École des mines de Paris
Centre de Recherche en Informatique

Thèse
présentée
pour obtenir

le grade de Docteur en sciences
Informatique Temps Réel, Robotique et Automatique
de l'École des mines de Paris

par Mme **Béatrice CREUSILLET**

Sujet :

Analyses de régions de tableaux et applications

Date de soutenance : 5 décembre 1996

Lieu de soutenance : École des mines de Paris, 60, bd Saint-Michel, Paris

Devant le jury composé de :

MM.	Yves Rouchaleau	Président
	François Irigoin	Directeur
	Paul Feautrier	Rapporteur
	William Pugh	Rapporteur
	Patrice Quinton	Rapporteur
	François Bourdoncle	Examinateur
	Alain Darte	Examinateur
	Bernard Dion	Examinateur

ANALYSES DE RÉGIONS DE TABLEAUX ET APPLICATIONS

ARRAY REGION ANALYSES AND APPLICATIONS

Béatrice Creusillet

Décembre 1996 — *December 1996*

À mes parents, Françoise et Denis Apvrille

À mon mari, Jean Creusillet

Résumé

La complexité et la diversité des machines parallèles à mémoire distribuée ou à hiérarchie de mémoires rend leur programmation particulièrement délicate. La parallélisation automatique, qui consiste à transformer un programme séquentiel exprimé dans un langage de haut niveau en un programme parallèle, apparaît alors comme une solution d'avenir. Ce processus nécessite d'une part une analyse des dépendances pour exposer le parallélisme potentiel, et d'autre part des transformations de programme comme la privatisation de tableaux pour accroître la localité des références. Nous montrons dans cette thèse comment ceci peut être effectué de manière automatique à partir d'analyses de régions de tableaux.

Quatre types de régions de tableaux sont décrits. Les régions READ et WRITE résument les effets des instructions et des procédures, et sont utilisées dans le cadre de l'analyse des dépendances interprocédurales. Les régions IN contiennent les éléments de tableaux *importés* par l'instruction courante ; et les régions OUT les éléments *exportés* vers une utilisation ultérieure. Ces deux dernières analyses permettent de détecter la localité des calculs. Un nouvel algorithme de privatisation de tableaux permettant d'exploiter cette localité est présenté.

Ces analyses ne peuvent pas être effectuées de manière exacte à la compilation, et il est nécessaire de recourir à des approximations. Les analyses approchées inférieurement posent des problèmes particuliers lorsque le domaine choisi pour la représentation des ensembles d'éléments de tableaux n'est pas clos pour l'union. Nous proposons alors d'utiliser un critère d'exactitude calculable pour définir la sous-approximation à partir de la sur-approximation. Cette approche est appliquée à la représentation des ensembles d'éléments de tableaux sous forme de polyèdres convexes.

Les analyses de régions de tableaux sont bien adaptées aux analyses macroscopiques, au niveau des séquences d'instructions ou des procédures. Ceci nécessite de pouvoir traduire les régions de l'espace de nommage d'une procédure à delui d'une autre, malgré les différences entre les déclarations des variables (*array reshaping*). Nous présentons donc un nouvel algorithme de traduction permettant de traiter les différents problèmes dans un cadre linéaire unifié. Il apparaît comme une synthèse et une extension des techniques existantes.

Cette approche est implantée dans PIPS, le paralléliseur développé à l'Ecole des mines de Paris. Des expériences qualitatives sur des applications réelles ont permis de montrer la robustesse de cette implantation, ainsi que sa capacité à paralléliser des boucles contenant des appels de procédures et nécessitant une privatisation des tableaux à l'intérieur des boucles ou des procédures appelées.

Mots clefs : Parallélisation automatique, analyse sémantique, régions de tableaux, dépendances, privatisation de tableaux.

Abstract

Programming parallel computers with distributed or hierarchical memories is particularly difficult, because of their complexity and diversity. Automatic parallelization, which converts high level sequential applications into parallel programs appears as a promising solution. This technique requires an analysis of dependences to expose potential parallelism, and program transformations such array privatization to enhance the locality of references. We show in this thesis how this can be achieved using array region analyses.

Four types of array regions are described: READ and WRITE regions summarize the effects of instructions and procedures, and are used for the analysis of interprocedural dependences. IN regions contain the array elements *imported* by the current statement; whereas OUT regions contain the array elements *exported* towards the program continuation. These two analyses characterize the computation locality, and are the basis of a new privatization algorithm which exploits this locality.

Exact array regions cannot be computed at compile time, and approximations must be used instead. However, when the domain chosen for the representation of array element sets is not closed under set union, the definition of under-approximations is problematic. We therefore propose to use an exactness criterion to compute the under-approximation from the over-approximation. This technique is applied to the representation of array regions as convex polyhedra.

Array region analyses are well adapted to coarse grain analyses, at the level of instruction sequences or procedures. Array regions must then be translated from the name space of a procedure into another, which is particularly difficult in case of array reshaping. A new algorithm exclusively based on linear algebra is therefore presented; it appears as a synthesis and extension of existing techniques.

The whole framework is implemented in PIPS, the parallelizer developed et Ecole des mines de Paris. Qualitative experiments have shown its robustness, as well as its ability to parallelize loops containing procedure calls, and requiring array privatization inside loops or called procedures.

Key words: Automatic parallelization, semantic analysis, array regions, dependences, array privatization.

Remerciements

Je tiens à remercier ici le Professeur Robert MAHL, directeur du Centre de recherche en informatique de l'École des mines de Paris, pour m'avoir accueillie dans son laboratoire; et le Professeur Yves ROUCHALEAU, responsable de la formation doctorale *informatique temps réel, robotique et automatique*, qui me fait l'honneur de présider mon jury de thèse.

Les Professeurs Paul FEAUTRIER, William PUGH, et Patrice QUINTON m'ont fait le plaisir et l'honneur d'accepter de rapporter ma thèse. Leurs commentaires m'ont été d'une grande utilité pour rédiger la version finale de ce mémoire. Qu'ils trouvent ici l'expression de ma gratitude.

Je désire remercier plus particulièrement le Professeur William PUGH dont les commentaires et questions ont largement contribué à l'amélioration de la partie théorique de cette thèse.

Mes remerciements vont aussi à Messieurs François BOURDONCLE, Alain DARTE et Bernard DION pour avoir accepté de faire partie de mon jury.

Ce travail n'aurait pas été possible sans mon directeur de thèse, François IRIGOIN. Il m'a proposé un sujet déjà bien défini, mais suffisamment large pour laisser place à la recherche, et m'a laissé une grande liberté. Il m'a également poussée à étudier de manière plus approfondie la théorie sous-tendant les analyses de régions; et ses questions et les discussions que nous avons eues se sont toujours avérées très pertinentes pour la poursuite de mon travail. Je tiens également à le remercier de m'avoir incitée à publier mes travaux de manière régulière.

La rédaction du chapitre 4 de ce mémoire m'a été grandement facilitée par des discussions avec Pierre JOUVELOT, François BOURDONCLE, et les Professeur William PUGH et Jean SERRA, que je tiens à remercier ici. Pierre JOUVELOT et William PUGH m'ont par ailleurs été d'un grand secours pour toute la partie théorique de ma thèse.

Un grand merci à tous les membres de l'équipe PIPS. Corinne ANCOURT m'a notamment apporté une aide inestimable par sa connaissance des polyèdres convexes. J'ai aussi eu de nombreuses discussions passionnantes et passionnées avec Fabien COELHO sur les analyses de régions et leur représentations, et ceci m'a indubitablement aidée à bien séparer les problèmes. Ronan KERYELL a aussi fait preuve d'une grande disponibilité lors d'expériences avec PIPS ou de problèmes avec LaTeX. Merci aussi à un ancien membre de l'équipe, Denis BARTHOU, pour les discussions très intéressantes que nous avons eues sur les relations entre PAF et les analyses de régions. Enfin, Arnauld LESERVOT, qui travaillait sur PIPS au CEA, a fourni la partie de la bibliothèque linéaire sur les listes de polyèdres que j'ai utilisée pour mon implantation; il m'a fourni un support technique non négligeable.

Ma reconnaissance va également à l'ensemble des membres du Centre de recherche en informatique pour la gentillesse de leur accueil, et tout particulièrement à Madame Jacqueline ALTIMIRA, Mademoiselle Annie PECH-RIPAUD et Monsieur Jean-Marie KESSIS pour leur support technique.

Je tiens aussi à remercier le Professeur Michel LENCI pour m'avoir fait connaître le CRI, ainsi que Monsieur Henri DELEBECQUE qui a guidé mes premiers pas dans la recherche.

Enfin, les derniers mais non les moindres, je voudrais remercier ici mes parents, Françoise et Denis APVRILLE, et mon mari Jean CREUSILLET pour m'avoir soutenue lors de mes études. C'est un plaisir pour moi de leur dédier ce travail qui leur doit tant.

Acknowledgments

I would like to thank Professor Robert MAHL, director of the *Centre de Recherche en informatique*, who received me in his department; and Professor Yves ROUCHALEAU, responsible for the doctoral formation *informatique temps réel, robotique et automatique* for doing me the honor of presiding my thesis committee.

I am very grateful to Professors Paul FEAUTRIER, William PUGH and Patrice QUINTON who have accepted the task of reporting my thesis. This is an honor for me. Their comments have been very helpful for the redaction of the final version of my dissertation.

I would like to more particularly thank Professor William PUGH, whose challenging comments and questions played a great part in the enhancement of the theoretical part of this thesis.

My sincerest thanks also to François BOURDONCLE, Alain DARTE and Bernard DION who have accepted to be in my thesis committee.

This work would not have been possible without my advisor, François IRIGOIN. He proposed me an already well-defined subject, which left room for research and freedom. He also incited me to study more thoroughly the theoretical foundations of array region analyses. His questions and remarks always proved useful, as well as the discussions we had together. I am also grateful for his constant incitations to publish the current state of my work.

Writing Chapter 4 has been a difficult task. I got help from many people to whom I am truly thankful: François BOURDONCLE, Pierre JOUVELOT, and Professors William PUGH and Jean SERRA. Besides, I had very interesting and challenging discussions with Pierre JOUVELOT, William PUGH and my advisor about the whole theoretical part of this work.

Many thanks to all the members of the PIPS team for their friendly support. Corinne ANCOURT's invaluable knowledge about convex polyhedra proved very useful. I had several discussions with Fabien COELHO about array region analyses and their representation. Ronan KERYELL was always ready to help when it was necessary for experiments with PIPS or for problems with LaTeX. Thanks also to a former PIPS team member, Denis BARTHOU, for the interesting discussions we had about array region analyses and PAF. Arnauld LESERVOT, who used to work on PIPS at CEA, provided a part of PIPS linear library I use in my implementation; his support was very helpful.

I would also like to acknowledge the technical support provided by Jacqueline ALTIMIRA, Annie PECH-RIPAUD and Jean-Marie KESSIS.

I am also indebted to Professor Michel LENCI, who let me know about CRI, and to Henri DELEBECQUE who guided my first steps in the world of research.

Last, but not least, I would like to thank my parents, Françoise and Denis APVRILLE, and my husband Jean Creusillet, for their support all along my studies. It is a pleasure for me to dedicate this work to them, for it owes them so much.

Table des Matières
Table of Contents

Liste des Figures
List of Figures

Liste des Tables
List of Tables

I

INTRODUCTION

Chapitre 1

Contexte et Problématique

"S'il-vous-plait, pourriez-vous m'indiquer dans quelle direction je dois aller à partir d'ici ?
- Cela dépend essentiellement de l'endroit où vous voulez arriver, dit le Chat.
- Je me soucie peu de l'endroit..., dit Alice.
- Alors peu importe la direction, dit le Chat.
- ... à condition d'arriver quelquepart, ajouta Alice.
- Oh, vous pouvez être sûre de ça, dit le Chat, si vous marchez suffisamment longtemps. "

Alice au pays des merveilles (Lewis CAROLL*)*

1.1 Contexte

Malgré des progrès constants dans les performances des technologies CMOS qui sont à la base de la plupart des processeurs actuels, tant au niveau de la fréquence d'horloge que de la densité d'intégration, les machines séquentielles ne répondent plus à une demande toujours croissante en puissance et en mémoire; d'autant que les temps d'accès à cette dernière ne suivent pas cette progression, empêchant ainsi d'atteindre les performances crête des processeurs. Les *Grands Challenges* ne sont pas la seule raison de cette course aux performances : la simulation logicielle prend peu à peu le pas sur les expériences réelles dans de nombreux domaines comme l'automobile, l'aéronautique ou la formation des pilotes par exemple ; pour atteindre un niveau de précision acceptable, ces simulations requièrent un grand nombre de données d'entrée, qui doivent ensuite être traitées. L'avènement de l'ère de l'information et du multi-média nécessite également un traitement rapide de volumes de données très importants.

Pour résoudre ces problèmes, l'idée est d'effectuer ces traitements en parallèle. Les deux dernières décennies ont donc vu l'émergence d'architectures dites parallèles. Ce terme générique cache en réalité de grandes disparités. Les premières machines, comme le CRAY 1, étaient essentiellement des processeurs vectoriels, c'est-à-dire des unités de calcul appliquant une série d'opérations sur des vecteurs de données, un peu comme sur une chaîne de fabrication de voitures. Ces systèmes ont permis des gains de performance appréciables dans le domaine des applications scientifiques. Mais le modèle de parallélisme sous-jacent manque de généralité, et la tendance actuelle est à d'autres types d'architectures parallèles ; le parallélisme s'exprime alors soit au niveau du processeur :

- architecture VLIW (pour *Very Long Instruction Word*) où un grand nombre d'instructions est effectué simultanément;
- architecture MTA (pour *Multi Threaded Architecture*) où plusieurs flots d'instructions sont exécutés quasi simultanément;

soit en multipliant le nombre de processeurs. La mémoire est alors soit physiquement partagée (mais éventuellement divisée en bancs pour réduire les effets de contention) comme dans les CRAY XMP ou YMP; soit physiquement distribuée comme dans les IBM SP1 et SP2, mais avec un espace d'adressage qui peut être partagé (CRAY T3D ou CONVEX SPP-1000 par exemple). A ceci s'ajoutent éventuellement un ou plusieurs niveaux de caches qui masquent les latences dues aux accès mémoire, mais augmentent la complexité du modèle.

Le problème pour l'utilisateur final est alors de tirer parti de ces architectures de manière efficace, simple, et portable, étant donné la vitesse d'obsolescence de ces machines. Il a alors le choix entre deux approches :

- *Programmer dans un langage explicitement parallèle* comme OCCAM, CM FORTRAN [160], HPF [76], FORTRAN D [78], VIENNA FORTRAN [181], ... L'avantage est que cela permet au programmeur d'implanter des algorithmes intrinsèquement parallèles, et d'avoir un meilleur contrôle du parallélisme. Par contre, les programmes sont plus difficiles à concevoir : il faut parfois ajouter des synchronisations à la main, ainsi que des communications inter-processeurs ; il faut aussi généralement spécifier le placement des données sur les processeurs en tenant compte de la topologie du réseau, des caractéristiques des processeurs, de la hiérarchie mémoire, ... Les langages les plus récents, comme HPF, soulagent le programmeur d'une partie de ces tâches, qui sont alors effectuées par le compilateur.

  De plus, comme il n'existe pas de modèle universel de programmation parallèle, indépendant de toute architecture, le programme résultant a peu de chances d'être portable ...

- *Programmer dans un langage séquentiel, et utiliser un paralléliseur automatique.* Cette approche qui paraissait utopique il y a peu de temps encore, commence à donner des résultats encourageants, comme en témoignent des études récentes [5].

  L'avantage majeur de cette approche est de permettre au programmeur d'écrire ses programmes dans un langage séquentiel familier, de haut niveau, et donc de concevoir plus facilement son application, et d'en assurer la portabilité. Le langage de choix reste FORTRAN, essentiellement pour deux types de raisons :

  - Des raisons *historiques* tout d'abord. Il existe une vaste communauté de chercheurs et de numériciens habitués à programmer en FORTRAN ; le passage à d'autres langages se fait lentement, au rythme du renouvellement des générations. De plus, il existe un très grand nombre de gros codes scientifiques éprouvés — les fameux *dusty decks* — écrits en FORTRAN, optimisés à la main pour une architecture particulière, et peu commentés ; ils sont difficiles à maintenir, et à plus forte raison à paralléliser à la main.
  - Des raisons *techniques* ensuite. FORTRAN a été conçu pour être relativement facile à analyser et à optimiser de façon à concevoir des exécutables efficaces.

> Il est donc naturellement devenu la source des paralléliseurs, qui ne sont qu'un cas particulier d'optimiseurs.

Toutefois, de plus en plus de langages deviennent l'objet d'études similaires : C, C++, Lisp (ou autres langages fonctionnels), Prolog, ...

La parallélisation automatique n'a pas que des avantages. Tout ne peut pas être parallélisé, et ce pour des questions de décidabilité. Même si les paralléliseurs sont de plus en plus puissants, les problèmes qu'ils tentent de résoudre ne peuvent généralement pas l'être en un temps fini, et des solutions correctes mais non optimales doivent souvent être adoptées. Tout comme ils ont appris à écrire du code vectorisable, les programmeurs devront donc apprendre à écrire du code parallélisable ; ceci, bien sûr, ne résoudra pas le problème des *dusty decks* qui ne pourront être qu'imparfaitement optimisés. Certains algorithmes sont aussi intrinsèquement séquentiels ; un retour au modèle sous-jacent est alors nécessaire pour détecter un éventuel degré de parallélisme ; mais ceci dépasse de beaucoup les capacités des compilateurs actuels, même si certaines recherches vont dans ce sens [150].

Enfin, la parallélisation automatique reste une technique assez coûteuse. Il est difficile de connaître *a priori* la complexité des codes à analyser, et l'on est souvent tenté d'implanter les techniques les plus performantes. De plus, les prototypes de recherche reposent souvent sur des outils génériques — par exemple, le paralléliseur interprocédural PIPS [105, 116] développé à l'École des mines de Paris repose sur une bibliothèque d'algèbre linéaire — qui permettent d'expérimenter des méthodes variées, au détriment parfois de la vitesse de compilation. Le temps et l'expérience permettront de faire le tri entre ce qui est coûteux mais véritablement utile, et ce qui est inutile ; des compromis seront certainement nécessaires. Mais en l'état actuel des connaissances et avec le peu de recul dont nous disposons, force nous est d'envisager toutes les solutions possibles.

La parallélisation automatique ne fournira jamais la même souplesse que l'utilisation directe d'un langage explicitement parallèle. Mais nous pensons qu'à terme elle permettra de traiter de manière satisfaisante la plupart des cas courants, et que son usage ne peut donc que se développer. Des paralléliseurs commerciaux existent déjà (comme KAP [118], FORGE [15], et le compilateur d'IBM, XL HPF), et ils feront certainement un jour partie de l'offre logicielle standard sur toute machine parallèle.

1.2 Régions de Tableaux

Plusieurs problèmes se posent lors de la génération automatique de code pour machines parallèles à mémoire distribuée ou à mémoire partagée (mais à accès non uniforme) ou hiérarchisée. Il faut tout d'abord pouvoir détecter le parallélisme existant, soit entre les itérations des boucles, soit entre des sections séquentielles ; voire exposer du parallélisme par des transformations de programme. Et il faut exploiter et privilégier la localité des références pour éviter les accès distants et donc coûteux.

Dans beaucoup de codes scientifiques, les tableaux sont la principale structure de données utilisée en dehors des variables scalaires. Ils sont souvent initialisés et référencés dans des boucles ou des portions de programmes qui constituent l'essentiel du temps total d'exécution. Il est donc primordial de pouvoir analyser ces accès aux éléments de tableaux pour étudier les dépendances qu'ils engendrent, et pouvoir ultérieurement générer du code parallèle efficace.

> Par exemple, dans le programme de la figure 1.1 extrait d'une application réelle de l'ONERA, la boucle sur `J` ne peut être parallélisée si l'on ne prouve pas que d'une itération à l'autre il n'y a pas de dépendance entre les diverses références à des éléments du tableau `T`, soit directement dans la procédure `EXTR`, soit à travers les appels à la fonction `D`.
>
> De même, dans l'exemple de la figure 1.2 — extrait du programme du Perfect Club [25] `OCEAN` — la boucle la plus externe ne peut être parallélisée si l'on ne s'aperçoit pas qu'à chaque itération les éléments du tableau `WORK` ne sont utilisés qu'après avoir été définis. Si l'on suppose en plus qu'ils ne sont pas utilisés par la suite, allouer une copie de `WORK` à chaque itération[1] élimine les dépendances sur `WORK`, et la boucle peut être parallélisée s'il n'existe pas d'autre dépendance. Cette transformation est couramment appelée *privatisation de tableau* et est cruciale dans le processus de parallélisation [29].

Beaucoup de travail a déjà été effectué dans le domaine de l'analyse des programmes comportant des références à des éléments de tableaux. Les premiers travaux ont pendant longtemps concerné l'analyse des dépendances, c'est-à-dire des conflits mémoire pouvant exister entre deux références, indépendamment des éventuelles autres références pouvant apparaître entre leurs exécutions. Il est maintenant couramment admis que si cette analyse donne des résultats positifs indéniables, elle n'en est pas moins insuffisante du fait de son incapacité à tenir compte de l'ordre dans lequel les références sont effectuées, ou flot des données. Deux approches sont actuellement utilisées dans les prototypes de recherche :

- Analyse des dépendances directes, ou encore fonctions sources, dépendances de valeurs, ... Ces différents termes désignent les dépendances qui ne peuvent être construites transitivement à partir d'autres dépendances portant sur la même case mémoire. Ces analyses donnent des résultats précis, voire exacts [72]. Malheureusement les langage d'entrée du paralléliseur est généralement très restreint malgré de récents progrès. Et ces analyses sont très coûteuses.

- Analyse basée sur les régions de tableaux. Les analyses de régions consistent à collecter des informations sur la manière dont les éléments de tableaux sont lus et écrits par le programme. Ces analyses furent introduites par TRIOLET [161] pour étudier les dépendances interprocédurales. Ces analyses étant insensibles au flot de contrôle du programme, elles ne permettent pas une analyse du flot des éléments de tableaux. Toutefois, d'autres types de régions de tableaux ont été ultérieurement introduits, et sont effectivement utilisés dans des paralléliseurs pour effectuer des transformations requérant une certaine connaissance du flot des éléments de tableaux, comme la privatisation. Ces analyses sont beaucoup moins précises que celles basées sur les dépendances directes du fait des nécessaires approximations. Par contre elles ne nécessitent aucune restriction sur le langage d'entrée du paralléliseur, et sont beaucoup moins coûteuses.

[1] En réalité à chaque processeur.

```
      PROGRAM EXTRMAIN
      DIMENSION T(52,21,60)
      COMMON/CT/T
      COMMON/CI/I1,I2,IMAX,I1P1,I1P2,I2M1,I2M2,IBF
      COMMON/CJ/J1,J2,JMAX,J1P1,J1P2,J2M1,J2M2,JA,JB,JAM1,JBP1
      COMMON/CK/K1,K2,KMAX,K1P1,K1P2,K2M1,K2M2
      COMMON/CNI/L

      READ(NXYZ) I1,I2,J1,JA,K1,K2
      REWIND NXYZ
C
      IF(J1.GE.1.AND.K1.GE.1) THEN
         N4=4
         J1=J1+1
         J2=2*JA+1
         JA=JA+1
         K1=K1+1
         CALL EXTR(NI,NC)
      ENDIF
      END

      SUBROUTINE EXTR(NI,NC)
      DIMENSION T(52,21,60)
      COMMON/CT/T
      COMMON/CI/I1,I2,IMAX,I1P1,I1P2,I2M1,I2M2,IBF
      COMMON/CJ/J1,J2,JMAX,J1P1,J1P2,J2M1,J2M2,JA,JB,JAM1,JBP1
      COMMON/CK/K1,K2,KMAX,K1P1,K1P2,K2M1,K2M2
      COMMON/CNI/L
      L=NI
      K=K1
      DO 300 J=J1,JA
         S1=D(J,K  ,J,K+1)
         S2=D(J,K+1,J,K+2)+S1
         S3=D(J,K+2,J,K+3)+S2
         T(J,1,NC+3)=S2*S3/((S1-S2)*(S1-S3))
         T(J,1,NC+4)=S3*S1/((S2-S3)*(S2-S1))
         T(J,1,NC+5)=S1*S2/((S3-S1)*(S3-S2))
         JH=J1+J2-J
         T(JH,1,NC+3)=T(J,1,NC+3)
         T(JH,1,NC+4)=T(J,1,NC+4)
         T(JH,1,NC+5)=T(J,1,NC+5)
 300  CONTINUE
      END

      REAL FUNCTION D(J,K,JP,KP)
      DIMENSION T(52,21,60)
      COMMON/CT/T
      COMMON/CNI/L
C
      D=SQRT((T(J,K,L  )-T(JP,KP,L  ))**2
     1     +(T(J,K,L+1)-T(JP,KP,L+1))**2
     2     +(T(J,K,L+2)-T(JP,KP,L+2))**2)
      END
```

Figure 1.1: Extrait du programme AILE

```
do J = 1, N1, 2
   do I = 1, N2P
      II = I + I
      WORK(II-1) = ...
      WORK(II) = ...
   enddo
   do I = 1, N2P
      ... = WORK(I)
   enddo
   do I = 1, N2P
      ... = WORK(I+N2P)
   enddo
enddo
```

Figure 1.2: Exemple

Notre étude se situe dans cette dernière classe. En effet, si les analyses de régions de tableaux ont fait l'objet de très nombreux articles ces dix dernières années, la plupart d'entre eux concernaient le type de régions à calculer pour effectuer telle ou telle transformation, ou encore le choix de la représentation des ensembles d'éléments de tableaux en fonction du type d'application considéré, de la complexité souhaitée, de la prise en compte d'informations symboliques comme le contexte, ... Par contre, très peu d'études se sont penchées sur le cadre théorique de la sémantique des analyses de régions ; ou alors pour certaines représentations [37], et toujours pour les analyses approchées supérieurement [108]. Mais jamais indépendamment de toute représentation et pour des analyses approchées supérieurement *et* inférieurement, qui sont toutes deux nécessaires pour tenir compte de l'ordre des références aux éléments de tableaux comme le montre l'exemple suivant:

> Nous avons vu précédemment que pour paralléliser la boucle la plus externe de la figure 1.2, il faut vérifier que les références en lecture sont *couvertes* par les références en écriture à la même itération. Pour cela, lors d'une analyse en arrière, on collecte les références en lecture sous la forme de régions, et l'on enlève de ces ensembles d'éléments de tableaux les références en écriture rencontrées au fur et à mesure de l'analyse.
>
> Ici, la dernière boucle utilise les éléments `WORK(N2P+1:2*N2P)` ; en remontant dans le corps de la boucle sur `J`, on ajoute à cet ensemble les éléments utilisés par l'avant-dernière boucle, soit `WORK(1:N2P)` ; on obtient donc `WORK(1:2*N2P)`. Il faut ensuite enlever de cet ensemble les éléments initialisés par la première boucle, soit `WORK(1:2*N2P)`; on obtient ainsi l'ensemble vide.
>
> Les analyses exactes étant impossibles à effectuer en un temps raisonnable, il est nécessaire de recourir à des approximations. Ici, les ensembles d'éléments référencés en lecture sont des sur-approximations des ensembles réels. Comme retrancher un ensemble sur-estimé d'un autre n'aurait aucun sens, l'ensemble des éléments référencés en écriture que l'on retranche est forcément une sous-approximation de l'ensemble réel.

La première partie de cette thèse est donc consacrée à l'étude du cadre théorique des analyses de régions de tableaux. Nous montrerons notamment les difficultés que posent les sous-approximations, et nous proposerons une méthode permettant de calculer des solutions satisfaisantes. La sémantique dénotationnelle de plusieurs types de régions de tableaux nécessaires pour les analyses de dépendance et de localité sera alors présentée, ainsi que quelques analyses préliminaires. Les régions READ et WRITE que nous définirons expriment les effets des instructions et des procédures sur les éléments de tableaux ; elles ne prennent donc pas en compte l'ordre des références, et ne permettent pas de calculer le flot des éléments de tableaux. Nous introduirons donc également les régions IN et OUT qui permettent de résoudre ces problèmes.

La sémantique dénotationnelle étant indépendante de toute représentation des régions de tableaux, nous présenterons ensuite dans une deuxième partie le choix effectué dans le paralléliseur PIPS où quasiment toutes les analyses reposent sur les polyèdres convexes en nombre entier.

La troisième partie abordera alors les problèmes posés par la présence d'appels de procédure dans les programmes. Nous donnerons à cet effet un algorithme de traduction des régions de tableaux de l'espace de nommage d'une procédure dans celui d'une autre, algorithme étendant ceux existants par ailleurs.

Deux applications possibles des analyses de régions de tableaux seront ensuite présentés dans la quatrième partie : l'analyse des dépendance, et la détection des régions de tableaux privatisables, pour laquelle un nouvel algorithme sera présenté. De nombreuses autres applications seraient par ailleurs envisageables. Nous avons l'intention d'utiliser les régions IN et OUT pour effectuer l'allocation de mémoire lors de la compilation de spécifications de traitement du signal basées sur une forme à assignation unique dynamique. Dans les systèmes à passage de message, elles pourraient être utilisées pour calculer les communications précédant et suivant l'exécution d'un fragment de code. Dans les machines à hiérarchie de mémoire, elles fournissent l'ensemble des élément de tableaux qui sont utilisés ou réutilisés, et pourraient donc être pré-chargés (régions IN) ou gardés (régions OUT) dans les caches ; les éléments de tableaux qui n'apparaissent pas dans ces ensembles sont des temporaires, et devraient être traités comme tels. Dans les systèmes à tolérance de pannes dans lesquels l'état courant est sauvegardé régulièrement par un composant logiciel (technique de *checkpointing* [107]), les régions IN et OUT pourraient fournir l'ensembles des données qui seront réutilisées lors de calculs ultérieurs, et pourraient donc permettre de réduire la quantité de données à sauvegarder. D'autres exemples d'applications sont la vérification de spécifications ou la compilation de programmes dont les données ne tiennent pas en mémoire [137].

La dernière partie récapitulera les résultats obtenus, et analysera les perspectives ouvertes par nos travaux. Plusieurs annexes se situent également à la fin de ce mémoire : la première est une description détaillée du paralléliseur PIPS ; la deuxième donne la formalisation d'un détail de la première partie ; les suivantes récapitulent les notations et fonctions sémantiques présentées dans la première partie.

Étant donné la diversité des sujets abordés (théorie, implantation, applications), les travaux relatifs aux nôtres ne font pas l'objet d'une partie distincte, mais sont examinés au fur et à mesure des développements.

Enfin, pour des raisons de diffusion externe, les principaux chapitres de cette thèse sont rédigés en langue anglaise. Toutefois, en reconnaissance de l'origine française de ce travail, chaque partie est préfacée par un chapitre en langue française présentant sa problématique et résumant les principaux résultats.

Chapter 2

Context and Open Issues

"Would you tell me, please, which way I ought to go from here?
- That depends a good deal on where you want to get to, said the Cat.
- I don't much care where..., said Alice
- Then it doesn't matter which way you go, said the Cat.
- ... so long as I get somewhere, Alice added as an explanation.
- Oh, you're sure to do that, said the Cat, if you only walk long enough."

Alice in Wonderland (Lewis CAROLL*)*

2.1 Context

Despite constant improvements in the performances of the technologies involved in processors, sequential machines do not meet the current requirements of speed and memory space, more especially as memory access times do not follow the progression of processor speeds, and thus prevent reaching their peak performances. The famous *Grand Challenges* are not the sole reason for this increasing demand: Software simulation tends to replace experiments in many fields, such as motoring, aeronautic, or pilot formation for instance; but to reach a sufficient level of accuracy, these simulations require a large amount of input data, which have then to be processed! Beside the resolution of scientific problems, multimedia applications, which play an increasing role, also necessitate fast processing of large amounts of data.

To solve these problems, the idea is to make these computation in parallel. The last two decades have therefore seen the emergence of many parallel commercial machines. In fact, this generic term hides great disparities. The first machines, such as the CRAY 1, generally were vector processors, which could apply a series of computation to data vector, much like on an assembly line. The performance gains were perceptible in the field of scientific applications. But the underlying model is quite limited, and the current tendency shows its preference for other types of parallel architectures. The parallelism is then achieved either at the processor level:

- *Very Long Instruction Word* (VLIV) architectures execute a large number of instructions in parallel.
- In *Multi Threaded Architectures* (MTA), several instruction streams are executed almost at the same time in a single processor.

or by increasing the number of processors. The memory is then either physically shared amongst the processors (but possibly divided in several banks to avoid bottle necks), as in the CRAY XMP or YMP; or physically distributed on the different nodes of the machine as in the IBM SP1 and SP2, the address space being possibly shared (CRAY T3D or CONVEX SPP-1000 for instance). Sometimes one or several fast cache levels are also added to hide the latency due to memory accesses; but this increases the complexity of the architectural model, and has to be taken into account by programmers or compilers.

For the user, the problem is then to exploit theses architecture in a simple, efficient and portable way, this last criterion being necessary because of the fast obsolescence of parallel systems. Two approaches are available to implement an application:

- The first is to *program in an explicitly parallel language* such as OCCAM, CM FORTRAN [160], HPF [76], FORTRAN D [78], VIENNA FORTRAN [181], ... This allows a good control on parallelism, and is the best way to implement intrinsically parallel programs. However, such programs are difficult to build: Synchronization must be added by hand, as well as data placements, and inter-processor communications. The most recent languages, such as HPF, remove some of these tasks from the burden of the programmer, part of the work being done by the compiler.

  Another drawback of this approach is that, since there exists no universal parallel programming model, explicitly parallel programs have little chance to be fully portable to any possible parallel architecture.

- The second approach consists in *programming in a sequential language, and using an automatic parallelizing compiler*. The most recent results are quite encouraging [5].

  The major advantage of this method is that it provides a mean to build programs in a familiar, high level language; and thus facilitates the conception of applications, and ensures their portability. The language of choice remains FORTRAN, for two types of reasons:

  - *Historical reasons* first.There is a vast community of researchers and numericians which are used to FORTRAN, and the transition to other languages follows the rhythm of generation renewal. Moreover, there exists a large number of old, extensively tested programs — the famous *dusty decks* — written in Fortran, optimized for a particular architecture, and with few, if any, comments; they are difficult to maintain, and even more to parallelize by hand!
  - *Technical reasons* also. The conception of FORTRAN makes it relatively easy to analyze and optimize, in order to generate efficient executables. It has thus naturally become the source of parallelizers, which are particular instances of optimizing compilers.

  However, more and more languages become the object of similar studies: C, C++, Lisp-like languages, Prolog, ...

  But automatic parallelization is not necessarily the panacea. Everything cannot be parallelized, because of decidability issues. Even if parallelizers are more and more powerful, the problems they try to solve are very difficult, and sub-optimal solutions must often be adopted. But, in the very same way programmers have learnt how to write vectorizable programs, they will learn how to write parallelizable code. Of course, this does not solve the problem of *dusty decks*. Some algorithms are also intrinsically sequential; considering the underlying model is then necessary to detect a possible degree of parallelism; this is far beyond the capabilities of current parallelizers, even if some studies go in this direction [150].

Finally, automatically parallelizing programs remains an expensive technique. It is difficult to know in advance the complexity of the source code, and the temptation is to implement the best possible optimizations. Moreover, research prototypes often rely on generic tools — for instance, the interprocedural parallelizer developed at Ecole des mines (PIPS [104, 116]) relies on a linear algebra library — to allow the experimentation of various methods, sometimes at the expense of compilaton speed. Time and experience will decide between expensive yet necessary techniques, and unuseful ones; compromises will certainly by necessary. But in the current state of knowledge, all solutions must be considered.

Automatic parallelization will never provide the same flexibility as the direct use of explicitly parallel languages. But we believe that it will one day be mature enough to handle most current programs, and that its usage can only spread out. Some commercial parallelizers are already available (such as KAP [118], FORGE [15], XL IBM HPF compiler). And they will certainly be part of the standard software package for parallel systems in the future.

2.2 Array Regions

Several problems need to be solved when generating code for parallel machines with distributed, shared but with non-uniform access, or hierarchical memories. The existing parallelism has to be detected, either between loop iterations, or even between sequential sections. Some parallelism can also be exhibited by program transformations. And the locality of references must be exploited and promoted to avoid distant and thus expensive memory accesses.

In many scientific programs, arrays are the main data structure, apart from scalar variables. They are often initialized and referenced within loop nests or program fragments which amount for a great part of the total execution time. It is therefore of the utmost importance to analyze these array accesses to study the dependences they generate, and be able to generate efficient parallel code.

For instance, in the program of Figure 2.1 (which is taken from a real application from the French institute ONERA), the `J` loop cannot be parallelized if it is not proven that from one iteration to another there is no dependence between the several references to array `T`, either in the procedure `EXTR`, or through the calls to function `D`.

Also, in the example of Figure 2.2 — extracted from the Perfect Club benchmark [25] `OCEAN` — the outermost loop cannot be parallelized if we do not find out that at each iteration the elements of `WORK` are used only after being defined. If we assume that they are not used in the program continuation, we can allocate a different copy of array `WORK` to each iteration[1], hence eliminate the dependences

[1] In fact to each processor.

```
      PROGRAM EXTRMAIN
      DIMENSION T(52,21,60)
      COMMON/CT/T
      COMMON/CI/I1,I2,IMAX,I1P1,I1P2,I2M1,I2M2,IBF
      COMMON/CJ/J1,J2,JMAX,J1P1,J1P2,J2M1,J2M2,JA,JB,JAM1,JBP1
      COMMON/CK/K1,K2,KMAX,K1P1,K1P2,K2M1,K2M2
      COMMON/CNI/L

      READ(NXYZ) I1,I2,J1,JA,K1,K2
      REWIND NXYZ
C
      IF(J1.GE.1.AND.K1.GE.1) THEN
         N4=4
         J1=J1+1
         J2=2*JA+1
         JA=JA+1
         K1=K1+1
         CALL EXTR(NI,NC)
      ENDIF
      END

      SUBROUTINE EXTR(NI,NC)
      DIMENSION T(52,21,60)
      COMMON/CT/T
      COMMON/CI/I1,I2,IMAX,I1P1,I1P2,I2M1,I2M2,IBF
      COMMON/CJ/J1,J2,JMAX,J1P1,J1P2,J2M1,J2M2,JA,JB,JAM1,JBP1
      COMMON/CK/K1,K2,KMAX,K1P1,K1P2,K2M1,K2M2
      COMMON/CNI/L
      L=NI
      K=K1
      DO 300 J=J1,JA
         S1=D(J,K  ,J,K+1)
         S2=D(J,K+1,J,K+2)+S1
         S3=D(J,K+2,J,K+3)+S2
         T(J,1,NC+3)=S2*S3/((S1-S2)*(S1-S3))
         T(J,1,NC+4)=S3*S1/((S2-S3)*(S2-S1))
         T(J,1,NC+5)=S1*S2/((S3-S1)*(S3-S2))
         JH=J1+J2-J
         T(JH,1,NC+3)=T(J,1,NC+3)
         T(JH,1,NC+4)=T(J,1,NC+4)
         T(JH,1,NC+5)=T(J,1,NC+5)
 300  CONTINUE
      END

      REAL FUNCTION D(J,K,JP,KP)
      DIMENSION T(52,21,60)
      COMMON/CT/T
      COMMON/CNI/L
C
      D=SQRT((T(J,K,L  )-T(JP,KP,L  ))**2
     1     +(T(J,K,L+1)-T(JP,KP,L+1))**2
     2     +(T(J,K,L+2)-T(JP,KP,L+2))**2)
      END
```

Figure 2.1: Excerpt from program AILE

due to WORK. The loop can be parallelized if there exists no other dependence. This transformation is usually called *array privatization* and is determinant in the parallelization process [29].

```
do J = 1, N1, 2
   do I = 1, N2P
      II = I + I
      WORK(II-1) = ...
      WORK(II) = ...
   enddo
   do I = 1, N2P
      ... = WORK(I)
   enddo
   do I = 1, N2P
      ... = WORK(I+N2P)
   enddo
enddo
```

Figure 2.2: Example

A lot of work has already been done on the analysis of programs involving array references. The first studies were essentially devoted to the analysis of dependences, that is to say of the memory conflicts existing between two array references, independently of the other references which may be executed in between. It is currently admitted that, even is this analysis gives undeniable positive results, it is however insufficient because it does not take into account the order in which references are executed (or array data flow). Two approaches are presently used in research prototypes:

- The analysis of *direct dependences*, or source functions, value dependences, ... These various terms denote the dependences which cannot be transitively built from other dependences on the same memory cell. These analyses give precise, or even exact results [72]. Unfortunately, the source language is generally very restricted despite recent extensions; and these analyses are quite expensive.

- Analyses based on *array regions*. Array region analyses consist in collecting information about the way array elements are used and defined by the program. These analyses were introduced by TRIOLET [161, 163] to study interprocedural dependences. These analyses being flow insensitive, they do not allow an analysis of array data flow. However, other types of array region analyses have subsequently been introduced, and are effectively used in parallelizers to perform program transformations which require a knowledge of array data flow, such as array privatization. Such analyses are much less precise than those based on direct dependences, because of the necessary approximations. But they do not restrict the source language, and are far less expensive.

Our study belongs to the last class. Indeed, if array regions have been extensively investigated in the last decade, most studies were devoted to the type of analysis necessary to perform such program transformation; or to the representation of array element sets, the usual issues being the type of application, the desired complexity, or the handling of symbolic information such as the context. But very few studies have brooded over the theoretical framework underlying array region analyses; and only for a particular representation [37], and always for over-approximate analyses [108]. But never independently of any representation, and for under-approximate as well as over-approximate analyses, which are both necessary to take into account the order in which array references are executed, as shown in the following example:

> We have seen that in order to parallelize the outermost loop in Figure 2.2, we must check that all the read references are covered by written references at the same iteration. This requires a backward analysis, which collects read references in the form of array regions, and removes written references from these array element sets when they are encountered.
>
> Here, the last loop uses the elements `WORK(N2P+1:2*N2P)`; going backwards in the main loop body, we add to this set the elements used by the middle loop, `WORK(1:N2P)`; we thus get `WORK(1:2*N2P)`. We must then remove the elements initialized by the first inner loop (`WORK(1:2*N2P)`); this gives the empty set.
>
> Exact analyses are not decidable, and approximations are necessary. Here, the sets of read array elements are over-approximations of the actual sets. Since subtracting an over-approximation from another one would not make much sense, the subtracted set of written array elements is necessarily an under-approximation of the actual set.

The first part of this thesis is therefore devoted to the study of the theoretical framework underlying array region analyses. We will show the problems which arise for under-approximate analyses, and we will propose a method to compute satisfying solutions. The denotational semantics of several types of array regions necessary for dependence and locality analyses will then be presented, as well as some preliminary analyses. The READ and WRITE regions which will be defined represent the effects of statements and procedures upon array elements; but they do not take into account the order of references, and thereby do not allow array data flow analyses. We will thus also introduce IN and OUT array regions to solve these problems.

The denotational semantics is independent of any representation of array element sets. The second part will therefore present the choices made in the implementation of array regions in the PIPS parallelizer, where almost all the analyses are based on the theory of convex polyhedra.

The third part will then show how to tackle procedure boundaries. In particular, we will give a new algorithm to translate array region from the name space of a procedure into another; this algorithm extends previously existing ones.

Two possible applications of array region analyses will then be presented in the fourth part: Dependence analysis, and the detection of privatizable array regions for which an original algorithm will be given. The other possible applications are numerous. We intend to use IN and OUT regions for memory allocation when compiling signal processing specifications based on dynamic single assignment. In massively parallel or heterogeneous systems, they can also be used to compute the communications before and after the execution of a piece of code. For a hierarchical memory machine, they provide the sets of array elements that are used or reused, and hence could be prefetched (IN regions) or kept (OUT regions) in caches; the array elements that do not appear in these sets are only temporaries, and should be handled as such. In fault-tolerant systems where the current state is regularly saved by a software component (*checkpointing*[107]) IN or OUT regions could provide the set of elements that will be used in further computations, and thus could be used to reduce the amount of data to be saved. Examples of other applications are software specification verification or compilation of out-of-core computations[137].

The last part will summarize the main results of this study, and the perspective they open. Several appendices are also provided at the end of this thesis: The first is a detailed description of the PIPS parallelizer; the second gives the formalization of a technical detail of the first part; the next appendices summarize the notations and semantic functions also presented in the first part.

Because of the diversity of the subjects examined in this work (theory, implementation, applications), the presentation of the related works is not done in a separate chapter, but is spread amongst the different parts, according to the current developments.

Lastly, as an acknowledgment to the French origins of this work, each part is preceded by a foreword in French, presenting the involved issues and summarizing the main results.

II

THÉORIE

THEORY

Chapitre 3

Sémantique des Régions de Tableaux

(Résumé des chapitres 4, 5, et 6)

Comme nous l'avions annoncé dans l'introduction, cette partie est consacrée à l'étude du cadre théorique des analyses de régions de tableaux, et à la sémantique de quatre types de régions : READ, WRITE, IN et OUT, qui ont été introduites lors d'études antérieures [161, 162, 17, 65]. En effet, si de nombreuses études se sont intéressées à de telles analyses, peu se sont penchées sur les problèmes théoriques sous-jacents, et jamais d'une façon aussi complète et uniforme que nous le proposons ici.

Ce chapitre est un résumé des développements de cette première partie, et est organisé en fonction des trois chapitres en anglais la constituant. Une première section est consacrée aux problèmes théoriques posés par les analyses de régions indépendamment de leur sémantique. La section 3.2 présentera ensuite notre langage d'étude et les analyses préliminaires nécessaires à la définition de la sémantique des régions, qui sera présentée dans la section 3.3.

3.1 Régions Exactes ou Approximations ?

Plusieurs études [29, 79] ont montré la nécessité de recourir à des optimisations de programme puissantes pour pouvoir gérer la mémoire de manière efficace lors de la compilation pour des machines à mémoire distribuée ou à hiérarchie de mémoires. Par exemple, BLUME et EIGENMANN [29] ont démontré par l'expérience que la privatisation de tableaux pouvait augmenter de manière cruciale le degré de parallélisme potentiel des programmes séquentiels. Le principe de la privatisation de tableaux est de découvrir les régions de tableaux qui sont utilisées comme des temporaires dans les boucles, et peuvent donc être remplacées par des copies locales sur chaque processeur sans changer la sémantique du programme. Détecter ces régions de tableaux privatisables requiert une analyse précise du flot des éléments de tableaux.

Plusieurs algorithmes de privatisation ou d'expansion[1] on déjà été proposées [70, 130, 123, 166, 93, 17, 63]. Ils reposent sur différents types d'analyse du flot des éléments de tableaux. La première approche [70, 130] effectue une analyse exacte, mais à partir

[1] Une technique similaire pour les machines à moire partagée.

d'un langage source restreint, dit langage monoprocédural à contrôle statique. La plupart des autres méthodes utilise des analyses de régions de tableaux approchées, sur- ou sous-estimées selon un ordre prédéfini. Au contraire, la dernière approche [17, 63] consiste à calculer des solutions exactes tant que cela est possible, et à n'effectuer des approximations que lorsque l'exactitude ne peut être conservée.

Cette section montre les avantages de cette dernière méthode. Pour cela, nous montrerons tout d'abord les limites des analyses approchées dans le cas des régions approchées inférieurement. Nous présenterons alors notre solution et ses conséquences. Nous terminerons la présentation du cadre général de nos analyses par les propriétés que devront vérifier les opérateurs que nous emploierons.

3.1.1 Analyses sémantiques et régions de tableaux

Habituellement, les analyses sémantiques sont définies sur des treillis ; c'est-à-dire que les solutions appartiennent à un ensemble partiellement ordonné muni de deux opérateurs internes $\wedge$ et $\vee$ donnant pour toute paire d'éléments leur plus grande borne inférieure et leur plus petite borne supérieure dans le domaine considéré, et selon l'ordre partiel[2]. Dans ce cadre, les fonctions sémantiques sont parfaitement définies, même lorsqu'elles sont récursives, ce qui est le cas dès lors que le langage source possède des structures de boucle. La meilleure solution est alors donnée par le plus petit point fixe de la fonctionnelle correspondante. Toutefois, ces solutions ne sont généralement pas calculables en un temps fini, et ne sont pas toutes représentables en machine. Deux approches permettent d'éviter ces problèmes : restreindre le langage source (comme dans [70, 72, 159, 130]) ; ou effectuer des analyses approchées sur des programmes réels.

Pour les analyses de régions, les solutions exactes appartiennent au treillis $(\wp(\mathbb{Z}^n), \varnothing, \mathbb{Z}^n, \cup, \cap)$. Les analyses approchées supérieurement sont aussi définies sur des treillis : treillis des polyèdres convexes dans PIPS, treillis des sections régulières dans le cas des RSDs [37], ... Ainsi, tout comme la sémantique exacte, la sémantique approchée des régions de tableaux est parfaitement définie.

