

[Alle 83] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of Control Dependence to Data Dependence," in Proceedings of the 10th Annual ACM Symposium on Principles of Programming Languages (1983).

[Alle 84] F. E. Allen and K. Kennedy, "Automatic Loop Interchange," in Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction, Montreal (Jun 1984).

[Andr 83] G. R. Andrews and F. B. Schneider, "Concepts and Notations for concurrent Programming," ACM Computing Surveys Vol. 15(1), pp.3-43 (Mar 1983).

[Baer 73] J. L. Baer, "A Survey of some Theoritical Aspects of Multiprocessing.," ACM Journal Vol. 5(1), pp.31-80 (Mar 1973).

[Bake 77] B. S. Baker, "An Algorithm for Structuring Flowgraphs," ACM Journal Vol. 24(1), pp.98-120 (Jan 1977).

[Bill X] P. Billaud, J. C. Cottet, P. Feautrier, C. Renvoise, C. Ross, G. Rouquie, and D. Sciamma, "Génération automatique de programmes parallèles," in Rapport interne du Centre de Recherche C.I.I. Honeywell Bull. lot 1: Nov 1980, lot 2: Aug 1981, lot 3: Dec 1982, lot 4: Aug 1983

[Boss 84] A. Bossavit, "Le type abstrait vecteur et les méthodes de programmation des ordinateurs vectoriels," in Proceedings of the 6th International Symposium on Programming, Toulouse (Apr 1984).

[Brod 82] B. Q. Brode, VAST: A Vectorization Tool for the CYBER-205, Pacific-Sierra Research Corporation (Jun 82). Internal Report

[Coop 84] K. D. Cooper and K. Kennedy, "Efficient Computation of Flow Insensitive Interprocedural Summary Information," in Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction, Montreal (Jun 1984).

[Cott 84] J. Cottet, C. Renvoise, and D. Sciamma, "Vesta: Vectorisation automatique et paramétrée de programmes," in Proceedings of the 6th International Symposium on Programming, Toulouse (Apr 1984).

[Cous 78] P. Cousot, "Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes," in Thèse d'état, Grenoble I.N.P. (1978).

[Crow 83] T. R. Crowley, "Array Features in Fortran 8X.," in Actes de la Conférence Internationale OUTILS, METHODES ET LANGAGES ADAPTES AU CALCUL SCIENTIFIQUE (Paris) (May 1983).

[Cytr 82] R. G. Cytron, "Improved Compilation Methods for Multiprocessors," UIUCDCS-R-82-1088, University of Illinois at Urbana-Champaign (D). Technical Report

(i) Attention ! toutes les références ne sont pas triées

[Cytr 83a] R. Cytron, "Augmenting Fortran-66 to Support CEDAR Constructs," in CEDAR Document no 14 (Aug 1983).

[Cytr 83b] R. Cytron, D. Gajski, D. J. Kuck, D. Lawrie, and A. Sameh, "The Architecture and Programming of the CEDAR System," in CEDAR Document no 21 (Aug 1983).

[Davi 81] J. R. B. Davies, "Parallel Loop Constructs for Multiprocessors," UIUCDCS-R-81-1070, University of Illinois at Urbana-Champaign (D). Technical Report

[Dong 83] J. J. Dongarra and R. E. Hiromoto, "A Collection of Parallel Linear Equations Routines for the Denelcor HEP," 15, Argonne National Laboratory, Argonne Illinois (D).

[Evan 83] D. J. Evans, "The Parallel Solution of Triangular Systems of Equations," IEEE TOC Vol. 32(2) (Feb 1983).

[Feau 84] P. Feautrier, "Projet Vesta: Outil de calcul symbolique," in Proceedings of the 6th International Symposium on Programming, Toulouse (Apr 1984).

[Fish 84] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau, "Parallel Processing: A Smart Compiler and a Dumb Machine," in Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction, Montreal (Jun 1984).

[Gajs 83] D. Gajski, D. J. Kuck, D. Lawrie, and A. Sameh, "CEDAR: Construction of a Large Scale Multiprocessor," in CEDAR Document no 5 & 6 (Feb 1983).

[Gord 79] M. J.C. Gordon, The Denotational Description of Programming Languages, Springer-Verlag New-York Heidelberg Berlin (1979).

[Gott 83] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer: Designing an MIMD Shared Memory Parallel Computer," IEEE TOC Vol. 32(2) (Feb 1983).

[Halb 79] N. Halbwachs, "Détermination automatique de relations linéaires vérifiées par les variables d'un programme," in Thèse 3ème cycle, Grenoble I.N.P. (1979).

[Hans 78] B. Hansen, "Distributed Processes: A Concurrent Programming Concept," ACM Communications Vol. 21(11) (Nov 1978).

[Hech 77] M. S. Hecht, Flow Analysis of Computer Programs, North-Holland (1977).

[Hoar 78] C. A. R. Hoare, "Communicating sequential processes.," ACM Communications Vol. 21(8), pp.666-677 (Aug 1978).

[Jonh 75] D. B. Jonhson, "Finding all the Elementary Circuits of a Directed Graph," SIAM Journal on Computing Vol. 4, pp.77-84 (1975).

[Kam 76] J. B. Kam and J. D. Ullman, "Global Data Flow Analysis and Iterative Algorithms.," ACM Journal Vol. 23(1), pp.158-171 (Jan 1976).

[Kell 73a] R. M. Keller, "Parallel Program Schemata and Maximal Parallelism I: Fundamental Results," ACM Journal Vol. 20(3), pp.514-537 (Jul 1973).

[Kell 73b] R. M. Keller andG. A. Kildall, "Global Expression Optimization During Compilation.," ACM Journal Vol. 20(4), pp.696-710 (Oct 1973).

[Knig 83] J. C. Knight and D. D. Dunlop, "On the Design of a Special-purpose Scientific Programming language," Software-Practice and Experience Vol. 13(10), pp.893-907 (Oct 1983).

[Kuck 72] D. J. Kuck, "On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup," IEEE TOC Vol. 21(12) (Dec 1972).

[Kuck 74] D. J. Kuck, "Measurement of Parallelism in Ordinary FORTRAN Programs," Computer Vol. 7(1) (Jan 1974).

[Kuck 79a] D. J. Kuck, "Time and Parallel Processor Bounds for Fortran-Like Loops," IEEE TOC Vol. 28(9) (Sep 1979).

[Kuck 79b] D. J. Kuck, "High-Speed Machines and Their Compilers," in Proceedings of the International Conference on Parallel Processing (1979).

[Kuck 80a] D. J. Kuck, "High-Speed Multiprocessors and Compilation Techniques," IEEE TOC Vol. 29(9) (Sep 1980).

[Kuck 80b] D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe, "The Structure of an Advanced Vectorizer for Pipelined Processors," in Proceedings of the 4th International Conference on Computer Software and Applications (Oct 1980).

[Kuck 81a] D. J. Kuck, "Automatic Program Restructuring for High-Speed Computation," pp. 66-84 in Proceedings of COMPAR 81, Conf. on Analysing Problem-Classes and Programmingfor Parallel Computing, ed. W. Handler, Springer-Verlag (June 1981).

[Kuck 81b] D. J. Kuck, "Languages and Compilers for Parallel and Pipeline Machines," in Proceedings of the CREST Conference on Design on Numerical Algorithms for Parallel Processing, Bergamo, Italy (Jun 1981). Invited Paper

[Kuck 81c] D. J. Kuck, "Dependence Graphs and Compiler Optimizations," in Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages (1981).

[Kuck 82] D. J. Kuck, "High-Speed Multiprocessors and Their Compilers," in Parallel Processing Systems, ed. D. J. Evans, Cambridge University Press (1982).