Toutefois, comme cela sera montré dans la section 4.1.3, les solutions des analyses sous-estimées de régions de tableaux posent des problèmes particuliers lorsque le domaine choisi pour la représentation des ensembles d'éléments de tableaux n'est pas clos pour l'union. Dans ce cas, la meilleure sous-approximation n'est pas définie. Le cas se produit avec des représentation très répandues comme les polyèdres convexes ou les RSDs. Ceci est illustré par la figure 3.1, qui donne plusieurs sous-approximations possibles et non comparables dans ces deux représentations. Une conséquence directe est que si l'on choisit un tel treillis, l'opérateur sous-estimant l'union ne peut être associatif[3]. Les fonctions récursives faisant intervenir cet opérateur ne sont donc pas continues ; et l'on ne peut définir leur plus petit point fixe, et donc en donner une meilleure définition.

Ce problème est connu dans le domaine de l'interprétation abstraite [33, 57], et pas seulement pour la sous-approximation des ensembles à coordonnées dans $\mathbb{Z}$. Une solution consiste à changer de domaine d'étude. Par exemple, dans le cadre des analyses

[2] Le lecteur pourra se référer à [27] pour plus de détails sur la théorie des treillis, et à [132, 88] pour une description plus complète des domaines sémantiques.

[3] Sauf si l'on prend l'intersection, mais on ne peut en attendre de résultats intéressants.

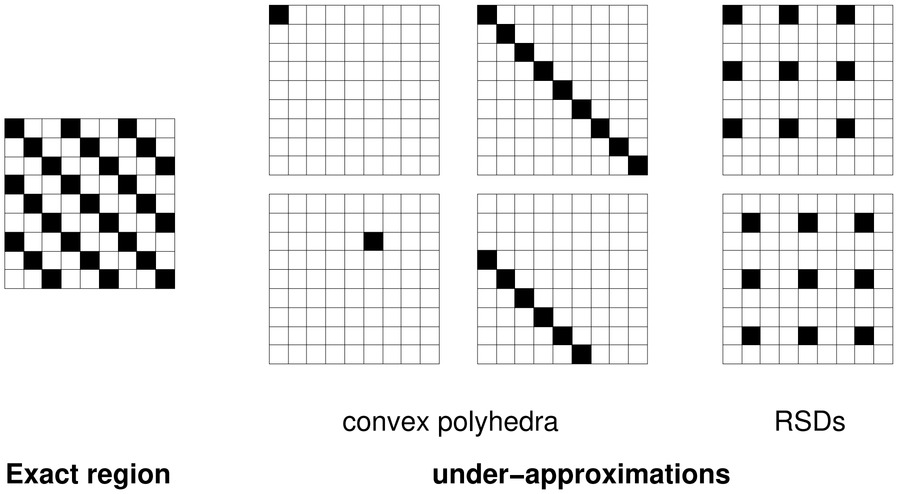


Figure 3.1: Sous-approximations possibles

de régions de tableaux, Wonnacott [176] propose d'utiliser le domaine des formules de Presburger qui est clos pour l'union. Mais si les résultats sont encourageants en terme de complexité dans un cadre monoprocédural, il faudrait plus d'expériences pour valider l'utilisation de cette représentation dans un cadre interprocédural. Bourdoncle propose deux autres solutions :

- Étendre le cadre habituel, dans lequel l'analyse approchée est liée à l'analyse exacte par une *fonction d'abstraction* α et une *fonction de concrétisation* γ (voir figure 3.2) possédant certaines bonnes propriétés ; notamment, α doit être monotone. La généralisation proposée consiste à lever cette dernière propriété, et à

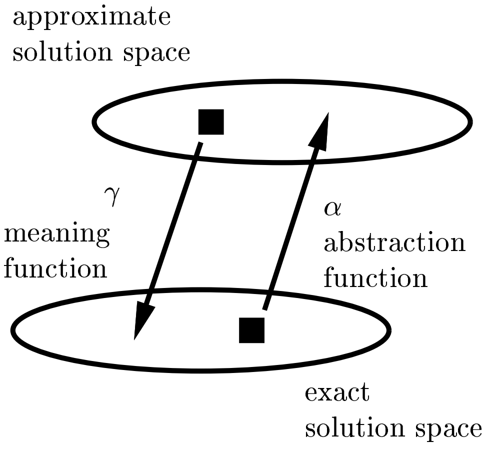


Figure 3.2: Connexion de Galois

doter le domaine des solutions approchées d'une séquence d'opérateurs de *rétrécissement* permettant d'obtenir une sous-approximation sûre en un temps fini. Une présentation plus formelle de cette méthode est proposée en annexe B. Ces limites sont illustrées par l'exemple suivant :

> Nous supposons que la séquence d'opérateurs de rétrécissement n'a qu'une seule itération : la région sous-estimant la région exacte correspondant à l'exécution d'une boucle est donc la région de la première itération de la boucle.
>
> Ainsi, pour la boucle la plus interne de la figure 3.3, la sous-approximation obtenue est l'ensemble {`A(I)`}, qui est identique à la région exacte. Si nous réitérons ce processus pour la boucle externe, nous obtenons {`A(1)`}, qui est une approximation très grossière de la région exacte {`A(1)`, `A(2)`, `A(3)`, `A(4)`, `A(5)`}.

```
do I = 1,5
   do J = 1,5
      A(I) =  ...
   enddo
enddo
```

Figure 3.3: Un petit exemple

Cet exemple est volontairement très simple. Avec une séquence de cinq opérateurs de rétrécissement, on obtiendrait le résultat souhaité. Mais que se passerait-il avec une borne supérieure de boucle égale à 10000, ou même symbolique ? Pourtant, dans tous ces cas, les régions exactes sont calculables, et leur exactitude est décidable [65]. Ce sera le point de départ de notre solution.

- La deuxième solution, appelée *partitionnement dynamique* [32], revient à déterminer au vol le treillis adéquat. Chaque région exacte est alors approchée inférieurement par une liste de sous-régions représentables. Mais nous pensons que dans le cas des analyses de régions cette méthode serait trop coûteuse, à la fois en temps et en mémoire.

3.1.2 Approximation et exactitude

Nous avons vu précédemment que, lorsque le domaine choisi n'est pas clos pour l'union ensembliste, les méthodes existantes d'approximation des analyses de régions pouvaient donner des résultats décevants, alors même que l'on sait pouvoir calculer la solution exacte sous certaines conditions [65]. Notre solution repose sur ce constat : nous proposons, à chaque étape de l'analyse, de calculer la sur-approximation ; de vérifier son exactitude à l'aide d'un *critère d'exactitude calculable* ; si ce critère est favorable, alors on peut choisir comme sous-approximation la région sur-estimée ; sinon, on peut utiliser toute autre méthode précédemment exposée.

> Ainsi, dans l'exemple de la figure 3.3, si les régions sont représentées par des polyèdres convexes, la région du corps de la boucle interne est $\{\Phi_1 : \Phi_1 = i\}$, Φ_1 étant le descripteur de la première dimension du tableau `A`.
>
> Une sur-approximation de la région de la boucle interne est alors obtenue en éliminant j du polyèdre précédant après lui avoir ajouté les contraintes induites par les bornes de boucle : $\{\Phi_1 : \Phi_1 = i \wedge 1 \leq j \leq 5\} \rightsquigarrow \{\Phi_1 : \Phi_1 = i\}$. Cette opération est exacte car les bornes de boucle ont des expressions affines et la projection est exacte selon [12, 144].

La sur-approximation obtenue pour la boucle externe est $\{\Phi_1 : 1 \leq \Phi_1 \leq 5\}$. En utilisant le critère d'exactitude précédant, on peut conclure que cette opération est exacte, et que la région est une sous-approximation valide. Le résultat serait encore exact avec des bornes non-numériques, mais toujours affines.

L'optimalité de cette approche est discutée dans la section 4.2.2 et dans [64].

Dans les autres paralléliseurs comme FIAT/SUIF [92], POLARIS [164] ou PANORAMA [135, 87], les analyses de régions de tableaux sous-estimées sont généralement basées sur un critère d'exactitude implicite qui consiste à éviter les opérations inexactes. Ceci nécessite bien sûr une représentation *a priori* infinie. En réalité, cette caractéristique n'est exploitée que pour les séquences d'instructions, pour lequel le nombre de références à des éléments de tableaux est borné par un nombre connu d'avance. Pour les boucles `do`, des solutions *ad hoc* sont utilisées. Par contre, les boucles générales sont mal traitées puisque l'opérateur d'union est approché inférieurement par l'opérateur d'intersection ; ce qui permet d'utiliser les techniques *normales* d'analyse de flot de données, $(\wp_{\text{convex}}(\mathbb{Z}^n), \varnothing, \cap)$ étant un semi-treillis.

Il faut noter aussi l'existence des travaux de PUGH et WONNACOTT [144, 171, 176], qui sur- et sous-estiment les parties de $\mathbb{Z}^n$ par des formules de PRESBURGER.

3.1.3 Opérateurs

La sémantique exacte des régions, qui sera présentée dans la section 3.2 et développée au chapitre 6, met en œuvre différents opérateurs : opérateurs internes, points fixes, et lois de composition externes. Si pour des analyses exactes, reposant sur un treillis, la définition de ces opérateurs ne pose pas de problème, il n'en est pas de même pour les analyses sur Z^n approchées, lorsque le domaine choisi n'est pas clos pour l'union ensembliste. De plus, de manière à éviter les détails dus à telle représentation particulière des régions, nous souhaitons utiliser des opérateurs approchés abstraits, leur instanciation se faisant au moment du choix effectif de la représentation. Ceci permet également de repousser les preuves de correction de la sémantique approchée au choix des opérateurs. Toutefois, pour assurer des approximations sûres, il convient d'imposer à ces opérateurs abstraits certaines propriétés.

Convention Les opérateurs et analyses sur-estimés seront désignés par des lettres et des symboles surlignés ; les sous-approximations seront désignés par des lettres et des symboles soulignés.

Opérateurs internes Les propriétés à imposer aux approximations des opérateurs d'union et d'intersection d'ensemble permettent simplement de garantir des approximations correctes, selon l'ordre choisi. Si $\boldsymbol{D}$ est le domaine des solutions exactes, $\boldsymbol{D}'$ et $\boldsymbol{D}''$ les domaines des solutions approchées supérieurement et inférieurement (tous deux inclus dans $\boldsymbol{D}$ pour simplifier la discussion), alors :

$$\forall a, b \in \boldsymbol{D}',\ a \cup b \sqsubseteq a \,\overline{\cup}\, b \text{ et } a \cap b \sqsubseteq a \,\overline{\cap}\, b$$
$$\forall a, b \in \boldsymbol{D}'',\ a \,\underline{\cup}\, b \sqsubseteq a \cup b \text{ et } a \,\underline{\cap}\, b \sqsubseteq a \cap b$$

C'est un peu plus difficile pour les opérateurs unaires et les opérateurs non commutatifs comme la différence, notée $\boxminus$, car cela n'a aucun sens de retrancher un ensemble sur-estimé d'un autre. Par contre une sur-approximation sûre est donnée par $\overline{a} \boxminus \underline{b}$. Dans ce cas, les éléments de $\boldsymbol{D}'$ et $\boldsymbol{D}''$ doivent pouvoir être combinés, et l'on a généralement $\boldsymbol{D}' = \boldsymbol{D}''$.

Points fixes Si les domaines des solutions approchées sont des treillis, les points fixes des fonctions définies sur ces treillis sont bien définies. Par contre si ce n'est pas le cas, nous devons utiliser les techniques décrites section 3.1.2. Pour éviter des notations trop lourdes, nous noterons $\overline{\mathrm{lfp}}(\Phi)$ et $\underline{\mathrm{lfp}}(\Phi)$ les sur- et sous-approximations du plus petit point fixe d'une fonctionnelle, quelque soit la manière dont elles sont calculées, et avec pour seule contrainte :

$$\Phi'' \sqsubseteq \Phi \sqsubseteq \Phi' \implies \underline{\mathrm{lfp}}(\Phi'') \sqsubseteq \mathrm{lfp}(\Phi) \sqsubseteq \overline{\mathrm{lfp}}(\Phi')$$

Lois de composition externes Comme nous le verrons dans les sections suivantes, les analyses de régions de tableaux reposent sur les résultats d'analyses précédentes, avec lesquels elles sont composées. Deux cas peuvent se présenter :

1. *Le codomaine*[4] *de l'analyse préliminaire est le même que le domaine de l'analyse de régions.* L'utilisation de la loi de composition usuelle ($\circ$) peut alors donner des solutions non représentables.

   Par exemple, si les régions sont représentées par des RSDs [37], alors la région de l'instruction (S) dans :

   ```
       if (i.eq.3) then
   (S) ...= A(i)
       endif
   ```

   est $A(i:i)$. Pour l'ensemble du programme, on doit composer ce résultat avec l'évaluation de la condition : **si** (i.eq.3) **alors** $A(i:i)$ **sinon** $\varnothing$. Mais ceci ne peut être représenté par un RSD, et la région doit être approchée supérieurement par $A(i:i)$ et inférieurement par $\varnothing$.

   Nous devrons donc, pour chaque paire d'analyses $(\mathcal{A}_1, \mathcal{A}_2)$ à composer, définir des lois de composition approchées $\overline{\circ}_{\overline{\mathcal{A}_1\mathcal{A}_2}}$ et $\underline{\circ}_{\underline{\mathcal{A}_1\mathcal{A}_2}}$, qui seront abrégées par $\overline{\circ}$ et $\underline{\circ}$ en l'absence d'ambiguïté, et vérifiant :

   $$\underline{\mathcal{A}_1} \,\underline{\circ}\, \underline{\mathcal{A}_2} \sqsubseteq \mathcal{A}_1 \circ \mathcal{A}_2 \sqsubseteq \overline{\mathcal{A}_1} \,\overline{\circ}\, \overline{\mathcal{A}_2}$$

2. *Le codomaine de l'analyse préliminaire n'est pas le même que le domaine des régions ($\boldsymbol{\Sigma}$, l'ensemble des états mémoire).* De nouvelles lois de composition doivent alors être définies. Le cas que nous rencontrerons sera celui où le codomaine de l'analyse préliminaire est l'ensemble $\wp(\boldsymbol{\Sigma})$ des parties de $\boldsymbol{\Sigma}$. Chaque élément du résultat de l'analyse préliminaire peut alors être composé avec la région considérée ; puis l'union ou l'intersection des régions obtenues fournit l'approximation souhaitée (voir la discussion détaillée page 58). Les lois de composition ainsi définies seront notées $\overline{\bullet}$ et $\underline{\bullet}$ selon le sens de l'approximation. Elles devront bien sûr vérifier, pour une paire d'analyses $\mathcal{A}_1, \mathcal{A}_2$:

   $$\underline{\mathcal{A}_1} \,\underline{\bullet}\, \underline{\mathcal{A}_2} \sqsubseteq \mathcal{A}_1 \circ \mathcal{A}_2 \sqsubseteq \overline{\mathcal{A}_1} \,\overline{\bullet}\, \overline{\mathcal{A}_2}$$

Ayant présenté le cadre de nos analyses, nous décrivons dans la section suivante notre langage d'étude et les analyses préliminaires nécessaires au calcul des régions de tableaux.

[4] Ou ensemble d'arrivée.

3.2 Analyses Préliminaires

Le langage source de PIPS est FORTRAN 77, avec quelques restrictions mineures. Mais pour ce mémoire, nous nous restreignons à un sous-ensemble du langage comportant les structures de contrôle principales (séquence, instruction conditionnelle, boucle `do`, `stop`), les affectations, les procédures, les expressions sans effet de bord et des instructions d'entrée/sortie simples. A ce sous-ensemble, nous avons choisi d'ajouter la boucle `do while` qui existe dans de nombreux langages et est à ce titre incontournable. La syntaxe retenue est fournie par la figure 5.1 (page 64). Elle met en œuvre un certain nombre de domaines sémantiques que nous utiliserons par la suite :

$\boldsymbol{C}$	constantes
$\boldsymbol{N}$	noms de variables, de procédures, ...
$\boldsymbol{R}$	références
$\boldsymbol{V}$	valeurs (entiers sur 8 bits, ...)
$\boldsymbol{S}$	instructions
$\boldsymbol{O}_1$	opérateurs unaires
$\boldsymbol{O}_2$	opérateurs binaires
$\boldsymbol{E}$	expressions
$\boldsymbol{E}_B$	expressions booléennes ($\boldsymbol{E}_B \subseteq \boldsymbol{E}$)

Nous utiliserons également le domaine des valeurs booléennes $\boldsymbol{B}$ et le domaine des états mémoire $\boldsymbol{\Sigma}$. Un état mémoire représente l'état courant des valeurs des variables, et est donc une fonction du domaine des références dans celui des valeurs : $\boldsymbol{\Sigma} : \boldsymbol{R} \longrightarrow \boldsymbol{V}$. Une instruction peut alors être vue comme une transformation d'un état mémoire en un autre, que nous appellerons un *transformeur*.

Dans ce chapitre, les fonctions sémantiques ne sont décrites que pour les affectations, les séquences d'instructions et les instructions conditionnelles, car ces constructions permettent de donner l'idée principale sous-tendant la sémantique des analyses, tout en conservant une complexité limitée, mieux adaptée à un exposé rapide. Le lecteur est renvoyé aux chapitres suivants pour les autres cas, notamment celui de la boucle.

3.2.1 Nécessité des analyses préliminaires

Notre but ultime est l'analyse des références aux éléments de tableaux afin d'effectuer d'autres analyses ou des transformations de programme comme la privatisation de tableaux. Cette opération n'est pas aussi simple que l'on pourrait le penser, du fait des caractéristiques du langage choisi. De petits exemples vont nous permettre de voir que des analyses préliminaires sont nécessaires.

Le premier exemple (figure 3.4) nous montre le problème principal des analyses de régions. Le but est ici de privatiser le tableau `WORK` sur la boucle `I`. Pour cela, l'une des conditions est que toutes les références en lecture sont précédées par une référence en écriture à la même itération. Il nous faut donc comparer la région correspondant à la deuxième boucle sur `J` avec celle correspondant à la première boucle sur `J`. La description de ces deux ensembles dépend forcément de la valeur de `K` ; il faut donc pouvoir modéliser la transformation de cette valeur par l'appel à `INC1`. Ce sera le rôle des *transformeurs* ou des *transformeurs inverses* selon le sens de la propagation.

```
K = FOO()
do I = 1,N
   do J = 1,N
      WORK(J,K) = J + K
   enddo
   call INC1(K)
   do J = 1,N
      WORK(J,K) = J*J - K*K
      A(I) = A(I)+WORK(J,K)+WORK(J,K-1)
   enddo
enddo
```

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

Figure 3.4: Exemple de travail.

Mais ceci peut être encore insuffisant pour comparer des régions de tableaux. Prenons comme exemple la fonction D du programme de la figure 1.1. Nous ne pouvons comparer ou combiner en un résumé les régions correspondant aux six références au tableau T, si nous ne connaissons pas les valeurs relatives de J et JP, et de K et KP. Ces informations peuvent être déduites à partir des trois sites d'appel : J=JP et KP=K+1. Ce type d'information est appelé *précondition* dans PIPS. La relation exacte avec les régions de tableau n'était pas claire jusqu'à présent et sera examinée dans la section 3.3.

Le dernier problème provient des instructions stop. Dans le programme de la figure 3.5, une approche naïve consisterait à ajouter A(K) à l'ensemble des éléments de tableaux référencés par le programme P_STOP. C'est une sur-approximation valide, quoique grossière du résultat attendu. Par contre, cela ne peut en aucun cas constituer une sous-approximation, puisque A(K) n'est référencé que si $K \leq 10$. Une analyse préliminaire, appelée *conditions de continuation*, est donc nécessaire pour vérifier les conditions sous lesquelles il existe effectivement un chemin d'exécution du début du programme à l'instruction considérée.

```
program P_STOP
integer A(5),K
...
call S_STOP(K)
A(K) = ...
...
end
```

```
subroutine S_STOP(K)
if (K.gt.10) then
   stop
endif
end
```

Figure 3.5: Problème de l'arrêt

Ces trois analyses préliminaires, *transformeurs*, *préconditions* et *conditions de continuation* sont décrites ci-dessous. Elles reposent toutes trois (ainsi que les analyses de régions) sur une autre analyse, l'*évaluation des expressions*.

3.2.2 Évaluation des expressions

La plupart des analyses reposent sur la représentation des expressions du programme : bornes de boucle, conditions de tests, indices de tableaux, ... Ces valeurs peuvent être fournies par une analyse que nous appelons *évaluation des expressions* et que nous désignerons par la lettre $\mathcal{E}$. Pendant l'exécution d'un programme, les expressions sont évaluées dans l'état mémoire courant. La sémantique d'une expression est donc une fonction du domaine des états mémoire $\mathbf{\Sigma}$ dans celui des valeurs $\boldsymbol{V}$:

$$\mathcal{E} : \boldsymbol{E} \longrightarrow (\mathbf{\Sigma} \longrightarrow \boldsymbol{V})$$

Cependant, tous les types d'expression ne peuvent être représentés lors d'une analyse statique. D'une part les domaines de valeur sont généralement très grands ; d'autre part, les valeurs des variables ne sont pas toutes connues à la compilation, et il devient alors nécessaire de traiter des expressions symboliques, ce qui est beaucoup trop complexe dans le cas général. On a donc recours à des approximations $\overline{\mathcal{E}}$ et $\underline{\mathcal{E}}$ dont le domaine cible restreint le type d'expressions symboliques que l'on peut représenter :

$$\overline{\mathcal{E}} : \boldsymbol{E} \longrightarrow \widetilde{\boldsymbol{Symb}}$$
$$\underline{\mathcal{E}} : \boldsymbol{E} \longrightarrow \widetilde{\boldsymbol{Symb}}$$

avec $\boldsymbol{Symb} = (\mathbf{\Sigma} \longrightarrow \boldsymbol{V})$, et $\widetilde{\boldsymbol{Symb}} \subseteq \boldsymbol{Symb}$. Nous renvoyons le lecteur à la section 5.3 pour le détail de la construction de ces fonctions.

$\mathcal{E}$ et ses approximations sont essentiellement utilisées pour évaluer les expressions d'indice de tableau, de bornes et de pas de boucle, ou encore les conditions des instructions `if` et `do while`. Dans ce dernier cas, il nous faudra composer le résultat de $\mathcal{E}$, $\overline{\mathcal{E}}$ ou $\underline{\mathcal{E}}$ (dont le codomaine est $\boldsymbol{V}$ ou un de ses sous-ensembles) avec d'autre analyses. Le domaine de la plupart de celles-ci étant $\mathbf{\Sigma}$, une composition directe n'est pas possible. Nous introduisons donc une autre analyse, qui est en réalité la fonction caractéristique de la restriction de $\mathcal{E}$ au domaine des booléens. nous la noterons $\mathcal{E}_c$ pour rappeler son lien avec $\mathcal{E}$, c signifiant *caractéristique*.

$$\begin{aligned} \mathcal{E}_c :& \boldsymbol{E}_B \longrightarrow (\mathbf{\Sigma} \longrightarrow \mathbf{\Sigma}) \\ & \quad \mathit{bool\_exp} \longrightarrow \mathcal{E}_c[\![\mathit{bool\_exp}]\!] = \lambda\sigma.\mathbf{if}\ \mathcal{E}[\![\mathit{bool\_exp}]\!]\sigma\ \mathbf{then}\ \sigma\ \mathbf{else}\ \perp \end{aligned}$$

$\overline{\mathcal{E}}_c$ et $\underline{\mathcal{E}}_c$ sont pareillement définies à partir de $\overline{\mathcal{E}}$ et $\underline{\mathcal{E}}$.

3.2.3 Transformeurs et transformeurs inverses

Les transformeurs décrivent la sémantique des instructions, vues comme des transformation d'un état mémoire dans un autre. Comme nous l'avons vu précédemment, deux types d'analyses sont en réalité nécessaires : les *transformeurs* modélisent la transformation de l'état mémoire d'entrée en un état mémoire de sortie ; et les *transformeurs inverses* modélisent la transformation inverse.

Transformeurs D'après la définition informelle que nous en avons donné, le prototype de cette analyse est le suivant :

$$\mathcal{T} : \boldsymbol{S} \longrightarrow (\mathbf{\Sigma} \longrightarrow \mathbf{\Sigma})$$

La sémantique de cette analyse est fournie dans la table D.1. Ces fonctions sémantiques ne sont généralement pas calculable, sauf pour des programmes très simples. Pour pouvoir traiter des programmes réels, il faut utiliser des approximations. Dans le cadre des analyses de régions, seule la sur-approximation nous intéresse (ceci sera détaillé dans la section 5.4.1). Pour tout état mémoire d'entrée, elle donne un ensemble d'états mémoire de sortie possibles, c'est-à-dire contenant l'état de sortie réel. Le domaine ciblé est donc celui des parties de $\mathbf{\Sigma}$. En réalité, ce sera un sous-ensemble noté $\tilde{\wp}(\mathbf{\Sigma})$, car il est impossible de représenter n'importe quel élément de $\wp(\mathbf{\Sigma})$ avec une machine finie :

$$\overline{\mathcal{T}} : \boldsymbol{S} \longrightarrow (\mathbf{\Sigma} \longrightarrow \tilde{\wp}(\mathbf{\Sigma}))$$

Ces fonctions sont fournies dans la table D.2 ; leur construction est détaillée dans la section 5.4.1.

Transformeurs inverses Cette analyse répond à la question : *étant donné une instruction et un état mémoire de sortie, quel est l'état d'entrée correspondant ?* C'est donc la fonction inverse de $\mathcal{T}$. Malheureusement, $\mathcal{T}$ n'est généralement pas inversible. Pour s'en convaincre, il suffit de considérer les instructions `K=5` ou `read K`. Ce problème est très similaire à celui du traitement des affectations dans la logique de HOARE [102, 124]. Et de la même manière, nous ne pouvons donner que des conditions nécessaires pour définir une analyse approchée :

$$\begin{aligned} \overline{\mathcal{T}}^{-1} :& \boldsymbol{S} \longrightarrow (\mathbf{\Sigma} \longrightarrow \tilde{\wp}(\mathbf{\Sigma})) \\ & S \longrightarrow \overline{\mathcal{T}}^{-1}[\![S]\!] : \forall \sigma \in \mathbf{\Sigma}, \exists \sigma', \sigma' \in \overline{\mathcal{T}}^{-1}[\![S]\!]\sigma \wedge \mathcal{T}[\![S]\!]\sigma' = \sigma \end{aligned}$$

Une définition complète de $\overline{\mathcal{T}}$ vérifiant ces conditions est donnée dans la table D.3.

3.2.4 Préconditions

Nous avons vu précédemment la nécessité de disposer d'informations sur la valeur relative des variables pour comparer ou combiner des régions de tableaux. Cette information est appelée *précondition* dans PIPS, car pour chaque instruction, elle fournit une condition qui est vraie avant son exécution. Nous pourrions donc définir les préconditions sous la forme d'une analyse booléenne. Toutefois, notre but étant de les composer avec des régions dont le domaine est celui des états mémoire, nous préférons définir cette analyse sous la forme d'un filtre d'état mémoire :

$$\mathcal{P} : \boldsymbol{S} \longrightarrow \mathbf{\Sigma} \longrightarrow \mathbf{\Sigma}$$

C'est strictement équivalent : le filtre est la fonction caractéristique de l'analyse booléenne, comme $\mathcal{E}_c$ est la fonction caractéristique de $\mathcal{E}$.

Pour de plus amples détails sur la définition de $\mathcal{P}$ et de sa sur-approximation $\overline{\mathcal{P}}$, nous renvoyons le lecteur à la section 5.6. C'est une analyse maintenant classique, fortement inspirée de l'algorithme fournit par COUSOT et HALBWACHS [59].

3.2.5 Conditions de continuation

Nous avons vu au début de cette section que la présence potentielle d'instruction `stop` dans un programme nécessitait une prise en charge particulière lors des analyses de régions de tableaux. On pourrait penser que les transformeurs fournissent l'information nécessaire, car l'instruction `stop` est modélisée par une fonction indéfinie, qui, lors de la propagation, est restreinte par les conditions du chemin d'accès à travers le programme.

Toutefois, cette information est généralement impossible à calculer, et les inévitables approximations masquent le résultat escompté. Il nous faut donc définir une nouvelle analyse donnant les *conditions de continuation* du programme. La fonction sémantique correspondante est aussi définie sous la forme d'une fonction de filtrage :

$$\mathcal{C} : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \boldsymbol{B})$$

ainsi que ses approximations, $\overline{\mathcal{C}}$ et $\underline{\mathcal{C}}$. Toutes ces fonctions sont décrites dans les tables D.6 et D.7.

3.3 Sémantique Intraprocédurale des Régions

Le concept d'analyse de régions fut introduit par TRIOLET [161] dans le but d'analyser les dépendances interprocédurales. Ses régions READ et WRITE représentent donc les effets en lecture et en écriture des instructions et des procédures sous la forme de résumés pouvant être ensuite utilisés au niveau des sites d'appel. Cependant, la méthode utilisée ne prend pas en compte le flot de contrôle et le flot des valeurs du programme, et les régions résultantes ne sont pas assez précises pour permettre des optimisations intraprocédurales. De plus, les régions READ et WRITE ne tiennent pas compte de l'ordre dans lequel les références aux éléments de tableaux sont effectuées, c'est-à-dire du flot des éléments de tableaux. Or, comme le montre l'exemple suivant, cela est nécessaire pour des optimisations de programme avancées comme la privatisation de tableaux.

> Dans le programme de la figure 3.4, des parties du tableau `WORK` sont réutilisées d'une itération sur l'autre (voir figure 3.4). Ceci induit des dépendances entre itérations, et empêche donc la parallélisation de la boucle sur `I`. Mais ce ne sont pas des dépendances directes car chaque référence en lecture est précédée par une référence en écriture à la même itération. En privatisant le tableau `WORK`, on élimine ces dépendances, et la boucle peut être parallélisée.

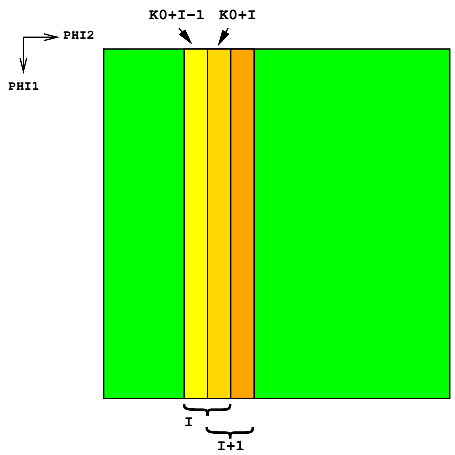


Figure 3.6: Dependances sur `WORK` dans le programme de la figure 3.4

> Avec les régions READ et WRITE du corps de la boucle externe, on ne peut pas s'apercevoir que chaque lecture est précédée d'une écriture à la même itération.

Cette information est donnée par une autre analyse, qui collecte les références en lecture qui ne sont pas *couvertes* par une référence en écriture dans le fragment de programme considéré. Dans notre cas, nous obtiendrions l'ensemble vide. Dans PIPS, cette analyse s'appelle analyse des régions IN.

Il reste encore un problème non résolu. Les régions IN fournissent l'ensemble des éléments de tableaux dont le calcul est local à une itération. Mais ces éléments peuvent éventuellement être réutilisés dans la suite du programme. Pour vérifier que ce n'est pas le cas, ou adopter une autre politique dans le cas contraire, nous avons introduit les régions OUT qui contiennent les éléments de tableaux définis par le fragment de programme courant *et* réutilisés par la suite.

Cette section décrit donc la sémantique de ces quatre types de régions (READ WRITE, IN et OUT), mais en tenant compte du flot de contrôle et du flot des valeurs du programme de manière à conserver, au moins localement, une précision suffisante.

3.3.1 Domaines et fonctions sémantiques

Une région d'un tableau T représente un ensemble de ses éléments, décrits par les valeurs que peuvent prendre ces indices, qui peuvent être vus comme des coordonnées dans $\mathbb{Z}^d$ si d est le nombre de dimensions de T. Une région peut donc être représentée par une partie de Z^d, et appartient de ce fait à $\wp(Z^d)$. Chaque dimension du tableau est décrite par un descripteur de région, noté Φ_i^T pour la i-ème dimension de T ; le vecteur des Φ_i^T sera noté Φ^T.

Le but des analyses de régions de tableaux est d'associer à chaque instruction (simple ou complexe) et à chaque état mémoire un ensemble d'éléments de tableaux. Si n tableaux sont accessibles dans la procédure analysée, de dimensions respectives $d_1, \ldots, d_n$, les fonctions sémantiques des analyses de régions de tableaux sont du type suivant :

$$\begin{aligned}\mathcal{R}: \quad & \tilde{\boldsymbol{\mathcal{L}}} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \textstyle\prod_{i=1,n} \wp(\mathbb{Z}^{d_i}))\\ & l \longrightarrow \mathcal{R}[\![l]\!] = \lambda\sigma.(\{\Phi^{T_1} : r_1(\Phi^{T_1}, \sigma)\}, \ldots, \{\Phi^{T_n} : r_n(\Phi^{T_n}, \sigma)\})\end{aligned}$$

r_i donnant la relation existant entre les coordonnées Φ^{T_i} des éléments de la région du tableau T_i et l'état mémoire σ.

Par exemple, `A(I)` pourrait être représenté par $\lambda\sigma.\{\Phi_1^A : \Phi_1^A = \sigma(\texttt{I})\}$.

Note Les notations ci-dessus semblent impliquer que nous utiliseront le formalisme de la sémantique dénotationnelle pour décrire nos analyses de régions de tableaux. Ce sera effectivement le cas pour les analyses en arrière, qui définissent les fonctions sémantiques de la structure globale à partir de celles des composants. Par contre, le formalisme des grammaires attribuées est mieux adapté à la description de l'implantation dans PIPS des analyses en arrière. Ce sera donc ce formalisme que nous utiliseront, tout en conservant les notations de la sémantique dénotationnelle (c'est-à-dire que le texte du programme sera fourni entre double-crochets).

Cependant, il n'est pas possible de représenter n'importe quelle partie de Z^d tout en conservant une complexité raisonnable. Une représentation particulière, sous-ensemble de $\wp(\mathbb{Z}^d)$ noté $\tilde{\wp}(Z^d)$, doit donc être choisie. Nous supposerons également que $\varnothing \in$

$\tilde{\wp}(Z^d)$ et $\mathbb{Z}^d \in \tilde{\wp}(Z^d)$. Nos analyses sur- et sous-estimées auront également le même domaine image, de façon à pouvoir les combiner facilement :

$$\begin{array}{ll}
\overline{\mathcal{R}}: & \tilde{\boldsymbol{\mathcal{L}}} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \tilde{\wp}(\mathbb{Z}^{d_i})) \\
 & l \longrightarrow \overline{\mathcal{R}}[\![l]\!] = \lambda\sigma.(\{\Phi^{T_1} : \overline{r}_1(\Phi^{T_1},\sigma)\},\ldots,\{\Phi^{T_n} : \overline{r}_n(\Phi^{T_n},\sigma)\}) \\
\underline{\mathcal{R}}: & \tilde{\boldsymbol{\mathcal{L}}} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \tilde{\wp}(\mathbb{Z}^{d_i})) \\
 & l \longrightarrow \underline{\mathcal{R}}[\![l]\!] = \lambda\sigma.(\{\Phi^{T_1} : \underline{r}_1(\Phi^{T_1},\sigma)\},\ldots,\{\Phi^{T_n} : \underline{r}_n(\Phi^{T_n},\sigma)\})
\end{array}$$

Rappelons que, comme nous l'avons vu dans la section 3.1, $\prod_{i=1,n} \tilde{\wp}(\mathbb{Z}^{d_i})$ ne peut pas, pour les types de représentations qui nous intéressent, être muni d'une structure de treillis dans le cadre de la sous-approximation.

3.3.2 Régions READ et WRITE

Les régions READ et WRITE représentent les effets du programme sur les variables : elles contiennent les éléments de tableaux lus ou écrits par les instructions (simples ou composées) lors de leur exécution. Ces deux types d'analyses étant très similaires, nous ne décrivons ici que l'analyse des régions READ.

$\mathcal{R}_r$, $\overline{\mathcal{R}}_r$ et $\underline{\mathcal{R}}_r$ définissent les sémantiques exactes et approchées des régions READ. Ces sémantiques étant définies aussi bien pour les expressions que pour les instructions, le domaine source est la somme disjointe de $\boldsymbol{E}$ et $\boldsymbol{S}$:

$$\mathcal{R}_r : \boldsymbol{S} \oplus \boldsymbol{E} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \wp(\mathbb{Z}^{d_i}))$$
$$\overline{\mathcal{R}}_r : \boldsymbol{S} \oplus \boldsymbol{E} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \tilde{\wp}(\mathbb{Z}^{d_i}))$$
$$\underline{\mathcal{R}}_r : \boldsymbol{S} \oplus \boldsymbol{E} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \tilde{\wp}(\mathbb{Z}^{d_i}))$$

Ces fonctions sont décrites dans les tables E.1, E.2 et E.3. Nous donnons ci-dessous le détail de la construction des plus importantes.

Expressions Les expressions de notre langage n'ayant pas d'effet de bord, leurs effets en lecture sont simplement l'union des effets en lecture de chacun de leurs composants. L'initialisation se fait au niveau des constantes ($\lambda\sigma.\varnothing$) ou des références. Nous représentons le résultat par $\lambda\sigma.(\{var\})$ pour une variable scalaire ou par $\lambda\sigma.(\{var(\mathcal{E}[\![exp_1,\ldots,exp_k]\!]\sigma)\})$ pour une variable tableau. Les approximations correspondantes dépendent de la représentation choisie : les définitions présentées dans les tables E.2 et E.3 ne reflètent donc que les propriétés qui doivent être vérifiées par l'implantation.

Affectation La partie droite d'une affectation est une expression sans effet de bord, qui est donc lue. La partie gauche écrit la variable ou l'élément de tableau référencé, mais a des effets en lecture sur les indices de tableau :

$$\begin{array}{lcl}
\mathcal{R}_r[\![var(exp_1,\ldots,exp_k) = exp]\!] & = & \mathcal{R}_r[\![exp_1,\ldots,exp_k]\!] \cup \mathcal{R}_r[\![exp]\!] \\
\mathcal{R}_r[\![var = exp]\!] & = & \mathcal{R}_r[\![exp]\!]
\end{array}$$

Les approximations correspondantes sont obtenues en utilisant les opérateurs abstraits approchés à la place des opérateurs exacts.

Séquence Les effets en lecture de $S_1; S_2$ sont l'union des effets de S_1 et de ceux de S_2. Il faut toutefois tenir compte, lors de l'évaluation des effets de S_2 de la modification de l'état mémoire par S_1 :

$$\mathcal{R}_r[\![S_1; S_2]\!] = \mathcal{R}_r[\![S_1]\!] \cup \mathcal{R}_r[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]$$

Les approximations ne sont pas directement dérivées de cette définition, car nous avons vu la nécessité de tenir compte des conditions de continuation du programme. Ainsi, composer le terme $\underline{\mathcal{R}}_r[\![S_2]\!] \underline{\bullet} \overline{\mathcal{T}}[\![S_1]\!]$ par $\underline{\mathcal{C}}[\![S_1]\!]$ garantit que le résultat est une région non vide uniquement si le programme ne s'arrête pas dans S_1 ; on a bien alors une sous-approximation de $\mathcal{R}_r[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]$. Pour la sur-approximation, l'utilisation de $\overline{\mathcal{C}}[\![S_1]\!]$ n'est pas obligatoire, car cela ne fait que restreindre l'ensemble des états mémoire pour lesquels le programme ne s'arrête pas dans S_1. Le résultat est bien sûr plus précis, mais l'intérêt de l'utilisation de $\overline{\mathcal{C}}[\![S_1]\!]$ apparaît surtout lorsque l'on veut vérifier l'exactitude de l'une des deux approximations ($\underline{\mathcal{R}}_r[\![S_2]\!] \underline{\bullet} \overline{\mathcal{T}}[\![S_1]\!] \underline{\circ} \underline{\mathcal{C}}[\![S_1]\!] = \overline{\mathcal{R}}_r[\![S_2]\!] \overline{\bullet} \overline{\mathcal{T}}[\![S_1]\!] \overline{\circ} \overline{\mathcal{C}}[\![S_1]\!]$), car $\underline{\mathcal{C}}[\![S_1]\!]$ peut être égal à $\overline{\mathcal{C}}[\![S_1]\!]$ mais pas à $\lambda\sigma.\sigma$, qui est le choix implicite si l'on n'utilise pas explicitement $\overline{\mathcal{C}}[\![S_1]\!]$. Les définitions retenues sont donc :

$$\begin{array}{rcl}
\overline{\mathcal{R}}_r[\![S_1; S_2]\!] & = & \overline{\mathcal{R}}_r[\![S_1]\!] \mathbin{\overline{\cup}} \overline{\mathcal{R}}_r[\![S_2]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{T}}[\![S_1]\!] \mathbin{\overline{\circ}} \overline{\mathcal{C}}[\![S_1]\!] \\
\underline{\mathcal{R}}_r[\![S_1; S_2]\!] & = & \underline{\mathcal{R}}_r[\![S_1]\!] \mathbin{\underline{\cup}} \underline{\mathcal{R}}_r[\![S_2]\!] \mathbin{\underline{\bullet}} \overline{\mathcal{T}}[\![S_1]\!] \mathbin{\underline{\circ}} \underline{\mathcal{C}}[\![S_1]\!]
\end{array}$$

Instruction conditionnelle Les éléments lus par une instruction conditionnelle sont ceux lus lors de l'évaluation de la condition, et, selon le résultat de cette évaluation, les éléments lus par la première branche ou la deuxième :

$$\mathcal{R}_r[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] = \mathcal{R}_r[\![C]\!] \cup (\mathcal{R}_r[\![S_1]\!] \circ \mathcal{E}[\![C]\!]) \cup (\mathcal{R}_r[\![S_2]\!] \circ \mathcal{E}[\![\texttt{.not.}C]\!])$$

La sur-approximation est dérivée directement de cette définition, tandis que la sous-approximation contient un terme supplémentaire qui exprime le fait qu'au moins l'une des branches est exécutée, et donc que si un élément de tableau est référencé dans les deux branches, il est certainement référencé par l'instruction globale :

$$\begin{array}{l}
\underline{\mathcal{R}}_r[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] = \\
\qquad \underline{\mathcal{R}}_r[\![C]\!] \mathbin{\underline{\cup}} (\underline{\mathcal{R}}_r[\![S_1]\!] \mathbin{\underline{\circ}} \underline{\mathcal{E}}_c[\![C]\!]) \mathbin{\underline{\cup}} (\underline{\mathcal{R}}_r[\![S_2]\!] \mathbin{\underline{\circ}} \underline{\mathcal{E}}_c[\![\texttt{.not.}C]\!]) \\
\qquad\qquad \mathbin{\underline{\cup}} (\underline{\mathcal{R}}_r[\![S_1]\!] \mathbin{\underline{\cap}} \underline{\mathcal{R}}_r[\![S_2]\!])
\end{array}$$

3.3.3 Régions IN

Informellement, les régions IN d'une instruction représentent les éléments de tableaux qu'elle importe : ce sont ceux qu'elle lit avant de les redéfinir éventuellement. Ceci est illustré par la figure 3.7 : `A` `B` et `C` sont lus dans la partie de code sombre ; mais `C` n'est pas importé par celle-ci car sa valeur est définie avant d'être lue ; ainsi, seuls `A` et `B` appartiennent aux régions IN du fragment de code sombre.

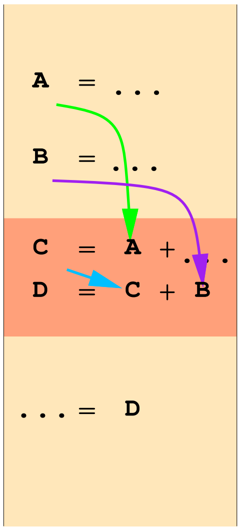


Figure 3.7: Régions READ et IN

$\mathcal{R}_i$, $\overline{\mathcal{R}}_i$ et $\underline{\mathcal{R}}_i$ définissent les sémantiques exactes et approchées des régions IN sur le domaine des instructions.

$$\mathcal{R}_i : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \wp(\mathbb{Z}^{d_i}))$$
$$\overline{\mathcal{R}}_i : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \tilde{\wp}(\mathbb{Z}^{d_i}))$$
$$\underline{\mathcal{R}}_i : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \tilde{\wp}(\mathbb{Z}^{d_i}))$$

Ces fonctions sémantiques sont décrites dans les tables E.6 et E.7 aux pages 296 et 297. Nous détaillons ci-dessous leur construction pour l'affectation, la séquence et l'instruction conditionnelle.

Affectation La norme FORTRAN précise que la partie droite d'une affectation est évaluée avant la partie gauche. La valeur des éléments référencés en lecture par la partie droite ne peut donc venir de la partie gauche, et ces valeur sont ainsi nécessairement importées. Comme nos expressions n'ont pas d'effet de bord, les régions IN d'une affectation sont donc ses régions READ :

$$\mathcal{R}_i[\![ref = exp]\!] = \mathcal{R}_r[\![ref = exp]\!]$$

Les approximations sont directement dérivées de cette définition.

Séquence Les régions IN de la séquence $S_1; S_2$ contiennent les éléments de tableaux importés par S_1 ($\mathcal{R}_i[\![S_1]\!]$), plus ceux importés par S_2 après l'exécution de S_1 ($\mathcal{R}_i[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]$) mais qui ne sont pas définis par S_1 ($\mathcal{R}_w[\![S_1]\!]$) :

$$\mathcal{R}_i[\![S_1; S_2]\!] = \mathcal{R}_i[\![S_1]\!] \cup \big((\mathcal{R}_i[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]) \boxminus \mathcal{R}_w[\![S_1]\!]\big)$$

Comme pour les régions READ et WRITE, les approximations doivent prendre en compte les conditions de continuation de S_1 :

$$\begin{array}{rcl}\overline{\mathcal{R}}_i[\![S_1;S_2]\!] & = & \overline{\mathcal{R}}_i[\![S_1]\!]\,\overline{\cup}\big((\overline{\mathcal{R}}_i[\![S_2]\!]\,\overline{\bullet}\,\overline{\mathcal{T}}[\![S_1]\!]\,\overline{\circ}\,\overline{\mathcal{C}}[\![S_1]\!])\,\overline{\boxminus}\,\underline{\mathcal{R}}_w[\![S_1]\!]\big)\\ \underline{\mathcal{R}}_i[\![S_1;S_2]\!] & = & \underline{\mathcal{R}}_i[\![S_1]\!]\,\underline{\cup}\,\big((\underline{\mathcal{R}}_i[\![S_2]\!]\,\underline{\bullet}\,\overline{\mathcal{T}}[\![S_1]\!]\,\underline{\circ}\,\underline{\mathcal{C}}[\![S_1]\!])\,\underline{\boxminus}\,\overline{\mathcal{R}}_w[\![S_1]\!]\big)\end{array}$$

Instruction conditionnelle Les régions IN d'une instruction conditionnelle contiennent les éléments lus lors de l'évaluation de la condition, plus les éléments importés par la première ou la deuxième branche, selon le résultat de cette évaluation :

$$\mathcal{R}_i[\![\texttt{if }C\texttt{ then }S_1\texttt{ else }S_2]\!] = \mathcal{R}_r[\![C]\!]\cup(\mathcal{R}_i[\![S_1]\!]\circ\mathcal{E}_c[\![C]\!])\cup(\mathcal{R}_i[\![S_2]\!]\circ\mathcal{E}_c[\![\texttt{.not.}C]\!])$$

La sur-approximation est directement dérivée de la sémantique exacte, tandis que la sous-approximation comporte un terme supplémentaire, comme pour les régions READ :

$$\begin{array}{l}\overline{\mathcal{R}}_i[\![\texttt{if }C\texttt{ then }S_1\texttt{ else }S_2]\!] = \\ \qquad\overline{\mathcal{R}}_r[\![C]\!]\,\overline{\cup}\,(\overline{\mathcal{R}}_i[\![S_1]\!]\,\overline{\circ}\,\overline{\mathcal{E}}_c[\![C]\!])\,\overline{\cup}\,(\overline{\mathcal{R}}_i[\![S_2]\!]\,\overline{\circ}\,\overline{\mathcal{E}}_c[\![\texttt{.not.}C]\!])\\ \underline{\mathcal{R}}_i[\![\texttt{if }C\texttt{ then }S_1\texttt{ else }S_2]\!] = \\ \qquad\underline{\mathcal{R}}_r[\![C]\!]\,\underline{\cup}\,(\underline{\mathcal{R}}_i[\![S_1]\!]\,\underline{\circ}\,\underline{\mathcal{E}}_c[\![C]\!])\,\underline{\cup}\,(\underline{\mathcal{R}}_i[\![S_2]\!]\,\underline{\circ}\,\underline{\mathcal{E}}_c[\![\texttt{.not.}C]\!])\,\underline{\cup}\,(\underline{\mathcal{R}}_i[\![S_1]\!]\,\underline{\cap}\,\underline{\mathcal{R}}_i[\![S_2]\!])\end{array}$$

3.3.4 Régions OUT

Les régions OUT d'une instruction représentent les éléments de tableaux qu'elle exporte : ce sont ceux qu'elle définit et qui sont utilisés dans la suite de l'exécution du programme, ce qui les différencie des régions WRITE. Ceci est illustré dans la figure 3.8 : `C` et `D` sont tous deux écrits dans la partie sombre du code ; `C` est réutilisé localement, mais pas après l'exécution de la partie sombre ; au contraire, `D` est réutilisé par la suite, et appartient donc aux régions OUT de la portion de code sombre.