[Lamp 74] L. Lamport, "The Parallel Execution of DO Loops," ACM Communications Vol. 17(2) (Feb 1974).

[Leas 76] B. R. Leasure, "Compiling Serial Languages For Parallel Machines," UIUCDCS-R-76-805, University of Illinois at Urbana-Champaign (D). Technical Report

[Lign ??] J. C. Lignac, "Présentation du CYBER+," in Actes de la journée parallélisme LRI-AFCET du 30 novembre 1984 (à paraitre).

[Lord 83] R. E. Lord, J. S. Kowalic, and S. P. Kumar, "Solving Linear Algebraic Equation on an MIMD Computer," ACM Journal Vol. 30(1), pp.103-117 (Jan 1983).

[Meis 83] L. P. Meissner, "Array Extensions to Fortan 77. A Critical Summary of Proposals.," in Actes de la Conférence Internationale OUTILS, METHODES ET LANGAGES ADAPTES AU CALCUL SCIENTIFIQUE (Paris) (May 1983).

[Mate 76] P. Mateti and N. Deo, "On Algorithms for Enumerating All Circuits of a Graph," SIAM Journal on Computing Vol. 5(1) (Mar 1976).

[Pato 69] K. Paton, "An Algorithm for Finding a Fundammental Set of Cycles of a Graph," ACM Communications Vol. 12(9) (Sep 1969).

[Perr 83] R. H. Perrott, D. Crookes, and P. Milligan, "The Programming Language ACTUS.," Software-Practice and Experience Vol. 13(4), pp.305-322 (Apr 1983).

[Ploy 81] F. Ployette, "Un langage pour la programmation parallèle: YEZHKEN," BIGRE Vol. 25 (sep 1981).

[Renv 83] C. Renvoise, "AMOS: Architecture Multi Opérateurs Sequentiels," in Rapport interne du Centre de Recherche C.I.I. Honeywell Bull (Jan 1983).

[Rouc 78] G. Roucairol, "Contribution à l'étude des équivalences syntaxiques et transformations de programmes parallèles.," in Thèse d'état, PARIS VI (1978).

[Rouc 82] G. Roucairol, "Transformations of Sequential Programs into Parallel Programs," in Parallel Processing Systems, ed. D. J. Evans, Cambridge University Press (1982).

[Shos 81] R. Shostak, "Deciding Linear Inequalities by Computing Loop Residues," ACM Journal Vol. 28(4), pp.769-779 (Oct 1981).

[Sush 83] J. Sushil, L. Jian, and A. Peter, "A Scheme of Parallel Processing for MIMD Systems," IEEE TOC Vol. SE-9 (Jul 1983).

[Suzu 77] N. Suzuki, "Implementation of an Array Bound Checker," in Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages (1977).

[Tai 82] K. Tai, "Comments on Parameter Passing Techniques in Programming Languages," SIGPLAN Notices Vol. 17(2) (Feb 1982).

[Tarj 73] R. Tarjan, "Enumeration of the Elementary Circuits of a Directed Graph," SIAM Journal on Computing Vol. 2(3) (Sep 1973).

[Tayl 83] R. N. Taylor, "An Integrated Verification Environment," Software-Practice and Experience Vol. 13(8), pp.697-713 (Aug 1983).

[Tims 83] C. Timsit, "Le super ordinateur ISIS," in Actes de la journée parallélisme LRI-AFCET du 1er décembre 1983 (Dec 1983).

[Wedi 81] R. G. Wedig, "Dynamic Detection of Concurrency in DO-loops Using Ordering Matrices," 209, Stanford University (D). Technical Report

[Wolf 78] M. J. Wolfe, "Techniques for Improving the Inherent Parallelism in Programs," UIUCDCS-R-78-929, University of Illinois at Urbana-Champaign (D). Technical Report

[AFNO 83] S. AFNOR, Traitement de l'information: langage de programmation Fortran, norme française NF-Z-65-110, 1983.