Les régions OUT sont définies sur le domaine des instructions, car les expressions de notre langage n'ont pas d'effet de bord :

$$\mathcal{R}_o : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \wp(\mathbb{Z}^{d_i}))$$
$$\overline{\mathcal{R}}_o : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \tilde{\wp}(\mathbb{Z}^{d_i}))$$
$$\underline{\mathcal{R}}_o : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \tilde{\wp}(\mathbb{Z}^{d_i}))$$

Les régions OUT sur-estimées contiennent les régions *potentiellement* écrites ($\overline{\mathcal{R}}_w$) par l'instruction considérée, et *potentiellement utilisées* dans la suite du programme. Tandis que les régions OUT sous-estimées contiennent les éléments de tableaux *certainement* définis par l'instruction courante, et *certainement* utilisés par la suite du programme.

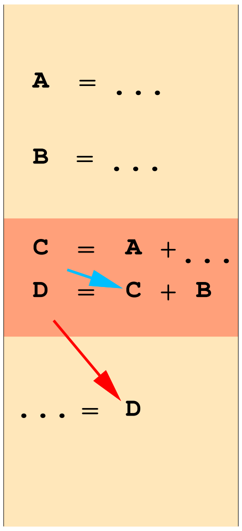


Figure 3.8: Régions WRITE et OUT

Les régions OUT d'une instruction dépendent, comme nous l'avons vu, du futur de l'exécution. Par exemple, les régions OUT d'une instruction S_1 d'un programme $S_1; S_2$ sont une fonction de $S_1; S_2$ considéré comme un tout, et de S_2. De ce fait, les régions OUT sont propagées de haut en bas dans la représentation du programme. Comme les opération d'entrée/sortie font partie du programme, les régions OUT du programme principal, desquelles sont dérivées les régions OUT des instructions qui le constituent, sont initialisées à $\lambda\sigma.\varnothing$.

Toutes les fonctions sémantiques sont décrites dans les tables E.8 et E.9. Nous détaillons ci-dessous leur construction dans le cas d'une séquence et d'une instruction conditionnelle.

Séquence Nous supposons que les régions OUT de la séquence $S_1; S_2$ sont connues, et notre but est de calculer les régions OUT de S_1 et de S_2, respectivement $\mathcal{R}_o[\![S_1]\!]$ et $\mathcal{R}_o[\![S_2]\!]$.

Intuitivement, les régions exportées par S_2 sont les régions exportées par la séquence, et écrite par S_2. Il faut cependant faire attention à l'effet de S_1 sur l'état mémoire : pour $\mathcal{R}_o[\![S_1; S_2]\!]$, l'état de référence est celui précédant S_1 ; alors que pour $\mathcal{R}_o[\![S_2]\!]$, c'est celui précédant S_2, obtenu en appliquant $\mathcal{T}[\![S_1]\!]$. $\mathcal{R}_o[\![S_2]\!]$ doit donc vérifier :

$$\mathcal{R}_o[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!] = \mathcal{R}_o[\![S_1; S_2]\!] \cap \mathcal{R}_w[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]$$

Nous ne pouvons pas donner une définition constructive de $\mathcal{R}_o[\![S_2]\!]$ car $\mathcal{T}[\![S_1]\!]$ n'est pas inversible. Mais l'on peut utiliser le *transformeur inverse*, $\overline{\mathcal{T}}^{-1}[\![S_1]\!]$ pour définir $\overline{\mathcal{R}}_o[\![S_2]\!]$ et $\underline{\mathcal{R}}_o[\![S_2]\!]$:

$$\overline{\mathcal{R}}_o[\![S_2]\!] = \overline{\mathcal{R}}_o[\![S_1; S_2]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}^{-1}[\![S_1]\!] \,\overline{\cap}\, \overline{\mathcal{R}}_w[\![S_2]\!]$$
$$\underline{\mathcal{R}}_o[\![S_2]\!] = \underline{\mathcal{R}}_o[\![S_1; S_2]\!] \,\underline{\bullet}\, \overline{\mathcal{T}}^{-1}[\![S_1]\!] \,\underline{\cap}\, \underline{\mathcal{R}}_w[\![S_2]\!]$$

Et l'on peut montrer que ce sont bien des approximations valides de $\mathcal{R}_o[\![S_2]\!]$ (propriété 6.4).

Les régions exportées par S_1 sont les régions exportées par la séquence, mais pas par S_2, et qui sont écrites par S_1. A ceci, il faut ajouter les éléments écrits par S_1 et exportés vers S_2, c'est-à-dire importés par S_2 :

$$\mathcal{R}_o[\![S_1]\!] = \mathcal{R}_w[\![S_1]\!] \cap \big((\mathcal{R}_o[\![S_1; S_2]\!] \boxminus \mathcal{R}_o[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]) \cup \mathcal{R}_i[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]\big)$$

Ici, les approximations sont directement dérivées de la sémantique exacte.

Instruction conditionnelle Pour une instruction conditionnelle, les régions OUT sont propagées de la structure globale vers les branches. Si la condition est vraie, les régions OUT de la première branche sont les régions exportées par l'instruction conditionnelle et écrites dans la première branche ; et vice-versa :

$$\begin{array}{rcl}\mathcal{R}_o[\![S_1]\!] & = & (\mathcal{R}_o[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] \cap \mathcal{R}_w[\![S_1]\!]) \circ \mathcal{E}_c[\![C]\!] \\ \mathcal{R}_o[\![S_2]\!] & = & (\mathcal{R}_o[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] \cap \mathcal{R}_w[\![S_2]\!]) \circ \mathcal{E}_c[\![\texttt{.not.}C]\!]\end{array}$$

Les approximations sont directement dérivées de ces définitions.

3.3.5 Détails d'implantation

Dans PIPS, le programme est représenté par son graphe de contrôle hiérarchisé. Il s'agit essentiellement de l'arbre de syntaxe abstrait dans lequel certains noeuds sont remplacés par un graphe de contrôle standard lorsque le code n'est pas structuré (c'est-à-dire contient des `GOTO`s). Les fonctions sémantiques des quatre types de régions présentés précédemment sont donc implantées directement.

Ces dernières induisent un ordre de calcul des différentes analyses : transformeurs, conditions de continuation, régions READ et WRITE, puis IN et enfin OUT. De cet ordre et des sens de propagation, on peut déduire que les transformeurs, les conditions de continuation et les régions READ, WRITE et IN pourraient être calculés lors d'un même parcours en arrière de la représentation du programme, alors que les régions OUT doivent être calculées ultérieurement, lors d'une phase de propagation en avant. En réalité, PIPS est un prototype permettant d'expérimenter différentes combinaisons d'analyses, et celles-ci sont donc effectuées indépendamment les unes des autres de manière à assurer une grande flexibilité, au détriment parfois de la rapidité. Il faut noter que les conditions de continuation ne sont pas encore implantées.

Nous avons présenté les sémantiques des régions sur- et sous-estimées. En réalité, nous ne calculons que des sur-approximations, et nous utilisons des critères d'exactitude pour les marquer comme *exactes* autant que possible. Les régions sous-estimées sont donc implicitement égales à la sur-approximation en cas d'exactitude, ou à l'ensemble vide. Nous avons vu à la section 3.1 les avantages de cette approche.

Jusqu'à présent, les préconditions n'ont pas été utilisées pour définir la sémantique des régions de tableaux. Pourtant, nous avons vu précédemment leur utilité pour la comparaison ou la combinaison des régions. En fait, on peut montrer que composer une région par la précondition courante ($\mathcal{R}[\![S]\!] \circ \mathcal{P}[\![S]\!]$) ne modifie pas la sémantique exacte, et n'apporte aucune information complémentaire. Par contre, les préconditions permettent de restreindre le domaine de définition des régions approchées.

> C'est ce qui se produit dans notre exemple de la figure 1.1, dans la fonction `D`. Si les relations entre `K` et `KP` et entre `J` et `JP` ne sont pas prises en compte, chaque référence au tableau `T` accède potentiellement à n'importe quel élément. Avec les préconditions, nous restreignons le domaine des états mémoire à ceux dans lesquels `J=JP` et `KP=K+1`, et nous pouvons trouver les positions relatives des six références à `T`.

Les préconditions peuvent donc faciliter les analyses de régions. Elles peuvent aussi donner de moins bons résultats, notamment si des conditions exclusives apparaissent dans les deux termes d'une intersection, comme c'est le cas pour les régions READ, WRITE et IN sous-estimées d'une instruction conditionnelle.

Dans PIPS, les préconditions sont actuellement ajoutées systématiquement, ce qui pose les problèmes évoqués ci-dessus. Une solution serait de ne les utiliser que si nécessaire. Mais ce critère de nécessité reste à définir.

3.4 Autres Travaux et Conclusion

De nombreuses études ont abordé les analyses de régions de tableaux depuis leur introduction par TRIOLET [161]. Les plus anciennes concernent généralement le choix de la représentation des éléments de tableaux [37, 19, 98, 99], problème que nous traiterons dans la partie III, pour des analyses approchées supérieurement, et insensibles au flot de contrôle du programme.

Dans les études plus récentes [151, 86, 92, 135, 164], nous retrouvons des types d'analyses similaires aux nôtres, à l'exception des régions OUT. Leurs sémantiques sont généralement présentées sous une forme dépendant de la représentation du programme, dans la plupart des cas un graphe de flot de contrôle ; seul JOUVELOT [108] avait déjà utilisé la sémantique dénotationnelle pour définir les régions, mais son étude était limitée aux sur-approximations. Les sous-approximations sont utilisées dans tous les paralléliseurs effectuant des optimisations comme la privatisation de tableaux [92, 135, 164, 176], et ce malgré l'absence d'étude préliminaire des fondements théoriques de ces analyses dans la plupart des cas. Seuls WONNACOTT et PUGH [171, 170, 176] se sont penchés sur ces problèmes dans le cas des formules de PRESBURGER.

Nous avions nous-mêmes présenté nos analyses de régions dans plusieurs rapports et articles [16, 18, 17, 63, 62, 60, 65]. Cependant, cette partie apporte de nouvelles contributions. Nous avons ainsi étudié le cadre théorique des analyses de régions de tableaux, indépendamment de leur type, et avons proposé une solution aux problèmes posés par les analyses approchées inférieurement. Nous avons ensuite défini la sémantique de quatre types de régions de tableaux (READ, WRITE, IN et OUT), en utilisant un formalisme indépendant de la représentation du programme et des régions, qui permet également de retarder une bonne partie des preuves de correction au choix effectif d'une représentation. Nous avons également identifié et présenté les analyses préliminaires nécessaires à la définition d'analyses de régions prenant en compte le contexte d'exécution, le flot de contrôle du programme et son flot de valeurs.

Dans cette partie, nous n'avons considéré que les analyses *intraprocédurales* des régions de tableaux, et indépendamment de toute représentation des ensembles d'éléments de tableaux. Dans la partie suivante, nous présenterons l'implantation effectuée dans PIPS ; et dans la partie IV, nous montrerons comment effectuer ces analyses en présence d'appels de procédures.

Chapter 4

May, Must or Exact?

Several studies [79, 29] have highlighted the need for advanced program optimizations to deal with memory management issues. For instance, Blume and Eigenmann [29] have shown that array privatization could greatly enhance the amount of potential parallelism in sequential programs. This technique basically aims at discovering array sections that are used as temporaries in loops, and can thus be replaced by local copies on each processor. An array section is said to be privatizable in a loop if each read of an array element is preceded by a write in the same iteration [123, 166][1]. Solving such problems requires a precise intra- and inter-procedural analysis of array data flow.

Several algorithms for array privatization or array expansion[2], based on different types of array data flow analyses, have already been proposed [70, 130, 123, 166, 93, 17, 63], . The first approach [70, 130] performs an *exact* analysis of array data flow, but for a restricted source language[3]. Most of the other methods use conservative approximations of array element sets, such as *MayBeDefined* and *MustBeDefined* sets. In fact, may sets are over-approximations of exact solutions, while must sets are under-approximations, according to a predefined approximation ordering. On the contrary, the last approach [17, 63] tries to compute exact solutions whenever possible, switching to conservative approximations only when exactness cannot be preserved anymore. However, except for specific applications [46, 47] requiring the knowledge of exactness, the advantages of our approach are still an open issue, which is discussed in this chapter.

Traditionally, semantic analyses are defined on lattices. This ensures that they are precisely defined even in case of recursive semantic equations, which appear whenever the source language contains looping constructs. For instance, exact solutions of array region analyses belong to the lattice $(\wp(\mathbb{Z}^n), \varnothing, \mathbb{Z}^n, \cup, \cap)$. Over-approximate analyses of array regions are also defined on lattices: Lattice of convex polyhedra in PIPS, or regular section lattice in the case of RSDs [37] for instance[4], or lattice of Presburger formulae as proposed in [176].

However, as will be shown in Section 4.1.3, under-approximate solutions are not uniquely defined when the domain is not closed under set union, e.g. for convex

[1] Additional conditions may also be used depending on the underlying architectural model, or on the target program transformation; this will be developed in Chapter 14.

[2] A similar technique for shared memory machines.

[3] Monoprocedural, static control language.

[4] A definition of RSDs is provided in Section 8.7.2. Very briefly, RSDs describe array regions with triplet notations $(l : u : s)$ on each array dimension.

polyhedra or RSDs. This means that, given an exact array region, no *best* under-approximation can be defined. This is illustrated in Figure 4.1 where several possible under-approximations of an array region are provided for two different representations. In this case, BOURDONCLE [33] suggests to approximate fixed points by decreasing under-approximate iterations[5], using a *narrowing operator*[6]. However, this approach may fail to compute under-approximations equal to the corresponding exact solutions whereas the computation of the latter is decidable, which is possible as shown in [17, 63].

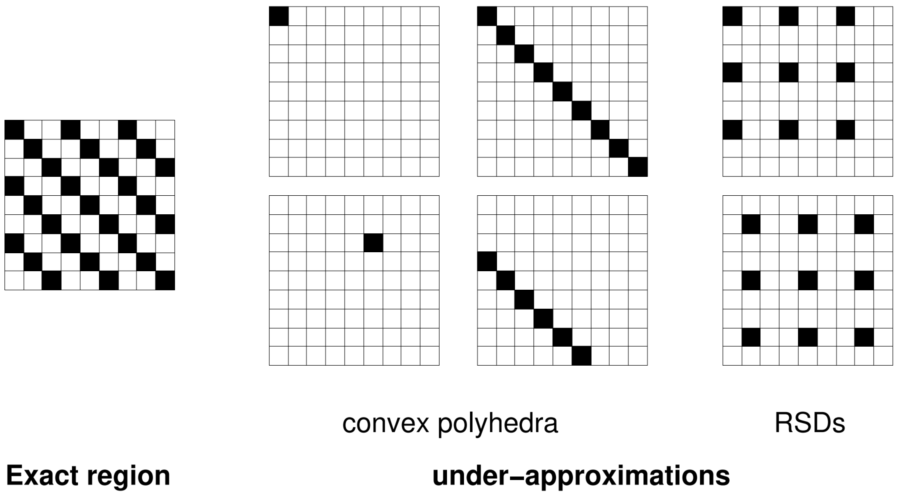


Figure 4.1: Possible under-approximations

We thus propose another approach, based on our previous experience of exact array data flow analyses [63, 62, 65]: We suggest to perform under- and over-approximate analyses at the same time, and to enhance the results of must analyses with those of may analyses, when the latter can be proved exact according to a *computable exactness criterion*. This method is very general and not solely limited to array region analyses. It can be used whenever one of the approximate domain is not a lattice, but also whenever exactness is specifically required, as in [46, 47].

This chapter is organized as follows. Section 4.1 presents the semantic analysis framework, and shows that some analysis solutions may not belong to a predefined lattice; existing ad-hoc solutions are also presented. Our solution and the *exactness criterion* are presented in Section 4.2 in relation with our previous work. We finally complete the description of our semantic framework, with a presentation of the properties of its main operators.

[5]He also proposes a more precise technique called *dynamic partitioning* [32]. Very briefly, it consists in dynamically determining an approximate solution built over simpler domains. But its complexity would be too high in the case of array region computations.

[6]More precisely the dual of his widening operator.

4.1 Semantic Analysis Frameworks

The semantic analysis of a program aims at statically discovering its properties, or the properties of the objects it uses (variables, ...). However, semantic equations may not have solutions computable in finite time; this is the case of recursive functions in an infinite height lattice for instance. Moreover, exactly describing the behavior of any program written in a given language may happen to be impossible, or at least of an overwhelming complexity; most languages have I/O instructions whose effects are not completely foreseeable; representing all types of expressions, particularly real expressions, is also generally considered too complex for an average compiler. Facing these obstacles, the solution is either to restrict the input language (as in [70, 72, 159, 130]), or to perform conservative approximate analyses [55] on real-life programs. This last characteristics has played a great role in the popularity of the method [2, 111, 55, 134, 153, 161, 37, 86]. Here, the complexity of the computation directly depends on the accuracy of the analysis. Up to now, no measurement of this accuracy has been proposed [55].

This section describes some usual characteristics of exact and approximate semantic analysis frameworks. It is then shown that some analyses do not respect these properties. Usual solutions are finally presented in the last subsection, and their limitations are discussed.

4.1.1 Sets and domains

Traditionally, mathematical spaces used for the denotation of programming constructs are called *semantic domains* and are *complete partial orders* or *lattices*.

Definition 4.1
A *complete partial order* (or cpo) is a set $\boldsymbol{E}$ partially ordered by a relation $\sqsubseteq_{\boldsymbol{E}}$; it has a least element $\perp_{\boldsymbol{E}}$ ($\boldsymbol{E}$ is *inductive*); and every countable increasing sequence $e_0 \sqsubseteq \cdots \sqsubseteq e_n \sqsubseteq \ldots$ has a limit in $\boldsymbol{E}$, denoted by $\bigsqcup_i e_i$.

Examples of basic domains are boolean values, natural integers $\mathbb{N}$, or relative integers $\mathbb{Z}$ (all of them augmented with an undefined element $\perp$). More complex domains are powerset domains, product domains, sum domains or continuous function domains[7].

Definition 4.2
A *complete lattice* is a cpo in which any two elements x and y have a greatest lower bound $(x \vee y)$ and a least upper bound $(x \wedge y)$.

The advantage of using cpo's as semantic domains is that the semantics of any recursive continuous semantic equation $f : \boldsymbol{D} \longrightarrow \boldsymbol{E}$ can be defined as the *least fixed point* of a continuous functional $\Phi : (\boldsymbol{D} \longrightarrow \boldsymbol{E}) \longrightarrow (\boldsymbol{D} \longrightarrow \boldsymbol{E})$.

Definition 4.3
Given cpo's $\boldsymbol{D}$ and $\boldsymbol{E}$, and a function $g : \boldsymbol{D} \longrightarrow \boldsymbol{E}$,
g is *monotone* if $x \sqsubseteq_{\boldsymbol{D}} y \implies g(x) \sqsubseteq_{\boldsymbol{E}} g(y)$.
g is *continuous* if $\forall\, e_0 \sqsubseteq \cdots \sqsubseteq e_n \sqsubseteq \ldots \quad g(\bigsqcup_i e_i) = \bigsqcup_i g(e_i)$.

[7] For a more complete description see [132, 88].

Definition 4.4
Let $\boldsymbol{D}$ be a cpo, and $g : \boldsymbol{D} \longrightarrow \boldsymbol{D}$ a continuous function.

$d \in \boldsymbol{D}$ is called a *fixed point* of g if $g(d) = d$. Moreover, d is called the *least fixed point* of g if $\forall\, d',\ g(d') = d' \implies d \sqsubseteq_{\boldsymbol{D}} d'$.

Theorem 4.1 (TARSKY)
Let $\boldsymbol{D}$ be a cpo, and $g : \boldsymbol{D} \longrightarrow \boldsymbol{D}$ a continuous function.

Then g has a least fixed point $\mathrm{lfp}(g) = \bigsqcup_{i>0} g^i(\perp)$.

In our framework, we require that exact semantic equations be continuous functions. This ensures that the exact semantics of the analysis is precisely defined. However, approximate semantic equations are not required to be continuous functions. This might be rather cumbersome in case of recursive definitions (which are usually needed to handle loops), since the existence of a fixed point is not guaranteed anymore. When the functions meet the weaker condition of monotonicity, GRAHAM and WEGMAN [82] define an *acceptable* solution. However, as will be shown in Section 4.1.3, some approximate analysis solutions do not belong to predefined lattices, and recursive functions may not even be monotone.

4.1.2 Semantic functions

Since we are interested in the semantic analysis of program properties, semantic functions share the same pattern. Let $\mathcal{A}$ be an exact semantic analysis of a program written in a programming language $\boldsymbol{\mathcal{L}}$. $\mathcal{A}$ is generally a function from a subset $\tilde{\boldsymbol{\mathcal{L}}}$ of $\boldsymbol{\mathcal{L}}$ to another set, usually a function set:

$$\begin{aligned} \mathcal{A} : \tilde{\boldsymbol{\mathcal{L}}} &\longrightarrow (\boldsymbol{D} \longrightarrow \boldsymbol{A}) \\ l &\longrightarrow \mathcal{A}[\![l]\!] \end{aligned}$$

For instance, the semantics of binary operators is given by a function mapping the cross-product of value domains to the domain of values ($\boldsymbol{V} \times \boldsymbol{V} \longrightarrow \boldsymbol{V}$).

And the exact semantics of array region analysis is described by a function

$$\mathcal{R} : \tilde{\boldsymbol{\mathcal{L}}} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \wp(\mathbb{Z}^n))$$

where $\boldsymbol{\Sigma}$ is the set of memory stores.

Note

- $\mathcal{A}$ is generally a partial function.
- The notation $\mathcal{A}[\![l]\!]$ used here suggests that we use denotational semantics [132] to define our analyses. This is the case for outward analyses, which define the non-standard semantics of a construct in terms of the semantics of its components.

  For example, the exact semantics of the READ regions of a sequence of instructions $S_1; S_2$ is:

  $$\mathcal{R}_r[\![S_1; S_2]\!] = \mathcal{R}_r[\![S_1]\!] \cup \mathcal{R}_r[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]$$

  as will be explained in Chapter 6.

However, this formalism cannot be used in the same way to define inward analyses, that is to say those defining the semantics of the components in terms of the characteristics of the global construct. We could use higher-order functions to define our inward analyses, and still use denotational semantics (an example can be found in [108]). But this would be more complicated than the existing implementation, whose axiomatization is closer to attribute grammar concepts than to denotational semantics. We therefore use the attribute grammar formalism for inward analyses. However, to keep uniform notations throughout this thesis, we keep the same kind of notations as for denotational semantics (i.e. the program text is provided inside double brackets).

> For example, the over-approximate OUT regions of a sequence of instructions are defined by (see Section6.6):
>
> $S_1; S_2$
> $$\begin{array}{lcl} \overline{\mathcal{R}}_o[\![S_2]\!] & = & \overline{\mathcal{R}}_o[\![S_1;S_2]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}^{-1}[\![S_1]\!] \,\overline{\sqcap}\, \overline{\mathcal{R}}_w[\![S_2]\!] \\ \overline{\mathcal{R}}_o[\![S_1]\!] & = & \overline{\mathcal{R}}_w[\![S_1]\!] \,\overline{\sqcap}\, ((\overline{\mathcal{R}}_o[\![S_1;S_2]\!] \,\overline{\boxminus}\, \underline{\mathcal{R}}_o[\![S_2]\!] \,\underline{\bullet}\, \overline{\mathcal{T}}[\![S_1]\!]) \,\overline{\sqcup}\, \overline{\mathcal{R}}_i[\![S_2]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}[\![S_1]\!]) \end{array}$$

- To describe our semantical functions, we use CHURCH's λ-calculus [22].

As stated in the introduction of this section, the solutions defined by $\mathcal{A}$ may be extremely difficult, indeed impossible, to compute. Hence the idea to replace $\mathcal{A}$ by approximate analyses, comparable to $\mathcal{A}$ by a partial order $\sqsubseteq$:

$$\begin{array}{rcl} \overline{\mathcal{A}} \text{ is an over-approximation of } \mathcal{A} & \Longleftrightarrow & \mathcal{A} \sqsubseteq \overline{\mathcal{A}} \\ \underline{\mathcal{A}} \text{ is an under-approximation of } \mathcal{A} & \Longleftrightarrow & \underline{\mathcal{A}} \sqsubseteq \mathcal{A} \end{array}$$

This *approximation ordering* [58] is a logical ordering, and is not necessarily related to a partial ordering between semantic values. However, since $\mathcal{A}$, $\overline{\mathcal{A}}$ and $\underline{\mathcal{A}}$ are defined on the same sub-domain of $\boldsymbol{\mathcal{L}}$, it is equivalent to an ordering of the solution functions:

$$\mathcal{A} \sqsubseteq \overline{\mathcal{A}} \iff \forall\, l \in \tilde{\boldsymbol{\mathcal{L}}} \quad \mathcal{A}[\![l]\!] \sqsubseteq \overline{\mathcal{A}}[\![l]\!]$$

Given this approximation ordering, a semantic analysis $\mathcal{A}$ can be surrounded by two new, and hopefully computable, approximate analyses: an over-approximation, denoted by $\overline{\mathcal{A}}$; and an under-approximation, denoted by $\underline{\mathcal{A}}$. The target sets of $\underline{\mathcal{A}}$ and $\overline{\mathcal{A}}$ are not necessarily the same as the target set of $\mathcal{A}$.

> For instance, array element index sets are parts of $\mathbb{Z}^n$. Due to the complexity of being able to represent any part of $\mathbb{Z}^n$, several compact representations have been defined: Convex polyhedra [161], RSD [37], DAD [19], ...
>
> Another example is the abstraction of statement effects on variables values (the so-called *transformers* in PIPS). The exact analysis is a function from the memory store set to itself, while the approximation maps the memory store set to its powerset:
> $$\overline{\mathcal{T}} : \boldsymbol{\Sigma} \longrightarrow \tilde{\wp}(\boldsymbol{\Sigma})$$

The approximate analyses we are interested in are then such that:

$$\overline{\mathcal{A}} : \tilde{\boldsymbol{\mathcal{L}}} \longrightarrow (\boldsymbol{D} \longrightarrow \boldsymbol{A}') \quad \text{and} \quad \underline{\mathcal{A}} : \tilde{\boldsymbol{\mathcal{L}}} \longrightarrow (\boldsymbol{D} \longrightarrow \boldsymbol{A}'')$$

Moreover, to simplify the discussion, we suppose that $\boldsymbol{A}' \subseteq \boldsymbol{A}$ and $\boldsymbol{A}'' \subseteq \boldsymbol{A}$, so that we can use the same ordering to compare their elements.

Note It does not necessarily make sense, or it may not be useful, to define both $\overline{\mathcal{A}}$ or $\underline{\mathcal{A}}$ for all type of semantic analysis. It is the case for the *transformers* which are presented in the next chapter.

4.1.3 Non-lattice frameworks: Why?

A natural question that arises is why we may need non-continuous, or even non-monotonous functions, while very friendly data-flow frameworks have been so carefully designed for years. In fact, this kind of problem is not unfrequent [57, 33]. And whereas we restrain our discussion to array region analyses in this section, the same argumentation would hold for other types of analyses.

As will be shown later in Chapters 5 and 6, array regions are functions from the set of memory stores $\mathbf{\Sigma}$ to the powerset $\wp(\mathbb{Z}^d)$, d being the dimension of the array:

$$\mathcal{R} : \mathbf{\Sigma} \longrightarrow \wp(\mathbb{Z}^d)$$

These analyses rely on a complete lattice, as shown by the next property.

Property 4.1
Let $L = (\mathbf{\Sigma} \longrightarrow \wp(\mathbb{Z}^d), \sqsubseteq, \lambda\sigma.\varnothing, \lambda\sigma.\mathbb{Z}^d, \sqcup, \sqcap)$ *with:*

$$\begin{array}{l} \mathcal{R}_1 \sqsubseteq \mathcal{R}_2 \iff \forall \sigma \in \mathbf{\Sigma},\ \mathcal{R}_1(\sigma) \subseteq \mathcal{R}_2(\sigma) \\ \forall \sigma \in \mathbf{\Sigma},\ (\mathcal{R}_1 \sqcup \mathcal{R}_2)(\sigma) = \mathcal{R}_1(\sigma) \cup \mathcal{R}_2(\sigma) \\ \forall \sigma \in \mathbf{\Sigma},\ (\mathcal{R}_1 \sqcap \mathcal{R}_2)(\sigma) = \mathcal{R}_1(\sigma) \cap \mathcal{R}_2(\sigma) \end{array}$$

$\subseteq$ *being the usual set inclusion,* $\cup$ *and* $\cap$ *the usual set operations.*

Then L *is a* complete lattice.

However, being able to represent all possible functions from $\mathbf{\Sigma}$ to $\wp(\mathbb{Z}^d)$ would be of an overwhelming complexity. This is why we usually restrain the domain of the analyses. For instance, we can use functions which are represented by PRESBURGER formulae: $\mathbf{\Sigma} \underset{\text{presb}}{\longrightarrow} \wp(\mathbb{Z}^d)$. And we have the following property:

Property 4.2
Let $L_{presb} = (\mathbf{\Sigma} \underset{presb}{\longrightarrow} \wp(\mathbb{Z}^d), \sqsubseteq, \lambda\sigma.\varnothing, \lambda\sigma.\mathbb{Z}^d, \sqcup, \sqcap)$

Then L_{presb} *is a complete sub-lattice of* L*.*

This is an important property; firstly because L_{presb} being complete implies that it has the necessary properties to define our analyses; secondly, because $\sqcup$ and $\sqcap$ are the same as in L and are internal operators; hence, as long as the analysis is confined to this domain, that is to say as long as the program is a *linear static control program*[8], the results are exact.

However, for several reasons which are beyond the scope of this section, and which will be presented in chapter 8, other types of representations are often used for array region analyses; for example convex polyhedra: $\mathbf{\Sigma} \underset{\text{conv}}{\longrightarrow} \wp(\mathbb{Z}^d)$. $L_{\text{conv}} = (\mathbf{\Sigma} \underset{\text{conv}}{\longrightarrow} \wp(\mathbb{Z}^d)$,

[8]That is to say a program made of sequences of `do` loops, and in which all subscripts and bound expressions are affine functions of the surroundings loop indices; affine `if` conditions can also be handled with PRESBURGER formulae.

$\sqsubseteq, \lambda\sigma.\varnothing, \lambda\sigma.\mathbb{Z}^d$, convex_hull, $\sqcap$) is a complete lattice, but is not a sub-lattice of L because in general convex_hull$(\mathcal{R}_1, \mathcal{R}_2) \neq \mathcal{R}_1 \sqcup \mathcal{R}_2$. (This is also the case for RSDs [37] or DADs [19] ...). However, since convex_hull$(\mathcal{R}_1, \mathcal{R}_2) \sqsupseteq \mathcal{R}_1 \sqcup \mathcal{R}_2$, the resulting analyses necessarily are over-approximations. But we also need under-approximations. The next theorem characterizes the complete lattices which allow such analyses.

Theorem 4.2
Let $\mathbf{\Sigma} \underset{?}{\longrightarrow} \wp(\mathbb{Z}^d)$ *be a subset of* $\mathbf{\Sigma} \longrightarrow \wp(\mathbb{Z}^d)$. *Let* $L_? = (\mathbf{\Sigma} \underset{?}{\longrightarrow} \wp(\mathbb{Z}^d), \sqsubseteq, \lambda\sigma.\varnothing, \lambda\sigma.\mathbb{Z}^d, \underline{\sqcup}, \sqcap)$ *with* $\mathcal{R}_1 \,\underline{\sqcup}\, \mathcal{R}_2 \sqsubseteq \mathcal{R}_1 \sqcup \mathcal{R}_2$, *and* $\underline{\sqcup}$ *and* $\sqcap$ *being internal operators*[9].

Then $L_?$ *is a complete lattice if and only if* $\underline{\sqcup} = \sqcup$.

Proof Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be two elements of $\mathbf{\Sigma} \underset{?}{\longrightarrow} \wp(\mathbb{Z}^d)$. Let $\mathcal{R}_3 = \mathcal{R}_1 \,\underline{\sqcup}\, \mathcal{R}_2$.

If we suppose that $L_?$ is a complete lattice, then

$$\mathcal{R}_1 \sqsubseteq \mathcal{R}_3 \tag{4.1}$$

$$\mathcal{R}_2 \sqsubseteq \mathcal{R}_3 \tag{4.2}$$

Since $\mathbf{\Sigma} \underset{?}{\longrightarrow} \wp(\mathbb{Z}^d) \subseteq \mathbf{\Sigma} \longrightarrow \wp(\mathbb{Z}^d)$, inclusions (4.1) and (4.2) imply that $\mathcal{R}_1 \sqcup \mathcal{R}_2 \sqsubseteq \mathcal{R}_3$. From the theorem hypothesis, we also know that $\mathcal{R}_3 \sqsubseteq \mathcal{R}_1 \sqcup \mathcal{R}_2$. Then:

$$\forall \mathcal{R}_1, \mathcal{R}_2 \in \mathbf{\Sigma} \underset{?}{\longrightarrow} \wp(\mathbb{Z}^d),\ \mathcal{R}_1 \,\underline{\sqcup}\, \mathcal{R}_2 = \mathcal{R}_1 \sqcup \mathcal{R}_2$$

Thus, $L_?$ is a complete lattice $\Longrightarrow\ \underline{\sqcup} = \sqcup$.

The converse is trivial. □

As a consequence, since the domain of convex polyhedra is not closed for $\sqcup$, this representation cannot be used as a regular framework for under-approximate semantic analyses. And this is also the case for RSDs, DADs, lists of finite length, ... It intuitively corresponds to the fact that the greatest convex polyhedron or RSD contained into any array region is not uniquely defined. An ad-hoc computable under-approximation must then be built in the chosen domain. Some possible choices have already been described in [33]. This will be the subject of the next subsection.

But before that, there is one more interesting question: What is the *smallest* sub-lattice of L which can be used to perform over- and under-approximate array region analyses? In fact, if we enforce that $\lambda\sigma.\varnothing$, $\lambda\sigma.\mathbb{Z}^d$, and affine functions belong to $\mathbf{\Sigma} \underset{?}{\longrightarrow} \wp(\mathbb{Z}^d)$, as it usually is the case for the representations used for array region analyses, then the smallest such domain is L_{presb}. There already exists several libraries to deal with PRESBURGER formulae [71, 112, 121], the most popular ones being the OMEGA and PIP libraries.

But the results are *exact* only as long as we deal with linear static control programs. If there are nonlinear array subscripts, conditions, assignments to scalar variables, the exact solutions of array region analyses do not belong to this framework anymore. Obviously, under- or over-approximations can be provided in the chosen representation, but there may exist several *equally good* approximations.

[9] Notice that the *greatest lower bound* is the same as in L. In fact, it would be possible to find a complete lattice with another operator $\underline{\sqcap}$, under-approximation of $\sqcap$. This would not affect the argumentation of the following proof. Moreover, the representations used for array region analyses are generally closed for $\sqcap$.

4.1.4 Usual ad-hoc solutions

When the approximate solution space $\boldsymbol{A}''$ is a lattice or cpo, its relations with the exact solution space $\boldsymbol{A}$ can be described by a GALOIS connection [27, 56], which defines two monotone functions: An *abstraction* function $\alpha : \boldsymbol{A} \longrightarrow \boldsymbol{A}''$, and a *concretization* or *meaning* function $\gamma : \boldsymbol{A}'' \longrightarrow \boldsymbol{A}$ (see Figure 4.2). The image of an exact continuous recursive function by α is then still continuous; its least fixed point is well defined, though its computability is not even decidable. COUSOT has shown that least fixed points can then be safely approximated by successive iterations relying on sequences of *narrowing* operators [56, 55].

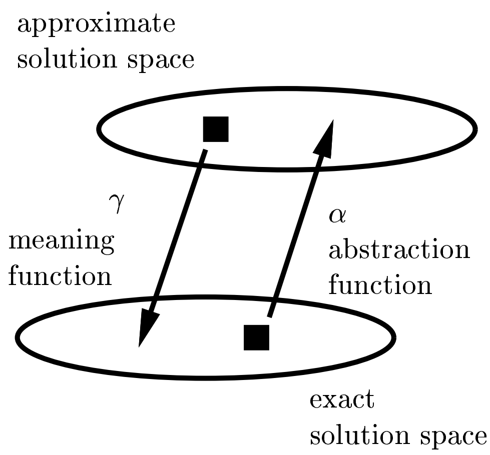


Figure 4.2: GALOIS Connection

To handle cases where the approximate solution space is not a lattice or a cpo, but a mere partial order, BOURDONCLE [33] generalizes the traditional approach of GALOIS connections by defining *representations*, in which α is not required to be a monotone function. In this framework, least fixed points of exact continuous recursive functions are safely under-approximated by sequences of narrowing operators, even if their images by α are not continuous (see Appendix B).

> Let us see how this technique performs for under-approximate array region analyses on trivial loops. We assume that the decreasing sequence of narrowing operators has only one iteration: Its result is the region corresponding to the first iteration of the loop.
>
> In the innermost loop body of the contrived example in Figure 4.3, the sole element `A(I)` is referenced. From our previous definition of the iterative process, we deduce that the summary region corresponding to the innermost loop is the set of array elements referenced at iteration `J` $= 1$, that is to say $\{$`A(I)`$\}$. It exactly describes the set of array elements read by the five iterations of the `J` loop. Repeating the process for the outermost loop, we obtain the set $\{$`A(1)`$\}$, which is far from the set of array elements actually referenced by the loop nest!

The previous example has been chosen for its simplicity, and it could be objected that the sequence of operators could be greatly enhanced. For instance, more iterations

```
do I = 1,5
   do J = 1,5
      A(I) =  ...
   enddo
enddo
```

Figure 4.3: A small contrived example

could be allowed: merging the sets of array elements during five iterations would give the exact result. But what if the loop upper bound is so great (`do I = 1,10000`) that the number of iterations would be overwhelming; or if it is unknown (`do I = 1,N`)? However, in all these cases, the exact set of array elements referenced within the previous loop nest is *computable* and *known* to be computable [65]. This is the basis for the solution we propose in the next section.

4.2 Approximations and Exactness

The previous section has just shown that existing under-approximate analyses may fail to expose interesting results, when the chosen domain is not closed under set union. This section describes a method to alleviate this problem, based on our previous experience of array region analyses [65]. It relies on the use of a *computable exactness criterion* which is introduced in a first subsection. Its optimality and applications are then discussed in Section 4.2.2.

4.2.1 Exactness of approximate analyses

From the definition of $\overline{\mathcal{A}}$ and $\underline{\mathcal{A}}$, we already know that:

$$\forall\, l \in \tilde{\boldsymbol{\mathcal{L}}}, \quad \underline{\mathcal{A}}[\![l]\!] \sqsubseteq \mathcal{A}[\![l]\!] \sqsubseteq \overline{\mathcal{A}}[\![l]\!]$$

An immediate consequence is:

$$(\underline{\mathcal{A}}[\![l]\!] \equiv \overline{\mathcal{A}}[\![l]\!]) \implies (\underline{\mathcal{A}}[\![l]\!] \equiv \mathcal{A}[\![l]\!] \equiv \overline{\mathcal{A}}[\![l]\!])$$

Hence, if the over-approximation is equal to the under-approximation, then both solutions are equal to the exact solution. However, it is not true that:

$$(\overline{\mathcal{A}}[\![l]\!] \equiv \mathcal{A}[\![l]\!]) \Rightarrow (\underline{\mathcal{A}}[\![l]\!] \equiv \mathcal{A}[\![l]\!])$$

A very simple counter-example is given by:

$$\overline{\mathcal{A}} : l \longrightarrow (\boldsymbol{D} \longrightarrow \top)$$
$$\underline{\mathcal{A}} : l \longrightarrow (\boldsymbol{D} \longrightarrow \bot)$$

It is not possible to have $\overline{\mathcal{A}}[\![l]\!] \equiv \underline{\mathcal{A}}[\![l]\!]$.

Thus a trivial way to check if the result of an approximation is exact is to verify that it is equal to the opposite approximation. A question that arises is whether this is the best possible criterion.

4.2.2 Optimality of the exactness criterion

The next theorem shows that if proving that $\overline{\mathcal{A}}[\![l]\!] \equiv \underline{\mathcal{A}}[\![l]\!]$ is not the best known criterion for checking the exactness of $\overline{\mathcal{A}}[\![l]\!]$ and $\underline{\mathcal{A}}[\![l]\!]$, then two other computable analyses $\overline{\mathcal{A}}'$ and $\underline{\mathcal{A}}'$ can be defined, which are better approximations of $\mathcal{A}$ than $\overline{\mathcal{A}}$ and $\underline{\mathcal{A}}$. A constructive proof is then given, which shows how to build $\overline{\mathcal{A}}'$ and $\underline{\mathcal{A}}'$ from $\overline{\mathcal{A}}$ and $\underline{\mathcal{A}}$. The consequences of this theorem are discussed afterwards.

Theorem 4.3
If there exists a computable *criterion* $C_{\underline{\mathcal{A}}\equiv\overline{\mathcal{A}}}$ *such that:*

$$\forall\, l \in \tilde{\boldsymbol{\mathcal{L}}},\ C_{\underline{\mathcal{A}}\equiv\overline{\mathcal{A}}}(l) \implies \overline{\mathcal{A}}[\![l]\!] \equiv \underline{\mathcal{A}}[\![l]\!]$$

then any computable *criterion* $C_{\underline{\mathcal{A}}\equiv\mathcal{A}}$ *(resp.* $C_{\overline{\mathcal{A}}\equiv\mathcal{A}}$*) such that:*

$$\forall\, l \in \tilde{\boldsymbol{\mathcal{L}}},\ C_{\underline{\mathcal{A}}\equiv\mathcal{A}}(l) \implies \underline{\mathcal{A}}[\![l]\!] \equiv \mathcal{A}[\![l]\!]$$

(resp. $\forall\, l \in \tilde{\boldsymbol{\mathcal{L}}},\ C_{\overline{\mathcal{A}}\equiv\mathcal{A}}(l) \implies \overline{\mathcal{A}}[\![l]\!] \equiv \mathcal{A}[\![l]\!]$*) is equivalent to* $C_{\underline{\mathcal{A}}\equiv\overline{\mathcal{A}}}$*, or there exists a* computable *approximation* $\underline{\mathcal{A}}'$ *(resp.* $\overline{\mathcal{A}}'$*) such that* $\underline{\mathcal{A}} \sqsubseteq \underline{\mathcal{A}}' \sqsubseteq \mathcal{A}$ *and* $C_{\underline{\mathcal{A}}'\equiv\mathcal{A}} \iff C_{\overline{\mathcal{A}}'\equiv\mathcal{A}}$ *(resp.*$\mathcal{A} \sqsubseteq \overline{\mathcal{A}}' \sqsubseteq \overline{\mathcal{A}}$ *and* $C_{\underline{\mathcal{A}}'\equiv\overline{\mathcal{A}}'} \iff C_{\overline{\mathcal{A}}'\equiv\mathcal{A}}$*).*

Proof

- If $C_{\underline{\mathcal{A}}\equiv\overline{\mathcal{A}}}(l)$ is true, it means that we know that $\overline{\mathcal{A}}[\![l]\!] \equiv \underline{\mathcal{A}}[\![l]\!] \equiv \mathcal{A}[\![l]\!]$. Then, necessarily, $C_{\underline{\mathcal{A}}\equiv\mathcal{A}}(l)$ is also true.
- On the contrary, let us assume that there exists $l \in \tilde{\boldsymbol{\mathcal{L}}}$ such that $C_{\overline{\mathcal{A}}\equiv\mathcal{A}}(l)$ is true and $C_{\underline{\mathcal{A}}\equiv\overline{\mathcal{A}}}(l)$ is false.

  $C_{\overline{\mathcal{A}}\equiv\mathcal{A}}(l) = \textit{true}$ implies that $\mathcal{A}[\![l]\!]$ is computable and is equal to $\overline{\mathcal{A}}[\![l]\!]$. Then, we can define a new approximation of $\mathcal{A}$, $\underline{\mathcal{A}}'$ by:

  $$\begin{aligned} \underline{\mathcal{A}}' : \tilde{\boldsymbol{\mathcal{L}}} &\longrightarrow (\boldsymbol{D} \longrightarrow \boldsymbol{A}' \cup \boldsymbol{A}'') \\ l &\longrightarrow \textbf{if } C_{\overline{\mathcal{A}}\equiv\mathcal{A}}(l) \textbf{ then } \overline{\mathcal{A}}[\![l]\!] \textbf{ else } \underline{\mathcal{A}}[\![l]\!] \end{aligned}$$

  And the equivalence criteria of $\underline{\mathcal{A}}'$ with the exact analysis ($C_{\underline{\mathcal{A}}'\equiv\mathcal{A}}$) and with the current over-approximation ($C_{\underline{\mathcal{A}}'\equiv\overline{\mathcal{A}}}$) are defined by:

  $$\forall\, l \in \tilde{\boldsymbol{\mathcal{L}}},\ C_{\underline{\mathcal{A}}'\equiv\mathcal{A}}(l) = \text{if } C_{\overline{\mathcal{A}}\equiv\mathcal{A}}(l) \text{ then } \textit{true} \text{ else } C_{\underline{\mathcal{A}}\equiv\mathcal{A}}(l)$$
  $$\forall\, l \in \tilde{\boldsymbol{\mathcal{L}}},\ C_{\underline{\mathcal{A}}'\equiv\overline{\mathcal{A}}}(l) = \text{if } C_{\overline{\mathcal{A}}\equiv\mathcal{A}}(l) \text{ then } \textit{true} \text{ else } C_{\underline{\mathcal{A}}\equiv\overline{\mathcal{A}}}(l)$$

  Since $C_{\underline{\mathcal{A}}\equiv\mathcal{A}}(l) = C_{\underline{\mathcal{A}}\equiv\overline{\mathcal{A}}}(l) = \textit{false}$, $C_{\underline{\mathcal{A}}'\equiv\overline{\mathcal{A}}}(l) = C_{\underline{\mathcal{A}}'\equiv\mathcal{A}}(l) = C_{\overline{\mathcal{A}}\equiv\mathcal{A}}(l)$.

The previous assumption that $\exists l : C_{\overline{\mathcal{A}}\equiv\mathcal{A}}(l) \wedge \neg C_{\underline{\mathcal{A}}\equiv\overline{\mathcal{A}}}(l)$ implies that for at least one element l of $\mathcal{L}$ we know that $\overline{\mathcal{A}}[\![l]\!] \equiv \mathcal{A}[\![l]\!] \equiv \underline{\mathcal{A}}'[\![l]\!]$, but we do not know whether $\underline{\mathcal{A}}[\![l]\!] \equiv \mathcal{A}[\![l]\!]$ or not. A consequence is that $\underline{\mathcal{A}}'$ is either identical to $\underline{\mathcal{A}}$ or more accurate:

$$\underline{\mathcal{A}} \sqsubseteq \underline{\mathcal{A}}' \sqsubseteq \mathcal{A}$$

Thus, we have shown that either $C_{\overline{\mathcal{A}}\equiv\mathcal{A}} \iff C_{\underline{\mathcal{A}}\equiv\overline{\mathcal{A}}}$ or there exists a computable under-approximation $\underline{\mathcal{A}}'$ of $\mathcal{A}$, more accurate than $\underline{\mathcal{A}}$, and such that $C_{\underline{\mathcal{A}}'\equiv\overline{\mathcal{A}}} \iff C_{\underline{\mathcal{A}}'\equiv\mathcal{A}}$.

The proof is identical for $C_{\underline{\mathcal{A}}\equiv\mathcal{A}}$ and $C_{\underline{\mathcal{A}}\equiv\overline{\mathcal{A}}}$. □

Figure 4.4 summarizes the relations between the different criteria of exactness, and the actual exactness of the approximate analyses. To sum up, if $C_{\underline{\mathcal{A}}\equiv\overline{\mathcal{A}}}$ is not the best

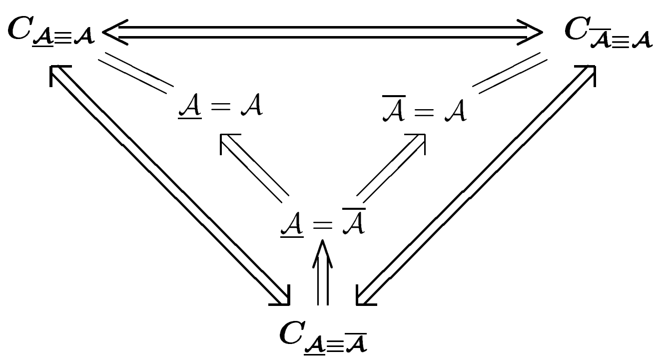


Figure 4.4: Relations between approximations

possible criterion for checking exactness, it is possible to define two other approximations, which are more accurate than the original ones. The new criterion $C_{\underline{\mathcal{A}}'\equiv\overline{\mathcal{A}}'}$ is then the best possible computable criterion (relatively to the available information about the program behavior). Moreover, from the previous proof, computing and testing the exactness of the new approximations is not more expensive than with the original solution.

Theorem 4.3 has two more consequences:

1. It may not always be useful to compute both $\overline{\mathcal{A}}$ and $\underline{\mathcal{A}}$; but we may well want to check the exactness of the chosen analysis, say $\overline{\mathcal{A}}$. However, it may sometimes be difficult to find a computable exactness criterion $C_{\overline{\mathcal{A}}\equiv\mathcal{A}}$: it may depend on $\mathcal{A}$, which is not computable! The above theorem gives a solution. If an opposite approximation $\underline{\mathcal{A}}$ is sufficiently well defined, $C_{\underline{\mathcal{A}}\equiv\overline{\mathcal{A}}}$ is the best possible exactness criterion for $\overline{\mathcal{A}}$. Thus, it is sufficient to define $\underline{\mathcal{A}}$, deduce the exactness criterion $C_{\underline{\mathcal{A}}\equiv\overline{\mathcal{A}}}$ and use it as $C_{\overline{\mathcal{A}}\equiv\mathcal{A}}$. It is not even necessary to explicitly compute $\underline{\mathcal{A}}$.

2. As shown in the previous section, defining an interesting approximation analysis is not always an easy task and general solutions may be disappointing. In this

case, if a computable exactness criterion for the opposite analysis ($C_{\overline{\mathcal{A}}\equiv\mathcal{A}}$) is available, then $\underline{\mathcal{A}}$ can be refined by using $\overline{\mathcal{A}}$ whenever $C_{\overline{\mathcal{A}}\equiv\mathcal{A}}$ is true.

> Let us consider the example of Figure 4.3 to illustrate this. We assume that array regions are represented by convex polyhedra. For the loop body, the regions are exactly represented by $\{\Phi_1 : \Phi_1 = i\}$, Φ_1 being the descriptor of the first dimension of array `A`.
>
> An over-approximation for the innermost loop nest is readily obtained by adding the loop bound constraints to the above polyhedron, and eliminating the loop index j:
>
> $$\{\Phi_1 : \Phi_1 = i \wedge 1 \leq j \leq 5\} \rightsquigarrow \{\Phi_1 : \Phi_1 = i\}$$
>
> This operation is exact because the loop bounds are affine, and the projection is exact according to [12, 144]j.
>
> For the outermost loop, computing the over-approximation gives:
>
> $$\{\Phi_1 : 1 \leq \Phi_1 \leq 5\}$$
>
> Again, the loop bounds are affine, and the elimination of i is exact. Since the previous regions are exact, they are valid under-approximations. We would obtain the same results with non-numerical but still affine loop bounds.

4.2.3 Relations with other approaches

Since the absence of a predefined lattice is a general characteristic of under-approximate array region analyses, how do parallelizing compilers other than PIPS handle the problem, either consciously or not?

In **Fiat/Suif** [92], the approach is to avoid inexact operations[10]. This is supported by an infinite representation, lists of polyhedra, which allows exact unions by simply appending regions to the list. For `do` loops with affine bounds, the loop index is merely replaced by a dummy variable, which is also an exact operation. For general loops, existing iterative techniques are used, but the underlying domain is the semi-lattice $(\wp_{\text{convex}}(\mathbb{Z}^n), \varnothing, \cap)$. This ensures safe approximations of fixed points, without having to use costly techniques such as dynamic partitioning. However, this may often result in coarse approximations, such as the empty set. And this prevents taking context information into account, since they are often exclusive in the several branches reaching join nodes of the control flow graph.

For example, for the following piece of code

```
if (i.le.2) then
   A(i) = ...
else
   A(i) = ...
endif
```

[10] This can be viewed as a *preventive* exactness criterion.

the regions of the two branches can respectively be represented by (taking the context into account):

$$\{\Phi_1 : \Phi_1 = i \wedge i \leq 2\}$$

and

$$\{\Phi_1 : \Phi_1 = i \wedge 3 \leq i\}$$

Their intersection is the empty set, because they contain exclusive conditions ($i \leq 2$ and $3 \leq i$). If the conditions are not taken into account, the regions are identical, and their intersection is:

$$\{\Phi_1 : \Phi_1 = i\}$$

In **Polaris** [164], a similar approach is adopted, but applied to lists of RSDs. The main difference lies in the choice of the meet operator, which is called a *gated intersection*, but seems to be a guarded union, which matches the intrinsic nature of the corresponding exact operation. How exponential growth of lists of regions is avoided is not specified.

In **Panorama** [87], lists of guarded RSDs are used to represent array regions. This allows to avoid inexact union and to handle `do` loops accurately. If during an operation a guard or RSD component cannot be computed, it is replaced by an unknown value, Ω. In fact, this is equivalent to computing exact regions whenever possible, and flag them as inexact otherwise. In this case, Ω can be interpreted as $\top$ or $\bot$ depending on the desired approximation (respectively *may* and *must*)[11]. For general loops, conservative approximations are performed. In particular, for loops with premature exits, an intersection operation is performed, as in Fiat/Suif.

To summarize, most under-approximate array region analyses implemented in existing parallelizing compilers are based on implicit exactness criteria, to avoid inexact operations. Infinite representations are used to handle sequences of instruction, for which union operations are involved. For `do` loops, ad-hoc solutions are used to compute exact solutions without using iterative techniques. But general loops for which fixed points would be required are conservatively handled, by switching to another lattice based on the same representation but associated with the intersection operator.

A lot of work has also been done in the *Omega Library* by PUGH and WONNACOTT [144, 171, 176] to under-approximate parts of Z^n using PRESBURGER formulae. The target domain was primarily the analysis of dependences, but their techniques can also be applied to array region analyses. Nonlinear conditions or references are represented either by *unknown* variables, or by dummy functions. The *dark shadow* defined in [144] could also provide under-approximations of projections, which are necessary for the array region analysis of `do` loops, as will be shown in Part III.

4.3 Operators

The semantics we define in the next chapters rely on several operators: Internal operators, fixed points, and external composition laws. For exact analyses, defined on lattices, the definition of these operators is generally straightforward; however, this is

[11] This is very similar to the use of the *unknown* variable proposed by WONNACOTT in his thesis [176]

no more the case for approximate analyses, especially when their domain is not a predefined lattice. Moreover, in order to avoid details due to particular representations when defining the semantics of our analyses, we wish to use abstract approximate operators; to ensure safe approximations, some properties must be enforced on these operators. This is the purpose of this section.

The first subsection is devoted to internal operators; the second to fixed points, in relation with the considerations of the previous two sections. Internal and external composition laws are then examined in Section 4.3.3. Finally, the last section shows the consequences of the properties enforced on operators in Sections 4.3.1, 4.3.2 and 4.3.3 on correctness proving.

4.3.1 Internal operators

To specify the exact semantics $\mathcal{A}$, we usually need several internal operators. We therefore provide the exact analysis cpo or lattice $(\boldsymbol{D} \longrightarrow \boldsymbol{A})$ with a set of k internal operators, $\{\mathrm{op}_i\}_{i=1,k}$. The first two operators are the well-known join and meet operators, respectively denoted by $\vee$ or $\wedge$ for boolean lattices $(\boldsymbol{D} \longrightarrow \boldsymbol{B})$, and $\cup$ or $\cap$ for powersets. When $\boldsymbol{D} \longrightarrow \boldsymbol{A}$ is a cpo, $\cup$ and $\cap$ share the following properties:

Property 4.3
In any lattice $\boldsymbol{D} \longrightarrow \boldsymbol{A}$, the operations $\cup$ and $\cap$ satisfy the following laws, for all $x, y, z \in (\boldsymbol{D} \longrightarrow \boldsymbol{A})$:

Idempotence $x \cup x = x$ *and* $x \cap x = x$.

Commutativity $x \cup y = y \cup x$ *and* $x \cap y = y \cap x$.

Associativity $x \cup (y \cup z) = (x \cup y) \cup z$ *and* $x \cap (y \cap z) = (x \cap y) \cap z$.

Absorption $x \cap (x \cup y) = x \cup (x \cap y) = x$.

$\cup$ and $\cap$ (as well as $\vee$ or $\wedge$) are binary operators, but are trivially extended to n-ary operators:

Finite join

$$\begin{array}{lll}
\text{if } k_2 < k_1 & \displaystyle\bigcup_{k=k_1}^{k_2} : \longrightarrow (\boldsymbol{D} \longrightarrow \boldsymbol{A}) & \\
& \qquad \longrightarrow \perp & \\
\text{if } k_2 \geq k_1 & \displaystyle\bigcup_{k=k_1}^{k_2} : (\boldsymbol{D} \longrightarrow \boldsymbol{A})^{k_2-k_1+1} \longrightarrow (\boldsymbol{D} \longrightarrow \boldsymbol{A}) & \\
& \qquad\qquad \bigcup_{k=k_1}^{k_2} x_k \longrightarrow x_1 \cup (\bigcup_{k=k_1+1}^{k_2} x_k) &
\end{array}$$

Finite meet

$$
\begin{aligned}
\text{if } k_2 < k_1 \quad & \bigcap_{k=k_1}^{k_2} : \longrightarrow (\boldsymbol{D} \longrightarrow \boldsymbol{A}) \\
& \qquad \longrightarrow \bot \\
\text{if } k_2 = k_1 \quad & \bigcap_{k=k_1}^{k_2} : (\boldsymbol{D} \longrightarrow \boldsymbol{A}) \longrightarrow (\boldsymbol{D} \longrightarrow \boldsymbol{A}) \\
& \qquad\qquad \textstyle\bigcap_{k=k_1}^{k_2} x_k \longrightarrow x_1 \\
\text{if } k_2 > k_1 \quad & \bigcap_{k=k_1}^{k_2} : (\boldsymbol{D} \longrightarrow \boldsymbol{A})^{k_2-k_1+1} \longrightarrow (\boldsymbol{D} \longrightarrow \boldsymbol{A}) \\
& \qquad\qquad\qquad \textstyle\bigcap_{k=k_1}^{k_2} x_k \longrightarrow x_{k_1} \cap (\bigcap_{k=k_1+1}^{k_2} x_k)
\end{aligned}
$$

In boolean lattices, the third operator usually is the negation operator (denoted by $\neg$). Whereas in powerset lattices, it is the difference operator (denoted by $\boxminus$). Notice that, unlike the preceding binary operators, the difference operator is neither commutative nor associative.

An easy way to define the approximate semantic equations, is to directly derive them from the exact analysis, by using approximate operators. So, for each *commutative* operator op_i from the original cpo, an over- ($\overline{\mathrm{op}}_i$) and an under-operator ($\underline{\mathrm{op}}_i$) are defined as such:

$$
\begin{aligned}
\text{if } \mathrm{op}_i &: (\boldsymbol{D} \longrightarrow \boldsymbol{A})^n \longrightarrow (\boldsymbol{D} \longrightarrow \boldsymbol{A}) \\
\text{then } \overline{\mathrm{op}}_i &: (\boldsymbol{D} \longrightarrow \boldsymbol{A}')^n \longrightarrow (\boldsymbol{D} \longrightarrow \boldsymbol{A}') \\
\underline{\mathrm{op}}_i &: (\boldsymbol{D} \longrightarrow \boldsymbol{A}'')^n \longrightarrow (\boldsymbol{D} \longrightarrow \boldsymbol{A}'')
\end{aligned}
$$

and $\underline{\mathrm{op}}_i \sqsubseteq \mathrm{op}_i \sqsubseteq \overline{\mathrm{op}}_i$. Thus, defining $\overline{\mathcal{A}}$ using $\{\overline{\mathrm{op}}_i\}$ instead of $\{\mathrm{op}_i\}$ automatically leads to an over-approximation ($\mathcal{A} \sqsubseteq \overline{\mathcal{A}}$).

For unary and non-commutative operators, such as $\boxminus$ and $\neg$, more work has to be done to preserve the approximation ordering. For instance, removing an over-approximation from another one does not make any sense as shown on Figure 4.5. Here, $\overline{E}_1 \,\overline{\boxminus}\, \overline{E}_2$ is not comparable with $E_1 \boxminus E_2$. A safe over-approximation of $E_1 \boxminus E_2$

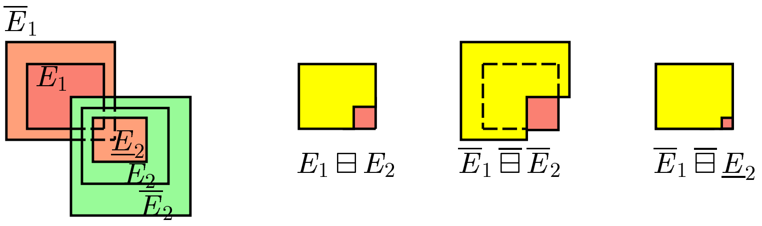


Figure 4.5: Over-approximation of $E_1 \boxminus E_2$

can be obtained with $\overline{E}_1 \,\overline{\boxminus}\, \underline{E}_2$. Similarly, a safe under-approximation is given by

$\underline{E}_1 \underline{\boxminus} \overline{E}_2$. The domain and codomains of these two operators are thus:

$$
\begin{aligned}
\overline{\boxminus} &: (\boldsymbol{D} \longrightarrow \boldsymbol{A}') \times (\boldsymbol{D} \longrightarrow \boldsymbol{A}'') \longrightarrow (\boldsymbol{D} \longrightarrow \boldsymbol{A}') \\
\underline{\boxminus} &: (\boldsymbol{D} \longrightarrow \boldsymbol{A}'') \times (\boldsymbol{D} \longrightarrow \boldsymbol{A}') \longrightarrow (\boldsymbol{D} \longrightarrow \boldsymbol{A}'')
\end{aligned}
$$

Of course, the elements of $\boldsymbol{A}'$ and $\boldsymbol{A}''$ must be combinable, and usually $\boldsymbol{A}' = \boldsymbol{A}''$ (from now on they are denoted as $\tilde{\boldsymbol{A}}$). Moreover, we require that:

$$
\forall \overline{E}_1, \underline{E}_2 \in \boldsymbol{D} \longrightarrow \tilde{\boldsymbol{A}}, \quad
\begin{aligned}
&\overline{E}_1 \boxminus \underline{E}_2 \sqsubseteq \overline{E}_1 \overline{\boxminus} \underline{E}_2 \\
&\underline{E}_2 \underline{\boxminus} \overline{E}_1 \sqsubseteq \underline{E}_2 \boxminus \overline{E}_1
\end{aligned}
$$

No other property is enforced on approximate operators, in particular on $\overline{\cup}$, $\underline{\cup}$, $\overline{\cap}$ and $\underline{\cap}$: They may still be idempotent, commutative and associative, but it is not compulsory. Of course, when they do not meet these properties, different implementations may give different, equally good, results. To garantee determinism, a careful parenthesizing of equation terms may be necessary. Also, finite union and intersection may become infinite operations, if their bounds depend on the result of other analyses; *ad hoc* solutions must then be found in each particular case, but we still use the usual notation ($\overline{\bigcup}$ for instance).

4.3.2 Fixed points

In our framework, we require that exact semantic equations be defined on cpos or complete lattices, and be continuous. This ensures that the exact semantic of the analysis is precisely defined, especially in case of recursive functions. However, we have seen in the previous sections that approximate analyses are not always defined on cpos or lattices. While this does not prevent the computability of non-recursive functions, it is rather cumbersome for recursive ones. We have seen in Section 4.2 how to alleviate this problem. Relying on these results, we can introduce two operators to approximate exact least fixed points: They are improperly denoted by $\overline{\mathrm{lfp}}$ and $\underline{\mathrm{lfp}}$ to recall their link with the least fixed point operator lfp.

Definition 4.5
Let l be an element of $\tilde{\boldsymbol{\mathcal{L}}}$. Let $\Phi' : (\boldsymbol{D} \longrightarrow \boldsymbol{A}') \longrightarrow (\boldsymbol{D} \longrightarrow \boldsymbol{A}')$ and $\Phi'' : (\boldsymbol{D} \longrightarrow \boldsymbol{A}'') \longrightarrow (\boldsymbol{D} \longrightarrow \boldsymbol{A}'')$ such that $\overline{\mathcal{A}}[\![l]\!] = \Phi'(\overline{\mathcal{A}}[\![l]\!])$ and $\underline{\mathcal{A}}[\![l]\!] = \Phi''(\overline{\mathcal{A}}[\![l]\!])$.

$\overline{\text{lfp}}$ and $\underline{\text{lfp}}$ are then defined by[12]:

$$
\begin{array}{lll}
\overline{\text{lfp}}_{\mathcal{A},l}(\Phi') = & \textbf{If} & ((\boldsymbol{D} \longrightarrow \boldsymbol{A}') \text{ is a lattice } \textbf{and } \Phi' \text{ is continuous}) \\
& \textbf{Then} & \text{lfp}(\Phi') \\
& \textbf{Else} & \begin{array}{ll} \textbf{if} & C_{\underline{\mathcal{A}}\equiv\mathcal{A}}(l) \\ \textbf{then} & \underline{\mathcal{A}}[\![l]\!] \\ \textbf{else} & \lim\limits_{i\to\infty} \left\{ \begin{array}{lll} a_0 & = & \perp_{(\boldsymbol{D}\longrightarrow\boldsymbol{A}')\longrightarrow(\boldsymbol{D}\longrightarrow\boldsymbol{A}')} \\ a_{i+1} & = & a_i \,\overline{\nabla}_i\, \Phi'(a_i) \end{array} \right. \end{array} \\
\underline{\text{lfp}}_{\mathcal{A},l}(\Phi'') = & \textbf{If} & ((\boldsymbol{D} \longrightarrow \boldsymbol{A}'') \text{ is a lattice } \textbf{and } \Phi'' \text{ is continuous}) \\
& \textbf{Then} & \text{lfp}(\Phi'') \\
& \textbf{Else} & \begin{array}{ll} \textbf{if} & C_{\overline{\mathcal{A}}\equiv\mathcal{A}}(l) \\ \textbf{then} & \overline{\mathcal{A}}[\![l]\!] \\ \textbf{else} & \lim\limits_{i\to\infty} \left\{ \begin{array}{lll} a_0 & = & \perp_{(\boldsymbol{D}\longrightarrow\boldsymbol{A}'')\longrightarrow(\boldsymbol{D}\longrightarrow\boldsymbol{A}'')} \\ a_{i+1} & = & a_i \,\underline{\Delta}_i\, \Phi''(a_i) \end{array} \right. \end{array}
\end{array}
$$

When no ambiguity is possible, $\overline{\text{lfp}}_{\mathcal{A},l}(\Phi')$ and $\underline{\text{lfp}}_{\mathcal{A},l}(\Phi'')$ will be abbreviated by $\overline{\text{lfp}}(\Phi')$ and $\underline{\text{lfp}}(\Phi'')$.

The next property, whose proof is trivial, shows that $\overline{\text{lfp}}$ and $\underline{\text{lfp}}$ are approximations of the fixed points of their arguments.

Property 4.4
$\overline{\text{lfp}}$ and $\underline{\text{lfp}}$ preserve the direction of the approximation, that is to say:

$$\overline{\mathcal{A}}[\![l]\!] = \Phi'(\overline{\mathcal{A}}[\![l]\!]) \implies \overline{\mathcal{A}}[\![l]\!] \sqsubseteq \overline{\text{lfp}}_{\mathcal{A}}[\![l]\!](\Phi')$$

and,

$$\underline{\mathcal{A}}[\![l]\!] = \Phi''(\underline{\mathcal{A}}[\![l]\!]) \implies \underline{\text{lfp}}_{\mathcal{A}}[\![l]\!](\Phi'') \sqsubseteq \underline{\mathcal{A}}[\![l]\!]$$

Thus, when $\underline{\mathcal{A}}$ (or $\overline{\mathcal{A}}$) has a recursive definition, a safe solution can always be defined. But in the case of a continuous function, its computability is not necessarily decidable [55]. Convergence acceleration methods, very similar to those presented in Section 4.1.4 for representations, can nevertheless be used when the cpo is also a complete lattice [55, 33]. They are not presented here, because they would provide no useful insights, but their use is assumed whenever necessary.

4.3.3 Composing analyses

As will be shown in the following chapters, many analyses rely on the results given by preceding analyses. For instance continuation conditions need an evaluation of `if` condition values; array region analyses need a representation of array index values, and of their modification by a statement, ...

[12] Using the notations introduced in Definitions B.1 and B.2

This section discusses the composition of the analysis $\mathcal{A}$, with the results of another one ($\mathcal{X} : \tilde{\boldsymbol{\mathcal{L}}}' \longrightarrow (\boldsymbol{E} \longrightarrow \boldsymbol{X})$), and of their respective approximations. $\overline{\mathcal{X}}$ and $\underline{\mathcal{X}}$ domains and codomains respectively are:

$$\overline{\mathcal{X}} : \tilde{\boldsymbol{\mathcal{L}}}' \longrightarrow (\boldsymbol{E} \longrightarrow \boldsymbol{X}')$$

and,

$$\underline{\mathcal{X}} : \tilde{\boldsymbol{\mathcal{L}}}' \longrightarrow (\boldsymbol{E} \longrightarrow \boldsymbol{X}'')$$

Composing exact analyses whose domain and codomain match ($\boldsymbol{X} \subseteq \boldsymbol{D}$) is done using the usual composition law $\circ$, possibly extended to partial functions. However, composing the corresponding approximate analyses may prove much more difficult:

1. The codomain of an approximate analysis is not necessarily the same as the corresponding exact analysis. Thus, the domain and codomain of the analyses to combine may not match anymore ($\boldsymbol{X}' \not\subseteq \boldsymbol{D}$ for $\overline{\mathcal{X}}$ and $\boldsymbol{X}'' \not\subseteq \boldsymbol{D}$ for $\underline{\mathcal{X}}$). New composition laws must then be defined for each particular case.

   An interesting case is when $\boldsymbol{X}' = \wp(\boldsymbol{X})$ or $\boldsymbol{X}'' = \wp(\boldsymbol{X})$ (the over-approximation of *transformers* will belong to this class). It is clear that it is not possible to directly compose $\overline{\mathcal{A}}[\![l]\!]$ or $\underline{\mathcal{A}}[\![l]\!]$ with $\overline{\mathcal{X}}[\![l']\!]$ or $\underline{\mathcal{X}}[\![l']\!]$. However, they can be composed with each x that belongs to the result of $\overline{\mathcal{X}}[\![l']\!]$ or $\underline{\mathcal{X}}[\![l']\!]$, and the results can be combined with each other. Much care must be given to the direction of the approximation. For instance, the following composition laws will be used in the following chapters:

$$\overline{\mathcal{A}}[\![l]\!] \,\overline{\bullet}_{\overline{\mathcal{A}}\overline{\mathcal{X}}}\, \overline{\mathcal{X}}[\![l']\!] = \lambda e. \overline{\bigcup_{x \in \overline{\mathcal{X}}[\![l']\!]e}} \overline{\mathcal{A}}[\![l]\!]x$$

$$\underline{\mathcal{A}}[\![l]\!] \,\underline{\bullet}_{\underline{\mathcal{A}}\overline{\mathcal{X}}}\, \overline{\mathcal{X}}[\![l]\!] = \lambda e. \underline{\bigcap_{x \in \overline{\mathcal{X}}[\![l']\!]e}} \underline{\mathcal{A}}[\![l]\!]x$$

   When there is no possible ambiguity, $\overline{\bullet}$ and $\underline{\bullet}$ will denote $\overline{\bullet}_{\overline{\mathcal{A}}\overline{\mathcal{X}}}$ and $\underline{\bullet}_{\underline{\mathcal{A}}\overline{\mathcal{X}}}$.

   It is clear from the previous definitions that on the domain of $\mathcal{X}[\![l']\!]$:

$$\underline{\mathcal{A}}[\![l]\!] \,\underline{\bullet}\, \overline{\mathcal{X}}[\![l']\!]\backslash \operatorname{dom}(\mathcal{X}[\![l']\!]) \sqsubseteq \mathcal{A}[\![l]\!] \circ \mathcal{X}[\![l']\!] \sqsubseteq \overline{\mathcal{A}}[\![l]\!] \,\overline{\bullet}\, \overline{\mathcal{X}}[\![l']\!]$$

   where $\overline{\mathcal{X}}[\![l']\!]\backslash \operatorname{dom}(\mathcal{X}[\![l']\!])$ denotes the restriction of $\overline{\mathcal{X}}[\![l']\!]$ to the domain of $\mathcal{X}[\![l']\!]$. A preliminary analysis to compute at least an under-approximation of this domain is then necessary.

   $\overline{\mathcal{X}}[\![l']\!]e$ and $\overline{\mathcal{X}}[\![l']\!]e$ may be infinite sets; $\overline{\bullet}$ and $\underline{\bullet}$ may thus involve an infinite sum or intersection. These compositions might not be computable in finite time, or may even not be defined if the approximations of $\bigcup$ and $\bigcap$ are not monotone. In the first case, further approximations must be made, while in the second an approach similar to what was done for least fixed points in the previous section must be adopted.

Note We do not describe the composition of $\overline{\mathcal{A}}$ and $\underline{\mathcal{A}}$ with $\underline{\mathcal{X}}$ because we will not use them in this thesis. With the type of representations we use (i.e. compact representations such as convex polyhedra), an under-approximation of $\mathcal{X}$ in $\wp(\boldsymbol{X})$ would be reduced to a single element of $\boldsymbol{X}$ or to the undefined element, and would not have much interest. This will be further detailed in due time (Section 5.4.1).

2. Even if the domain and codomain of the analyses to combine still match, using the usual composition law can lead to solutions that cannot be represented.

   Let us consider the analysis of the set of array elements read by a program; let us assume that array element sets are represented by RSDs [37]. And let us compute the solution for the following conditional instruction:

   ```
        if (i.eq.3) then
   (S)  ...= A(i)
        endif
   ```

   The RSD for instruction (S) is $A(i:i)$. It has to be composed with the result of the condition evaluation (`i.eq.3`): **if** (`i.eq.3`) **then** $A(i:i)$ **else** $\varnothing$. This cannot be represented by a RSD. Instead, an over- or an under-approximation must be used to represent this composition, respectively $A(i:i)$ and $\varnothing$.

   It is then advisable to define two approximate composition laws, $\overline{\circ}_{\overline{\mathcal{A}\mathcal{X}}}$ and $\underline{\circ}_{\underline{\mathcal{A}\mathcal{X}}}$, which must verify[13]:

$$\underline{\mathcal{A}}[\![l]\!] \,\underline{\circ}_{\underline{\mathcal{A}\mathcal{X}}}\, \underline{\mathcal{X}}[\![l']\!] \sqsubseteq \mathcal{A}[\![l]\!] \circ \mathcal{X}[\![l]\!] \sqsubseteq \overline{\mathcal{A}}[\![l]\!] \,\overline{\circ}_{\overline{\mathcal{A}\mathcal{X}}}\, \overline{\mathcal{X}}[\![l']\!]$$

   From now on, when no ambiguity is possible, $\overline{\circ}$ and $\underline{\circ}$ are respectively used instead of $\overline{\circ}_{\overline{\mathcal{A}\mathcal{X}}}$ and $\underline{\circ}_{\underline{\mathcal{A}\mathcal{X}}}$.

4.3.4 Consequences on correctness proving

When approximate analyses are defined, their correctness has to be proved: we must check if the approximation ordering is preserved for each definition. In our framework, if the exact semantic functions are defined using the operators $\{\mathrm{op}_i\}_{i=1,k}$ and the usual composition law $\circ$; and if the approximate semantic functions are derived from them by replacing each op_i either by $\overline{\mathrm{op}}_i$ or $\underline{\mathrm{op}}_i$, the composition law by either $\overline{\circ}$ or $\underline{\circ}$, or even $\overline{\bullet}$ or $\underline{\bullet}$, and the least fixed point operator by either $\overline{\mathrm{lfp}}$ or $\underline{\mathrm{lfp}}$; then the approximation ordering is preserved.

This means that the correctness of approximate semantics is ensured by the properties enforced on approximate operators. This does not mean that no correctness proof has to be made. But it is delayed until the choice of the representation: When the actual operators are defined, it must be proved that they preserve the approximation ordering. This approach allows us to separate the definition of the semantics from the choice of the representation and operators.

[13] Approximating $\mathcal{A} \circ \mathcal{X}$ by $\overline{\mathcal{A}} \,\overline{\circ}_{\overline{\mathcal{A}}\underline{\mathcal{X}}}\, \underline{\mathcal{X}}$ or $\underline{\mathcal{A}} \,\underline{\circ}_{\underline{\mathcal{A}}\overline{\mathcal{X}}}\, \overline{\mathcal{X}}$ would generally not make much sense.

4.4 Conclusion

Due to a considerable amount of efforts over the last ten years, over-approximate array regions analyses are now well-known techniques, even if debates about the choice of the representation are still alive: Time and space complexity *versus* accuracy is the main issue, but usefulness is widely acknowledged [163, 29, 101, 92, 65].

The need for under-approximations appeared only recently [165, 123, 92, 171], mainly for locality analysis or direct dependence analysis. PUGH and WONNACOTT have provided extended studies [171, 170, 176] when the underlying semantic analysis framework is the domain of PRESBURGER formulae. But no similar study was available for other types of framework, in particular convex polyhedra or RSDs, thus missing their inherent problems due to the fact that they are not closed under set union. This implies that, in these domains, under-approximate solutions of array region analyses are not uniquely defined, as over-approximations do. We show that traditional ad-hoc solutions based on iteration techniques do not give interesting results, even though problems can be lessened by more accurate representations and more complex approximations of fixed points.

We thus propose a method based on our previous experience of array region analyses [63, 65]. The idea is to perform corresponding may and must analyses at the same time, and to enhance the results of must analyses with those of may analyses according to an *exactness criterion*. In our implementation, whose results are already used in PIPS to privatize array regions [61], must regions are not even computed; instead, may regions are flagged as exact whenever the exactness criterion is true; under-approximations are thus always equal to the empty set, unless they are exact and equal to the corresponding over-approximations. In the few qualitative experiments we have already performed, this approach was sufficient. But more experiments would be necessary to know whether better under-approximations would improve array privatization in real programs. In the framework of dependence analysis for example, PUGH and WONNACOTT [171] have already shown that good under-approximations are necessary and can be obtained with PRESBURGER formulae.

The method presented in this chapter, though primarily introduced for array region analyses, is not solely restricted to them. It can be useful for every type of analysis where one of the approximation is not defined on a predefined lattice. But also when exact results are required for specific applications [46, 47]. Indeed, we show how to obtain an optimal *exactness criterion* valid for both may and must analyses. Furthermore, and from a theoretical point of view, being able to detect the exactness of an analysis gives a partial answer to a still unresolved question from COUSOT's thesis [55], about the distance between an approximation and its corresponding exact solution.

Finally, as the purpose of this chapter was to set the framework for our subsequent analyses, basic properties have been enforced on internal operators in approximate frameworks, in order to ensure that the approximation ordering is preserved; and new external composition laws have been defined to handle usual cases. Using these operators and composition laws to derive approximate semantics from the corresponding exact semantics ensures their correctness: Proofs for correctness are thereby delayed until the choice of the actual representation (for array regions: convex polyhedra, RSDs, DADs, ...), and of the actual operators.

The next two chapters rely on the framework presented in this section for the definition of several exact and approximate analyses, in particular array region analyses. The next chapter presents our subset language, and preliminary analyses necessary to define the semantics of array regions.

Chapter 5

Source Language and Preliminary Analyses

The purpose of the preceding chapter was to set the general frame of the semantic analyses involved in this thesis. The present chapter describes the small language used in our study (Section 5.1), and several preliminary analyses necessary for the definition of array region analyses: *Expression evaluation*, *transformers*, and *continuation conditions*, problems which are respectively met in Sections 5.3, 5.4 and 5.5. We will also present the analysis of *preconditions* in Section 5.6; they can be useful to enhance array region analyses. The necessity of using these analyses, which comes from the characteristics of the language, is informally shown using examples in Section 5.2.

5.1 Language

The source language of PIPS is FORTRAN 77, minus a few syntactic restrictions [23]. However, for the sake of clarity, only a small subset $\mathcal{L}$ of the language is considered here, with a few additional constructs. Its syntax provided in Figure 5.1.

Restrictions and additions to the language

The main structures of FORTRAN are kept: Assignments, sequences of instructions, conditional instructions, `DO` loops and calls to external subroutines. However, function calls are forbidden; they can be replaced by procedure calls without loss of generality, and it avoids the burden of expressions with side-effects. `if then` instructions are also left aside because they can be modeled by `if then else continue` constructs. The chosen `read` and `write` instructions are also less general than their FORTRAN counterparts, because they perform I/Os only from the standard input or to the standard output, and because they only read a single variable or write a single expression at a time.

One of the major restrictions is the lack of `goto` statements. The policy of PIPS is that program optimizations can seldom be performed on unstructured pieces of code. As a consequence, analyses and transformations of well-structured parts are carried to their best, while they are usually performed in a straightforward way on unstructured parts. This will be informally described in due time.

< *procedure* >	::=	< *header* > < *declaration* >* < *common* >* < *statement* > `end`
< *header* >	::=	`program` < *name* >
		| `subroutine` < *name* > (< *name* >*)
< *statement* >	::=	`continue`
		| `stop`
		| < *reference* > `=` < *expression* >
		| `if` < *expression* > `then` < *statement* > `else` < *statement* > `endif`
		| `do` < *range* > < *statement* > `enddo`
		| `do while` (< *expression* >) < *statement* > `enddo`
		| `read` < *reference* >
		| `write` < *expression* >
		| `call` < *name* > (< *expression* >*)
		| < *statement* >*
< *expression* >	::=	< *constant* >
		| < *reference* >
		| (< *expression* >)
		| < *unary_op* > < *expression* >
		| < *expression* > < *binary_op* > < *expression* >
< *range* >	::=	< *name* > `=` < *expression* >^{3}
< *unary_op* >	::=	`-` | `.not.`
< *binary_op* >	::=	`+` | `-` | `*` | `/` | `**`
		| `.lt.` | `.le.` | `.gt.` | `.ge.` | `.eq.` | `.neq.`
		| `.and.` | `.or.`
< *reference* >	::=	< *name* > | < *name* >(< *expression* >*)
< *declaration* >	::=	< *type* > < *name* > | < *type* > < *name* > (< *dimension* >*)
< *common* >	::=	`common/` < *name* > `/` < *name* >*
< *dimension* >	::=	< *expression* > `:` < *expression* >
< *type* >	::=	`integer` | `real` | `logical` | `double` | `complex`

Figure 5.1: Syntax of the language $\mathcal{L}$

Only one instruction has been added: `do while` loops, or their syntactic equivalents, are defined by so many languages, that they cannot be avoided. And although they are not defined in FORTRAN 77, and thus not handled by PIPS, they are available in many commercial FORTRAN 77 compilers. `do while` is in fact a very general loop instruction, which can be used to model a `do` loop:

<pre>do i = lb,ub,s
   ...
enddo</pre>
	$\equiv$	<pre>i=1
		dummy=MAX(INT((ub-lb+s)/s),0)
		do while(dummy.gt.0)
		   ...
		   i=i+s
		   dummy=dummy-1
		enddo</pre>

Hence, we will generally define our semantic analyses for the sole `do while` loops, except for array region analyses. In this case, we will show how the semantic functions can be simplified.

Semantic analyses are defined for `do` loops only for array regions and when a particular treatment may enhance the results. In this case, they are only given for normalized[1] `do` loops, to lighten the notations. Otherwise, they are only given for `do while` loops.

The syntax of variable declaration is also much simplified in $\mathcal{L}$. It mostly eliminates the syntactic sugar of FORTRAN declarations: There is no `dimension` key-word, commons are all declared after type and dimension declarations, and their declaration is made of the sole list of variable identifiers.

Equivalenced variables are not considered here. Problems due to intraprocedural aliasing of arrays are difficult problems we have chosen not to formally address in this study. However, they will be informally highlighted when necessary. This choice allows us not to use environments, but merely memory stores when defining semantic analyses. For an extension to environments, the reader is referred to [132].

Lastly, only a subset of the programs that can be written in $\mathcal{L}$ are considered: Those that meet the FORTRAN standard (except for the `do while` and `continue` instructions), and that generate no execution errors, such as integer overflows for instance.

Semantic domains

To define the semantics of this language and of additional analyses, we need several domains, which are presented in the next table.

[1] Lower bound and step equal to 1.

$\boldsymbol{C}$	constants
$\boldsymbol{N}$	names of variables, procedures, ...
$\boldsymbol{R}$	references
$\boldsymbol{V}$	values (8-bit integer, ...)
$\boldsymbol{S}$	statements
$\boldsymbol{O}_1$	unary operators
$\boldsymbol{O}_2$	binary operators
$\boldsymbol{E}$	expressions
$\boldsymbol{E}_B$	boolean expressions ($\boldsymbol{E}_B \subseteq \boldsymbol{E}$)
$\boldsymbol{\Sigma}$	memory stores
$\boldsymbol{B}$	boolean values

The first eight domains come from the definition of the language $\boldsymbol{\mathcal{L}}$. The domain $\boldsymbol{\Sigma}$ is used to represent the current state of the program, in fact the current state of the memory: It maps each reference to the value it represents[2].

$$\boldsymbol{\Sigma} : \boldsymbol{R} \longrightarrow \boldsymbol{V}$$

Any statement can thus be seen as a memory store *transformer* [55]: It converts its preceding memory store into another memory store.

Each domain is also provided with a greatest element $\top$ and a lowest element $\bot$ when necessary.

In the following sections and chapters, c denotes a constant ($c \in \boldsymbol{C}$); var a variable name ($var \in \boldsymbol{N}$), representing either a scalar variable or an array; ref a reference ($ref \in \boldsymbol{R}$); val a value ($val \in \boldsymbol{V}$); S a statement ($S \in \boldsymbol{S}$); op_1 a unary operator ($\mathrm{op}_1 \in \boldsymbol{O}_1$), and op_2 a binary operator ($\mathrm{op}_2 \in \boldsymbol{O}_2$); exp is an expression ($exp \in \boldsymbol{E}$); and lastly, σ is a memory store ($\sigma \in \boldsymbol{\Sigma}$).

Analyses

As stated in Chapter 4, the semantics of several analyses is defined in this thesis. However, some problems are left aside. First, as previously written on Page 65, we do not consider intra-procedural aliasing. Second, while the language contains procedure calls, inter-procedural issues are not addressed for preliminary analyses, because they are classical problems. For array regions, they are addressed in Part IV, apart from the intra-procedural semantics which is formally presented for intra-procedural statements in the next chapter. Most problems due to inter-procedural aliasing have already been addressed by others [161], and we prefer not to make our semantic equations heavier, but to primarily focus on problems due to control flow and context modification. This does not prevent us from dealing with problems due to encapsulation mechanisms provided by procedures, as will be illustrated in Section 5.2.

Note To reduce the length of semantic function definitions, we will omit end of statement markers (`endif` or `enddo`) whenever there is no ambiguity.

[2]Because of the no-alias hypothesis, a reference corresponds to a single memory location and vice-versa.

5.2 Necessity of Preliminary Analyses

Our ultimate goal is to analyze array accesses in order to perform program transformations, such as array privatization for instance. However, this operation is not as straightforward as one could think: It does not merely consist in scanning the subroutines to collect array accesses, because of some of the characteristics of the source language. This section provides three small examples to show that preliminary analyses are necessary.

The first example, in Figure 5.2, serves as a working example throughout this thesis. It is a contrived FORTRAN program which highlights the main difficulties of array region analysis.

```
K = FOO()
do I = 1,N
   do J = 1,N
      WORK(J,K) = J + K
   enddo
   call INC1(K)
   do J = 1,N
      WORK(J,K) = J*J - K*K
      A(I) = A(I)+WORK(J,K)+WORK(J,K-1)
   enddo
enddo
```

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

Figure 5.2: Working example.

The goal is to privatize array `WORK`. The condition is that any iteration of the `I` loop neither imports nor exports any element of the array `WORK`. In other words, if there is a read reference to an element of `WORK`, it has been previously initialized in the same iteration, and it is not reused in the subsequent iterations (we assume that the array `WORK` is not used anymore after the `I` loop).

In order to verify that the set of written references to `WORK` *covers* the set of subsequent read references, we need to compare them. However, both sets depend on the value of the variable `K`, which is unknown at the entry of the `I` loop, and is modified by the call. Thus, the modification of the value of `K` has to be modeled to be taken into account during the comparison[3]. This is represented by a *transformer* [105], which acts as a transfer function for other data flow problems, such as array region analyses. Depending on the direction of the data flow analysis, two transfer functions may be necessary: *Transformers* to model the modification by a statement execution of input values into output ones; and the reverse transformation to retrieve the input values from the output ones. These first analyses both rely on the static *evaluation of expressions*, which is treated as another analysis.

However, this may still be insufficient to compare two array regions. Consider for instance the function `D` in the program excerpt of Figure 2.1. Since we do not know the relative values of `J` and `JP`, and of `K` and `KP`, we cannot accurately compare the regions

[3]Of course, in this very simple case, a better solution would be to expand inline the call to `INC1`, and to propagate the value of `K`. However, this may not be always possible or even advisable in the general case. Indeed, the choice of the right transformations and their ordering is a very difficult problem [173, 168].

corresponding to the six array accesses, or compute an exact summary; the six regions must either be merged to form an approximate summary, or kept in a list of length six. But we can infer from the three call sites of `D` in `EXTR` that `J=JP` and `KP=K+1`. With this information, the six regions in `D` can be accurately compared, or merged to form a unique exact summary. Such information is called *precondition*. The precise relations with array region analysis was still unclear up to now. This will be discussed in the next chapter, Section 6.8.

Another problem is due to `stop` statements. In the program of Figure 5.3, a naive approach would add `A(K)` to the set of array elements accessed by the program `P_STOP`. This is a valid, though quite rough, over-approximation of the set of written elements; however, because of the `stop` statement hidden in the call to `S_STOP`, `A(K)` is accessed only when `K<=10`, and hence can belong to the under-estimate of the written element set only when this conditions is met. A preliminary analysis of *continuation conditions* is necessary to check under which conditions there exists an execution path from the beginning of the program to the statement in which the array element is referenced.

```
program P_STOP              subroutine S_STOP(K)
integer A(5),K              if (K.gt.10) then
...                            stop
call S_STOP(K)              endif
A(K) = ...                  end
...
end
```

Figure 5.3: Halting problem

These preliminary analyses are presented in the next sections.

5.3 Expression Evaluation

Many analyses rely on a representation of program expressions: loop bounds, test conditions, array subscripts, ... are examples of expressions whose values are often needed to deduce other types of information about the program behavior. These values can be provided by a preliminary analysis which we call *expression evaluation* and which is denoted by $\mathcal{E}$. During a program execution, expressions are evaluated in the current memory store. The semantics of an expression is thus a function from the domain of stores $\boldsymbol{\Sigma}$ to the domain of values $\boldsymbol{V}$:

$$\mathcal{E} : \boldsymbol{E} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \boldsymbol{V})$$

According to the syntax of $\boldsymbol{\mathcal{L}}$ previously described, expressions can be combined using unary and binary operators, to build more complex expressions. In the standard semantics, these operators take values resulting from the evaluation of their arguments as inputs, and they return another value. However, values are seldom available at compile time, and it is more convenient to handle symbolic expressions, that is to say functions from $\boldsymbol{\Sigma}$ to $\boldsymbol{V}$. This requires a non-standard semantics of operators, described by two functions, $\mathcal{O}_1$ for unary operators and $\mathcal{O}_2$ for binary operators. If we denote

$(\boldsymbol{\Sigma} \longrightarrow \boldsymbol{V})$ by $\boldsymbol{Symb}$, then:

$$\begin{aligned}
\mathcal{O}_1 &: \boldsymbol{O}_1 \longrightarrow (\boldsymbol{Symb} \longrightarrow \boldsymbol{Symb}) \\
\mathcal{O}_2 &: \boldsymbol{O}_2 \longrightarrow (\boldsymbol{Symb} \times \boldsymbol{Symb} \longrightarrow \boldsymbol{Symb})
\end{aligned}$$

Obviously, this non-standard semantics is closely related to the standard semantics described by the FORTRAN 77 standard [8].

The exact non-standard semantics of expressions can then be defined as:

$$\begin{array}{lcl}
\mathcal{E}[\![c]\!] & = & \lambda\sigma.c \\
\mathcal{E}[\![\mathit{ref}]\!] & = & \lambda\sigma.\sigma(\mathit{ref}) \\
\mathcal{E}[\![\mathrm{op}_1\ \mathit{exp}]\!] & = & \mathcal{O}_1[\![\mathrm{op1}]\!](\mathcal{E}[\![\mathit{exp}]\!]\sigma) \\
\mathcal{E}[\![\mathit{exp}_1\ \mathrm{op}_2\ \mathit{exp}_2]\!] & = & \mathcal{O}_2[\![\mathrm{op}_2]\!](\mathcal{E}[\![\mathit{exp}_1]\!], \mathcal{E}[\![\mathrm{exp}_2]\!])
\end{array}$$

In practice, all types of symbolic expressions are generally not handled, because of the resulting complexity. Real valued expressions are a typical example of expressions left aside by existing parallelizers. As a consequence, expression evaluation is abstracted by two approximate analyses $\overline{\mathcal{E}}$ and $\underline{\mathcal{E}}$, whose image domain $\widetilde{\boldsymbol{Symb}}$ is a subset of the domain of symbolic expressions:

$$\begin{aligned}
\overline{\mathcal{E}} &: \boldsymbol{E} \longrightarrow \widetilde{\boldsymbol{Symb}} \\
\underline{\mathcal{E}} &: \boldsymbol{E} \longrightarrow \widetilde{\boldsymbol{Symb}}
\end{aligned}$$

> For example, PIPS only handles expressions which can be represented by conjunctions of affine integer equalities or inequalities, that is to say convex polyhedra over $\mathbb{Z}$ (see Section 8.1).

We must provide $(\boldsymbol{\Sigma} \longrightarrow \boldsymbol{V})$ with a lattice structure to enable analyses. A possible, and very simple, ordering is the following:

$$\begin{aligned}
&\forall f_1, f_2 \in \boldsymbol{Symb}, \\
&\qquad f_1 \sqsubseteq f_2 \iff \mathrm{dom}(f_1) \subseteq \mathrm{dom}(f_2) \text{ and } \forall x \in \mathrm{dom}(f_1),\ f_1(x) \leq f_2(x)
\end{aligned}$$

$\boldsymbol{V}$ being provided with a flat lattice structure. The extremal points of the lattice of symbolic expressions can be defined as:

$$\begin{aligned}
\top &: \boldsymbol{Symb} \\
&\quad \sigma \longrightarrow \top_{\boldsymbol{V}} \\
&\quad \mathrm{dom}(\top) = \boldsymbol{\Sigma} \\
\bot &: \boldsymbol{Symb} \\
&\quad \mathrm{dom}(\bot) = \varnothing
\end{aligned}$$

In this lattice, a straightforward definition of the over-approximate semantics of operators can be:

$$\begin{aligned}
\overline{\mathcal{O}}_1 &: \widetilde{\boldsymbol{Symb}} \longrightarrow \widetilde{\boldsymbol{Symb}} \\
\overline{\mathcal{O}}_2 &: \widetilde{\boldsymbol{Symb}} \times \widetilde{\boldsymbol{Symb}} \longrightarrow \widetilde{\boldsymbol{Symb}}
\end{aligned}$$

$$
\begin{array}{lcl}
\overline{\mathcal{O}}_1[\![\mathrm{op}_1]\!](\top) & = & \top \\
\overline{\mathcal{O}}_1[\![\mathrm{op}_1]\!](s\_exp) & = & \textbf{If } \mathcal{O}_1[\![\mathrm{op}_1]\!](s\_exp) \in \widetilde{\boldsymbol{Symb}} \\
 & & \textbf{Then } \mathcal{O}_1[\![\mathrm{op}_1]\!](s\_exp) \\
 & & \textbf{Else} \top \\
\overline{\mathcal{O}}_2[\![\mathrm{op}_2]\!](\top,\top) & = & \top \\
\overline{\mathcal{O}}_2[\![\mathrm{op}_2]\!](\top, s\_exp) & = & \top \\
\overline{\mathcal{O}}_2[\![\mathrm{op}_2]\!](s\_exp, \top) & = & \top \\
\overline{\mathcal{O}}_2[\![\mathrm{op}_2]\!](s\_exp_1, s\_exp_2) & = & \textbf{If } \mathcal{O}_2[\![\mathrm{op}_2]\!](s\_exp_1, s\_exp_2) \in \widetilde{\boldsymbol{Symb}} \\
 & & \textbf{Then } \mathcal{O}_2[\![\mathrm{op}_2]\!](s\_exp_1, s\_exp_2) \\
 & & \textbf{Else} \top
\end{array}
$$

And similar definitions using $\perp$ instead of $\top$ could also be given for $\underline{\mathcal{O}}_1$ and $\underline{\mathcal{O}}_2$.