[Dene 82] S. Denelcor, HEP Fortran 77 User's Guide, Feb 1982.

[Jung 83] J. P. Jung, "Un prototype d'analyseur sémantique pour un sous-ensemble de Pascal", in Thèse 3ème Cycle, Metz I.N.P, (1983)

[Mino 83] M. Minoux, "Programmation Mathématiques - théorie et algorithmes (tome 2)", , Dunod (1983)

[Kill 73] G. Killdal, " A Unified Approch to Global Program Optimization ", in Proceedings of the ACM Symposium on Principles of Programming Languages, Palo-Alto, Mass, (Oct 1973)

[Trio 84] R. Triolet, " Problèmes posés par l'expansion de procédure en Fortran-77 ", in rapport interne, Centre d'Automatique et d'Informatique de l'Ecole des Mines de Paris, à paraitre

RESUME

Pour accélérer l'exécution d'un programme par l'utilisation d'une machine parallèle, il est nécessaire d'effectuer une suite de transformations destinées à l'adapter à la machine cible. L'axe de recherche visant à automatiser ces transformations s'est particulièrement développé ces dernières années, ce qui a conduit à la réalisation de compilateurs-vectoriseurs universitaires ou commerciaux performants (chapitre II).

Le problème posé par les instructions contenant des appels de procédure n'est cependant pas résolu par ces outils, si ce n'est par la méthode de l'expansion. Pourtant, des travaux récents ont montré que l'exécution parallèle de plusieurs procédures pouvait être intéressante pour les machines mutiprocesseurs (chapitre III). Le travail présenté dans cette thèse est une contribution à la résolution de ce problème.

Nous montrons tout d'abord que l'analyse des propriétés d'une procédure peut être effectuée avec plus ou moins de précision suivant que l'on tient compte ou non des différents appels qui en sont faits. Ceci nous conduit à définir la notion d'occurence statique de procédure et de graphe des occurences statiques d'appels (chapitre IV).

Les algorithmes de parallélisation automatique ont besoin de connaitre les ensembles de variables lues et modifiées par les instructions du programme. Nous montrons comment construire ces ensembles pour une instruction contenant des appels, en recensant les variables manipulées par les procédures appelées et en les reportant dans la procédure appelante. Le traitement efficace des tableaux est obtenu par l'introduction des régions, objets permettant de décrire des parties de tableaux grâce à un ensemble d'inéquations linéaires. Ces régions sont construites à partir d'assertions vérifiées par les variables simples (chapitres V & VI).

Ces assertions sont calculables par différentes méthodes que nous présentons. Celles-ci doivent être adaptées afin de prendre en compte aussi bien les problèmes spécifiques de Fortran-77 que les problèmes généraux des langages, notamment les appels de procédure (chapitre VII).

Principalement utilisées dans les régions, les assertions peuvent servir à d'autres fins: recherche de branches mortes, calcul de l'aliasing, calcul du graphe des dépendances, ... Différents traitements sont donc nécessaires: comparaison et évaluation d'expressions, test de faisabilité d'un système, ... Plusieurs algorithmes sont présentés et comparés (chapitre VIII).

L'aliasing est un problème bien connu, lié aux procédures. Nous proposons une méthode originale pour le calculer, particulièrement adaptée à Fortran-77 (chapitre IX).

Ces trois traitements - recherche des variables manipulées par chaque instruction, recherche d'assertions et calcul de l'aliasing - sont effectués sur le programme complet selon un algorithme itératif que nous proposons et dont nous prouvons la convergence en un nombre fini de pas (chapitre X).

MOTS-CLE

PARALLELISME, PARALLELISATION, VECTORISATION, MULTIPROCESSEUR, APPEL DE PROCEDURE, ASSERTION, INEQUATION LINEAIRE, ALIASING