The following definition of $\overline{\mathcal{E}}$ can then be derived from the previous definitions of $\overline{\mathcal{O}}_1$ and $\overline{\mathcal{O}}_2$:

$$
\begin{array}{lcl}
\overline{\mathcal{E}}[\![c]\!] & = & \textbf{if } \lambda\sigma.c \in \widetilde{\boldsymbol{Symb}} \textbf{ then } \lambda\sigma.c \textbf{ else } \top \\
\overline{\mathcal{E}}[\![ref]\!] & = & \textbf{if } \lambda\sigma.\sigma(ref) \in (\boldsymbol{\Sigma} \longrightarrow \boldsymbol{V})^{\sim} \textbf{ then } \lambda\sigma.\sigma(ref) \textbf{ else } \top \\
\overline{\mathcal{E}}[\![\mathrm{op}_1\ exp]\!] & = & \overline{\mathcal{O}}_1[\![\mathrm{op}_1]\!](\overline{\mathcal{E}}[\![exp]\!]\sigma) \\
\overline{\mathcal{E}}[\![exp_1\ \mathrm{op}_2\ exp_2]\!] & = & \overline{\mathcal{O}}_2[\![\mathrm{op}_2]\!](\overline{\mathcal{E}}[\![exp_1]\!], \overline{\mathcal{E}}[\![\mathrm{exp}_2]\!])
\end{array}
$$

And similarly for $\underline{\mathcal{E}}$:

$$
\begin{array}{lcl}
\underline{\mathcal{E}}[\![c]\!] & = & \textbf{if } \lambda\sigma.c \in \widetilde{\boldsymbol{Symb}} \textbf{ then } \lambda\sigma.c \textbf{ else } \perp \\
\underline{\mathcal{E}}[\![ref]\!] & = & \textbf{if } \lambda\sigma.\sigma(ref) \in (\boldsymbol{\Sigma} \longrightarrow \boldsymbol{V})^{\sim} \textbf{ then } \lambda\sigma.\sigma(ref) \textbf{ else } \perp \\
\underline{\mathcal{E}}[\![\mathrm{op}_1\ exp]\!] & = & \underline{\mathcal{O}}_1[\![\mathrm{op}_1]\!](\underline{\mathcal{E}}[\![exp]\!]\sigma) \\
\underline{\mathcal{E}}[\![exp_1\ \mathrm{op}_2\ exp_2]\!] & = & \underline{\mathcal{O}}_2[\![\mathrm{op}_2]\!](\underline{\mathcal{E}}[\![exp_1]\!], \underline{\mathcal{E}}[\![\mathrm{exp}_2]\!])
\end{array}
$$

Obviously, this last definition is a trivial definition, given here as an example. But it is far from being optimal, and more aggressive techniques can be used to improve the results. In PIPS, boolean expressions are handled more accurately by using the semantic properties of boolean operators:

> For example, the expression `(I.leq.3).and.(A(I).lt.1)` is over-approximated by $\sigma(\mathtt{I}) \leq 3$, instead of $\top$ as with the previous definition of $\overline{\mathcal{E}}$. However, it is still under-approximated by $\perp$.

PUGH and WONNACOTT [170, 176] have also proposed several techniques (*unknown* variables, and uninterpreted functions) to give better approximations of expressions with non-affine terms, such as references to array elements.

> An example provided by William PUGH[4] is the evaluation of the conditional expression `A(x,y).eq.A(i,j)` which can be under-approximated by $x = i \wedge y = j$.

Moreover, boolean expression evaluation can be refined by successive local iterations, as defined by GRANGER [84]: Sub-expressions are evaluated in a first iteration;

[4] Personal communication.

during the next iteration, each sub-expression evaluation is refined using the evaluation of the other expressions; and so on. The same kind of effect is obtained with PIPS by performing a partial evaluation phase, and a second evaluation of expressions afterwards.

In addition, we could extend the evaluation of expressions to generate sets of values instead of values:

$$\overline{\mathcal{E}} : \boldsymbol{E} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \wp(\boldsymbol{V}))$$

Let us consider the expression `J+M**2`. With sets of values represented by convex polyhedra, the different stages of its evaluation could be:

$$\begin{array}{lcl}
\overline{\mathcal{E}}[\![\mathtt{J}]\!] & = & \lambda\sigma.\{v : v = \sigma(\mathtt{J})\} \\
\overline{\mathcal{E}}[\![\mathtt{M**2}]\!] & = & \lambda\sigma.\{v : v \geq \sigma(\mathtt{M})\} \\
\overline{\mathcal{E}}[\![\mathtt{J+M**2}]\!] & = & \overline{\mathcal{E}}[\![\mathtt{J}]\!] \oplus \overline{\mathcal{E}}[\![\mathtt{M**2}]\!] \\
 & = & \lambda\sigma.\{v : \exists v1, v2,\ v = v_1 + v_2 \wedge v_1 = \sigma(\mathtt{J}) \wedge v_2 \geq \sigma(\mathtt{M})\} \\
 & = & \lambda\sigma.\{v : v \geq \sigma(\mathtt{J}) + \sigma(\mathtt{M})\}
\end{array}$$

$\oplus$ being appropriately defined.

Many other techniques could certainly be used to improve the evaluation of expressions at compile time. However, our purpose here is only to show how the preliminary analyses which are necessary for array region analyses could be defined. We therefore do not try to give a more powerful formal definition.

Composing other analyses with $\mathcal{E}$ We will use $\mathcal{E}$ and its approximations mainly to evaluate array indices, assignments or conditional expressions. In the case of conditional expressions, we will need to compose $\mathcal{E}$, $\overline{\mathcal{E}}$ or $\underline{\mathcal{E}}$ with other analyses, such as *transformers* or *array regions*.

For example, in the following piece of code

```
if (i.lt.3) then
   A(i) =
endif
```

we may need to know that `A(i)` is written only when `i.lt.3` evaluates to *true*.

However, as will be seen later, most of our analyses take memory stores as input. For example, we have seen that array regions are functions from the set of memory stores to $\wp(\mathbb{Z}^d)$:

$$\mathcal{R} : \boldsymbol{\Sigma} \longrightarrow \wp(\mathbb{Z}^d)$$

Hence, a direct composition with a ***Symb*** function ($\mathcal{R} \circ \mathcal{E}$) is not possible, because $\mathcal{E}$ is a function from the set of memory stores to the set of values: Its codomain does not match the domain of $\mathcal{R}$. We thus introduce a new analysis, which is in fact the characteristic function of $\mathcal{E}$ on the domain of boolean values. It is denoted by $\mathcal{E}_c$ to recall its relation with $\mathcal{E}$, c standing for *characteristic*.

$$\begin{array}{l}
\mathcal{E}_c : \boldsymbol{E}_B \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \boldsymbol{\Sigma}) \\
\qquad exp \longrightarrow \mathcal{E}_c[\![exp]\!] = \lambda\sigma.\mathbf{if}\ \mathcal{E}[\![exp]\!]\sigma\ \mathbf{then}\ \sigma\ \mathbf{else}\ \bot
\end{array}$$

$\underline{\mathcal{E}}_c$ and $\overline{\mathcal{E}}_c$ are similarly defined from $\underline{\mathcal{E}}$ and $\overline{\mathcal{E}}$. Their codomain is the set of memory stores, and their results can thus be used as the input of other analyses, such as transformers or array region analyses.

5.4 Transformers and Inverse Transformers

The current section is devoted to the semantics of statements. As stated before, and following a traditional point of view [55, 132, 108], statements can be seen as memory store *transformers*, that is to say functions mapping stores to stores. As stated in Section 5.2, two types of transfer functions will be necessary to define our array region analyses: We call *transformers* those modeling the transformation of an input store into an output store resulting of the execution of the considered statement; and *inverse transformers* give the input store from the output store. Both analyses are backward analyses.

5.4.1 Transformers

From its informal above definition, the prototype of the analysis is the following:

$$\mathcal{T} : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \boldsymbol{\Sigma})$$

The definition of $\mathcal{T}$ is provided in Appendix D, Tables D.1 and D.2, but some details are given below.

`stop` statement FORTRAN `stop`s cause the termination of the whole program, as opposed to the current scope or procedure; the program then does not exist anymore. We have chosen to model this situation by the unknown memory store:

$$\mathcal{T}[\![\texttt{stop}]\!] = \lambda\sigma.\bot$$

Assignment An assignment has the effect of replacing the value of the assigned reference (variable or array element) by a computed value. The resulting store is denoted by $\sigma_{[\mathit{ref} \leftarrow \cdots]}$ to show that it is identical to the previous store σ, but for the value of *ref*.

Sequence of instructions $S_1; S_2$ The execution goes trough S_1, and then through S_2. So, the initial memory store is first modified by $\mathcal{T}[\![S_1]\!]$, and then by $\mathcal{T}[\![S_2]\!]$:

$$\mathcal{T}[\![S_1; S_2]\!] = \mathcal{T}[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]$$

`if` statement Either the condition C evaluates to *true* ($\mathcal{E}_c[\![C]\!]\sigma = \sigma$) and the first branch is executed ($\mathcal{T}[\![S_1]\!]$), or ($\cup$) it evaluates to *false* ($\mathcal{E}_c[\![\texttt{.not.}C]\!]\sigma = \sigma$) and the execution goes through the second branch:

$$\mathcal{T}[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] = (\mathcal{T}[\![S_1]\!] \circ \mathcal{E}_c[\![C]\!]) \cup (\mathcal{T}[\![S_2]\!] \circ \mathcal{E}_c[\![\texttt{.not.}C]\!])$$

`do while` loop If the condition evaluates to *false* ($\mathcal{E}_c[\![\texttt{.not.}C]\!]\sigma = \sigma$), then the loop is not executed, and the memory store is not modified. On the contrary, if the condition evaluates to *true* ($\mathcal{E}_c[\![C]\!]\sigma = \sigma$), S is executed ($\mathcal{T}[\![S]\!]$), and the process loops:

$$\mathcal{T}[\![\texttt{do while(}C\texttt{) } S]\!] = \mathcal{T}[\![\texttt{do while(}C\texttt{) } S]\!] \circ \mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!] \cup \mathcal{E}_c[\![\texttt{.not.}C]\!]$$

This is a recursive function. Its best definition is given by:

$$\mathcal{T}[\![\texttt{do while(}C\texttt{) } S]\!] = \mathrm{lfp}(\lambda f. f \circ \mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!] \cup \mathcal{E}_c[\![\texttt{.not.}C]\!])$$

These semantic functions usually are not computable, except for very simple programs with no input instructions, no loops, and no complicated instructions. To be able to analyze complex applications, approximations must be used. One of the reasons for which $\mathcal{T}$ is not computable, comes from the fact that some variable values are unknown at compile time. It is therefore impossible to exactly represent the current memory store; but we can compute a set of stores containing the exact store. This is the definition of our over-approximation.

Henceforth, we define $\overline{\mathcal{T}}$ as a bunch of functions from the set of stores $\mathbf{\Sigma}$, to the powerset $\wp(\mathbf{\Sigma})$. In practice, it is often impossible to represent any set of stores $S_\sigma \in \wp(\mathbf{\Sigma})$, but only set of stores sharing common properties (for instance sets represented by convex polyhedra as in PIPS). $\overline{\mathcal{T}}$ functions are thereby functions to $\tilde{\wp}(\mathbf{\Sigma}) \subseteq \wp(\mathbf{\Sigma})$:

$$\overline{\mathcal{T}} : \boldsymbol{S} \longrightarrow (\mathbf{\Sigma} \longrightarrow \tilde{\wp}(\mathbf{\Sigma}))$$

These functions, which are provided in Table D.2 are derived from the definition of $\mathcal{T}$ using approximate operators, except for $\overline{\mathcal{T}}[\![\texttt{continue}]\!]$, $\overline{\mathcal{T}}[\![\texttt{stop}]\!]$, $\overline{\mathcal{T}}[\![ref = exp]\!]$, $\overline{\mathcal{T}}[\![\texttt{read}\ ref]\!]$ and $\overline{\mathcal{T}}[\![\texttt{write}\ exp]\!]$ which depend on the chosen representation: It explains their rather unprecise definition such as:

$$\overline{\mathcal{T}}[\![var = exp]\!] = \lambda\sigma.\overline{\{\sigma_{[var \leftarrow \mathcal{E}[\![exp]\!]\sigma]}\}}$$

This is more a property to ensure a safe approximation than a constructive definition, which is delayed until the choice of the representation. Notice also the use of $\overline{\bullet}$ in the definition of $\overline{\mathcal{T}}[\![S_1; S_2]\!]$:

$$\overline{\mathcal{T}}[\![S_1; S_2]\!] = \overline{\mathcal{T}}[\![S_2]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}[\![S_1]\!]$$

The result of $\overline{\mathcal{T}}[\![S_1]\!]$ is a set of stores, and $\overline{\mathcal{T}}[\![S_2]\!]$ takes a single store as input; as explained in Chapter 4, a particular composition law, denoted by $\overline{\bullet}$, must then be used.

The next step should now be to define the corresponding under-approximate analysis, $\underline{\mathcal{T}}$, using the same approximation ordering:

$$\forall S \in \boldsymbol{S},\ \forall \sigma \in \Sigma,\ \{\sigma' : \sigma' \in \underline{\mathcal{T}}[\![S]\!]\sigma\} \subseteq \{\sigma' : \sigma' = \mathcal{T}[\![S]\!]\sigma\}$$

Since $\{\sigma' : \sigma' = \mathcal{T}[\![S]\!]\sigma\}$ is reduced to a singleton, $\underline{\mathcal{T}}[\![S]\!]\sigma$ also contains a single element, and can thus be described as a function from $\mathbf{\Sigma}$ to $\mathbf{\Sigma}$. And we necessarily have:

$$\forall S \in \boldsymbol{S},\ \forall \sigma \in \mathrm{dom}(\underline{\mathcal{T}}[\![S]\!]),\ \underline{\mathcal{T}}[\![S]\!]\sigma = \mathcal{T}[\![S]\!]\sigma$$

Hence, either $\underline{\mathcal{T}}[\![S]\!]\sigma$ gives the same result as $\overline{\mathcal{T}}[\![S]\!]\sigma$, or it is undefined. For this type of information to be useful in practice, the chosen representation must be very precise, or the result would most of the time be the undefined store.

Let us consider the following piece of code as an example.

```
if (J.eq.N) then
   N = N+1
endif
if (N=5) then
   STOP
endif
```

The exact transformer corresponding to the whole sequence of instructions is:

$$\begin{array}{lll}
\mathcal{T}[\![...]\!] = \lambda\sigma.\sigma' : & \textbf{if} & \sigma(\mathtt{J}) = \sigma(\mathtt{N}) \\
& \textbf{then} & \textbf{if } \sigma(\mathtt{N}) + 1 = 5 \\
& & \textbf{then } \sigma' = \bot \\
& & \textbf{else } \sigma'(\mathtt{N}) = \sigma(\mathtt{N}) + 1 \wedge \forall r \in \boldsymbol{R}, r \neq \boldsymbol{N} \implies \sigma'(r) = \sigma(r) \\
& \textbf{else} & \textbf{if } \sigma(\mathtt{N}) = 5 \\
& & \textbf{then } \sigma' = \bot \\
& & \textbf{else } \sigma' = \sigma
\end{array}$$

This can only be represented by PRESBURGER formulae. But the representation is already quite large, considering the smallness of the previous piece of code. For real large applications, with complicated control flows, and several procedures, this would soon become unmanageable.

And with a less precise representation, such as convex polyhedra for instance, we would get the following result:

$$\underline{\mathcal{T}}[\![...]\!] = \lambda\sigma.\bot$$

which is not very interesting.

However, transformers are not our target analysis. They will be necessary only when propagating other analysis results, such as array regions. And we have already seen in Section 4.3.3 how to get a safe under-approximation from the composition of an under-approximation with an over-approximation:

$$\underline{\mathcal{R}}[\![S]\!] \underline{\bullet}_{\underline{\mathcal{R}}\overline{\mathcal{T}}} \overline{\mathcal{T}}[\![S']\!] = \lambda\sigma. \underline{\bigcap}_{\sigma' \in \overline{\mathcal{T}}[\![S']\!]\sigma} \underline{\mathcal{R}}[\![S]\!]\sigma'$$

We will show in Chapter 8 that this composition law can be effectively used to compute convex under-approximate regions using over-approximate transformers; and we therefore do not define under-approximate transformers.

However, there is a small problem with this definition. $\overline{\mathcal{T}}[\![S']\!]$ is an over-approximation, and as such potentially misses undefined stores resulting from the execution of S'.

In the following piece of code

```
if (R.gt.0.D0) then
   STOP
endif
```

the transformer given by PIPS is:

$$\overline{\mathcal{T}}[\![...]\!] = \lambda\sigma.\{\sigma' : \sigma' = \sigma\}$$

whereas some executions lead to the undefined store.

Thus, the undefined store potentially belongs to $\overline{\mathcal{T}}[\![S']\!]\sigma$. As a consequence, $\underline{\mathcal{R}}[\![S]\!]\bot = \emptyset$ is one of the arguments of the intersection $\underline{\bigcap}_{\sigma' \in \overline{\mathcal{T}}[\![S']\!]\sigma} \underline{\mathcal{R}}[\![S]\!]\sigma'$, which is then also equal to the empty set! Thus, before applying this composition law, we must be sure that the empty store cannot be the result of the execution of S' from the input store σ. This information will be provided by the *continuation condition* analysis, described in Section 5.5.

5.4.2 Inverse transformers

Transformers model the modification, by a statement execution, of an input memory store into an output memory store. We will see in the remainder of this chapter and in Chapter 6 for READ, WRITE and IN regions, that transformers are useful to define backward semantic analyses. But, for forward problems such as OUT regions, we need the reverse transformation:

> *Given a memory store reached after the execution of a particular statement, what is the corresponding input memory store?*

The solution would be to define the inverse function of $\mathcal{T}$:

$$\begin{aligned}\mathcal{T}^{-1} : \boldsymbol{S} &\longrightarrow (\boldsymbol{\Sigma} \longrightarrow \boldsymbol{\Sigma}) \\ S &\longrightarrow \mathcal{T}^{-1}[\![S]\!] : \mathcal{T}[\![S]\!] \circ \mathcal{T}^{-1}[\![S]\!] = \lambda\sigma.\sigma\end{aligned}$$

Unfortunately, $\mathcal{T}$ is not invertible in the general case.

> For instance, what is the value of `K` before instructions `read K` or `K=5` given the value of `K` after the execution of the statement?
>
> For a `do while` statement, how do we know whether the loop has been executed at least once, and thus that the input store is not identical to the output store?

Hence, we can only enforce necessary conditions on the inverse of $\mathcal{T}$. This leads to a definition of an over-approximation[5] of this inverse function:

$$\begin{aligned}\overline{\mathcal{T}}^{-1} : \boldsymbol{S} &\longrightarrow (\boldsymbol{\Sigma} \longrightarrow \tilde{\wp}(\boldsymbol{\Sigma})) \\ S &\longrightarrow \overline{\mathcal{T}}^{-1}[\![S]\!] : \forall\sigma \in \boldsymbol{\Sigma},\ \exists\sigma',\ \sigma' \in \overline{\mathcal{T}}^{-1}[\![S]\!]\sigma \wedge \mathcal{T}[\![S]\!]\sigma' = \sigma\end{aligned}$$

The complete definition of $\overline{\mathcal{T}}^{-1}$ is given in Table D.3 on Page 288. The most common control structures are detailed below.

Sequence of instructions $S_1; S_2$ The intuition is that the output store must first be propagated backwards through S_2 to get the output store of S_1, and then through S_1 to get the input store before S_1:

$$\overline{\mathcal{T}}^{-1}[\![S_1; S_2]\!] = \overline{\mathcal{T}}^{-1}[\![S_1]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}^{-1}[\![S_2]\!]$$

We must however check that $\overline{\mathcal{T}}^{-1}[\![S_1; S_2]\!]$ meets the property enforced on the general definition of $\overline{\mathcal{T}}^{-1}$:

Property 5.1
If

$$\forall\sigma \in \boldsymbol{\Sigma}, \exists\sigma',\ \sigma' \in \overline{\mathcal{T}}^{-1}[\![S_1]\!]\sigma \wedge \mathcal{T}[\![S_1]\!]\sigma' = \sigma$$

and

$$\forall\sigma \in \boldsymbol{\Sigma}, \exists\sigma',\ \sigma' \in \overline{\mathcal{T}}^{-1}[\![S_2]\!]\sigma \wedge \mathcal{T}[\![S_2]\!]\sigma' = \sigma$$

then

$$\forall\sigma \in \boldsymbol{\Sigma}, \exists\sigma',\ \sigma' \in \overline{\mathcal{T}}^{-1}[\![S_1; S_2]\!]\sigma \wedge \mathcal{T}[\![S_1; S_2]\!]\sigma' = \sigma$$

[5] The inverse transformation of $\mathcal{T}$ could be exactly and constructively defined by keeping track of the current context in the current execution path; for instance, a stack could be used as an argument to the new semantic function to gather the required information. But this would require a change of the domain of $\mathcal{T}^{-1}$ to take this additional argument into account. This is not done for the sake of simplicity. Besides, we only need an over-approximation of $\mathcal{T}^{-1}$, and it is readily provided.

Proof Let σ be a memory store. From the assumptions,

$$\exists \sigma_2, \sigma_2 \in \overline{\mathcal{T}}^{-1}[\![S_2]\!]\sigma \wedge \mathcal{T}[\![S_2]\!]\sigma_2 = \sigma$$

and similarly,

$$\exists \sigma_1, \sigma_1 \in \overline{\mathcal{T}}^{-1}[\![S_1]\!]\sigma_2 \wedge \mathcal{T}[\![S_1]\!]\sigma_1 = \sigma_2$$

We therefore simultaneously have:

$$\sigma_1 \in \overline{\mathcal{T}}^{-1}[\![S_1]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}^{-1}[\![S_2]\!]\sigma = \overline{\mathcal{T}}^{-1}[\![S_1; S_2]\!]\sigma$$

and

$$\mathcal{T}[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]\sigma_1 = \mathcal{T}[\![S_1; S_2]\!]\sigma_1 = \sigma$$

$\overline{\mathcal{T}}^{-1}[\![S_1; S_2]\!]$ therefore meets the properties enforced on the general definition of $\overline{\mathcal{T}}^{-1}_{\text{inv}}$. □

`if` statement With the same reasoning as for the sequence, the output store must be propagated backwards either through the first (S_1) or second branch (S_2) to get the input store. However, the output store comes from the input store of S_1 (resp. S_2) only if condition C (resp. `.not.`C) is met:

$$\overline{\mathcal{T}}^{-1}[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] = (\overline{\mathcal{E}}_c[\![C]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}^{-1}[\![S_1]\!]) \,\overline{\cup}\, (\overline{\mathcal{E}}_c[\![\texttt{.not.}C]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}^{-1}[\![S_2]\!])$$

Again, we must check that this is an over-approximation of the inverse function of $\mathcal{T}[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!]$.

Property 5.2
Let test = ***if*** C ***then*** S_1 ***else*** S_2*.*

If

$$\forall \sigma \in \mathbf{\Sigma}, \exists \sigma',\ \sigma' \in \overline{\mathcal{T}}^{-1}[\![S_1]\!]\sigma \wedge \mathcal{T}[\![S_1]\!]\sigma' = \sigma$$

and

$$\forall \sigma \in \mathbf{\Sigma}, \exists \sigma',\ \sigma' \in \overline{\mathcal{T}}^{-1}[\![S_2]\!]\sigma \wedge \mathcal{T}[\![S_2]\!]\sigma' = \sigma$$

then

$$\forall \sigma \in \mathbf{\Sigma}, \exists \sigma',\ \sigma' \in \overline{\mathcal{T}}^{-1}[\![test]\!]\sigma \wedge \mathcal{T}[\![test]\!]\sigma' = \sigma$$

Proof Let σ be a memory store resulting from the execution of test. From the assumptions, we know that:

$$\exists \sigma_1, \sigma_1 \in \overline{\mathcal{T}}^{-1}[\![S_1]\!]\sigma \wedge \mathcal{T}[\![S_1]\!]\sigma_1 = \sigma$$

and

$$\exists \sigma_2, \sigma_2 \in \overline{\mathcal{T}}^{-1}[\![S_2]\!]\sigma \wedge \mathcal{T}[\![S_2]\!]\sigma_2 = \sigma$$

However, S_1 is executed only if C evaluates to *true* in the store σ_1, that is to say if $\mathcal{E}[\![C]\!]\sigma_1 = \sigma_1$, which implies $\overline{\mathcal{E}}[\![C]\!]\sigma_1 = \sigma_1$. In this case,

$$\sigma_1 \in \overline{\mathcal{E}}[\![C]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}^{-1}[\![S_1]\!]\sigma \sqsubseteq \overline{\mathcal{T}}^{-1}[\![\text{test}]\!]\sigma$$

and

$$\mathcal{T}[\![S_1]\!] \circ \mathcal{E}[\![C]\!]\sigma_1 = {}^6\mathcal{T}[\![\text{test}]\!] = \sigma$$

We would have the same result for the second branch. Therefore, our property holds. □

`do while` loop The statement `do while(`C`)` S is equivalent to

`if` C `then` (S;`do while(`C`)` S) `else continue`

From the definitions of $\overline{\mathcal{T}}^{-1}$ for tests and sequences of instructions, we can deduce that:

$$\overline{\mathcal{T}}^{-1}[\![\texttt{do while(}C\texttt{)}\ S]\!] = (\overline{\mathcal{E}}[\![C]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}^{-1}[\![S]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}^{-1}[\![\texttt{do while(}C\texttt{)}\ S]\!]) \\ \overline{\cup}(\overline{\mathcal{E}}_c[\![\texttt{.not.}C]\!]) \,\overline{\bullet}\, \lambda\sigma.\overline{\{\sigma\}}\})$$

This is a recursive function whose solution is given by:

$$\overline{\mathcal{T}}^{-1}[\![\texttt{do while(}C\texttt{)}\ S]\!] = \overline{\text{lfp}}(\lambda f.(\overline{\mathcal{E}}[\![C]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}^{-1}[\![S]\!] \,\overline{\bullet}\, f)\,\overline{\cup}(\overline{\mathcal{E}}_c[\![\texttt{.not.}C]\!]) \,\overline{\bullet}\, \lambda\sigma.\overline{\{\sigma\}}\}))$$

Since this definition is built from the definitions of $\overline{\mathcal{T}}^{-1}$ for tests and sequences of instructions, we do not need to check that it is an over-approximation of the inverse function of $\mathcal{T}$ for the same construct. It is already ensured by Properties 5.1 and 5.2.

$\overline{\mathcal{T}}^{-1}$ is not an analysis specific to our OUT regions. It is also used by others. For instance, it is the transfer function informally used by COUSOT and HALBWACHS to derive linear relationships between variables in programs [59]. This is also the transformation used in PIPS to compute *preconditions* (see Section 5.6) and, as stated before, OUT regions (see Section 6.6).

5.4.3 Implementation details

It is worthwhile noticing that for each statement S both semantic functions $\overline{\mathcal{T}}[\![S]\!]$ and $\overline{\mathcal{T}}^{-1}[\![S]\!]$ can be provided by a single relation between the input and output memory store, $t(\sigma_{\text{in}}, \sigma_{\text{out}})$:

$$\overline{\mathcal{T}}[\![S]\!] = \lambda\sigma.\{\sigma_{\text{out}} : t(\sigma, \sigma_{\text{out}})\}$$
$$\overline{\mathcal{T}}^{-1}[\![S]\!] = \lambda\sigma.\{\sigma_{\text{in}} : t(\sigma_{\text{in}}, \sigma)\}$$

Let us take an example. In the program of Figure 5.2, the call to `INC1` increments the value of `K` by one. This is represented in PIPS by:

```
T(K) {K == K#init +1}
```

[6] Because of the semantics of the `if` construct: The two branches are exclusive.

K stands for the value of variable K in the output store, and K#init in the input store. Relations for non-modified variables (I==I#init) are omitted. To distinguish non-modified variables from modified variables for which the relation could not be computed, the list of modified variables is kept in the representation (here T(K) means that the sole variable K is modified).

From the previous relation, we can deduce the value of K from K#init and conversely. This case is particularly easy, because the transformation is invertible. For a statement such as `read K`, the relation would be:

```
T(K) {}
```

It means that K is modified, but that its value in the target store is not constrained, leading to a set of memory stores instead of a single store as in the previous case.

Another interesting point is the way loops are handled in PIPS. In fact, the transformer of the loop is further approximated by the loop invariant.

For instance, in the program of Figure 5.2, the transformer provided for the outermost loop is:

```
T(K) {K == K#init + I - 1}
```

K#init being the value of K before the execution of the loop.

This type of transformer will be very useful when defining array region analyses. From now on, it is denoted by $\overline{\mathcal{T}}_{\mathrm{inv}}[\![\ldots]\!]$; and the inverse transformation by $\overline{\mathcal{T}}_{\mathrm{inv}}^{-1}[\![\ldots]\!]$. However, when the loop is a `do` loop, this approximation does not reflect the transformation on the loop index value, which has to be added when it is known, to get a less coarse approximation.

For instance the transformation of the memory store performed after k iterations of the loop `do i = 1,n` S `enddo` can be over-approximated by:

$$\overline{\mathcal{E}}[\![i = k]\!] \,\overline{\circ}\, \overline{\mathcal{T}}_{\mathrm{inv}}[\![\texttt{do i = 1,n } S \texttt{ enddo}]\!]$$

5.5 Continuation Conditions

We have shown in Section 5.2 that `stop` statements must be carefully taken into account for array region analyses. It could be thought that transformers provide the necessary information. This is true for exact transformers: Whenever a `stop` is encountered, the unknown store is returned; even if it is encountered on a particular path, full information (`if` conditions, loop counters, ...) about this path is kept. This information is however impossible to compute or even keep in memory for real life programs: For a compound statement and a store σ, $\overline{\mathcal{T}}[\![S]\!]\sigma$ gives a set of stores which may result from the execution of S from the store σ; if this set contains the unknown store $\perp$, S *may* stop or not! Thus, $\overline{\mathcal{T}}$ does not provide the information necessary for under-approximate array region analyses:

Under which conditions does the program certainly *stop or not?*

These conditions could be called *halting* conditions. However, we found it more convenient to use the results from the dual analysis, which we call *continuation conditions*, and which is denoted by $\mathcal{C}$. As for *preconditions*, we define it as a filter function which returns the input store if and only if no `stop` statement is encountered in the remainder of the code:

$$\mathcal{C} : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \boldsymbol{\Sigma})$$

The definition of $\mathcal{C}$ is straightforward:

`stop` statement If a `stop` is encountered, then the program does not continue:

$$\mathcal{C}[\![\texttt{stop}]\!] = \lambda\sigma.\bot$$

`continue` statement It has no effect on the execution of the program:

$$\mathcal{C}[\![\texttt{continue}]\!] = \lambda\sigma.\sigma$$

Assignments, `read` and `write` statements Since we assume that there are no run-time errors, the program does not stop when executing such statements:

$$\begin{array}{lcl} \mathcal{C}[\![\mathit{ref} = \mathit{exp}]\!] & = & \lambda\sigma.\sigma \\ \mathcal{C}[\![\texttt{read}\ \mathit{ref}]\!] & = & \lambda\sigma.\sigma \\ \mathcal{C}[\![\texttt{write}\ \mathit{exp}]\!] & = & \lambda\sigma.\sigma \end{array}$$

Sequence of instructions $S_1; S_2$ If the program does not stop in S_1, *and* in S_2 after the execution of S_1, then it does not stop for the whole sequence:

$$\mathcal{C}[\![S_1; S_2]\!] = \mathcal{C}[\![S_1]\!] \cap \mathcal{C}[\![S_2]\!] \circ \mathcal{T}[\![S1]\!]$$

$\mathcal{T}[\![S_1]\!]$ is used here to model the modification of the store by S_1 before the execution of S_2. The continuation conditions of S_2 must be evaluated in this new store, and not in the store preceding S_1, which is the reference.

`if` statement If the condition is true, then the program does not stop if it does not stop in S_1. Conversely, if the condition is false, it does not stop if it does not stop in S_2:

$$\mathcal{C}[\![\texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] = (\mathcal{C}[\![S_1]\!] \circ \mathcal{E}_c[\![C]\!]) \cup (\mathcal{C}[\![S_2]\!] \circ \mathcal{E}_c[\![\texttt{.not.}C]\!])$$

`do while` statements If the condition evaluates to *true*, then the program does not stop at all if it does not stop in S, *and* if it does not stop in the iterations, after the execution of S (T[\![S]\!]). If the condition is *false*, then it does not stop:

$$\mathcal{C}[\![\texttt{do while(}C\texttt{)}\ S]\!] = (\mathcal{C}[\![S]\!] \cap \mathcal{C}[\![\texttt{do while(}C\texttt{)}\ S]\!] \circ \mathcal{T}[\![S]\!]) \circ \mathcal{E}_c[\![C]\!] \cup \mathcal{E}_c[\![\texttt{.not.}C]\!]$$

This is a recursive continuous function on a cpo. Its best definition is therefore:

$$\mathcal{C}[\![\texttt{do while(}C\texttt{)}\ S]\!] = \mathrm{lfp}(\lambda f.(\mathcal{C}[\![S]\!] \cap f \circ \mathcal{T}[\![S]\!]) \circ \mathcal{E}_c[\![C]\!] \cup \mathcal{E}_c[\![\texttt{.not.}C]\!])$$

As for expressions, preconditions and transformers, and as is now usual for the reader, we define approximations for $\mathcal{C}$. $\overline{\mathcal{C}}$, the over-approximation, gives the conditions under which the program may continue, or in other words may not stop. On the contrary, $\underline{\mathcal{C}}$ gives the conditions under which the program never stops during the execution of the current statement. As $\mathcal{C}$, both analyses are filtering analyses. The target domain is thus the same as the target domain of $\mathcal{C}$. Approximate semantics are directly derived from the exact semantics, using approximate operators, and are given in Appendix D, Tables D.6 and D.7.

Continuation conditions are very closely related to transformers. This is illustrated in the next property.

Property 5.3
Let S be a statement ($S \in \boldsymbol{S}$), and σ a memory store ($\sigma \in \boldsymbol{\Sigma}$). Then the following properties hold:

$$\begin{array}{rcl} \mathcal{C}[\![S]\!]\sigma = \sigma & \Longleftrightarrow & \mathcal{T}[\![S]\!]\sigma \neq \perp \\ \overline{\mathcal{C}}[\![S]\!]\sigma = \sigma & \Longleftarrow & \mathcal{T}[\![S]\!]\sigma \neq \perp \\ \underline{\mathcal{C}}[\![S]\!]\sigma = \sigma & \Longrightarrow & \mathcal{T}[\![S]\!]\sigma \neq \perp \end{array}$$

and,

$$\begin{array}{rcl} \mathcal{C}[\![S]\!]\sigma = \perp & \Longleftrightarrow & \mathcal{T}[\![S]\!]\sigma = \perp \\ \overline{\mathcal{C}}[\![S]\!]\sigma = \perp & \Longrightarrow & \mathcal{T}[\![S]\!]\sigma = \perp \\ \underline{\mathcal{C}}[\![S]\!]\sigma = \perp & \Longleftarrow & \mathcal{T}[\![S]\!]\sigma = \perp \end{array}$$

Proof

- The proof for $\mathcal{C}[\![S]\!]\sigma = \sigma \iff \mathcal{T}[\![S]\!]\sigma \neq \perp$ and $\mathcal{C}[\![S]\!]\sigma = \perp \iff \mathcal{T}[\![S]\!]\sigma = \perp$ is by structural induction on the domain of statements $\boldsymbol{S}$. We do not provide it here because it is quite simple.
- The other properties directly come from the previous properties, and from the definitions of $\overline{\mathcal{C}}$ and $\underline{\mathcal{C}}$, and the corresponding approximation ordering.

□

5.6 Preconditions

We have seen in Section 5.2 that we need information about the relations between variable values in order to compare, or even merge, array regions. This is called *precondition*, because for any statement, it provides a condition which holds true before its execution. We could thus describe *preconditions* as a boolean analysis. But since we will compose it with other analyses which take memory stores as input, we provide it as a filter function. This is strictly equivalent: The filter function would be the characteristic function of the boolean analysis, as $\mathcal{E}_c$ is the characteristic function of $\mathcal{E}$. Thus, the prototype of $\mathcal{P}$ is:

$$\mathcal{P} : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \boldsymbol{\Sigma})$$

$\mathcal{P}$ is a forward analysis. If a relation holds true before a statement, then it holds true before the next statement, provided that the involved variables are not modified. The information is therefore propagated from the root of the program representation to the leaves, that is to say from a sequence of instructions to its constituting instructions; from a `if` construct to its branches; from a loop to its body.

Note $\mathcal{P}$ is a forward analysis. As explained in Chapter 4, we use the formalism of attribute grammars to give its definition, but we use the notations of denotational semantics.

Sequence of instructions $S_1; S_2$

If a condition holds just before the execution of $S_1; S_2$, then it holds just before the execution of S_1. Thus, if a store is not filtered out by $\mathcal{P}[\![S_1; S_2]\!]$, then it is not filtered out by $\mathcal{P}[\![S_1]\!]$:

$$\mathcal{P}[\![S_1]\!] = \mathcal{P}[\![S_1; S_2]\!]$$

The corresponding approximations are easily derived from this equation, using approximate operators.

Now, if a store is not filtered out by $\mathcal{P}[\![S_2]\!]$ after S_1 has been executed, then the store just before S_1 from which it comes would not have been filtered out by $\mathcal{P}[\![S_1]\!]$, and conversely:

$$\mathcal{P}[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!] = \mathcal{T}[\![S_1]\!] \circ \mathcal{P}[\![S_1]\!] \tag{5.1}$$

This is not a constructive definition, but a mere property which must be met by $\mathcal{P}[\![S_2]\!]$. Since $\mathcal{T}[\![S_1]\!]$ is not invertible, we cannot give a constructive definition of the exact semantics. However, this is possible for the over-approximation[7]:

$$\overline{\mathcal{P}}[\![S_2]\!] = \overline{\mathcal{T}}[\![S_1]\!] \mathbin{\bar{\bullet}} \overline{\mathcal{P}}[\![S_1]\!] \mathbin{\bar{\bullet}} \overline{\mathcal{T}}^{-1}[\![S_1]\!] \tag{5.2}$$

This definition is not directly derived from the definition of the exact semantics. We must thereby verify that it actually is an over-approximation:

Property 5.4
Let $S_1; S_2$ be a sequence of instructions. Let $\mathcal{P}[\![S_2]\!]$ and $\overline{\mathcal{P}}[\![S_2]\!]$ be defined by:

$$\mathcal{P}[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!] = \mathcal{T}[\![S_1]\!] \circ \mathcal{P}[\![S_1]\!]$$

and

$$\overline{\mathcal{P}}[\![S_2]\!] = \overline{\mathcal{T}}[\![S_1]\!] \mathbin{\bar{\bullet}} \overline{\mathcal{P}}[\![S_1]\!] \mathbin{\bar{\bullet}} \overline{\mathcal{T}}^{-1}[\![S_1]\!]$$

Then $\mathcal{P}[\![S_2]\!] \sqsubseteq \overline{\mathcal{P}}[\![S_2]\!]$.

---

[7] Computing an under-approximation would not be possible from the above formula, since $\underline{\mathcal{T}}$ is not available because of its lack of interest.

Proof Let us first show that $\mathcal{P}\llbracket S_2\rrbracket \circ \mathcal{T}\llbracket S_1\rrbracket \sqsubseteq \overline{\mathcal{P}}\llbracket S_2\rrbracket \circ \mathcal{T}\llbracket S_1\rrbracket$. From the assumptions, we know that:

$$\overline{\mathcal{P}}\llbracket S_2\rrbracket \circ \mathcal{T}\llbracket S_1\rrbracket = \overline{\mathcal{T}}\llbracket S_1\rrbracket \,\bar{\bullet}\, \overline{\mathcal{P}}\llbracket S_1\rrbracket \,\bar{\bullet}\, \overline{\mathcal{T}}^{-1}\llbracket S_1\rrbracket \circ \mathcal{T}\llbracket S_1\rrbracket$$

From the definition of $\overline{\mathcal{T}}^{-1}$ (see Section 5.4.2), we have

$$\overline{\mathcal{T}}^{-1}\llbracket S_1\rrbracket \circ \mathcal{T}\llbracket S_1\rrbracket \sqsupseteq \lambda\sigma.\sigma$$

Thus,

$$\begin{aligned}\overline{\mathcal{P}}\llbracket S_2\rrbracket \circ \mathcal{T}\llbracket S_1\rrbracket &\sqsupseteq \overline{\mathcal{T}}\llbracket S_1\rrbracket \,\bar{\bullet}\, \overline{\mathcal{P}}\llbracket S_1\rrbracket \\ &\sqsupseteq \mathcal{T}\llbracket S_1\rrbracket \circ \mathcal{P}\llbracket S_1\rrbracket \\ &\sqsupseteq \mathcal{P}\llbracket S_2\rrbracket \circ \mathcal{T}\llbracket S_1\rrbracket \end{aligned} \tag{5.3}$$

Let σ be a memory store. If there exists σ' such that $\mathcal{T}\llbracket S_1\rrbracket\sigma' = \sigma$, then from (5.3) we have

$$\mathcal{P}\llbracket S_2\rrbracket\sigma \sqsubseteq \overline{\mathcal{P}}\llbracket S_2\rrbracket\sigma$$

If there exists no σ' such that $\mathcal{T}\llbracket S_1\rrbracket\sigma' = \sigma$, then σ cannot be reached during an execution of the program, and $\mathcal{P}\llbracket S_2\rrbracket\sigma = \perp$. In this case, whatever the definition of $\overline{\mathcal{P}}\llbracket S_2\rrbracket\sigma$ may be, we also have

$$\mathcal{P}\llbracket S_2\rrbracket\sigma \sqsubseteq \overline{\mathcal{P}}\llbracket S_2\rrbracket\sigma$$

□

Conditional `if` C `then` S_1 `else` S_2

A memory store is not filtered out by $\mathcal{P}\llbracket S_1\rrbracket$ if it is not filtered out by the precondition of the whole construct, and by the condition (and conversely):

$$\begin{aligned}\mathcal{P}\llbracket S_1\rrbracket &= \mathcal{E}_c\llbracket C\rrbracket \circ \mathcal{P}\llbracket \texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2\rrbracket \\ \mathcal{P}\llbracket S_2\rrbracket &= \mathcal{E}_c\llbracket \texttt{.not.}C\rrbracket \circ \mathcal{P}\llbracket \texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2\rrbracket\end{aligned}$$

The corresponding over-approximation is directly derived from the exact semantics.

`do while` loop

Since preconditions are propagated forwards in the program representation, there are two types of preconditions to compute from the preconditions of the loop:

- First, the precondition $\mathcal{P}_i\llbracket \texttt{do while(}C\texttt{)}\ S\rrbracket$ which holds before iteration i (including the test of the condition). From now on, it is denoted as $\mathcal{P}_i$. $\mathcal{P}_0$ denotes $\mathcal{P}\llbracket \texttt{do while(}C\texttt{)}\ S\rrbracket$, the initial precondition. Hence $\mathcal{P}_1 = \mathcal{P}_0$.

- Second, the preconditions $\mathcal{P}_i\llbracket S\rrbracket$ which holds before the inner statement of iteration i. In fact, $\mathcal{P}_i\llbracket S\rrbracket$ can be easily deduced from $\mathcal{P}_i$:

$$\mathcal{P}_i\llbracket S\rrbracket = \mathcal{E}_c\llbracket C\rrbracket \circ \mathcal{P}_i$$

To define $\mathcal{P}_i$, we can follow the same reasoning as for the sequence: If a memory store is not filtered out by $\mathcal{P}_i$ after being propagated through the condition evaluation and the transformation by S of the previous iteration, then it is also not filtered out by $\mathcal{P}_{i-1}$ followed by the condition evaluation and the transformation by S; and conversely. This is better expressed by:

$$\mathcal{P}_i \circ \mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!] = \mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!] \circ \mathcal{P}_{i-1}$$

Since $\mathcal{T}[\![S]\!]$ is not invertible in the general case, we cannot give a constructive deterministic definition of $\mathcal{P}_i$. However, we can define a conservative over-approximation:

$$\begin{aligned}\overline{\mathcal{P}}_1 &= \overline{\mathcal{P}}_0 = \overline{\mathcal{P}}[\![\texttt{do while(}C\texttt{)}\ S]\!] \\ \overline{\mathcal{P}}_i &= \overline{\mathcal{T}}[\![S]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{E}}_c[\![C]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{P}}_{i-1} \mathbin{\overline{\bullet}} \overline{\mathcal{E}}_c[\![C]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{T}}^{-1}[\![S]\!] \quad \forall i > 1\end{aligned}$$

Since this definition is not directly derived from the corresponding exact semantics, its correctness must be proved:

Property 5.5
With the above notations and definitions,

$$\forall i \geq 1,\ \overline{\mathcal{P}}_i \sqsupseteq \mathcal{P}_i$$

Proof The proof is by induction on i.

The property is trivial for $i = 1$, from the definition of $\overline{\mathcal{P}}_0$.

Let us assume that for one i, $\overline{\mathcal{P}}_{i-1} \sqsupseteq \mathcal{P}_{i-1}$. And let us prove that $\overline{\mathcal{P}}_i \sqsupseteq \mathcal{P}_i$.

$$\overline{\mathcal{P}}_i \circ \mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!] = \overline{\mathcal{T}}[\![S]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{E}}_c[\![C]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{P}}_{i-1} \mathbin{\overline{\bullet}} \overline{\mathcal{E}}_c[\![C]\!] \mathbin{\overline{\bullet}} \underbrace{\overline{\mathcal{T}}^{-1}[\![S]\!] \circ \mathcal{T}[\![S]\!]}_{\sqsupseteq \lambda\sigma.\sigma} \circ \mathcal{E}_c[\![C]\!]$$

Furthermore, since $\mathcal{E}_c[\![C]\!]$ is more restrictive than $\overline{\mathcal{E}}_c[\![C]\!]$, we have

$$\overline{\mathcal{P}}_i \circ \mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!] \sqsupseteq \overline{\mathcal{T}}[\![S]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{E}}_c[\![C]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{P}}_{i-1} \circ \mathcal{E}_c[\![C]\!]$$

(here we can use exact composition laws, because computability is not involved).

Since $\mathcal{E}_c[\![C]\!]$ and $\overline{\mathcal{P}}_{i-1}$ are filters, they commute:

$$\overline{\mathcal{P}}_i \circ \mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!] \sqsupseteq \overline{\mathcal{T}}[\![S]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{E}}_c[\![C]\!] \circ \mathcal{E}_c[\![C]\!] \circ \overline{\mathcal{P}}_{i-1}$$

And, again because $\mathcal{E}_c[\![C]\!]$ is more restrictive than $\overline{\mathcal{E}}_c[\![C]\!]$,

$$\begin{aligned}\overline{\mathcal{P}}_i \circ \mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!] &\sqsupseteq \overline{\mathcal{T}}[\![S]\!] \circ \mathcal{E}_c[\![C]\!] \circ \overline{\mathcal{P}}_{i-1} \\ &\sqsupseteq \mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!] \circ \mathcal{P}_{i-1} \\ &\sqsupseteq \mathcal{P}_i \circ \mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!]\end{aligned}$$

The proof is easily completed as for Property 5.4:

$$\overline{\mathcal{P}}_i \sqsupseteq \mathcal{P}_i$$

□

$\overline{\mathcal{P}}_i$ is not very convenient to use in subsequent analyses: First the iteration counter is not explicitly available in `do while` loops; second, even in `do` loops, computing $\mathcal{P}_{10000}$ may be of an overwhelming complexity! An invariant information is usually more appropriate. We therefore define $\overline{\mathcal{P}}_{\text{inv}}$ as:

$$\begin{aligned}&\overline{\mathcal{P}}_0 \sqsubseteq \overline{\mathcal{P}}_{\text{inv}}\\ &\overline{\mathcal{P}}_{\text{inv}} = \overline{\mathcal{T}}[\![S]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{E}}_c[\![C]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{P}}_{\text{inv}} \mathbin{\overline{\bullet}} \overline{\mathcal{E}}_c[\![C]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{T}}^{-1}[\![S]\!]\end{aligned}$$

which can be rewritten as:

$$\overline{\mathcal{P}}_{\text{inv}} = \overline{\text{lfp}}(\lambda f.\overline{\mathcal{T}}[\![S]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{E}}_c[\![C]\!] \mathbin{\overline{\bullet}} f \mathbin{\overline{\bullet}} \overline{\mathcal{E}}_c[\![C]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{T}}^{-1}[\![S]\!]) \mathbin{\overline{\cup}} \overline{\mathcal{P}}_0$$

The correctness of the definition of $\overline{\mathcal{P}}_{\text{inv}}$ derives from the following property and from Property 5.5:

Property 5.6
With the above notations and definitions,

$$\forall i \geq 1,\ \overline{\mathcal{P}}_i \sqsubseteq \overline{\mathcal{P}}_{inv}$$

Proof The proof is by induction on i.

For $i = 1$, the property directly derives from the definition of $\overline{\mathcal{P}}_{\text{inv}}$.

Let us assume that, for one i, $\overline{\mathcal{P}}_{i-1} \sqsubseteq \overline{\mathcal{P}}_{\text{inv}}$, and let us prove that $\overline{\mathcal{P}}_i \sqsubseteq \overline{\mathcal{P}}_{\text{inv}}$.

$$\begin{aligned}\overline{\mathcal{P}}_{\text{inv}} &= \overline{\mathcal{T}}[\![S]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{E}}_c[\![C]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{P}}_{\text{inv}} \mathbin{\overline{\bullet}} \overline{\mathcal{E}}_c[\![C]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{T}}^{-1}[\![S]\!]\\ &\sqsupseteq \underbrace{\overline{\mathcal{T}}[\![S]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{E}}_c[\![C]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{P}}_{i-1} \mathbin{\overline{\bullet}} \overline{\mathcal{E}}_c[\![C]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{T}}^{-1}[\![S]\!]}_{=\overline{\mathcal{P}}_i}\end{aligned}$$

□

Implementation details

Equation (5.2) on Page 81 provides a constructive definition of the precondition for the second statement of a sequence. However, it may not be very practical. To obtain the precondition of statement S_i in sequence $S_0, \ldots, S_n$, one has to go backwards for the precondition of S_{i-1}, and then for S_{i-2}, ..., up to S_0, and then back to S_i in a forward analysis:

$$\overline{\mathcal{P}}[\![S_i]\!] = \overline{\mathcal{T}}[\![S_{i-1}]\!] \mathbin{\overline{\bullet}} \ldots \mathbin{\overline{\bullet}} \overline{\mathcal{T}}[\![S_0]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{P}}[\![S_0]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{T}}^{-1}[\![S_0]\!] \mathbin{\overline{\bullet}} \ldots \mathbin{\overline{\bullet}} \overline{\mathcal{T}}^{-1}[\![S_{i-1}]\!]$$

What is called a *precondition* in PIPS is in fact the term

$$\overline{\mathcal{T}}[\![S_{i-1}]\!] \mathbin{\overline{\bullet}} \ldots \mathbin{\overline{\bullet}} \overline{\mathcal{T}}[\![S_0]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{P}}[\![S_0]\!]$$

It is a kind of cumulated *transformer* from the input store of the sequence to the point of interest[8]. But the initial precondition $\overline{\mathcal{P}}[\![S_0]\!]$ is taken into account. It avoids the backward propagation. The forward propagation is performed step by step, one statement after another. And the precondition can thereby be directly derived from the previous statement.

5.7 Conclusion and Related Work

The purpose of this chapter was to quickly set the bases for array region analyses. A small subset language $\mathcal{L}$ has been introduced for this study: It covers the main features of FORTRAN 77, minus a few restrictions, plus the `do while` construct which is present in many other structured languages, and is thus of great interest.

The natural semantics of this language has been defined using two semantic functions: $\mathcal{E}$ provides the semantics of expressions, and $\mathcal{T}$ the semantics of statements, which are viewed as memory store transformers. Two other analyses called *preconditions* and *continuation conditions* have also been introduced to help with array region analyses.

Approximations have been almost systematically derived from the exact specifications of all these analyses. They rely on the approximate operators introduced in Chapter 4. Their correctness, that is to say their respect of the approximation ordering, is not proved: It is ensured by the properties enforced on approximate operators in the previous chapter.

The material presented in this chapter is not original, nor optimal, but has been placed here with a regard to correctness and completeness, and with an eye to our target array region analyses. Transformers have already been defined as store transformers by many [55, 108, 132], and their over-approximation is a well-known technique [55, 108]. Transformers and continuation conditions are backward analyses, which were as such broadly studied on a theoretical and practical ground by COUSOT [55] and HALBWACHS [90]. The use of the denotational approach for forward analyses was mostly inspired by JOUVELOT [108]. It allows very clear specifications of semantic functions and the correctness of approximations is for free when they are directly derived from their exact counterparts using approximate operators, as stated in Chapter 4.

Transformers are already implemented in PIPS [115, 104, 105, 103], where they are represented by convex polyhedra. They are used to compute array regions, as will be shown in the next chapter, but also to compute *preconditions*, a forward analysis which computes predicates over integer scalar variables, holding just before the execution of the corresponding instruction. Continuation conditions are currently being implemented, also as convex polyhedra.

[8] *Preconditions* are of type `transformer` in PIPS internal representation, and were presented as *transformers* in [105].

Chapter 6

Array Region Analyses

Many scientific programs (see [25] for instance) make a wide use of array structures. This may render their compilation for complex machines (e.g. parallel or hierarchical memory machines) or even their maintenance, particularly tricky, especially when such data structures are accessed through procedure calls.

To alleviate this problem and allow interprocedural dependence analyses, TRIOLET [161, 163] introduced the concept of array region analysis. His approach consists in gathering intraprocedural read and written references to array elements into over-approximate summaries, which can be later used at call sites. However, this method is flow insensitive, and thus locally not precise enough to enable intraprocedural optimizations of the program. Besides, read and written sets do not reflect the order in which references to array elements are executed (*array data flow*), and are thus insufficient for many advanced optimizations which are used to enhance the locality of references, such as array privatization. We have therefore introduced IN and OUT regions [17, 63, 65] to enable such transformations, but their description was rather informal.

This chapter formally describes the flow and context sensitive analysis of READ, WRITE, IN and OUT regions. Their exact semantics are defined, as well as their under- and over-approximations. The first section shows on an example why READ and WRITE regions are not sufficient for some advanced optimizations, and why under-approximations may be necessary. Semantic domains are then introduced and discussed in Section 6.2. The next four sections successively present for each type of region its exact semantics and its approximations, which are summarized in Appendix E. Sections 6.8, 6.7, and 6.9 discuss some implementation issues: The impact of variable aliasing, the use of preconditions and the computation ordering of our analyses. The related work is finally presented in Section 6.10.

6.1 Motivating Example

READ and WRITE array regions do not reflect the order in which references to array elements are performed. This section uses an example (see Figure 5.2 on page 67) to show why this can prevent advanced program optimizations such as array privatization, and which kind of information is necessary.

In our sample program, parts of the array `WORK` are reused from one iteration of the outermost loop to another, as shown in Figure 6.1. This induces dependences between iterations, and prevents the parallelization of the loop. In fact, these dependences are not direct dependences: Each read reference to an array element is preceded (or covered) by a write in the same iteration, and the computation is locally performed. The solution is to privatize the array `WORK`, that is to say to declare it as local to each iteration. Hence, there is no more loop carried dependences, and the loop can be parallelized.

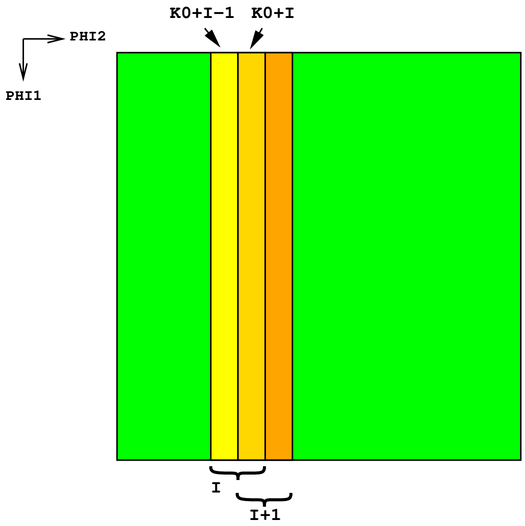


Figure 6.1: Dependences on array `WORK` in the program of Figure 5.2

If we want to solely consider read and write effects to detect privatizable arrays, we have to go backward through all statements, and to verify that each read reference is preceded by a write in the same iteration. However, this is impossible when there are procedure calls. Imagine that the body of the `I` loop is a call to an external subroutine. With summary READ and WRITE regions, we only know that some elements of `WORK` are read, and some others are written; but we know nothing about the order in which these accesses are performed, and in particular, we do not know whether the writes precede the reads. A second problem is due to the fact that early approaches [161, 37, 19] only performed over-approximate analyses of READ and WRITE regions. Using such regions, we can only answer the question:

> *Is it* possible *that the array element which is read in instruction S_2 be written by the preceding instruction S_1?*

But we cannot answer to the question involved when privatizing arrays:

> *Is the array element which is read in instruction S_2* certainly written *by the preceding instruction S_1?*

The last problem can be partially addressed by computing *under-approximations* of WRITE regions as in [164, 92] (see Chapter 14). But both problems require an analysis which takes into account the order in which array elements are referenced. This is the purpose of IN regions which contain the *locally upward exposed read*, or *imported*, array elements (see Figure 6.2). In our example, the IN region of WORK corresponding to the body of the I loop is the empty set.

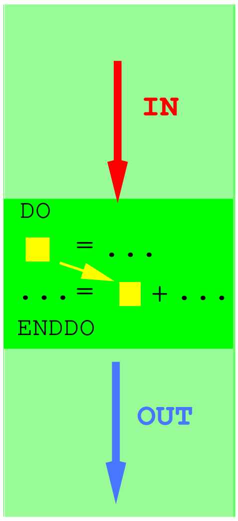


Figure 6.2: IN and OUT regions

Another problem is still pending. IN regions provide a mean to detect array elements whose computation is local to each iteration. But these elements may be reused in the program continuation. To declare them as local elements, we must either make sure that they are not reused (or *exported*) afterwards, or update the global array with the values of the local copies after the parallel execution of the loop. In this last case, for each privatized array element, we must know which is the last assigning iteration, that is to say which iteration *exports* its value towards the continuation of the program. This problem cannot be solved using WRITE regions, because this analysis does not take into account the order in which writes occur. This is why we introduced OUT regions [17]: They contain the *downward exposed written and used afterwards*, or *exported*, array elements (see Figure 6.2). In our example, assuming that WORK is not referenced in the continuation of the program, no element of the array is exported by any iteration of the I loop.

6.2 Domains and Semantic Functions

A *region* for an array T represents a set of some of its elements, which are described by their subscript values, or coordinates in $\mathbb{Z}^d$ if d is the array dimension. An array region can thus be viewed as a part of $\mathbb{Z}^d$, and belongs to $\wp(\mathbb{Z}^d)$. The coordinates of individual array elements are what TRIOLET introduced as *region descriptors*. They will be denoted by Φ^T variables, Φ_1^T representing the first coordinate, Φ_2^T the second, … [1]

[1]In FORTRAN, arrays have at most seven dimensions. But this limit is not taken into account in this study. Its sole effects are on the implementation of array regions.

The purpose of array region analyses is thus to associate to each statement (or sometimes even expressions) of a program, and to each possible memory store, a set of array elements (or array region) for each array which belongs to the current scope. If n arrays $(T_1, \ldots, T_n)$ are declared in the current procedure, of respective dimensions $d_1, \ldots, d_n$, then the target domain is the Cartesian product $\prod_{i=1,n} \wp(\mathbb{Z}^{d_i}) = \wp(\mathbb{Z}^{d_1}) \times \cdots \times \wp(\mathbb{Z}^{d_n})$. Array region semantic functions are thus of the following type:

$$
\begin{aligned}
\mathcal{R}: \quad & \tilde{\boldsymbol{\mathcal{L}}} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \textstyle\prod_{i=1,n} \wp(\mathbb{Z}^{d_i})) \\
& l \longrightarrow \mathcal{R}[\![l]\!] = \lambda\sigma.(\{\Phi^{T_1} : r_1(\Phi^{T_1}, \sigma)\}, \ldots, \{\Phi^{T_n} : r_n(\Phi^{T_n}, \sigma)\})
\end{aligned}
$$

r_i being the relation existing between the coordinates Φ^{T_i} of the elements of the region and any memory store σ.

Note

1. Using sets such as $\{\Phi^{T_1} : r_1(\Phi^{T_1}, \sigma)\}$ does not preclude any representation.

   In PIPS, $r_1, \ldots, r_n$ are convex polyhedra parameterized by the program variables.

   For RSD's [37], r_i is a conjunction of affine constraints with unit coefficients, which only depend on loop indices and symbolic constants:

   $$r_i(\Phi^{T_i}, \sigma) = \bigwedge_{j=1}^{j=d_i} (\Phi_j^{T_i} = \pm I_k + \alpha_j) \text{ or } r_i(\Phi^{T_i}, \sigma) = \bigwedge_{j=1}^{j=d_i} (\Phi_j^{T_i} = \alpha_j)$$

   where I_k are induction variable values, and α_j are symbolical constants. This is thus a constant function.

   And for lists of polyhedra as in [93], $r_1, \ldots, r_n$ would be disjunctions of convex polyhedra parameterized by the program variables.

2. Array region analyses are not limited to array variables, but are easily extended to scalar variables, which can be considered as arrays with no dimensions. In this case the relation r_i merely restrains the context (that is to say the set of memory stores σ) in which the variable is actually referenced.

It is not possible to represent any possible part of $\mathbb{Z}^d$. To define computable approximations, a particular representation, subset of $\wp(\mathbb{Z}^d)$, must be chosen. We denote it as $\tilde{\wp}(\mathbb{Z}^d)$, and we require that $\varnothing \in \tilde{\wp}(\mathbb{Z}^d)$ and $\mathbb{Z}^d \in \tilde{\wp}(\mathbb{Z}^d)$ to ensure that $\top$ and $\bot$ are the same as for the exact analyses. We also require that over- and under-approximate analyses use the same representation so that the results be combinable. For the previous analysis $\mathcal{R}$, $\overline{\mathcal{R}}$ and $\underline{\mathcal{R}}$ are defined as such:

$$
\begin{aligned}
\overline{\mathcal{R}}: \quad & \tilde{\boldsymbol{\mathcal{L}}} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \textstyle\prod_{i=1,n} \tilde{\wp}(\mathbb{Z}^{d_i})) \\
& l \longrightarrow \overline{\mathcal{R}}[\![l]\!] = \lambda\sigma.(\{\Phi^{T_1} : \overline{r}_1(\Phi^{T_1}, \sigma)\}, \ldots, \{\Phi^{T_n} : \overline{r}_n(\Phi^{T_n}, \sigma)\}) \\
\underline{\mathcal{R}}: \quad & \tilde{\boldsymbol{\mathcal{L}}} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \textstyle\prod_{i=1,n} \tilde{\wp}(\mathbb{Z}^{d_i})) \\
& l \longrightarrow \underline{\mathcal{R}}[\![l]\!] = \lambda\sigma.(\{\Phi^{T_1} : \underline{r}_1(\Phi^{T_1}, \sigma)\}, \ldots, \{\Phi^{T_n} : \underline{r}_n(\Phi^{T_n}, \sigma)\})
\end{aligned}
$$

Notice that $\prod_{i=1,n}\tilde{\wp}(\mathbb{Z}^{d_i})$ generally is not a lattice for under-approximate array region analyses ($\underline{\mathcal{R}}$), as seen in Section 4.1.3.

The next sections define the exact and approximate semantics of READ, WRITE, IN and OUT regions. The following notations will be used:

$\mathcal{R}_r$	array regions of read references
$\mathcal{R}_w$	array regions of written references
$\mathcal{R}_i$	IN array regions
$\mathcal{R}_o$	OUT array regions

In addition, the usual notations are used for the corresponding under- and over-approximations.

6.3 READ Regions

READ and WRITE regions represent the effects of the program on variables: They contain the array elements which are respectively read or written by simple or compound statements during their execution. In this section, we focus on READ regions. Additionnal details and differences for WRITE regions will be highlighted in Section 6.4.

$\mathcal{R}_r$, $\overline{\mathcal{R}}_r$ and $\underline{\mathcal{R}}_r$ define the exact and approximate semantics of READ regions. Each can be divided into two components: The semantics of read effects of expressions, which is given in the first part of Tables E.1, E.2 and E.3; and the semantics of read effects of statements, which is given in the second part of the previous tables. Since the semantics is defined for both expressions and statements, the source domain is the separated sum of the two domains, $\boldsymbol{S}$ and $\boldsymbol{E}$:

$$\mathcal{R}_r : \boldsymbol{S} \oplus \boldsymbol{E} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \wp(\mathbb{Z}^{d_i}))$$

$$\overline{\mathcal{R}}_r : \boldsymbol{S} \oplus \boldsymbol{E} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \tilde{\wp}(\mathbb{Z}^{d_i}))$$

$$\underline{\mathcal{R}}_r : \boldsymbol{S} \oplus \boldsymbol{E} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \tilde{\wp}(\mathbb{Z}^{d_i}))$$

This section discusses the main functions given in Tables E.1, E.2 and E.3: Expressions, assignments, sequences of instructions, `if` constructs and loops. The others are not detailed because of their simplicity.

6.3.1 Expression

The read effects of an expression merely are the union of the effects of its components. The initialization is made when a constant value is encountered ($\lambda\sigma.\varnothing$), or a reference. In this last case, since the representation has not yet been chosen, the result is represented by $\lambda\sigma.(\{var\})$ for a scalar variable, and $\lambda\sigma.(\{var(\mathcal{E}[\![exp_1,\dots,exp_k]\!]\sigma)\})$ for a reference to an array element, $\{var\}$ or $\{var(\dots)\}$ standing for *the region which contains the element var or var*$(\dots)$. The corresponding approximations directly depend on the chosen representation. As for over-approximate transformers (see Page 73), their definitions only reflect the properties which must be enforced.

We now define the semantics of READ array regions for statements. Intuitively, it is the union of all read references occurring during its execution. However, the context (i.e. the memory store) must be taken into account for the evaluation of expressions such as `if` or `do while` conditions, array subscripts, ...

6.3.2 Assignment

Assignment statements are quite easy to handle. The right hand side is an expression which is read. The left hand side defines the variable or array element it references, but the possible array subscripts are read expressions:

$$\begin{array}{lcl} \mathcal{R}_r[\![var(exp_1,\dots,exp_k) = exp]\!] & = & \mathcal{R}_r[\![exp_1,\dots,exp_k]\!] \cup \mathcal{R}_r[\![exp]\!] \\ \mathcal{R}_r[\![var = exp]\!] & = & \mathcal{R}_r[\![exp]\!] \end{array}$$

Corresponding approximations are straightforward, and the reader is referred to the tables in Appendix E.

6.3.3 Sequence of instructions

Let us consider the sequence of instructions $S_1; S_2$. Intuitively, its read effects are the union of the effects of S_1 and S_2. But the modification of the input memory store by S_1, modelled by $\mathcal{T}[\![S_1]\!]$, has to be taken into account for the evaluation of the effects of S_2:

$$\mathcal{R}_r[\![S_1; S_2]\!] = \mathcal{R}_r[\![S_1]\!] \cup \mathcal{R}_r[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]$$

$\overline{\mathcal{R}}_r[\![S_1; S_2]\!]$ and $\underline{\mathcal{R}}_r[\![S_1; S_2]\!]$ are not directly derived from $\mathcal{R}_r[\![S_1; S_2]\!]$: The continuation conditions of S_1 are also taken into account. For $\underline{\mathcal{R}}_r[\![S_1; S_2]\!]$, the use of $\underline{\mathcal{C}}[\![S_1]\!]$ ensures that $\underline{\mathcal{R}}_r[\![S_2]\!] \underline{\bullet} \overline{\mathcal{T}}[\![S_1]\!]$ gives a non-empty region only when the program does not stop in S_1, and thus provides an under-approximation of $\mathcal{R}_r[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]$. This was already explained in Section 5.4.1.

For $\overline{\mathcal{R}}_r[\![S_1; S_2]\!]$, using $\overline{\mathcal{C}}[\![S_1]\!]$ is not compulsory: It merely restricts the set of stores for which S_1 does not stop ($\overline{\mathcal{T}}[\![S_1]\!] \,\overline{\circ}\, \overline{\mathcal{C}}[\![S_1]\!] \sqsubseteq \overline{\mathcal{T}}[\![S_1]\!]$), and thereby gives more accurate regions. This is particularly important when exactness must be proved by checking that the under-approximation is equal to the over-approximation ($\underline{\mathcal{R}}_r[\![S_2]\!] \underline{\bullet} \overline{\mathcal{T}}[\![S_1]\!] \underline{\circ} \underline{\mathcal{C}}[\![S_1]\!] = \overline{\mathcal{R}}_r[\![S_2]\!] \overline{\bullet} \overline{\mathcal{T}}[\![S_1]\!] \,\overline{\circ}\, \overline{\mathcal{C}}[\![S_1]\!]$) because $\underline{\mathcal{C}}[\![S_1]\!]$ may be equal to $\overline{\mathcal{C}}[\![S_1]\!]$ but not to $\lambda\sigma.\sigma$, which is the implicit choice when the over-approximation is defined as $\overline{\mathcal{R}}_r[\![S_2]\!] \overline{\bullet} \overline{\mathcal{T}}[\![S_1]\!]$.

For example, let us consider the following piece of code:

```
S1 : if (i.gt.3) then
        stop
     endif
S2 : ... =  A(i)
```

The exact region for S_2 can be represented by:

$$\mathcal{R}_r[\![S_2]\!] = \lambda\sigma.\{\Phi_1 : \Phi_1 = \sigma(\mathtt{i})\}$$

If the over-approximate transformer for S_1 is:

$$\overline{\mathcal{T}}[\![S_1]\!] = \lambda\sigma.\{\sigma' : \sigma' = \sigma\}$$

then

$$\mathcal{R}_r[\![S_2]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{T}}[\![S_1]\!] = \lambda\sigma.\{\Phi_1 : \Phi_1 = \sigma(\mathtt{i})\}$$

The continuation condition for S_1 can be exactly described by:

$$\mathcal{C}[\![S_1]\!] = \lambda\sigma.\textbf{if } \sigma(\mathtt{i}) \leq 3 \textbf{ then } \sigma \textbf{ else } \perp$$

Thus

$$\mathcal{R}_r[\![S_2]\!] \mathbin{\underline{\bullet}} \overline{\mathcal{T}}[\![S_1]\!] \mathbin{\underline{\circ}} \mathcal{C}[\![S_1]\!] = \lambda\sigma.\{\Phi_1 : \Phi_1 = \sigma(\mathtt{i}) \wedge \sigma(\mathtt{i}) \leq 3\}$$

This is clearly different from $\mathcal{R}_r[\![S_2]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{T}}[\![S_1]\!]$. But it is equal to $\mathcal{R}_r[\![S_2]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{T}}[\![S_1]\!] \mathbin{\overline{\circ}} \mathcal{C}[\![S_1]\!]$.

The semantic functions finally are:

$$\begin{aligned}
\overline{\mathcal{R}}_r[\![S_1;S_2]\!] &= \overline{\mathcal{R}}_r[\![S_1]\!] \mathbin{\overline{\cup}} \overline{\mathcal{R}}_r[\![S_2]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{T}}[\![S_1]\!] \mathbin{\overline{\circ}} \overline{\mathcal{C}}[\![S_1]\!] \\
\underline{\mathcal{R}}_r[\![S_1;S_2]\!] &= \underline{\mathcal{R}}_r[\![S_1]\!] \mathbin{\underline{\cup}} \underline{\mathcal{R}}_r[\![S_2]\!] \mathbin{\underline{\bullet}} \overline{\mathcal{T}}[\![S_1]\!] \mathbin{\underline{\circ}} \underline{\mathcal{C}}[\![S_1]\!]
\end{aligned}$$

Since these definitions are not directly derived from the corresponding definition for $\mathcal{R}_r$, the correctness must be proved. This is done in the following properties for each approximation.

Property 6.1
For any sequence of instruction $S_1;S_2$, the following property holds:

$$\mathcal{R}_r[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!] \sqsubseteq \overline{\mathcal{R}}_r[\![S_2]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{T}}[\![S_1]\!] \mathbin{\overline{\circ}} \overline{\mathcal{C}}[\![S_1]\!]$$

Proof Let σ be a memory store. Either $\overline{\mathcal{C}}[\![S_1]\!]\sigma = \sigma$ or $\overline{\mathcal{C}}[\![S_1]\!]\sigma = \perp$.

$$\begin{aligned}
\overline{\mathcal{C}}[\![S_1]\!]\sigma = \sigma &\implies \overline{\mathcal{T}}[\![S_1]\!] \mathbin{\overline{\circ}} \overline{\mathcal{C}}[\![S_1]\!]\sigma = \overline{\mathcal{T}}[\![S_1]\!]\sigma \\
&\implies \mathcal{T}[\![S_1]\!]\sigma \sqsubseteq \overline{\mathcal{T}}[\![S_1]\!] \mathbin{\overline{\circ}} \overline{\mathcal{C}}[\![S_1]\!]\sigma
\end{aligned}$$

and, from Property 5.3,

$$\begin{aligned}
\overline{\mathcal{C}}[\![S_1]\!]\sigma = \perp &\implies \mathcal{T}[\![S_1]\!]\sigma = \perp \\
&\implies \overline{\mathcal{T}}[\![S_1]\!] \mathbin{\overline{\circ}} \overline{\mathcal{C}}[\![S_1]\!]\sigma = \mathcal{T}[\![S_1]\!]\sigma
\end{aligned}$$

From both case, we can conclude that:

$$\mathcal{T}[\![S_1]\!] \sqsubseteq \overline{\mathcal{T}}[\![S_1]\!] \mathbin{\overline{\circ}} \overline{\mathcal{C}}[\![S_1]\!]$$

and therefore,

$$\mathcal{R}_r[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!] \sqsubseteq \overline{\mathcal{R}}_r[\![S_2]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{T}}[\![S_1]\!] \mathbin{\overline{\circ}} \overline{\mathcal{C}}[\![S_1]\!]$$

□

Property 6.2
For any sequence of instruction $S_1;S_2$, the following property holds:

$$\underline{\mathcal{R}}_r[\![S_2]\!] \mathbin{\underline{\bullet}} \overline{\mathcal{T}}[\![S_1]\!] \mathbin{\underline{\circ}} \underline{\mathcal{C}}[\![S_1]\!] \sqsubseteq \mathcal{R}_r[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]$$

Proof Let σ be a memory store. Either $\underline{\mathcal{C}}[\![S_1]\!]\sigma = \sigma$ or $\underline{\mathcal{C}}[\![S_1]\!]\sigma = \perp$.

$$\begin{aligned}\underline{\mathcal{C}}[\![S_1]\!]\sigma = \sigma &\Longrightarrow \overline{\mathcal{T}}[\![S_1]\!] \underline{\circ} \underline{\mathcal{C}}[\![S_1]\!]\sigma = \overline{\mathcal{T}}[\![S_1]\!]\sigma \\ &\Longrightarrow \overline{\mathcal{T}}[\![S_1]\!] \underline{\circ} \underline{\mathcal{C}}[\![S_1]\!]\sigma \sqsubseteq \mathcal{T}[\![S_1]\!]\sigma\end{aligned}$$

and, from Property 5.3,

$$\begin{aligned}\underline{\mathcal{C}}[\![S_1]\!]\sigma = \perp &\Longrightarrow \overline{\mathcal{T}}[\![S_1]\!] \underline{\circ} \underline{\mathcal{C}}[\![S_1]\!]\sigma = \perp \\ &\Longrightarrow \overline{\mathcal{T}}[\![S_1]\!] \underline{\circ} \underline{\mathcal{C}}[\![S_1]\!]\sigma \sqsubseteq \mathcal{T}[\![S_1]\!]\sigma\end{aligned}$$

From both case, we can conclude that:

$$\overline{\mathcal{T}}[\![S_1]\!] \underline{\circ} \underline{\mathcal{C}}[\![S_1]\!] \sqsubseteq \mathcal{T}[\![S_1]\!]$$

and therefore,

$$\underline{\mathcal{R}_r}[\![S_2]\!] \underline{\bullet} \overline{\mathcal{T}}[\![S_1]\!] \underline{\circ} \underline{\mathcal{C}}[\![S_1]\!] \sqsubseteq \mathcal{R}_r[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]$$

□

The correctness of the remaining parts of the definitions is ensured by the use of approximate operators derived from the exact operators.

6.3.4 Conditional instruction

Because expressions do not have side effects in $\boldsymbol{\mathcal{L}}$, the scalar variables and array elements read by a conditional instruction are those read during the evaluation of the conditions ($\mathcal{R}_r[\![C]\!]$), plus the elements read during the execution of one of the branches, depending on the result of the condition evaluation:

$$\mathcal{R}_r[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] = \mathcal{R}_r[\![C]\!] \cup (\mathcal{R}_r[\![S_1]\!] \circ \mathcal{E}[\![C]\!]) \cup (\mathcal{R}_r[\![S_2]\!] \circ \mathcal{E}[\![\texttt{.not.}C]\!])$$

If expressions had side effects, the definition would have to take into account the possible modification of the input memory store by the evaluation of the condition; it would be modeled by $\mathcal{T}[\![C]\!]$, assuming that $\mathcal{T}$ has been extended to the domain of expressions:

$$\begin{aligned}&\mathcal{R}_r[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] = \\ &\qquad \mathcal{R}_r[\![C]\!] \cup \big((\mathcal{R}_r[\![S_1]\!] \circ \mathcal{E}[\![C]\!]) \cup (\mathcal{R}_r[\![S_2]\!] \circ \mathcal{E}[\![\texttt{.not.}C]\!])\big) \circ \mathcal{T}[\![C]\!]\end{aligned}$$

The over-approximation $\overline{\mathcal{R}_r}[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!]$ is merely derived from the exact semantics using the corresponding approximate operators. Whereas the under-approximation contains an additional term, $(\underline{\mathcal{R}_r}[\![S_1]\!] \underline{\cap} \underline{\mathcal{R}_r}[\![S_2]\!])$:

$$\begin{aligned}&\underline{\mathcal{R}_r}[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] = \\ &\quad \underline{\mathcal{R}_r}[\![C]\!] \underline{\cup} (\underline{\mathcal{R}_r}[\![S_1]\!] \underline{\circ} \underline{\mathcal{E}_c}[\![C]\!]) \underline{\cup} (\underline{\mathcal{R}_r}[\![S_2]\!] \underline{\circ} \underline{\mathcal{E}_c}[\![\texttt{.not.}C]\!]) \underline{\cup} (\underline{\mathcal{R}_r}[\![S_1]\!] \underline{\cap} \underline{\mathcal{R}_r}[\![S_2]\!]) \quad (6.1)\end{aligned}$$

It has been added to better reflect the semantics of `if` statements, which ensures that at least one of the branches is executed, and thus that if an array element is certainly referenced in both branches (that is to say belongs to $\underline{\mathcal{R}_r}[\![S_1]\!] \underline{\cap} \underline{\mathcal{R}_r}[\![S_2]\!]$), it is certainly referenced by the `if`, whatever the condition may be. Whereas it may fail to be detected by the remaining part of the equation, because of too narrow under-approximations of the condition and its opposite.

Because of this additional term, the correctness must be proved:

Property 6.3
Using the above notations and definitions, the following property holds:

$$\begin{aligned}\underline{\mathcal{R}_r}[\![C]\!] \,\underline{\cup}\, (\underline{\mathcal{R}_r}[\![S_1]\!] \,\underline{\circ}\, \underline{\mathcal{E}_c}[\![C]\!]) \,\underline{\cup}\, (\underline{\mathcal{R}_r}[\![S_2]\!] \,\underline{\circ}\, \underline{\mathcal{E}_c}[\![\texttt{.not.}C]\!]) \,\underline{\cup}\, (\underline{\mathcal{R}_r}[\![S_1]\!] \,\underline{\cap}\, \underline{\mathcal{R}_r}[\![S_2]\!]) \\ \sqsubseteq\ \mathcal{R}_r[\![C]\!] \cup (\mathcal{R}_r[\![S_1]\!] \circ \mathcal{E}[\![C]\!]) \cup (\mathcal{R}_r[\![S_2]\!] \circ \mathcal{E}[\![\texttt{.not.}C]\!])\end{aligned}$$

.

Proof It is clear from the properties enforced on under-approximate operators that:

$$\begin{aligned}\underline{\mathcal{R}_r}[\![C]\!] \,\underline{\cup}\, (\underline{\mathcal{R}_r}[\![S_1]\!] \,\underline{\circ}\, \underline{\mathcal{E}_c}[\![C]\!]) \,\underline{\cup}\, (\underline{\mathcal{R}_r}[\![S_2]\!] \,\underline{\circ}\, \underline{\mathcal{E}_c}[\![\texttt{.not.}C]\!]) \\ \sqsubseteq\ \mathcal{R}_r[\![C]\!] \cup (\mathcal{R}_r[\![S_1]\!] \circ \mathcal{E}[\![C]\!]) \cup (\mathcal{R}_r[\![S_2]\!] \circ \mathcal{E}[\![\texttt{.not.}C]\!])\end{aligned}$$

From the semantics of `if` statements, we know that:

$$\mathcal{R}_r[\![S_1]\!] \cap \mathcal{R}_r[\![S_2]\!] \sqsubseteq \mathcal{R}_r[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!]$$

Moreover, from the properties enforced on operators:

$$\underline{\mathcal{R}_r}[\![S_1]\!] \,\underline{\cap}\, \underline{\mathcal{R}_r}[\![S_2]\!] \sqsubseteq \mathcal{R}_r[\![S_1]\!] \cap \mathcal{R}_r[\![S_2]\!]$$

And therefore,

$$\underline{\mathcal{R}_r}[\![S_1]\!] \,\underline{\cap}\, \underline{\mathcal{R}_r}[\![S_2]\!] \sqsubseteq \mathcal{R}_r[\![C]\!] \cup (\mathcal{R}_r[\![S_1]\!] \circ \mathcal{E}[\![C]\!]) \cup (\mathcal{R}_r[\![S_2]\!] \circ \mathcal{E}[\![\texttt{.not.}C]\!])$$

The property derives from this last equation and the first given in this proof. □

6.3.5 `do while` loop

Another interesting statement is the `do while` construct. To understand the definition of $\mathcal{R}_r[\![\texttt{do while(}C\texttt{)}\ S]\!]$, the best approach is to consider its sequential execution from the store σ. First, the condition C is evaluated; it may reference some variables whose regions are represented by $\mathcal{R}_r[\![C]\!]\sigma$. If its evaluation $\mathcal{E}[\![C]\!]\sigma$ is false, the loop stops, and no array element is referenced within the loop body; if it is true, S is executed, and the elements $\mathcal{R}_r[\![S]\!]\sigma$ are referenced. Up to now, the regions are given by:

$$\mathcal{R}_r[\![C]\!] \cup (\mathcal{R}_r[\![S]\!] \circ \mathcal{E}_c[\![C]\!])$$

After that, the whole process loops, but in the store resulting from the previous execution of S after C has been evaluated to *true*, $\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!]\sigma$. The semantics of the `do while` loop is thus given by:

$$\mathcal{R}_r[\![\texttt{do while(}C\texttt{)}\ S]\!] = \mathcal{R}_r[\![C]\!] \cup \big(\mathcal{R}_r[\![S]\!] \cup \mathcal{R}_r[\![\texttt{do while(}C\texttt{)}\ S]\!] \circ \mathcal{T}[\![S]\!]\big) \circ \mathcal{E}_c[\![C]\!]$$

This is a continuous recursive function, defined on a lattice. Its best definition is therefore given by the least fixed point of the corresponding functional:

$$\mathcal{R}_r[\![\texttt{do while}(C)\ \ S]\!] = \text{lfp}\big(\lambda f.\mathcal{R}_r[\![C]\!] \cup (\mathcal{R}_r[\![S]\!] \cup f \circ \mathcal{T}[\![S]\!]) \circ \mathcal{E}_c[\![C]\!]\big)$$

The corresponding approximations (cf. Pages 292 and 293) are built from the exact semantics as for the sequence of instructions: Approximate operators are used whenever possible, but continuation conditions are also taken into account. The correctness is ensured by this process and by Properties 6.1 and 6.2.

6.3.6 The special case of `do` loops

As stated in Chapter 5, `do` loops are a particular case of `do while` loops. Their peculiarity lies in the fact that the number of iterations is known at run-time, and is even actually evaluated before the execution of the loop. An equivalent definition for the exact read array regions[2] of a normalized `do` loop is thus:

$$\mathcal{R}_r[\![\texttt{do i=1,n}\ \ S\ \ \texttt{enddo}]\!] = \lambda\sigma.\mathcal{R}_r[\![\texttt{n}]\!]\sigma \cup \bigcup_{k=1}^{k=\mathcal{E}[\![\texttt{n}]\!]\sigma} \big(\mathcal{R}_r[\![S]\!] \circ \mathcal{T}[\![\texttt{do i=1,}k-1\ \ S\ \ \texttt{enddo}]\!]\sigma\big)$$

Since the number of iterations is known at run-time, $\bigcup_{k=1}^{k=\mathcal{E}[\![\texttt{n}]\!]}$ is a finite union, and the region is thereby well defined. Let us further explicit this expression. Regions represent sets of array elements. And a finite union over k of a set parameterized by k, with $k \in [1..\mathcal{E}[\![\texttt{n}]\!]]$, is mathematically equivalent to a projection along k of the previous set. Thus,

$$\begin{aligned}&\bigcup_{k=1}^{k=\mathcal{E}[\![\texttt{n}]\!]\sigma} \mathcal{R}_r[\![S]\!] \circ \mathcal{T}[\![\texttt{do i=1,}k-1\ \ S\ \ \texttt{enddo}]\!]\sigma = \\ &\qquad \text{proj}_k\big(\mathcal{R}_r[\![S]\!] \circ \mathcal{T}[\![\texttt{do i=1,}k-1\ \ S\ \ \texttt{enddo}]\!] \circ \mathcal{E}_c[\![\texttt{(1.lt.}k\texttt{).and.(}k\texttt{.lt.n)}]\!]\big)\sigma\end{aligned}$$

We now have three equivalent definitions for the read regions of a `do` loop:

$$\begin{aligned}&\mathcal{R}_r[\![\texttt{do i=1,n}\ \ S\ \ \texttt{enddo}]\!] \\ &\quad = \text{lfp}\big(\lambda f.\mathcal{R}_r[\![C]\!] \cup (\mathcal{R}_r[\![S]\!] \cup f \circ \mathcal{T}[\![S]\!]) \circ \mathcal{E}_c[\![C]\!]\big) \\ &\quad = \lambda\sigma.\mathcal{R}_r[\![\texttt{n}]\!]\sigma \cup \bigcup_{k=1}^{k=\mathcal{E}[\![\texttt{n}]\!]\sigma} \mathcal{R}_r[\![S]\!] \circ \mathcal{T}[\![\texttt{do i=1,}k-1\ \ S\ \ \texttt{enddo}]\!]\sigma \\ &\quad = \lambda\sigma.\mathcal{R}_r[\![\texttt{n}]\!]\sigma \cup \text{proj}_k\big(\mathcal{R}_r[\![S]\!] \circ \mathcal{T}[\![\texttt{do i=1,}k-1\ \ S\ \ \texttt{enddo}]\!] \\ &\qquad\qquad\qquad\qquad \circ\, \mathcal{E}_c[\![\texttt{(1.lt.}k\texttt{).and.(}k\texttt{.lt.n)}]\!]\big)\sigma\end{aligned}$$

They are not mere mathematical toys. For a particular representation (e.g. convex polyhedra, RSDs, DADs, ...), it might be difficult to define *good* approximations for one of these definitions, while it is relatively easy for another definition, as will be shown in Chapter 8 for convex polyhedra. The alternative definitions for approximations

[2]Except for the read effects on the loop variable, which can trivially be added, but would make the formula unnecessarily heavy.

are given below, assuming that approximate operators are available for the finite sum (which may become infinite when $\mathcal{E}[\![\mathtt{n}]\!] = \top$ for instance), or for the projection operator. We also further approximate the transformer of the loop by its invariant form, as shown on Page 78.

$$\begin{aligned}
&\overline{\mathcal{R}}_r[\![\texttt{do i=1,n } S \texttt{ enddo}]\!] \\
&\quad = \overline{\text{lfp}}\big(\lambda f.\overline{\mathcal{R}}_r[\![C]\!] \,\overline{\cup}\, (\overline{\mathcal{R}}_r[\![S]\!] \,\overline{\cup}\, f \,\overline{\bullet}\, \overline{\mathcal{T}}[\![S]\!] \,\overline{\circ}\, \overline{\mathcal{C}}[\![S]\!]) \,\overline{\circ}\, \overline{\mathcal{E}}_c[\![C]\!]\big) && (6.2)\\
&\quad = \lambda\sigma.\overline{\mathcal{R}}_r[\![\mathtt{n}]\!]\sigma \,\overline{\cup} \overline{\bigcup_{k=1}^{k=\overline{\mathcal{E}}[\![\mathtt{n}]\!]\sigma}} \overline{\mathcal{R}}_r[\![S]\!] \,\overline{\bullet}\, \overline{\mathcal{E}}_c[\![\texttt{i=}k]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}_{\text{inv}}[\![\texttt{do i=1,n } S \texttt{ enddo}]\!]\sigma && (6.3)\\
&\quad = \lambda\sigma.\overline{\mathcal{R}}_r[\![\mathtt{n}]\!]\sigma \,\overline{\cup}\, \overline{\text{proj}}_k\big(\overline{\mathcal{R}}_r[\![S]\!] \,\overline{\bullet}\,\overline{\bullet}\, \overline{\mathcal{E}}_c[\![\texttt{i=}k]\!] \overline{\mathcal{T}}_{\text{inv}}[\![\texttt{do i=1,n } S \texttt{ enddo}]\!]\sigma \\
&\qquad\qquad \overline{\circ}\, \overline{\mathcal{E}}_c[\![\texttt{(1.le.}k\texttt{).and.(}k\texttt{.le.n)}]\!]\big)\sigma && (6.4)
\end{aligned}$$

$$\begin{aligned}
&\underline{\mathcal{R}}_r[\![\texttt{do i=1,n } S \texttt{ enddo}]\!] \\
&\quad = \underline{\text{lfp}}\big(\lambda f.\underline{\mathcal{R}}_r[\![C]\!] \,\underline{\cup}\, (\underline{\mathcal{R}}_r[\![S]\!] \,\underline{\cup}\, f \,\underline{\bullet}\, \overline{\mathcal{T}}[\![S]\!] \,\underline{\circ}\, \underline{\mathcal{C}}[\![S]\!]) \,\underline{\circ}\, \underline{\mathcal{E}}_c[\![C]\!]\big) && (6.5)\\
&\quad = \lambda\sigma.\underline{\mathcal{R}}_r[\![\mathtt{n}]\!]\sigma \,\underline{\cup} \underline{\bigcup_{k=1}^{k=\underline{\mathcal{E}}[\![\mathtt{n}]\!]\sigma}} \underline{\mathcal{R}}_r[\![S]\!] \,\underline{\bullet}\, \overline{\mathcal{E}}_c[\![\texttt{i=}k]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}_{\text{inv}}[\![\texttt{do i=1,n } S \texttt{ enddo}]\!]\sigma && (6.6)\\
&\quad = \lambda\sigma.\underline{\mathcal{R}}_r[\![\mathtt{n}]\!] \,\underline{\cup}\, \underline{\text{proj}}_k\big(\underline{\mathcal{R}}_r[\![S]\!] \,\underline{\bullet}\, \overline{\mathcal{E}}_c[\![\texttt{i=}k]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}_{\text{inv}}[\![\texttt{do i=1,n } S \texttt{ enddo}]\!] \\
&\qquad\qquad \underline{\circ}\, \underline{\mathcal{E}}_c[\![\texttt{(1.le.}k\texttt{).and.(}k\texttt{.le.n)}]\!]\big)\sigma && (6.7)
\end{aligned}$$

6.3.7 Unstructured parts of code

As stated in the previous chapter, unstructured parts of code are not formally treated in this study. This is reflected in the definition of $\boldsymbol{\mathcal{L}}$ (Figure 5.1) by the lack of `goto` statements. The major reason for which `goto` statements are not considered is that, except for hidden regular (`do` and `do while`) loops, it is highly improbable that some interesting parallelism be discovered in unstructured fragments of code. This is also the reason for PIPS hierarchical control flow graph, which represents structured constructs by their abstract syntax tree, and unstructured parts of code by standard control flow graphs.

The computation of regions on structured parts is performed as described above by the semantic functions. For unstructured parts, iterative methods could be used. The equations giving READ regions for a node n would informally be something like:

$$\begin{aligned}
\overline{\mathcal{R}}_r(n) &= \overline{\mathcal{R}}_r[\![\text{code}(n)]\!] \,\overline{\cup}\, \overline{\bigcup}_{n' \in \text{succ}(n)}(\overline{\mathcal{R}}_r(n') \,\overline{\bullet}\, \overline{\mathcal{T}}[\![\text{code}(n)]\!])\\
\underline{\mathcal{R}}_r(n) &= \underline{\mathcal{R}}_r[\![\text{code}(n)]\!] \,\underline{\cup}\, \underline{\bigcup}_{n' \in \text{succ}(n)}(\underline{\mathcal{R}}_r(n') \,\underline{\bullet}\, \overline{\mathcal{T}}[\![\text{code}(n)]\!])
\end{aligned}$$

where $\text{code}(n)$ gives the text of the code at node n, and $\text{succ}(n)$ gives its successors in the local control flow graph.

However, these equations, though widely used in the literature, do not take into account the conditions under which each successor is executed. For that purpose, *flags* or *continuation semantics* [132, 157] should be used.

In PIPS, since no parallelism is expected from such constructs, a very simple approach is taken: Variables which may be modified by any node are eliminated (see Chapter 8) from the regions of each node to convert them into regions relative to the input store; then, they are merged to form a summary for the global construct. Less drastic solutions should however be devised for `do while` loops built with `goto` statements, which are common in *dusty decks*. These hidden loops could be discovered using normalization techniques such as those described in [7].

6.4 WRITE Regions

The definition of WRITE regions is very similar to READ regions. Except that, since the expressions in $\boldsymbol{\mathcal{L}}$ have no side effects, WRITE regions are solely defined on the domain of statements $\boldsymbol{S}$. The semantics is almost the same for the control structures of the language: `if`, `do while`, and the sequence; for the first two constructs, the only difference is the way the condition is handled (expressions have only read effects; they do not have write effects). But the main differences are for the assignment, and for I/O instructions. The exact and approximate semantics are given in Appendix E for completeness. The same remarks as for READ regions could be done for correctness proofs and for `do` loops.

6.5 IN Regions

Informally, the IN regions of a statement represent its *imported* or *locally upward exposed* array elements: They contain the array elements which are read by the considered statements, before being possibly redefined by another instruction in the same statement. They differ from READ regions because they take into account the order in which array elements are defined. This is illustrated in Figure 6.3: `A`, `B` and `C` are all read by the dark part of the code; but `C` is not imported, because its value is defined before being read in the fragment of code which is considered. Only `A` and `B` belong to its IN regions.

> Also, in Figure 5.2 on Page 67, the body of the second `J` loop reads the element `WORK(J,K)`, but does not imports its value because it is previously defined in the same iteration. On the contrary, the element `WORK(J,K-1)` is imported from the first `J` loop.

$\mathcal{R}_i$, $\overline{\mathcal{R}}_i$ and $\underline{\mathcal{R}}_i$ define the exact and approximate semantics of IN regions. Because our expressions have no side-effects, there is no need to define IN regions on the domain of expressions: They are identical to READ regions. They are thereby solely defined on

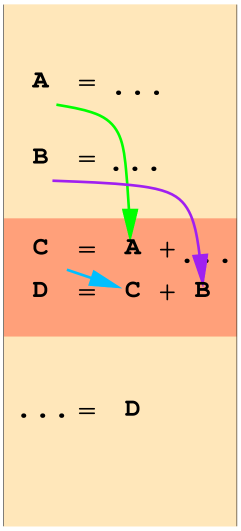


Figure 6.3: READ and IN regions

the domain of statements $\boldsymbol{S}$:

$$\mathcal{R}_i : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \wp(\mathbb{Z}^{d_i}))$$
$$\overline{\mathcal{R}}_i : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \tilde{\wp}(\mathbb{Z}^{d_i}))$$
$$\underline{\mathcal{R}}_i : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \tilde{\wp}(\mathbb{Z}^{d_i}))$$

These semantic functions are provided in Tables E.6 and E.7 on Pages 296 and 297. The most important and complicated are detailed below.

6.5.1 Assignment

The IN regions of an assignment are identical to the corresponding READ regions because the values of the referenced elements cannot come from the assignment itself, according to the FORTRAN standard [8]. Henceforth, the write on the left hand side cannot *cover* the reads performed while evaluating the right hand side and the subscript expressions of the left hand side:

$$\mathcal{R}_i[\![ref = exp]\!] = \mathcal{R}_r[\![ref = exp]\!]$$

Corresponding approximations are straightforwardly derived from the exact semantics.

If right hand side expressions had side-effects (through external function calls for instance), IN regions should be defined on the domain of expressions, and the definition would be[3]:

$$\begin{array}{lcl} \mathcal{R}_i[\![var(exp_1,\ldots,exp_k) = exp]\!] & = & \mathcal{R}_r[\![exp_1,\ldots,exp_k]\!] \cup \mathcal{R}_i[\![exp]\!] \\ \mathcal{R}_r[\![var = exp]\!] & = & \mathcal{R}_i[\![exp]\!] \end{array}$$

6.5.2 Sequence of instructions

Intuitively, the IN regions of the sequence $S_1; S_2$ contain the elements imported by instruction S_1 ($\mathcal{R}_i[\![S_1]\!]$); plus the elements imported by S_2 after S_1 has been executed ($\mathcal{R}_i[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]$), but which are not defined by S_1 ($\mathcal{R}_w[\![S_1]\!]$):

$$\mathcal{R}_i[\![S_1; S_2]\!] = \mathcal{R}_i[\![S_1]\!] \cup \big((\mathcal{R}_i[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]) \boxminus \mathcal{R}_w[\![S_1]\!]\big)$$

As for READ regions of a sequence of instructions, approximations have to take continuation conditions into account:

$$\begin{array}{lcl} \overline{\mathcal{R}}_i[\![S_1; S_2]\!] & = & \overline{\mathcal{R}}_i[\![S_1]\!] \,\overline{\cup}\, \big((\overline{\mathcal{R}}_i[\![S_2]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}[\![S_1]\!] \,\overline{\circ}\, \overline{\mathcal{C}}[\![S_1]\!]) \,\overline{\boxminus}\, \underline{\mathcal{R}}_w[\![S_1]\!]\big) \\ \underline{\mathcal{R}}_i[\![S_1; S_2]\!] & = & \underline{\mathcal{R}}_i[\![S_1]\!] \,\underline{\cup}\, \big((\underline{\mathcal{R}}_i[\![S_2]\!] \,\underline{\bullet}\, \overline{\mathcal{T}}[\![S_1]\!] \,\underline{\circ}\, \underline{\mathcal{C}}[\![S_1]\!]) \,\underline{\boxminus}\, \overline{\mathcal{R}}_w[\![S_1]\!]\big) \end{array}$$

The correctness is again ensured by the use of derived approximate operators, and by Properties 6.1 and 6.2.

6.5.3 Conditional instruction

The case of an `if` statement is very similar to READ and WRITE regions. IN regions contain the elements read when executing the condition ($\mathcal{R}_r[\![C]\!]$), plus the elements imported by S_1 or S_2, depending on the result of the condition evaluation:

$$\mathcal{R}_i[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] = \mathcal{R}_r[\![C]\!] \cup (\mathcal{R}_i[\![S_1]\!] \circ \mathcal{E}_c[\![C]\!]) \cup (\mathcal{R}_i[\![S_2]\!] \circ \mathcal{E}_c[\![\texttt{.not.}C]\!])$$

The over-approximation is directly derived from this definition, whereas the under-approximation contains a corrective term to better take into account the semantics of `if` statements, as for READ regions (see Property 6.3):

$$\begin{array}{l} \overline{\mathcal{R}}_i[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] = \\ \qquad \overline{\mathcal{R}}_r[\![C]\!] \,\overline{\cup}\, (\overline{\mathcal{R}}_i[\![S_1]\!] \,\overline{\circ}\, \overline{\mathcal{E}}_c[\![C]\!]) \,\overline{\cup}\, (\overline{\mathcal{R}}_i[\![S_2]\!] \,\overline{\circ}\, \overline{\mathcal{E}}_c[\![\texttt{.not.}C]\!]) \\ \underline{\mathcal{R}}_i[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] = \\ \qquad \underline{\mathcal{R}}_r[\![C]\!] \,\underline{\cup}\, (\underline{\mathcal{R}}_i[\![S_1]\!] \,\underline{\circ}\, \underline{\mathcal{E}}_c[\![C]\!]) \,\underline{\cup}\, (\underline{\mathcal{R}}_i[\![S_2]\!] \,\underline{\circ}\, \underline{\mathcal{E}}_c[\![\texttt{.not.}C]\!]) \,\underline{\cup}\, (\underline{\mathcal{R}}_i[\![S_1]\!] \,\underline{\cap}\, \underline{\mathcal{R}}_i[\![S_2]\!]) \end{array}$$

If expressions had side effects, the definition would have to take into account the modification of the memory store by the execution of C ($\mathcal{T}[\![C]\!]$), and the fact that write effects during the condition evaluation could cover the reads in the branches. The definition would thus become:

$$\begin{array}{l} \mathcal{R}_i[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] = \\ \qquad \mathcal{R}_i[\![C]\!] \cup \big((\mathcal{R}_i[\![S_1]\!] \circ \mathcal{E}_c[\![C]\!] \cup \mathcal{R}_i[\![S_2]\!] \circ \mathcal{E}_c[\![\texttt{.not.}C]\!]) \circ \mathcal{T}[\![C]\!] \boxminus \mathcal{R}_w[\![C]\!]\big) \end{array}$$

[3] It is always assumed that the value resulting from the evaluation of the right hand side is assigned to the left hand side reference after the evaluation of the right hand side is completed.

6.5.4 `do while` loop

When executing a `do while` statement, the condition is evaluated first; it may import some array elements ($\mathcal{R}_i[\![C]\!]$). Then, if $\mathcal{E}[\![C]\!]$ is *true*, S is executed, and it possibly imports some elements $\mathcal{R}_i[\![S]\!]$. Up to now the IN regions are given by:

$$\mathcal{R}_i[\![C]\!] \cup \mathcal{R}_i[\![S]\!] \circ \mathcal{E}_c[\![C]\!]$$

Then the `do while` loop is executed again, in the memory store resulting from the execution of S. It imports the elements $\mathcal{R}_i[\![\texttt{do while(}C\texttt{)}\ S]\!]$. However, some of these elements may have been written by the first execution of S ($\mathcal{R}_w[\![S]\!]$), and must not be included in the IN regions of the global computation. The semantics is thus:

$$\begin{aligned}\mathcal{R}_i[\![\texttt{do while(}C\texttt{)}\ S]\!] = \mathcal{R}_i[\![C]\!] \cup \\ \big(\mathcal{R}_i[\![S]\!] \cup (\mathcal{R}_i[\![\texttt{do while(}C\texttt{)}\ S]\!] \circ \mathcal{T}[\![S]\!] \boxminus \mathcal{R}_w[\![S]\!])\big) \circ \mathcal{E}[\![C]\!]\end{aligned}$$

This is a continuous recursive function defined on a lattice. Its best definition is thereby given by the least fixed point of a functional:

$$\mathcal{R}_i[\![\texttt{do while(}C\texttt{)}\ S]\!] = \text{lfp}\big(\lambda f.\mathcal{R}_i[\![C]\!] \cup \big\{\mathcal{R}_i[\![S]\!] \cup \big((f \circ \mathcal{T}[\![S]\!]) \boxminus \mathcal{R}_w[\![S]\!]\big)\big\} \circ \mathcal{E}_c[\![C]\!]\big)$$

Approximations are directly derived from this definition using approximate operators, except, as usual, for the use of continuation conditions.

6.5.5 The special case of `do` loops

The previous section defined the semantics of IN regions for general `do while` loops. This section examines the impact of the particularities of `do` loops on this definition, as already done in Section 6.3.6 for READ regions.

The contribution of an iteration k to the IN regions of the `do` loop is equal to its own IN regions, minus the elements which are written in the previous iterations k' ($1 \leq k' < k$):

$$\begin{aligned}&\mathcal{R}_i[\![\texttt{do i=1,n}\ S\ \texttt{enddo}]\!] = \\ &\qquad \lambda\sigma.\mathcal{R}_r[\![\texttt{n}]\!]\sigma \cup \bigcup_{k=1}^{k=\mathcal{E}[\![\texttt{n}]\!]\sigma} \big\{\mathcal{R}_i[\![S]\!] \circ \mathcal{T}[\![\texttt{do i=1,}k-1\ S\ \texttt{enddo}]\!]\sigma \boxminus \\ &\qquad\qquad \bigcup_{k'=1}^{k'=k-1} \big(\mathcal{R}_w[\![S]\!] \circ \mathcal{T}[\![\texttt{do i=1,}k'-1\ S\ \texttt{enddo}]\!]\sigma\big)\big\}\end{aligned}$$

Using the same equivalences as in Section 6.3.6, we obtain three alternative definitions:

$$\begin{aligned}
&\mathcal{R}_i[\![\texttt{do i=1,n } S \texttt{ enddo}]\!] \\
&\quad = \mathrm{lfp}\big(\lambda f.\mathcal{R}_i[\![C]\!] \cup \big\{\mathcal{R}_i[\![S]\!] \cup \big((f \circ \mathcal{T}[\![S]\!]) \boxminus \mathcal{R}_w[\![S]\!]\big)\big\} \circ \mathcal{E}_c[\![C]\!]\big) &(6.8)\\
&\quad = \lambda\sigma.\mathcal{R}_r[\![\texttt{n}]\!]\sigma \cup \bigcup_{k=1}^{k=\mathcal{E}[\![\texttt{n}]\!]\sigma} \big\{\mathcal{R}_i[\![S]\!] \circ \mathcal{T}[\![\texttt{do i=1,}k-1\ S\texttt{ enddo}]\!]\sigma \boxminus \\
&\qquad\qquad \bigcup_{k'=1}^{k'=k-1} \big(\mathcal{R}_w[\![S]\!] \circ \mathcal{T}[\![\texttt{do i=1,}k'-1\ S\texttt{ enddo}]\!]\sigma\big)\big\} &(6.9)\\
&\quad = \lambda\sigma.\mathcal{R}_r[\![\texttt{n}]\!]\sigma \cup \mathrm{proj}_k(\mathcal{R}_i[\![S]\!] \circ \mathcal{T}[\![\texttt{do i=1,}k-1\ S\texttt{ enddo}]\!] \\
&\qquad\qquad \circ\, \mathcal{E}_c[\![\texttt{(1.le.}k\texttt{).and.(}k\texttt{.le.n)}]\!] \\
&\qquad\qquad \boxminus \mathrm{proj}_{k'}(\mathcal{R}_w[\![S]\!] \circ \mathcal{T}[\![\texttt{do i=1,}k'-1\ S\texttt{ enddo}]\!] \\
&\qquad\qquad\qquad \circ\, \mathcal{E}_c[\![\texttt{(1.le.}k'\texttt{).and.(}k'\texttt{.lt.}k\texttt{)}]\!]))\sigma &(6.10)
\end{aligned}$$

We do not provide the corresponding approximations, because it would provide no useful insight, since they are built similarly to those for READ regions of `do` loops.

6.5.6 Unstructured parts of code

As for READ and WRITE regions, IN regions of unstructured parts of code are computed by PIPS in a very straightforward fashion: Over-approximate regions are equal to the union of the regions imported by all the nodes (in which all the variables modified by the nodes have been eliminated to model store modifications); under-approximate regions are systematically set to the safe empty set. Again, it would be quite expensive to compute accurate IN regions for general control flow graphs; however, `do while` loops built with `goto` statements should be handled.

6.6 OUT Regions

The OUT regions of a statement are its *exported* or *live and used afterwards* regions. They contain the array elements which are defined, and are used in the continuation of the program, that is to say in the instructions whose execution follows the current statement. They differ from WRITE regions because they take into account whether the array elements are reused afterwards or not. This is illustrated in Figure 6.4: Both `C` and `D` are written in the dark part of the code; `C` is reused locally, but not after the considered part of the code has been executed; on the contrary, `D` is reused afterwards, and thus belongs to the OUT regions.

In the program of Figure 5.2, the first `J` loop exports all the elements of the array `WORK` it defines towards the second `J` loop, whereas the elements of `WORK` defined in the latter are not exported toward the next iterations of the `I` loop.

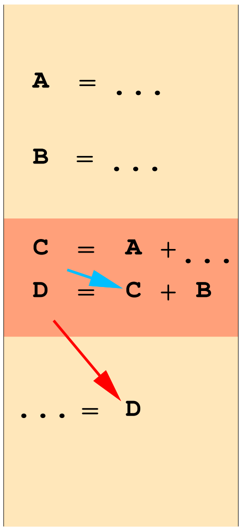


Figure 6.4: WRITE and OUT regions

Notice that OUT and IN regions are not symmetrical: IN regions do not depend on the definitions preceding the current piece of code. This is because the program is supposed to be correct; hence, every use of a variable is preceded by a definition, and there is no need to check the existence of the definition. On the contrary, OUT regions give information about the scope in which the definition of a variable is used.

OUT regions are solely defined on the domain of statements $\boldsymbol{S}$. It would have no sense to define them on the domain of expressions since our expressions have no side-effects.

$$\mathcal{R}_o : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \wp(\mathbb{Z}^{d_i}))$$
$$\overline{\mathcal{R}}_o : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \tilde{\wp}(\mathbb{Z}^{d_i}))$$
$$\underline{\mathcal{R}}_o : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \tilde{\wp}(\mathbb{Z}^{d_i}))$$

The over-approximate OUT regions of a statement contain the regions *potentially* written by this statement, and *potentially* read in the program continuation. Whereas under-approximate OUT regions contain the array elements *certainly* written by the current statement, and *certainly* read in the program continuation.

The OUT region of a statement is not defined *per se*, but depends on the future of the computation. For instance, the OUT region of S_1 in program $S_1; S_2$ is a function of $S_1; S_2$ as a whole, and of S_2. Thus, OUT regions are propagated in a top-down fashion along the call graph and hierarchical control flow graph of the program. Since I/O operations are part of the program, the OUT region of the main program, from which the other OUT regions are derived, is initialized to $\lambda\sigma.\varnothing$[4].

[4]As a consequence, programs without output instructions are considered as no side-effect programs, and could be flagged as dead code using OUT regions!

Note Because of this inward propagation, we use the formalism of attribute grammars. But, as explained in Chapter 4, we keep the notations of denotational semantics (i.e. the statements are provided inside double brackets) for uniformization purposes.

The next sections detail the exact and approximate semantics of OUT regions for the main structures of the language $\boldsymbol{\mathcal{L}}$. They are summarized in Tables E.8 and E.9.

6.6.1 Sequence of instructions

The OUT region corresponding to the sequence $S_1; S_2$ is supposed to be known, and our purpose is to find the OUT regions corresponding to S_1 and S_2, respectively $\mathcal{R}_o[\![S_1]\!]$ and $\mathcal{R}_o[\![S_2]\!]$.

Intuitively, the regions exported by S_2 are the regions exported by the whole sequence ($\mathcal{R}_o[\![S_1; S_2]\!]$), and written by S_2 ($\mathcal{R}_w[\![S_2]\!]$). Great care must be taken of the modification of the memory store by S_1: The reference is the store just before the execution of the sequence, and the store before S_2 is given by applying $\mathcal{T}[\![S_1]\!]$ to the reference store. $\mathcal{R}_o[\![S_2]\!]$ must therefore verify the following constraint:

$$\mathcal{R}_o[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!] = \mathcal{R}_o[\![S_1; S_2]\!] \cap \mathcal{R}_w[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]$$

We cannot give a constructive definition of $\mathcal{R}_o[\![S_2]\!]$, because $\mathcal{T}[\![S_1]\!]$ is not invertible. The same problem arises with assignments in HOARE logic [102, 124]:

> In HOARE logic, the assignment axiom is: $p(a, x \leftarrow t)\{\texttt{x:=t}\}p(a, x)$, $p(a, x \leftarrow t)$ being the predicate $p(a, x)$ where x is replaced by t. When $p(a, x \leftarrow t)$ is provided and the purpose is to find $p(a, x)$, $p(a, x)$ must be *guessed* so that it meets the axiom requirements.

However, safe approximations can be defined using $\overline{\mathcal{T}}^{-1}$:

$$\overline{\mathcal{R}}_o[\![S_2]\!] = \overline{\mathcal{R}}_o[\![S_1; S_2]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{T}}^{-1}[\![S_1]\!] \mathbin{\overline{\cap}} \overline{\mathcal{R}}_w[\![S_2]\!] \tag{6.11}$$

$$\underline{\mathcal{R}}_o[\![S_2]\!] = \underline{\mathcal{R}}_o[\![S_1; S_2]\!] \mathbin{\underline{\bullet}} \overline{\mathcal{T}}^{-1}[\![S_1]\!] \mathbin{\underline{\cap}} \underline{\mathcal{R}}_w[\![S_2]\!] \tag{6.12}$$

Notice that the continuation conditions of S_1 are not necessary in the definition of $\underline{\mathcal{R}}_o[\![S_2]\!]$, because if S_1 contained a `stop` statement (possibly guarded by a condition), then, by construction, $\underline{\mathcal{R}}_o[\![S_1; S_2]\!]$ would be the empty set.

Property 6.4
Let $S_1; S_2$ be a sequence of instructions. If $\mathcal{R}_o[\![S_2]\!]$, $\overline{\mathcal{R}}_o[\![S_2]\!]$ and $\underline{\mathcal{R}}_o[\![S_2]\!]$ are defined as above, then

$$\underline{\mathcal{R}}_o[\![S_2]\!] \sqsubseteq \mathcal{R}_o[\![S_2]\!] \sqsubseteq \overline{\mathcal{R}}_o[\![S_2]\!]$$

Proof The proof is similar for $\overline{\mathcal{R}}_o[\![S_2]\!]$ and $\underline{\mathcal{R}}_o[\![S_2]\!]$. We only provide it for $\overline{\mathcal{R}}_o[\![S_2]\!]$.

By composing (6.11) by $\mathcal{T}[\![S_1]\!]$, and given that $\overline{\mathcal{T}}^{-1}[\![S_1]\!] \circ \mathcal{T}[\![S_1]\!] = \overline{\{\lambda\sigma.\sigma\}}$, we obtain:

$$\begin{aligned}
\overline{\mathcal{R}}_o[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!] &= \overline{\mathcal{R}}_o[\![S_1; S_2]\!] \mathbin{\overline{\bullet}} \overline{\{\lambda\sigma.\sigma\}} \mathbin{\overline{\cap}} \overline{\mathcal{R}}_w[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!] \\
&\sqsupseteq \mathcal{R}_o[\![S_1; S_2]\!] \cap \mathcal{R}_w[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!] \\
&\sqsupseteq \mathcal{R}_o[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]
\end{aligned} \tag{6.13}$$

Let σ be a memory store. If $\exists \sigma' \in \mathbf{\Sigma} : \mathcal{T}[\![S_1]\!]\sigma' = \sigma$, Equation (6.13) gives:

$$\overline{\mathcal{R}}_o[\![S_2]\!]\sigma \sqsupseteq \mathcal{R}_o[\![S_2]\!]\sigma$$

On the contrary, if $\nexists \sigma' \in \mathbf{\Sigma} : \mathcal{T}[\![S_1]\!]\sigma' = \sigma$, then σ cannot be reached during an execution, and $\mathcal{R}_o[\![S_2]\!]\sigma$ is undefined ($= \perp$) and is therefore over-approximated by any definition of $\overline{\mathcal{R}}_o[\![S_2]\!]\sigma$. Hence,

$$\forall \sigma \in \mathbf{\Sigma}, \overline{\mathcal{R}}_o[\![S_2]\!]\sigma \sqsupseteq \mathcal{R}_o[\![S_2]\!]\sigma$$

$\square$

The regions exported by S_1 are the regions exported by the whole sequence but not by S_2 ($\mathcal{R}_o[\![S_1; S_2]\!] \boxminus \mathcal{R}_o[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]$), and which are written by S_1. To these regions must be added the elements written by S_1 and exported towards S_2, that is to say imported by S_2 ($\mathcal{R}_i[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]$):

$$\mathcal{R}_o[\![S_1]\!] = \mathcal{R}_w[\![S_1]\!] \cap \big((\mathcal{R}_o[\![S_1; S_2]\!] \boxminus \mathcal{R}_o[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]) \cup \mathcal{R}_i[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]\big)$$

Another equivalent possibility is to use $\mathcal{R}_w[\![S_2]\!]$ instead of $\mathcal{R}_o[\![S_2]\!]$: If an array element is written by S_2 and is exported by $S_1; S_2$, it is systematically exported by S_2; it cannot be exported by S_1 because the writes in S_1 are covered by the writes in S_2.

The approximations corresponding to $\mathcal{R}_o[\![S_1]\!]$ can be directly derived from the previous definition using approximate operators.

6.6.2 Conditional instruction

For an `if` C `then` S_1 `else` S_2 statement, OUT regions are propagated from the global construct towards the branches. For instance, when the condition is true, the OUT regions for S_1 are the regions exported by the whole construct, and written by S_1. $\mathcal{R}_o[\![S_1]\!]$ and $\mathcal{R}_o[\![S_2]\!]$ therefore are:

$$\begin{aligned}
\mathcal{R}_o[\![S_1]\!] &= (\mathcal{R}_o[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] \cap \mathcal{R}_w[\![S_1]\!]) \circ \mathcal{E}_c[\![C]\!] \\
\mathcal{R}_o[\![S_2]\!] &= (\mathcal{R}_o[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] \cap \mathcal{R}_w[\![S_2]\!]) \circ \mathcal{E}_c[\![\texttt{.not.}C]\!]
\end{aligned}$$

Both approximations are easily derived from these definitions using approximate operators. They can be found in Tables E.8 and E.9.

If side-effect expressions were allowed, we would have the same problems as for deriving $\mathcal{R}_o[\![S_2]\!]$ from $\mathcal{R}_o[\![S_1; S_2]\!]$, because $\mathcal{T}[\![C]\!]$ would have to be taken into account.

6.6.3 `do while` loops

The OUT regions of the whole loop ($\mathcal{R}_o[\![\texttt{do while(}C\texttt{)}\ S]\!]$) are supposed to be known, and we are looking for the OUT regions of the body. In fact, we are looking for the OUT regions of the k-th iteration, provided that it is executed, that is to say that $k-1$ iterations have already been executed, and that the condition in the resulting store evaluates to *true*:

$$\mathcal{E}_c[\![C]\!] \circ \big(\underbrace{(\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!]) \circ \ldots \circ (\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!])}_{k-1}\big) \neq \perp$$

The definition is very similar to the definition for a sequence of instructions. The OUT regions for the k-th iteration contain the elements it writes and which are:

- Either exported by the whole loop and not overwritten in the subsequent iterations (that is to say after k executions of the loop body $\mathcal{T}[\![S]\!]$, provided that the condition evaluates to *true* at each step):

$$\mathcal{R}_o[\![\texttt{do while(}C\texttt{)}\ S]\!] \boxminus \mathcal{R}_w[\![\texttt{do while(}C\texttt{)}\ S]\!] \circ \underbrace{\big((\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!]) \circ \ldots \circ (\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!])\big)}_{k}$$

- Or exported towards the next iterations, that is to say imported by the next iterations (thus after k iterations; see, as above):

$$\mathcal{R}_i[\![\texttt{do while(}C\texttt{)}\ S]\!] \circ \underbrace{\big((\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!]) \circ \ldots \circ (\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!])\big)}_{k}$$

The constraint verified by $\mathcal{R}_o[\![S]\!]$ at the k-th iteration therefore is:

$$\begin{aligned}
&\mathcal{R}_o[\![S]\!]_k \circ \mathcal{E}_c[\![C]\!] \circ \underbrace{\big((\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!]) \circ \ldots \circ (\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!])\big)}_{k-1} = \\
&\quad \mathcal{R}_w[\![S]\!] \circ \mathcal{E}_c[\![C]\!] \circ \underbrace{\big((\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!]) \circ \ldots \circ (\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!])\big)}_{k-1} \cap \\
&\quad \Big((\mathcal{R}_o[\![\texttt{do while(}C\texttt{)}\ S]\!] \boxminus \\
&\quad \mathcal{R}_w[\![\texttt{do while(}C\texttt{)}\ S]\!] \circ \underbrace{\big((\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!]) \circ \ldots \circ (\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!])\big)}_{k}) \\
&\quad \cup \mathcal{R}_i[\![\texttt{do while(}C\texttt{)}\ S]\!] \circ \underbrace{\big((\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!]) \circ \ldots \circ (\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!])\big)}_{k} \Big)
\end{aligned} \tag{6.14}$$

As for sequences of instructions, this is not a constructive definition, but a constraint enforced on the exact semantics. This is not surprising, since `do while` loops are considered as sequences of guarded iterations. However, using inverse transformers, constructive definitions can be given for over- and under-approximations:

$$\begin{aligned}
\overline{\mathcal{R}}_o[\![S]\!]_k &= \overline{\mathcal{R}}_w[\![S]\!] \mathbin{\overline{\cap}} \\
&\Big((\overline{\mathcal{R}}_o[\![\texttt{do while(}C\texttt{)}\ S]\!] \mathbin{\overline{\bullet}} \underbrace{\big((\overline{\mathcal{E}}_c[\![C]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{T}}^{-1}[\![S]\!]) \mathbin{\overline{\bullet}} \ldots \mathbin{\overline{\bullet}} (\overline{\mathcal{E}}_c[\![C]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{T}}^{-1}[\![S]\!])\big)}_{k-1} \mathbin{\overline{\circ}} \overline{\mathcal{E}}_c[\![C]\!] \\
&\quad \mathbin{\overline{\boxminus}} \underline{\mathcal{R}}_w[\![\texttt{do while(}C\texttt{)}\ S]\!] \mathbin{\underline{\bullet}} \overline{\mathcal{T}}[\![S]\!]) \mathbin{\overline{\cup}} \overline{\mathcal{R}}_i[\![\texttt{do while(}C\texttt{)}\ S]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{T}}[\![S]\!] \Big) \\
\underline{\mathcal{R}}_o[\![S]\!]_k &= \underline{\mathcal{R}}_w[\![S]\!] \mathbin{\underline{\cap}} \\
&\Big((\underline{\mathcal{R}}_o[\![\texttt{do while(}C\texttt{)}\ S]\!] \mathbin{\underline{\bullet}} \underbrace{\big((\underline{\mathcal{E}}_c[\![C]\!] \mathbin{\underline{\bullet}} \overline{\mathcal{T}}^{-1}[\![S]\!]) \mathbin{\underline{\bullet}} \ldots \mathbin{\overline{\bullet}} (\underline{\mathcal{E}}_c[\![C]\!] \mathbin{\underline{\bullet}} \overline{\mathcal{T}}^{-1}[\![S]\!])\big)}_{k-1} \mathbin{\underline{\circ}} \underline{\mathcal{E}}_c[\![C]\!] \\
&\quad \mathbin{\underline{\boxminus}} \overline{\mathcal{R}}_w[\![\texttt{do while(}C\texttt{)}\ S]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{T}}[\![S]\!]) \mathbin{\underline{\cup}} \underline{\mathcal{R}}_i[\![\texttt{do while(}C\texttt{)}\ S]\!] \mathbin{\underline{\bullet}} \overline{\mathcal{T}}[\![S]\!] \Big)
\end{aligned}$$

$\overline{\mathcal{R}}_o[\![S]\!]_k$ and $\underline{\mathcal{R}}_o[\![S]\!]_k$ are not very convenient to use in subsequent analyses or program transformations: For `do while` loops, the iteration counter is not explicitly known, and an invariant information is usually more appropriate. We therefore define $\overline{\mathcal{R}}_o[\![S]\!]_{\rm inv}$ and $\underline{\mathcal{R}}_o[\![S]\!]_{\rm inv}$ as:

$$\begin{aligned}
\overline{\mathcal{R}}_o[\![S]\!]_{\rm inv} = {} & \overline{\mathcal{R}}_w[\![S]\!] \mathbin{\overline{\cap}} \\
& \Big((\overline{\mathcal{R}}_o[\![\texttt{do while(}C\texttt{)}\ S]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{T}}^{-1}_{\rm inv}[\![\texttt{do while(}C\texttt{)}\ S]\!] \mathbin{\overline{\circ}} \overline{\mathcal{E}}_c[\![C]\!] \\
& \quad \mathbin{\overline{\boxminus}} \underline{\mathcal{R}}_w[\![\texttt{do while(}C\texttt{)}\ S]\!] \mathbin{\underline{\bullet}} \overline{\mathcal{T}}[\![S]\!]) \mathbin{\overline{\cup}} \overline{\mathcal{R}}_i[\![\texttt{do while(}C\texttt{)}\ S]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{T}}[\![S]\!] \Big) \\
\underline{\mathcal{R}}_o[\![S]\!]_{\rm inv} = {} & \underline{\mathcal{R}}_w[\![S]\!] \mathbin{\underline{\cap}} \\
& \Big((\underline{\mathcal{R}}_o[\![\texttt{do while(}C\texttt{)}\ S]\!] \mathbin{\underline{\bullet}} \overline{\mathcal{T}}^{-1}_{\rm inv}[\![\texttt{do while(}C\texttt{)}\ S]\!] \mathbin{\underline{\circ}} \underline{\mathcal{E}}_c[\![C]\!] \\
& \quad \mathbin{\underline{\boxminus}} \overline{\mathcal{R}}_w[\![\texttt{do while(}C\texttt{)}\ S]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{T}}[\![S]\!]) \mathbin{\underline{\cup}} \underline{\mathcal{R}}_i[\![\texttt{do while(}C\texttt{)}\ S]\!] \mathbin{\underline{\bullet}} \overline{\mathcal{T}}[\![S]\!] \Big)
\end{aligned}$$

It is easy to verify that these invariant sets further approximate $\underline{\mathcal{R}}_o[\![S]\!]_k$ and $\overline{\mathcal{R}}_o[\![S]\!]_k$, for any k:

$$\forall k,\ \underline{\mathcal{R}}_o[\![S]\!]_{\rm inv} \sqsubseteq \underline{\mathcal{R}}_o[\![S]\!]_k \text{ and } \overline{\mathcal{R}}_o[\![S]\!]_k \sqsubseteq \overline{\mathcal{R}}_o[\![S]\!]_{\rm inv}$$

6.6.4 The special case of `do` loops

When dealing with a `do while` loop, we never know whether the k-th iteration we are interested in actually exists. This is why each iteration execution is guarded by the evaluation of the condition. This is the reason for using the terms

$$\mathcal{E}_c[\![C]\!] \circ \big(\underbrace{(\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!]) \circ \ldots \circ (\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!])}_{k-1}\big)$$

in Equation (6.14). However, for a loop `do i=1,n` S `enddo`, the number of iterations is known at compile time, and checking whether iteration k is an actual iteration is easily done with:

$$\mathcal{E}_c[\![\texttt{(1.le.}k\texttt{).and.(}k\texttt{.le.n)}]\!]$$

And the memory store just before iteration k is thereby given by

$$\big(\underbrace{\mathcal{T}[\![S]\!] \circ \ldots \circ \mathcal{T}[\![S]\!]}_{k-1}\big) \circ \mathcal{E}_c[\![\texttt{(1.le.}k\texttt{).and.(}k\texttt{.le.n)}]\!]$$

or,

$$\mathcal{T}[\![\texttt{do i=1,}k-1\ S\ \texttt{enddo}]\!] \circ \mathcal{E}_c[\![\texttt{(1.le.}k\texttt{).and.(}k\texttt{.le.n)}]\!]$$

With these equivalences, Equation (6.14) becomes:

$$
\begin{aligned}
&\mathcal{R}_o\llbracket S\rrbracket_k \circ \mathcal{T}\llbracket \mathtt{do\ i=1,}k-1\ S\ \mathtt{enddo}\rrbracket \circ \mathcal{E}_c\llbracket \mathtt{(1.le.}k\mathtt{).and.(}k\mathtt{.le.n)}\rrbracket = \\
&\quad \mathcal{R}_w\llbracket S\rrbracket \circ \mathcal{T}\llbracket \mathtt{do\ i=1,}k-1\ S\ \mathtt{enddo}\rrbracket \circ \mathcal{E}_c\llbracket \mathtt{(1.le.}k\mathtt{).and.(}k\mathtt{.le.n)}\rrbracket \cap \\
&\qquad \Big((\mathcal{R}_o\llbracket \mathtt{do\ i=1,n}\ S\ \mathtt{enddo}\rrbracket \boxminus \\
&\qquad \mathcal{R}_w\llbracket \mathtt{do\ i=}k+1\mathtt{,n}\ S\ \mathtt{enddo}\rrbracket \circ \mathcal{T}\llbracket \mathtt{do\ i=1,}k\ S\ \mathtt{enddo}\rrbracket \circ \\
&\qquad \mathcal{E}_c\llbracket \mathtt{(1.le.}k\mathtt{).and.(}k\mathtt{.lt.n)}\rrbracket) \cup \\
&\qquad \mathcal{R}_i\llbracket \mathtt{do\ i=}k+1\mathtt{,n}\ S\ \mathtt{enddo}\rrbracket \circ \mathcal{T}\llbracket \mathtt{do\ i=1,}k\ S\ \mathtt{enddo}\rrbracket \circ \\
&\qquad \mathcal{E}_c\llbracket \mathtt{(1.le.}k\mathtt{).and.(}k\mathtt{.lt.n)}\rrbracket \Big)
\end{aligned}
$$

As usual, constructive definitions can be given for the corresponding approximations, by using inverse transformers.

$$
\begin{aligned}
\overline{\mathcal{R}}_o\llbracket S\rrbracket_k = {}& \overline{\mathcal{R}}_w\llbracket S\rrbracket \,\overline{\cap} \\
& \Big\{ \big(\overline{\mathcal{R}}_o\llbracket \mathtt{do\ i=1,n}\ S\ \mathtt{enddo}\rrbracket \,\overline{\bullet}\, \overline{\mathcal{T}}^{-1}_{\text{inv}}\llbracket \mathtt{do\ i=1,n}\ S\ \mathtt{enddo}\rrbracket \\
& \quad \overline{\circ}\, \overline{\mathcal{E}}_c\llbracket \mathtt{(1.le.}k\mathtt{).and.(}k\mathtt{.le.n)}\rrbracket \\
& \quad \overline{\boxminus}\, \underline{\mathcal{R}}_w\llbracket \mathtt{do\ i=}k+1\mathtt{,n}\ S\ \mathtt{enddo}\rrbracket \,\underline{\bullet}\, \overline{\mathcal{T}}^{-1}\llbracket S\rrbracket \big) \\
& \quad \overline{\cup}\, \overline{\mathcal{R}}_i\llbracket \mathtt{do\ i=}k+1\mathtt{,n}\ S\ \mathtt{enddo}\rrbracket \,\overline{\bullet}\, \overline{\mathcal{T}}^{-1}\llbracket S\rrbracket \Big\}
\end{aligned}
$$

$$
\begin{aligned}
\underline{\mathcal{R}}_o\llbracket S\rrbracket_k = {}& \underline{\mathcal{R}}_w\llbracket S\rrbracket \,\underline{\cap} \\
& \Big\{ \big(\underline{\mathcal{R}}_o\llbracket \mathtt{do\ i=1,n}\ S\ \mathtt{enddo}\rrbracket \,\underline{\bullet}\, \overline{\mathcal{T}}^{-1}_{\text{inv}}\llbracket \mathtt{do\ i=1,n}\ S\ \mathtt{enddo}\rrbracket \\
& \quad \underline{\circ}\, \underline{\mathcal{E}}_c\llbracket \mathtt{(1.le.}k\mathtt{).and.(}k\mathtt{.le.n)}\rrbracket \\
& \quad \underline{\boxminus}\, \overline{\mathcal{R}}_w\llbracket \mathtt{do\ i=}k+1\mathtt{,n}\ S\ \mathtt{enddo}\rrbracket \,\overline{\bullet}\, \overline{\mathcal{T}}^{-1}\llbracket S\rrbracket \big) \\
& \quad \underline{\cup}\, \underline{\mathcal{R}}_i\llbracket \mathtt{do\ i=}k+1\mathtt{,n}\ S\ \mathtt{enddo}\rrbracket \,\underline{\bullet}\, \overline{\mathcal{T}}^{-1}\llbracket S\rrbracket \Big\}
\end{aligned}
$$

By further expliciting the terms using intermediate WRITE and IN regions (see Sections 6.3.6 and 6.5.5), we get the final forms which are implemented in PIPS:

$$
\begin{aligned}
\overline{\mathcal{R}}_o\llbracket S\rrbracket_k = {} & \lambda\sigma.\overline{\mathcal{R}}_w\llbracket S\rrbracket\sigma\,\overline{\cap} \\
& \Big\{ \big(\overline{\mathcal{R}}_o\llbracket \texttt{do i=1,n } S \texttt{ enddo}\rrbracket \,\overline{\bullet}\, \overline{\mathcal{T}}_{\mathrm{inv}}^{-1}\llbracket \texttt{do i=1,n } S \texttt{ enddo}\rrbracket \\
& \qquad \overline{\circ}\, \overline{\mathcal{E}}_c\llbracket \texttt{(1.le.}k\texttt{).and.(}k\texttt{.le.n)}\rrbracket\sigma \\
& \quad \overline{\boxminus} \underline{\bigcup}_{k'=k+1}^{k'=\underline{\mathcal{E}}\llbracket n\rrbracket\sigma} \underline{\mathcal{R}}_w\llbracket S\rrbracket \,\underline{\bullet}\, \overline{\mathcal{E}}_c\llbracket \texttt{i=}k'\rrbracket \,\overline{\bullet}\, \overline{\mathcal{T}}_{\mathrm{inv}}\llbracket \texttt{do i=1,n } S \texttt{ enddo}\rrbracket\sigma \big) \\
& \quad \overline{\cup} \overline{\bigcup}_{k'=k+1}^{k'=\overline{\mathcal{E}}\llbracket n\rrbracket\sigma} \{ \overline{\mathcal{R}}_i\llbracket S\rrbracket \,\overline{\circ}\, \overline{\mathcal{E}}_c\llbracket \texttt{i=}k'\rrbracket \,\overline{\bullet}\, \overline{\mathcal{T}}_{\mathrm{inv}}\llbracket \texttt{do i=1,n } S \texttt{ enddo}\rrbracket\sigma \\
& \qquad \overline{\boxminus} \underline{\bigcup}_{k''=k+1}^{k''=k'-1} \underline{\mathcal{R}}_w\llbracket S\rrbracket \,\underline{\bullet}\, \overline{\mathcal{E}}_c\llbracket \texttt{i=}k''\rrbracket \,\overline{\bullet}\, \overline{\mathcal{T}}_{\mathrm{inv}}\llbracket \texttt{do i=1,n } S \texttt{ enddo}\rrbracket\sigma \} \Big\}
\end{aligned}
$$

$$
\begin{aligned}
\underline{\mathcal{R}}_o\llbracket S\rrbracket_k = {} & \lambda\sigma.\underline{\mathcal{R}}_w\llbracket S\rrbracket\sigma\,\underline{\cap} \\
& \Big\{ \big(\underline{\mathcal{R}}_o\llbracket \texttt{do i=1,n } S \texttt{ enddo}\rrbracket \,\underline{\bullet}\, \overline{\mathcal{T}}_{\mathrm{inv}}^{-1}\llbracket \texttt{do i=1,n } S \texttt{ enddo}\rrbracket \\
& \qquad \underline{\circ}\, \underline{\mathcal{E}}_c\llbracket \texttt{(1.le.}k\texttt{).and.(}k\texttt{.le.n)}\rrbracket\sigma \\
& \quad \underline{\boxminus} \overline{\bigcup}_{k'=k+1}^{\overline{\mathcal{E}}\llbracket n\rrbracket\sigma} \overline{\mathcal{R}}_w\llbracket S\rrbracket \,\overline{\bullet}\, \overline{\mathcal{E}}_c\llbracket \texttt{i=}k'\rrbracket \,\overline{\bullet}\, \overline{\mathcal{T}}_{\mathrm{inv}}^{-1}\llbracket \texttt{do i=1,n } S \texttt{ enddo}\rrbracket\sigma \big) \\
& \quad \underline{\cup} \underline{\bigcup}_{k'=k+1}^{k'=\underline{\mathcal{E}}\llbracket n\rrbracket\sigma} \{ \underline{\mathcal{R}}_i\llbracket S\rrbracket \,\underline{\bullet}\, \overline{\mathcal{E}}_c\llbracket \texttt{i=}k'\rrbracket \,\overline{\bullet}\, \overline{\mathcal{T}}_{\mathrm{inv}}\llbracket \texttt{do i=1,n } S \texttt{ enddo}\rrbracket\sigma \\
& \qquad \underline{\boxminus} \overline{\bigcup}_{k''=k+1}^{k''=k'-1} \underline{\mathcal{R}}_w\llbracket S\rrbracket \,\underline{\bullet}\, \overline{\mathcal{E}}_c\llbracket \texttt{i=}k''\rrbracket \,\overline{\bullet}\, \overline{\mathcal{T}}_{\mathrm{inv}}\llbracket \texttt{do i=1,n } S \texttt{ enddo}\rrbracket\sigma \} \Big\}
\end{aligned}
$$

Here, there is no need for an invariant, since the loop index is explicit. This does not mean that we will compute $\overline{\mathcal{R}}_o\llbracket S\rrbracket_1, \ldots, \overline{\mathcal{R}}_o\llbracket S\rrbracket_{10000}, \ldots$: The loop index remains a symbolic variable, and the result can be seen as a function of the loop index.

6.6.5 Unstructured parts of code

Unlike READ, WRITE and IN regions, OUT regions of an unstructured piece of code are propagated from the global construct to the inner nodes, but still in a straightforward fashion:

- Under-approximate regions are systematically set to the safe empty set.
- To compute over-approximations, the variables modified throughout the unstructured piece of code are eliminated from the initial OUT region. This gives a region which is invariant in this fragment of code. The chosen over-approximate OUT region of each node is then the intersection of the OUT region for the whole structure and of the local WRITE region.

6.7 Impact of Variable Aliasing

Variable aliasing takes place when two different scalar variable names or two references to individual array elements with different names refer to the same memory location. In FORTRAN, this can be due to two reasons: An `equivalence` statement; or the same variable (or two *equivalent* variables) passed as an argument to two distinct formal parameters. In the first case, the aliasing is perfectly known during any intraprocedural analysis. However, the second case requires an interprocedural analysis to know which formal parameters are aliased. This can be represented by an *Aliased* set, consisting of pairs of aliased variable names (scalars or arrays), with an information about the corresponding elements in the case of arrays. Since array elements are stored in column order in FORTRAN, this is sufficient to describe the complete aliased memory suites.

But in fact, this set is not computable in the general case, due to interprocedural aliasing, and it must be approximated [33]: $MayBeAliased$ and $MustBeAliased$ sets respectively are its over- and under-approximations. Since the smallest variable declaration scope in FORTRAN is the procedure, both sets are constant during intraprocedural analyses.

Up to now, our array region analyses do not take into account variable aliasing, because it would have required using environments instead of memory stores, and it would have greatly complicated our data-flow equations. For READ and WRITE regions, this is not disturbing if subsequent analyses and transformations deal with variable aliasing. This is due to the fact that READ and WRITE analyses only involve union operations: Adding a region for a variable aliased with another one is strictly equivalent to adding the corresponding region for the aliased variable; adding both regions would not modify the semantics of the analyses, and would just result in a redundant solution.

On the contrary, IN and OUT regions involve subtractions and intersections. If variable aliasing is not taken into account, we may, for instance, fail to discover that a read is covered by a write, or we may underestimate the over-approximation of an intersection, which would yield an erroneous result. In fact, in programs respectful of the FORTRAN standard [8], both problems cannot be due to an interprocedural aliasing because the standard forbids the aliasing of arguments which are modified in the called procedure, and over-approximate intersections are only used to combine WRITE regions. The intraprocedural aliasing remains, but it is perfectly known, and aliased sets do not need to be approximated.

There are several methods to handle this kind of problem:

- The first one is to have a single representative for any set of may or must aliased variables. A well-known technique [35] is to introduce a new one-dimensional array to represent the corresponding memory suite, and to provide the offset of the first element of each aliased variable from the first element of the memory suite. Each reference to an aliased variable, and thus each region, can be translated into a reference (or region) to the corresponding memory suite array. This operation is not always possible, because of interprocedural aliasing, but we have seen that this is not a problem in FORTRAN.

The advantage obviously is that there is no more aliasing between the different memory suite arrays. However, this method has severe limiting factors. First, the aliasing must be perfectly known at compile time; we have seen that this is the case in FORTRAN for intraprocedural aliasing, and that interprocedural aliasing does not have any effect on the results of our analyses; but this may not be the case in other languages. The second drawback is due to the fact that all aliased array references must be linearized, hence possibly resulting in non-linear array subscripts, and thereby in unknown regions ($\top$ or $\perp$ depending on the approximation), whereas the regions of the actual references could be accurately represented. The third drawback is that the resulting regions do not correspond to actual arrays, and thus would not be very useful in an interactive tool, even if the correspondences between actual arrays and memory suites are provided to the user.

- Another method is to proceed as usual whenever aliasing is not a problem. When two regions concerning two aliased (possibly or certainly, depending on the approximation) variables must be combined, then each region is translated into a region concerning the other variable using techniques described in Chapter 11, and the operation can be performed as usual.

  The advantage of this technique is that linearization is not systematically required (see details in Chapter 11). For instance if two two-dimensional arrays `A` and `B` are totally aliased, then references `A(I,J)` and `B(I,J)` do not need to be linearized to check that they refer to the same memory units. Moreover, the method is adapted to the languages in which the aliasing is not necessarily exactly known at compile time. The last advantage is that the resulting regions correspond to the program variables, and are thereby meaningful to an eventual user of an interactive tool.

  However, a major drawback is the increasing complexity of the analyses. When combining two regions, we must check whether the corresponding variables are aliased or not, test which was avoided with the first method. Moreover, we have seen that to combine two aliased regions we must generate two additional translated regions, and perform two combinations, instead of one if unique memory suites were used. In fact, this is not required for all types of combination. As stated before, unions can be directly performed: Only subtractions and intersection can yield problems.

  Another problem is due to the translation process. Even if aliasing is perfectly known at compile time, it may not be possible to exactly represent the translated regions in the chosen representation.

  > For instance, if array regions are represented by RSDs, and if linearization is required, the resulting region cannot be exactly represented.

Our implementation of array region analyses does not currently deal with aliased variables. Thus IN and OUT regions may be erroneous for procedures with **`equivalent`** statements. We plan to integrate this dimension in a near future, since the translation algorithm which will be presented in Chapter 11 can also be used to translate *equivalent* regions. A consequence is that subsequent analyses must currently tackle this issue. This is coarsely done for instance in PIPS dependence test, *coarsely* meaning that no translation between aliased array regions is done, and that such potential dependences are assumed.

6.8 Regions and Preconditions

Up to now, *preconditions* (see Section 5.6) have not been used to define the semantics of array regions, whatever their type is. However, we have seen in Section 5.2 that they may be useful to compare or merge regions. Moreover, in PIPS (as in other tools, see Section 6.10) preconditions are systematically used to compute over-approximate regions. The natural question which arises and which is discussed in this section is whether using preconditions modifies the nature of our several array region analyses or not.

Using preconditions for array region analyses is in fact composing regions by preconditions. What is the effect of composing $\mathcal{R}\llbracket S\rrbracket$ by $\mathcal{P}\llbracket S\rrbracket$ for instance? $\mathcal{P}\llbracket S\rrbracket$ is a mere filter: It rejects those memory stores which never reach S, converting them into the unknown store. The result of $\mathcal{R}\llbracket S\rrbracket \circ \mathcal{P}\llbracket S\rrbracket\sigma$ in this case is $\perp$. However, if σ cannot be reached, then $\mathcal{R}\llbracket S\rrbracket\sigma = \perp$. And thus

$$\mathcal{R}\llbracket S\rrbracket \circ \mathcal{P}\llbracket S\rrbracket = \mathcal{R}\llbracket S\rrbracket$$

Composing *exact* regions by *preconditions* does not bring any information: It is absolutely unnecessary. This corroborates TRIOLET's intuition [161]. But what about approximations?

When computing approximate array regions, we know nothing about the stores which can reach a particular statement. Array regions must therefore be defined on a very large domain, including stores which will never actually occur. Composing regions by preconditions ($\overline{\mathcal{P}}$) partially alleviates this problem: Preconditions filter out some unreachable stores, and consequently reduce the definition domain of array regions.

> This is what happens in our example of Figure 2.1, in Function `D`. If the relations between `K` and `KP` and between `J` and `JP` are not considered, each reference to array `T` potentially accesses any array element. By taking preconditions into account, we restrain the domain of stores to those in which `J==JP` and `KP==K+1`, and we are able to find the relative positions of the six array references.

Using preconditions may however be harmful for under-approximate array region analyses. Consider for instance the computation of READ regions of a conditional instruction:

$$\begin{aligned}&\underline{\mathcal{R}}_r\llbracket \texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2\rrbracket = \\ &\qquad \underline{\mathcal{R}}_r\llbracket C\rrbracket \,\underline{\cup}\, (\underline{\mathcal{R}}_r\llbracket S_1\rrbracket \,\underline{\circ}\, \underline{\mathcal{E}}_c\llbracket C\rrbracket) \,\underline{\cup}\, (\underline{\mathcal{R}}_r\llbracket S_2\rrbracket \,\underline{\circ}\, \underline{\mathcal{E}}_c\llbracket \texttt{.not.}C\rrbracket) \,\underline{\cup}\, (\underline{\mathcal{R}}_r\llbracket S_1\rrbracket \,\underline{\cap}\, \underline{\mathcal{R}}_r\llbracket S_2\rrbracket)\end{aligned}$$

If we compose $\underline{\mathcal{R}}_r\llbracket S_1\rrbracket$ and $\underline{\mathcal{R}}_r\llbracket S_2\rrbracket$ by their corresponding preconditions, we obtain:

$$\begin{aligned}&\underline{\mathcal{R}}_r\llbracket \texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2\rrbracket = \\ &\qquad \underline{\mathcal{R}}_r\llbracket C\rrbracket \,\underline{\cup}\, (\underline{\mathcal{R}}_r\llbracket S_1\rrbracket \,\underline{\circ}\, \overline{\mathcal{P}}\llbracket S_1\rrbracket \,\underline{\circ}\, \underline{\mathcal{E}}_c\llbracket C\rrbracket) \,\underline{\cup}\, (\underline{\mathcal{R}}_r\llbracket S_2\rrbracket \,\underline{\circ}\, \overline{\mathcal{P}}\llbracket S_2\rrbracket \,\underline{\circ}\, \underline{\mathcal{E}}_c\llbracket \texttt{.not.}C\rrbracket) \,\underline{\cup} \\ &\qquad\qquad (\underline{\mathcal{R}}_r\llbracket S_1\rrbracket \,\underline{\circ}\, \overline{\mathcal{P}}\llbracket S_1\rrbracket \,\underline{\cap}\, \underline{\mathcal{R}}_r\llbracket S_2\rrbracket \,\underline{\circ}\, \overline{\mathcal{P}}\llbracket S_2\rrbracket)\end{aligned}$$

$\overline{\mathcal{P}}[\![S_1]\!]$ may contain the filter associated to condition C and $\overline{\mathcal{P}}[\![S_2]\!]$ the filter associated to condition `.not.`C (see their definition in Table D.5). Thus, if they filter memory stores exactly as the exact condition evaluation does (which is the optimality), then:

$$\underline{\mathcal{R}}_r[\![S_1]\!] \underline{\circ} \overline{\mathcal{P}}[\![S_1]\!] \underline{\cap} \underline{\mathcal{R}}_r[\![S_2]\!] \underline{\circ} \overline{\mathcal{P}}[\![S_2]\!] = \varnothing$$

It means that the last term of the equation does not play its role anymore. This is not cumbersome if $\underline{\cup} \equiv \cup$, because in this case the result is optimal. However, in many parallelizers, $\cup$ is under-approximated by $\cap$ (see Sections 6.10 and 4.2.3), and we have seen that the result may be the empty set in many cases.

So, composing regions with preconditions does not change their nature. And though not strictly necessary on a semantic ground, it *may* help over-approximate analyses, but is discouraged for under-approximations if $\cup$ is under-approximated by $\cap$. In PIPS we systematically compose our over-approximate array regions with preconditions. We do not explicitly compute under-approximations (more details are given in Chapter 6), but we instead use an exactness criterion. However, since the exactness criterion for the `if` construct is built from the definition for the under-approximation, it may fails in the situation described in the previous paragraph. This drawback had been reported in [17]. One solution would be to use preconditions only when they are necessary. In addition, it could decrease the complexity of analyses. But a criterion to test whether it is necessary to use preconditions or not would have to be defined.

6.9 Computation Ordering

Regions, transformers, and continuation conditions whose semantics has been previously defined can be seen as attributes of $\mathcal{L}$ grammar [131]: Transformers, continuation conditions, READ, WRITE and IN regions being synthesized attributes, and OUT regions inherited attributes. For a sequence of instruction, the dependences between the computation of these several attributes are displayed on Figure 6.5. We can deduce from these dependences that transformers, continuation conditions, READ, WRITE and IN regions can be computed during a same bottom-up traversal of the program abstract syntax tree, whereas OUT regions have to be computed afterwards during a top-down traversal.

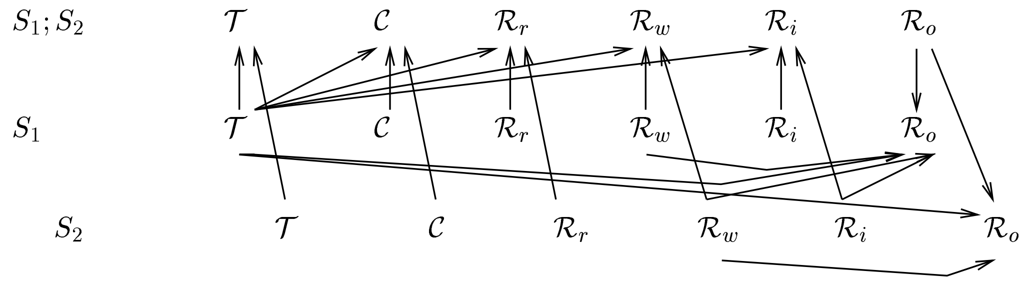


Figure 6.5: Sequence of instructions: Dependences between attributes

In PIPS, phases are independent from each other to allow more flexibility, compatible with a demand-driven approach. Hence, transformers are computed first; then READ and WRITE regions are computed simultaneously because they have the same nature (effects on compound variables). Afterwards, IN regions are independently computed; and finally OUT regions, to respect the previous dependences. In a commercial compiler where compilation time is a choice criterion, all the previous analyses, except OUT regions, should be performed simultaneously, to avoid multiple traversals of the program representation.

6.10 Related Work

Traditionally, data flow analysis techniques were first devoted to scalar variables [117, 82, 113, 1, 153]; accesses to array elements were treated as accesses to the whole array. This was however insufficient to efficiently parallelize many scientific programs dealing with structured data.

The first improvement was the introduction of array data dependence analysis [21]. But this was still insufficient to deal with interprocedural boundaries, and to achieve optimizations such as array expansion or privatization, which require the knowledge of the flow of array elements.

To solve the first problem, a new technique, based on the summarization of the array elements accessed by procedures, was introduced by TRIOLET [161], soon followed by many others [37, 19, 98, 99] who introduced different representations of array element sets.

For the second problem, two approaches were proposed:

- Extensions of classical data flow frameworks to array element sets. Earliest works were due to ROSENE or GROSS and STEENKISTE [151, 86]. This idea encountered a great success because of its ability to deal with any program control flow, and because of its low complexity due to the additional use of summarization techniques. IN and OUT regions belong to this class.

- Extensions of classical dependence analysis techniques to compute *direct* or *value-based* dependences, that is to say dependences which cannot be transitively built from other dependences. The first studies were due to BRANDES [34] and FEAUTRIER [70]. This approach also encountered a frank success for the quality of information it can provide. However, early implementations only dealt with very limited subset languages. Recent extensions lessen this drawback, sometimes at the expense of accuracy.

The next two sections more thoroughly describe these two approaches: Each work is analyzed, and differences with our approach are emphasized. But we restrain the following discussions to the semantics of the information which is computed, and do not consider representation problems at all: They will be tackled in the next part.

6.10.1 Extensions of data flow frameworks

The purpose of data flow analyses is to collect information about programs at compile time. This is usually done by propagating information on the representation of the program, either a control flow graph, its abstract syntax tree, or its call graph. The way the information is propagated is described by data flow equations. This is the case of most of the works described in this section. However, such data flow equations depend on the chosen program representation. Using denotational semantics for outward analyses and attributed grammars for inward analyses, which directly and only depends on the program text, further abstracts the description, delaying implementation details and allowing comparisons between different works. Unfortunately, this approach is seldom used in the field of array data flow analysis. This renders comparisons more difficult; but, we will try in the following discussion to solely focus on the nature of the propagated information, or to formulate data flow equations in our own framework.

One of the most used analysis of scalar variables is the *reaching definition* problem, which gives at each point of the program the reference responsible for the last definition of each variable. It thus provides a convenient way to represent the flow of values. Several studies [151, 86, 110, 85] proposed a direct extension of this type of analyses to compute the flow of individual array element values. The advantage of this approach is that it gives the precise program statement responsible for each value; this information may be useful during the code generation phase of the parallelizer, to find a reasonable schedule. However, we think it is too expensive to handle whole programs: Summarization of array element sets defined in multiple statements cannot be done, because the actual references responsible for the definition would be lost.

The second approach, the most commonly used at this date, is to propagate summaries (array regions), to compute different types of information, the most popular being the equivalent for our READ, WRITE and IN regions. The remainder of this section presents several works in this area.

TRIOLET [**161, 163**]

The concept of array region analyses was originally introduced by TRIOLET to represent the effects of procedures upon sets of array elements. This is almost equivalent to our over-approximation of READ and WRITE regions, except that we generalize his approach to statements, by better taking into account the intraprocedural control flow, and the execution context (modification of memory stores modeled by transformers, `if` and `do while` conditions). TRIOLET already uses some context information, given by *preconditions*, that is to say variable values. However, local modifications of variable values are not taken into account. While this is almost equivalent to our approach at the interprocedural level, this is insufficient at the intraprocedural level.

His work also relies heavily on the chosen representation of array regions as convex polyhedra, whereas the semantic equations presented in the current chapter are independent of the representation. Moreover, we have introduced other types of regions (IN and OUT) to represent the flow of array elements, as well as under-approximate regions, and the notion of exactness.

JOUVELOT **[108]**

In his thesis, JOUVELOT defines intraprocedural over-approximate READ and WRITE array regions by giving their denotational semantics. While others had already have the feeling that the context surrounding array references had to be taken into account, he formalizes it, and defines regions as functions from the set of memory stores to array element sets.

We have not yet defined the interprocedural semantics of array regions; so, except for the use of continuation conditions, our definitions of $\overline{\mathcal{R}}_r$ and $\overline{\mathcal{R}}_w$ are very similar to his regions; we have also shown the necessity of using approximate operators and composition laws. In addition, we have also defined their under-approximate counterparts, as well as IN and OUT regions.

HALL ***et al.*** **[94, 93, 92]**

FIAT/Suif includes an intra- and inter-procedural framework for the analysis of array variables. Different types of regions are available:

- $MustWrite$ regions are similar to $\underline{\mathcal{R}}_w$ regions.
- $Write$ regions are similar to $\overline{\mathcal{R}}_w$ regions.
- *Read* regions are similar to $\overline{\mathcal{R}}_r$ regions.
- *ExposedRead* regions also seem to be similar to $\overline{\mathcal{R}}_i$ regions.

However, there are no equivalent for our OUT regions, which are (among other applications) useful for the interprocedural resolution of the *copy-out* or *finalization* problem in array privatization (see Chapter 14).

Approximate equations for array region analyses are provided for general graphs, whereas we only formally defined the semantics of regions for structured constructs. However, when loops contain multiple exits, or when graphs are irreducible, they also take a straightforward approach.

For sequences of complex or simple instructions, their approach for handling context modifications is unclear. Information similar to transformers are actually computed, but their use during region propagation is not specified. The context information which is added to array regions is restricted to loop bounds[5], thus reducing the complexity of subsequent operations; but they do not fully exploit the ability of convex polyhedra to convey symbolic context information.

The meet function chosen for under-approximate write regions ($MustWrite$) is the intersection, which allows them to apply traditional data flow techniques since the chosen representation of array element sets is closed under intersection. Using our notations, their definition for `if` constructs is thus equivalent to:

$$MustWrite[\![\texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] = MustWrite[\![S_1]\!] \underline{\cap}^6 MustWrite[\![S_2]\!]$$

[5]Personal communication from Saman AMARASINGHE and Brian MURPHY.

whereas ours is:

$$\underline{\mathcal{R}}_w[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] = \\ (\underline{\mathcal{R}}_w[\![S_1]\!] \underline{\circ} \mathcal{E}_c[\![C]\!]) \underline{\cup} (\underline{\mathcal{R}}_w[\![S_2]\!] \underline{\circ} \mathcal{E}_c[\![\texttt{.not.}C]\!]) \underline{\cup} (\underline{\mathcal{R}}_w[\![S_1]\!] \underline{\cap} \underline{\mathcal{R}}_w[\![S_2]\!])$$

There are two differences:

1. Their definition is only based on the intersection operator, whereas ours also uses an under-approximation of the union operator; notice that our definition of the exact semantics of regions shows that the actual operation intrinsically is a union.

2. They do not take `if` conditions into account, since it would lead to an empty result in most cases.

Finally, in FIAT/Suif as in PIPS, the case of `do` loops is handled separately, to take their particular semantics into account. It is implemented as an exact projection (see Equation (6.4)): Dummy variables are introduced to replace the actual loop index whenever its elimination would lead to an over-approximation.

Li *et al.*[135, 87]

In the PANORAMA interprocedural compiler, two types of array regions are computed:

- MOD sets are similar to WRITE regions.
- UE sets are similar to IN regions.

But as in Fiat/Suif, there is no equivalent for our OUT regions.

They do not compute under-approximation of WRITE regions, because they only keep exact information: The computed region components (here array dimensions indices, and a guard representing the context) are either exact or unknown. In fact, this is equivalent to having only one representation for both over- and under-approximations, unknown components being interpreted either as $\top$ or $\perp$.

The way variable modifications are dealt with is unclear. A preliminary phase rewrites each array subscript expression in terms of surrounding loop variables, and loop invariant expressions. But what is exactly done with symbolical variables modified by procedure calls for instance is not specified.

One of the strength of array region analyses in the PANORAMA compiler is that it takes `if` conditions into account, thus avoiding the use of the intersection operator to compute under-approximations, and preserving the exact semantics of array regions.

For loops other than `do` loops and unstructured parts of code, they have conservative approaches (use of $\cap$ instead of $\cup$). `do` loops are handled separately to take their semantics into account.

[6] In fact, they use the actual set intersection $\cap$, because in their representation (lists of convex polyhedra), $\cap$ is an exact operator ($\underline{\cap} = \cap$).

TU **and** PADUA **[165, 166, 164]**

In the POLARIS interprocedural compiler, quite different sets are computed:

- $DEF(S)$ contains the variables which have an outward exposed definition in S.
- $USE(S)$ is similar to our IN regions.
- $KILL(S)$ is similar to our WRITE regions.
- $MRD_{\text{in}}(S)$ is the set of variables which are always defined upon entering S, that is to say the *must reaching definition*.
- $MRD_{\text{out}}(S)$ is the set of variables which are always defined upon exiting S.

The last two sets are obviously under-approximate regions, as well as DEF and $KILL$ sets. The USE set is an over-approximation. We do not compute *reaching definitions* to handle the finalization problem which may appear when privatizing arrays, but OUT regions. Both methods are difficult to compare: *Reaching definitions* provide a more precise information than OUT regions, that is to say at a lower granularity; but whether the array elements they refer to are reused afterwards or not is not considered.

Context information are taken into account with a demand-driven approach. The program is first converted into an extended SSA form (*Gated Single Assignment* form or GSA), which includes `if` conditions: Each scalar variable value is represented by a single name. When combining or comparing two regions, if the variables they involve do not match, their values are searched backwards until they can be compared.

In [166] however, `if` conditions were not taken into account while propagating regions: The meet operator chosen to propagate under-approximations was a mere intersection as in [93] (see above). In [164], TU introduces *conditional data flow analysis*, which settles the problem by using *gated functions*: In fact they are regions including context information. The new meet operator (denoted by $\cap_G$), is still called an intersection, whereas it essentially is a union operator. However, what is done when the condition cannot be exactly represented is unclear; in particular, it is not specified whether the actual intersection of regions is also taken into account as in Equation (6.1).

6.10.2 Extensions of dependence analyses techniques

Dependence analysis techniques are based on pair-wise comparisons of individual references, either definitions or uses. Direct or value-based dependence computations are based on the same principle. In addition, the order in which instructions are executed must be taken into account, which is achieved through different methods [34, 70, 147].

The differences between this approach and the former one (extensions of standard data flow analyses), are not easy to isolate. The closer approach certainly is the computation of *reaching definitions* [151, 86, 164]: Direct dependence analyses provide for each individual use its corresponding definition; instead, *reaching definition* analyses annotate each statement with the last defining statement of each variable. Other works are much more difficult to compare, because the nature of the computed information is not the same: On one hand, direct dependences provide very precise information about the flow of array data, but on limited subset languages; on the other hand, summarization methods [87, 93, 65] do not propagate information about the precise statement in which references are performed, but can be applied on real life programs.

The remainder of this section examines in more details the works done in this area. Our main concern are the class of programs which are treated, the accuracy of the computed information, the way the order in which instructions are executed is handled, and how the context (variable values, variable modifications, `if` conditions) is taken into account.

BRANDES [**34**]

BRANDES was one of the first to propose to compute direct dependences, that is to say dependences that cannot be transitively represented by other dependences. The method directly derives from this definition: Usual *true*, *anti* and *output* dependences are first computed; direct dependences are then found by combining the previous dependences, which is equivalent to removing non-direct dependences from them. This implies that the order in which statements are executed has been taken into account to some extent when computing memory dependences.

This method can be applied on programs represented by very general control flow graphs, but has not been extended to handle function calls. On this domain, the solutions are over- and under-approximations of the actual dependences, which are not always computable. The resulting dependences constitute a data flow graph, which can be used for optimizations such as array renaming or expansion (see Chapter 14).

No details are given about the way the context of array references is handled.

FEAUTRIER [**70, 72**]

FEAUTRIER proposes to compute for each use of a scalar or array element its *source function*, which gives its corresponding definitions as a function of its iteration vector. The method considers each potential producer-consumer pair at a time, and computes the source function of the consumer relative to the current producer. The solution is expressed in terms of the lexicographical maximum of an expression depending on the sequencing predicate of the program, and the iteration vectors of the producer and the consumer; it is computed using PIP, a parametric integer programming solver. Then, individual solutions are combined into a single decision tree (or *quast*); precedence constraints between statements are taken into account in the meantime, thus eliminating non-direct dependences by removing output dependences. Techniques to reduce the complexity of this approach have been presented by others:

MAYDAN et al. *[130]* propose more efficient solutions on a restricted domain, and switch to the original algorithm only when necessary.

MASLOV *[127]* does not consider all possible producer-consumer pairs. Instead, his method consists in recording the instances of the current instruction whose sources have already been found; the process stops when all sources have been found. To further reduce the cost, matching producers are considered in reverse lexicographical order; this relies on the assumption that definitions are usually performed near the corresponding uses.

The purpose, and advantage, of FEAUTRIER's approach is to give an exact answer to the problem of source functions, and thus to exactly describe the flow of compound variables. This enables powerful program optimizations such as array expansion with initialization and finalization (or copy-in and copy-out, see Chapter 14), or more recently the detection of scans and reductions [150].

However, the initial studies [70, 72] put overwhelming restrictions on the input language, which has to be a monoprocedural *static control* program: The only control structures of the language are the sequence, and FORTRAN-like `do` loops, whose limits are linear expressions in the surrounding loop variables and in the program structure parameters (symbolic constants); array indices must also be affine functions of the loop variables and structure parameters. Some of the intra-procedural restrictions have since been partially removed, sometimes at the expense of accuracy:

MASLOV *[127]* extends this method to handle, to some extent, `if` constructs and general variables.

COLLARD *[49, 50]* proposes an extension for `do while` constructs.

COLLARD et al. *[51, 73, 24]* remove some linearity limitations, but at the expense of accuracy.

COLLARD *and* GRIEBL *[52]* finally extend the framework to explicitly (control) parallel programs containing `doall` and `doacross` loops.

Interprocedural extensions have been formulated by LESERVOT: This is described below.

LESERVOT **[121]**

LESERVOT has extended FEAUTRIER's array data flow analysis [72] to handle *generalized instructions* that is to say complex instructions involving definitions and uses of array regions, and such that all uses take place before any definition. The resulting graph is named *Region Flow Graph*, and one of its applications is the extension of FEAUTRIER's method to handle procedure calls by generating an *Interprocedural Array Data Flow Graph*.

To build the *Region Flow Graph*, two abstractions are used:

- *Input effects* give for each array element its first use in the considered fragment of code. This is similar to our IN regions except that the precise statement instance in which the reference is performed is kept in the summary.

- *Output effects* give for each array element its last definition in the considered fragment of code (the solution is ϵ whenever there is no definition). This is equivalent to a local *reaching definition* analysis, and is performed by searching for the *source* of every array element in the current code fragment.

The generalized instruction can then be manipulated as a single instruction using FEAUTRIER's approach, except that *array regions* are manipulated instead of single array elements.

The source program must be a static control program, extended to procedure calls (see Section 11.4); It remains a strong assumption for interprocedural real life programs. LESERVOT however claims that his approach could be extended to handle more general programs by approximating array regions.

PUGH **and** WONNACOTT **[146, 147, 176]**

Last but not the least, PUGH and WONNACOTT have built a framework to compute value-based dependences. It is essentially based on the same equations as BRANDES [34], but is formulated using dependence relations, and can handle many more cases.

It is quite different from FEAUTRIER's approach. First, instead of using a lexicographical maximum, they express the ordering constraints using negated existentially quantified predicates. Second, they take output dependences into account (*array kills*), to reduce the number of negated terms in the complete dependence relation; while FEAUTRIER's method handles this problem when building the final *quast*, that is to say after each producer-consumer pair has been investigated.

Another difference lies in the fact that some non-linear constraints (for instance in `if` conditions) and array subscripts can be handled, either by replacing them by functions in dependence relations, or by *unknown* variables. Such variables can be used to represent non-exact sets, that is to say over- or under-approximations, depending on the logical operator used to link them with the dependence relation.

> For instance, to represent an access to the array element `A(I)` under a non-linear constraint, two sets can be used:
>
> $$\{\phi_1 : \phi_1 = \mathtt{I} \wedge unknown\}$$
>
> represents a set of array elements over-approximated by $\phi_1 = \mathtt{I}$. And
>
> $$\{\phi_1 : \phi_1 = \mathtt{I} \vee unknown\}$$
>
> represents a set of array elements under-approximated by $\phi_1 = \mathtt{I}$.

The modification of scalar variables is handled by a preliminary partial evaluation phase. Thus, no analysis similar to transformers is used.

`stop` statements are not taken into account, but `break` instructions in loops are handled by using function symbols whose value is the iteration in which the break is performed. However, no attempt is done to actually represent the conditions under which the `break` is performed, as we do with continuation conditions (see Page 5.5).

And lastly, WONNACOTT proposes an interprocedural extension.

6.11 Conclusion

We have already presented our array region analyses in several reports or papers [16, 18, 17, 63, 62, 60, 65]. However, the present chapter is new in several respects.

First, array regions are presented using denotational semantics or attributed grammars. It provides an abstraction from both the actual program representation (AST, control flow graph, ...) and from array region representation (convex polyhedra, RDS's, ...). In previous papers, the semantic equations we presented, though quite general, were not totally independent from the underlying representation of array regions as convex polyhedra. In particular, we directly used Equation (6.4) for the definition of the READ regions of `do` loops, assuming that the projection operator was available in the chosen representation.

The formalism we use to define the semantics of our analyses is very convenient to compare several approaches, as done in Section 6.10. Furthermore, approximations can most of the time be directly derived from the exact semantics using approximate operators as shown in Chapter 4, therefore ensuring their correctness, and preserving the intrinsic nature of involved operators. We have seen with `if` constructs how intuitive approaches could lead to use intersection operators instead of unions. Another interest is that sources of approximations are clearly identified, which can be useful to improve a particular implementation.

Second, in previous studies we did not distinguish over-approximate from under-approximate analyses, and instead directly described *exact* analyses: They are in fact over-approximate analyses, whose results are flagged as exact whenever possible, depending on exactness criteria such as introduced in Chapter 4. In this framework, under-approximations were either exact or empty. We now clearly distinguish over- from under-approximate semantics.

We have also clarified, and presented in a uniform framework the relations between array regions and the execution context. Following JOUVELOT [108], we define array regions as functions from the set of memory stores to the powerset of $\mathbb{Z}^n$. This does not preclude any representation: RSD's for instance are constant functions; and our representation consists in convex polyhedra parameterized by the program variables. At the level of semantic functions, taking the context into account involves using the results of several preliminary analyses:

Expression evaluation to evaluate array subscript expressions, and to take `if` conditions into account.

Transformers to model the effect of statements on memory stores, that is to say on variable values.

Continuation conditions which are shown to be necessary to compute safe under-approximations when `stop` statements are allowed by the language. This is a unique feature of our work: To our knowledge, no other study has been published on the impact of possible `stop` statements on array region analyses. WONNACOTT [176] handles `break` statements by adding *function symbols* representing the iterations in which the `break` occurs. But he does not actually represent the conditions under which it happens.

Preconditions have been shown to be unnecessary on a semantic ground, but can be an implementation issue: Adding them may be helpful when using particular representations (see [103, 61]), and harmful when computing under-approximate regions of `if` constructs (see Section 6.10). An open issue is whether it could be possible and interesting to use them only *when it is necessary*, a criterion which has to be defined.

READ, WRITE, IN and OUT region analyses are implemented in PIPS, with some restrictions. The current implementation does not handle `do while` constructs, and continuation conditions are being implemented. Also, aliased variables are not taken into account.

This part was related to the theoretical background of intraprocedural array region analyses. Part III deals with implementation details. The representation chosen in PIPS is presented, as well as the operators to manipulate this representation. Then Part IV deals with interprocedural issues of array region analyses.

III

RÉGIONS POLYÉDRIQUES

ARRAY REGIONS AS CONVEX POLYHEDRA

Chapitre 7

Implémentation dans PIPS : Régions Convexes

(Résumé du chapitre 8)

Au cours de ces dix dernières années, de nombreuses études ont été consacrées à la représentation des ensembles d'éléments de tableaux propagés durant les analyses de régions. La complexité en temps et en espace mémoire, ainsi que la précision des résultats, en dépendent ; et tout l'art consiste à trouver un équilibre satisfaisant entre ces paramètres. Dans PIPS, il a été choisi d'utiliser la convexité pour réduire la demande en espace mémoire, tout en conservant une bonne précision : les régions de tableaux sont donc représentées par des polyèdres convexes paramétrés, et des résumés sont systématiquement construits à chaque étape de l'analyse.

Comme nous l'avons vu dans la partie précédente, les analyses de régions reposent sur plusieurs opérateurs : sur- et sous-approximations des opérateurs ensemblistes (union, intersection, différence) ; approximations de points fixes ; mais aussi lois de composition avec d'autres analyses. L'instanciation de ces opérateurs avait été laissée en suspend jusqu'au choix effectif de la représentation des régions. Seules quelques propriétés avaient été imposées pour garantir la correction des analyses. Nous devrons donc nous assurer que les opérateurs que nous allons définir vérifient bien ces propriétés.

Les sous-approximations sont nécessaires dès lors que des soustractions apparaissent dans les fonctions sémantiques, comme cela est le cas pour les régions IN et OUT. Cependant, nous avons vu les problèmes posés par de telles analyses lorsque le domaine choisi n'est pas clos pour l'union ensembliste, ce qui est le cas pour les ensembles convexes : on ne peut pas définir une meilleure solution pour leurs fonctions sémantiques. Pour conserver une bonne précision, nous préconisons donc de calculer une sur-approximation, et de tester son exactitude ; si elle est exacte, c'est une sous-approximation valide ; sinon plusieurs sous-approximations non comparables sont possibles ; dans PIPS nous avons choisi l'ensemble vide pour ne pas accroître la complexité.

Il faut donc définir un critère d'exactitude calculable pour nos analyses de régions. Toutefois, au lieu d'en fournir un pour chaque fonction sémantique, nous avons trouvé plus pratique de le faire au niveau des opérateurs, sauf cas particulier[1]. Les deux approches ne sont pas totalement équivalentes car certains opérateurs, comme l'union,

[1]Lorsque les approximations ne sont pas directement dérivées de la sémantique exacte, comme pour les instructions conditionnelles.

ne sont pas associatifs. Mais cela permet de réutiliser les tests d'exactitude d'une analyse à l'autre, et cet avantage dépasse les inconvénients précédents.

L'organisation de ce chapitre est la suivante. Nous donnerons tout d'abord quelques rappels sur les polyèdres convexes. La section 7.2 définira ensuite notre représentation des régions de tableaux. Les opérateurs internes seront alors décrits dans la section 7.3, et les lois de composition externes dans la section 7.4.

7.1 Quelques Rappels

7.1.1 Définitions

Polyèdres convexes

Définition 7.1
Un *polyèdre convexe en nombres entiers* est un ensemble de points de $\mathbb{Z}^n$ défini par :

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

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

P peut être vu comme l'intersection des demi-espaces définis par ses inégalités. Un polyèdre peut aussi être représenté par son *système générateur* [9, 59].

Définition 7.2
Soit P un polyèdre convexe. Son *système générateur* est constitué de trois ensembles : un ensemble de sommets $V = \{v_1, \ldots, v_\sigma\}$, un ensemble de rayons $R = \{r_1, \ldots, r_\rho\}$, et un ensemble de droites $L = \{l_1, \ldots, l_\delta\}$.

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

$$\begin{cases} v = \sum_{i=1}^{p} \nu_i x_i \\ \forall\, i \in [1,p] \;\; x_i \in P \\ \forall\, i \in [1,p] \;\; \nu_i \geq 0 \\ \forall\, i,j \in [1,p] \;\; i \neq j \Rightarrow x_i \neq x_j \\ \sum_{i=1}^{p} \nu_i = 1 \end{cases} \qquad \Longrightarrow \quad (\forall\, i \in [1,p],\; \nu_i = 0 \text{ ou } v = x_i)$$

- Un *rayon* est un vecteur $r \in \mathbb{R}^n$ tel qu'il existe une demi-droite parallèle à r, et entièrement contenue dans P:

$$\forall\, x \in P,\; \forall\, \rho \in \mathbb{Z}^+,\; x + \rho r \in P$$

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

$$\forall\, x \in P \quad \forall\, \lambda \in Z \quad x + \lambda l \in P$$

P est alors défini par :

$$P = \left\{ x : \left| \begin{array}{l} \exists\, \nu_1, \dots, \nu_\alpha \in [0,1] \\ \sum_{i=1}^{\alpha} \nu_i = 1 \\ \exists\, \rho_1, \dots, \rho_\beta \in \mathbb{Z}^+ \\ \exists\, \lambda_1, \dots, \lambda_\delta \in \mathbb{Z} \end{array} \right. \quad \text{et } x = \sum_{i=1}^{\alpha} \nu_i v_i + \sum_{i=1}^{\delta} \rho_i r_i + \sum_{i=1}^{\delta} \lambda_i l_i \right\}$$

Ces deux représentations sont équivalentes, et des algorithmes permettent de passer de l'une à l'autre [90, 74]. Certaines opérations comme l'intersection sont plus faciles à réaliser sur les systèmes d'inégalités, d'autres, comme l'enveloppe convexe, sur les systèmes générateurs.

Formes normales disjonctives et formes conjoncto-négatives

Le choix d'une représentation convexe n'empêche pas d'utiliser des intermédiaires de calculs plus complexes si nécessaires, comme les formules de PRESBURGER sans quantificateur universel. Une formule de PRESBURGER est un prédicat du premier ordre sur $\mathbb{Z}$. LESERVOT [121] a montré que deux représentations équivalentes peuvent être utilisées : les formes normales disjonctives (FND) et les formes conjoncto-négatives (FCN).

Propriété 7.1
Toute formule de PRESBURGER F *peut se réécrire en une* FND*, c'est-à-dire en une disjonction de polyèdres convexes :*

$$F = \{x \in \mathbb{Z}^m : \bigvee_i P_i(x)\}$$

où $P_i(x)$ est un polyèdre convexe de $\mathbb{Z}^m$.

Propriété 7.2
Toute formule de PRESBURGER F *peut se réécrire en une* FCN*, qui est la conjonction d'un polyèdre convexe et de plusieurs complémentaires de polyèdres :*

$$F = \{x \in \mathbb{Z}^m : P_o(x) \wedge \ (\bigwedge_i \neg P_i(x)\}$$

où $P_i(x)$ est un polyèdre convexe de $\mathbb{Z}^m$.

La différence entre ces deux représentations est illustrée dans la figure7.1. LESERVOT a proposé un algorithme efficace pour générer la FND la plus courte à partir d'une FCN. Cet algorithme est aussi un test de faisabilité efficace (voir [121]).

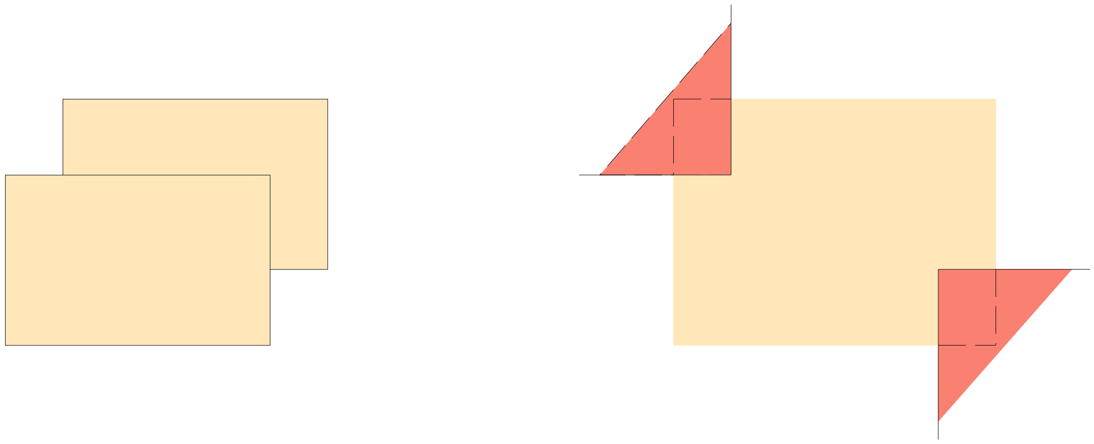


Figure 7.1: FND et FCN

7.1.2 Opérateurs sur les polyèdres convexes

Les polyèdres convexes sont des ensembles de points, et l'on peut donc effectuer sur eux la plupart des opérations ensemblistes habituelles : existence d'un élément (ou faisabilité), union, intersection, différence, projection. Cependant, certains d'entre eux ne préservent pas la convexité, et doivent donc être remplacés par des approximations.

Test de faisabilité Un polyèdre est dit faisable lorsqu'il contient au moins un élément dans $\mathbb{Z}$. Il existe plusieurs algorithmes pour effectuer ces tests. Dans PIPS, nous utilisons soit la méthode d'élimination de FOURIER-MOTZKIN, soit une méthode basée sur le simplexe lorsque le nombre de contraintes du système devient trop important, ou lorsqu'elles sont très liées. Ce sont des tests approchés.

Élimination d'une variable Nous avons vu dans la partie précédente que les régions sont des fonctions de l'état mémoire, et qu'elles doivent être traduites dans le même état mémoire avant de pouvoir être combinées. Cette opération implique l'élimination des variables de l'état mémoire d'origine. De même, le calcul des régions d'une boucle nécessite une projection selon l'indice de boucle, c'est-à-dire l'élimination de celui-ci.

L'algorithme de FOURIER-MOTZKIN permet d'effectuer cette élimination. En réalité, il calcule l'enveloppe convexe de la projection des points entiers du polyèdre de départ. Le résultat est donc une sur-approximation de l'ensemble recherché. ANCOURT [12] et PUGH [144] ont proposé une condition nécessaire et suffisante pour tester l'exactitude de cette opération (théorème 8.1).

Intersection L'intersection de deux polyèdres convexes est un polyèdre convexe. Un polyèdre convexe est l'intersection de plusieurs demi-espaces. L'intersection de deux polyèdres convexes est donc l'intersection de tous les demi-espaces qui les composent.

Union et enveloppe convexe L'union de deux polyèdres convexes n'est pas nécessairement un polyèdre convexe. La plus petite sur-approximation de l'union de deux polyèdres convexes est leur *enveloppe entière*, qui peut-être remplacée par leur enveloppe convexe dans $\mathbb{Q}^n$. Cette opération est plus facile à effectuer à partir des systèmes générateurs.

Propriété 7.3
Soient P_1 et P_2 deux polyèdres convexes, de systèmes générateurs :

$$\begin{cases} V_1 = \{v_1^1, \dots, v_{\alpha_1}^1\} \\ R_1 = \{r_1^1, \dots, r_{\beta_1}^1\} \\ L_1 = \{l_1^1, \dots, l_{\delta_1}^1\} \end{cases} \quad et \quad \begin{cases} V_2 = \{v_1^2, \dots, v_{\alpha_2}^2\} \\ R_2 = \{r_1^2, \dots, r_{\beta_2}^2\} \\ L_2 = \{l_1^2, \dots, l_{\delta_2}^2\} \end{cases}$$

Leur enveloppe convexe, notée convex_hull(P_1, P_2)*, est définie par :*

$$\begin{cases} V = \{v_1^1, \dots, v_{\alpha_1}^1, v_1^2, \dots, v_{\alpha_2}^2\} \\ R = \{r_1^1, \dots, r_{\beta_1}^1, r_1^2, \dots, r_{\beta_2}^2\} \\ L = \{l_1^1, \dots, l_{\delta_1}^1, l_1^2, \dots, l_{\delta_2}^2\} \end{cases}$$

Pour tester l'exactitude d'une enveloppe convexe, c'est-à-dire son égalité avec l'union, nous utiliserons les FCNs, comme le montre la propriété suivante :

Propriété 7.4
Soient P_1 et P_2 deux polyèdres convexes. Soit $P_c =$ convex_hull(P_1, P_2).
Alors convex_hull$(P_1, P_2) = P_1 \cup P_2$ *ssi $P_c \vee \neg P_1 \vee \neg P_2$ n'est pas faisable.*

Différence La différence de deux polyèdres convexes n'est pas nécessairement un polyèdre convexe. Des algorithmes [9, 121] permettent de représenter de manière exacte cette différence sous la forme d'une liste de polyèdres convexes, ou FND : $P_1 - P_2 = P_1 \wedge \neg P_2$ est une FCN qui peut être convertie en une FND. Nous noterons l'opérateur correspondant $-_{\mathrm{dnf}}$. Une sur-approximation convexe peut alors être obtenue en prenant l'enveloppe convexe de tous les éléments de la FND (notée convex_hull$(P_1 -_{\mathrm{dnf}} P_2)$) ; l'exactitude de cette opération peut être testée de la même manière que pour deux polyèdres.

7.2 Régions Convexes

Intuitivement, une région convexe est une fonction représentée par un *polyèdre convexe paramétré*, les *paramètres* étant les valeurs des variables du programme dans l'état mémoire courant. Plus formellement :

Définition 7.3
Une région convexe $\mathcal{R}$ d'un tableau A de dimension d est une fonction de l'ensemble des états mémoire $\mathbf{\Sigma}$ vers l'ensemble des parties convexes de $\mathbb{Z}^d$, $\wp_{\mathrm{convex}}(\mathbb{Z}^d)$, telle que :

$$\begin{aligned} \mathcal{R} : \mathbf{\Sigma} &\longrightarrow \wp_{\mathrm{convex}}(\mathbb{Z}^d) \\ \sigma &\longrightarrow \{\Phi : r(\Phi, \sigma)\} \end{aligned}$$

et r est représenté par un polyèdre convexe paramétré. ■

Par exemple, la région correspondant à l'instruction `WORK(J,K) = J+K` est représentée par le polyèdre $\{\phi_1 = \sigma(\mathtt{J}), \phi_2 = \sigma(\mathtt{K})\}$. ϕ_1 et ϕ_2 représentent les indices de la première et seconde dimension de `WORK`. Le polyèdre est paramétré par les valeurs de `J` et `K` dans l'état mémoire courant.

Dans la suite de ce chapitre, $\mathcal{R}$, $\overline{\mathcal{R}}$ et $\underline{\mathcal{R}}$ désigneront une région exacte pour une instruction S quelconque ($[\![S]\!]$ est omis), et ses approximations.

7.3 Opérateurs Internes

Plusieurs types d'opérateurs internes sont requis par les analyses de régions de tableaux définies dans la partie II. Ce sont les approximations des opérateurs ensemblistes usuels ($\overline{\cup}$, $\underline{\cup}$, $\overline{\cap}$, $\underline{\cap}$, $\overline{\boxminus}$, $\underline{\boxminus}$), et des unions finies, implantées comme des projections, comme suggéré dans la section 6.3.6. Ces opérateurs sont décrits ci-dessous. Les sur-approximations sont généralement définies directement à partir des opérateurs de base présentés dans la section précédente ; par contre, les sous-approximations nécessitent plus d'attention, à cause du manque de treillis sous-jacent : la plupart d'entre eux sont définis à partir de la sur-approximation correspondante, à l'aide d'un critère d'exactitude. Les preuves de correction sont données dans le chapitre 8.

Union La plus petite sur-approximation convexe de l'union de deux polyèdres convexes est leur enveloppe entière; mais l'on peut sans perte de précision utiliser l'enveloppe convexe rationnelle. Par contre, la plus grande sous-approximation convexe ne peut pas être définie. Nous définissons donc la sous-approximation de l'union de deux régions convexes à l'aide du test d'exactitude fourni section 7.1.2.

Définition 7.4

$$
\begin{array}{l}
\overline{\cup} : (\mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d)) \times (\mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d)) \longrightarrow (\mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d)) \\
\quad \mathcal{R}_1 = \lambda\sigma.\{\Phi : r_1(\Phi,\sigma)\}, \mathcal{R}_2 = \lambda\sigma.\{\Phi : r_2(\Phi,\sigma)\} \\
\qquad \longrightarrow \lambda\sigma.\{\Phi : \text{convex\_hull}(r_1(\Phi,\sigma), r_2(\Phi,\sigma))\} \\
\underline{\cup} : (\mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d)) \times (\mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d)) \longrightarrow (\mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d)) \\
\quad \mathcal{R}_1 = \lambda\sigma.\{\Phi : r_1(\Phi,\sigma)\}, \mathcal{R}_2 = \lambda\sigma.\{\Phi : r_2(\Phi,\sigma)\} \\
\qquad \longrightarrow \begin{array}[t]{ll}
\mathbf{Si} & \forall\sigma, \text{convex\_hull}(r_1(\Phi,\sigma), r_2(\Phi,\sigma)) = r_1(\Phi,\sigma) \cup r_2(\Phi,\sigma) \\
\mathbf{Alors} & \lambda\sigma.\{\Phi : r_1(\Phi,\sigma) \cup r_2(\Phi,\sigma)\} \\
\mathbf{Sinon} & \lambda\sigma.\varnothing
\end{array}
\end{array}
$$

Intersection L'intersection de deux polyèdres convexes étant toujours un polyèdre convexe, nous pouvons prendre $\overline{\cap} = \underline{\cap} = \cap$. Il n'y a pas besoin de critère d'exactitude : cette opération est toujours exacte.

Différence La différence de deux polyèdres convexes n'est pas, en général, un polyèdre convexe. Nous avons vu dans la section 7.1.2 comment calculer une représentation exacte de cette différence sous forme d'une liste de polyèdres, qui peut être surestimée par son enveloppe convexe. La meilleure sous-approximation n'étant pas définie, nous utilisons le critère d'exactitude de l'enveloppe convexe pour fournir une sous-approximation satisfaisante.

Définition 7.5

$$
\begin{array}{l}
\overline{\boxminus} : (\mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d)) \times (\mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d)) \longrightarrow (\mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d)) \\
\quad \mathcal{R}_1 = \lambda\sigma.\{\Phi : r_1(\Phi,\sigma)\}, \mathcal{R}_2 = \lambda\sigma.\{\Phi : r_2(\Phi,\sigma)\} \\
\qquad \longrightarrow \lambda\sigma.\{\Phi : \text{convex\_hull}(r_1(\Phi,\sigma) -_{\text{dnf}} r_2(\Phi,\sigma))\} \\
\underline{\boxminus} : (\mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d)) \times (\mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d)) \longrightarrow (\mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d)) \\
\quad \mathcal{R}_1 = \lambda\sigma.\{\Phi : r_1(\Phi,\sigma)\}, \mathcal{R}_2 = \lambda\sigma.\{\Phi : r_2(\Phi,\sigma)\} \\
\qquad \longrightarrow \begin{array}[t]{ll}
\textbf{Si} & \forall\sigma,\ \text{convex\_hull}(r_1(\Phi,\sigma) -_{\text{dnf}} r_2(\Phi,\sigma)) \\
 & \qquad\qquad\qquad = r_1(\Phi,\sigma) -_{\text{dnf}} r_2(\Phi,\sigma) \\
\textbf{Alors} & \lambda\sigma.\{\Phi : \text{convex\_hull}(r_1(\Phi,\sigma) -_{\text{dnf}} r_2(\Phi,\sigma))\} \\
\textbf{Sinon} & \lambda\sigma.\varnothing
\end{array}
\end{array}
$$

Union sur un intervalle La sémantique des régions d'une boucle `do` est définie à partir d'opérations du type $\bigcup_{k=1}^{k=\mathcal{E}[\![exp]\!]} \mathcal{R}_k$, avec $\mathcal{R}_k = \lambda\sigma.\{\Phi : r(\Phi,\sigma,k)\}$. Dans la pratique, la valeur de $\mathcal{E}[\![\mathtt{n}]\!]$ peut être très grande, voire même inconnue, et le calcul de cette quantité devient trop complexe ou même impossible. Cependant, une union finie sur k est mathématiquement équivalente à une projection le long de k:

$$
\bigcup_{k=1}^{k=\mathcal{E}[\![exp]\!]} \mathcal{R}_k = \lambda\sigma.\,\text{proj}_k\big(\{\Phi : r(\Phi,\sigma,k) \wedge (1 \leq k \leq \mathcal{E}[\![exp]\!])\}\big)
$$

Malheureusement, cette expression n'est pas, en général, représentable de manière exacte. Il y a trois sources d'approximation :

1. la régions $\mathcal{R}_k$ (et donc r), qui n'est souvent pas calculable, et doit être sur- ou sous-estimée par $\overline{\mathcal{R}}$ ou $\underline{\mathcal{R}}$;
2. $\mathcal{E}[\![exp]\!]$, qui pose le même problème ;
3. et la projection, qui n'est pas toujours représentable par un polyèdre convexe (voir section 7.1.2), et dont nous pouvons définir une meilleure sur-approximation, mais pas une meilleure sous-approximation.

La définition retenue utilise donc un critère d'exactitude pour définir la sous-approximation. Celui-ci peut se résumer par : $\overline{\mathcal{R}}$ est exacte, les bornes de boucle sont affines, et la projection est exacte.

Définition 7.6

$$\overline{\bigcup_{k=1}^{k=\mathcal{E}[\![exp]\!]}} \mathcal{R}_k = \lambda\sigma.\,\overline{\text{proj}}_k\big(\{\Phi : r(\Phi,\sigma,k) \wedge (1 \leq k \leq \overline{\mathcal{E}}[\![exp]\!])\}$$

$$\begin{array}{llll} \underline{\bigcup_{k=1}^{k=\mathcal{E}[\![exp]\!]}} \mathcal{R}_k = & \textbf{Si} & \forall\sigma, & \underline{\mathcal{E}}[\![exp]\!] = \overline{\mathcal{E}}[\![exp]\!] \\ & & & \wedge\, \overline{\text{proj}}_k\big(\{\Phi : r(\Phi,\sigma,k) \wedge (1 \leq k \leq \underline{\mathcal{E}}[\![exp]\!])\}\big) \\ & & & \quad = \text{proj}_k\big(\{\Phi : r(\Phi,\sigma,k) \wedge (1 \leq k \leq \underline{\mathcal{E}}[\![exp]\!])\}\big) \\ & \textbf{Alors} & \multicolumn{2}{l}{\lambda\sigma.\,\overline{\text{proj}}_k\big(\{\Phi : r(\Phi,\sigma,k) \wedge (1 \leq k \leq \underline{\mathcal{E}}[\![exp]\!])\}\big)} \\ & \textbf{Sinon} & \multicolumn{2}{l}{\lambda\sigma.\varnothing} \end{array}$$

7.4 Lois de Composition Externes

Les fonctions sémantiques définissant les analyses de régions de tableaux comportent leur composition avec les résultats d'autres analyses. Celles-ci appartiennent à deux classes qui se différencient par leur codomaine : les *filtres*, qui retournent des états mémoire, et les *transformeurs*, qui retournent des ensembles d'états mémoire. Les lois de composition sont de ce fait différentes pour chacune des classes.

Filtres Un filtre est une fonction sémantique qui prend en entrée un état mémoire, et le retourne s'il vérifie une certaine condition, ou retourne l'état mémoire indéfini :

$$\begin{aligned} \mathcal{F} : & \boldsymbol{\Sigma} \longrightarrow \boldsymbol{\Sigma} \\ & \sigma \longrightarrow \textbf{if } f(\sigma) \textbf{ then } \sigma \textbf{ else } \bot \\ \overline{\mathcal{F}} : & \boldsymbol{\Sigma} \longrightarrow \boldsymbol{\Sigma} \\ & \sigma \longrightarrow \textbf{if } \overline{f}(\sigma) \textbf{ then } \sigma \textbf{ else } \bot \\ \underline{\mathcal{F}} : & \boldsymbol{\Sigma} \longrightarrow \boldsymbol{\Sigma} \\ & \sigma \longrightarrow \textbf{if } \underline{f}(\sigma) \textbf{ then } \sigma \textbf{ else } \bot \end{aligned}$$

Dans PIPS, cette condition ($\overline{f}$ ou $\underline{f}$) est représentée par un polyèdre convexe, et l'on peut montrer (voir section 8.4.1) que :

$$\begin{aligned} \overline{\mathcal{R}} \,\overline{\circ}_{\overline{\mathcal{R}\mathcal{F}}}\, \overline{\mathcal{F}}(\sigma) &= \{\Phi : \overline{r}(\Phi,\sigma)\} \,\overline{\cap}\, \{\Phi : \overline{f}(\sigma)\} \\ \underline{\mathcal{R}} \,\underline{\circ}_{\underline{\mathcal{R}\mathcal{F}}}\, \underline{\mathcal{F}}(\sigma) &= \{\Phi : \underline{r}(\Phi,\sigma)\} \,\underline{\cap}\, \{\Phi : \underline{f}(\sigma)\} \end{aligned}$$

Comme dans la représentation choisie $\overline{\cap} = \underline{\cap} = \cap$, cette opération est exacte, et nous avons $\overline{\circ}_{\overline{\mathcal{R}\mathcal{F}}} = \underline{\circ}_{\underline{\mathcal{R}\mathcal{F}}} = \circ$. L'exactitude de la composition ne dépend donc que de l'exactitude des opérandes.

Transformeurs La composition d'une région avec un transformeur[2] est plus difficile à définir, car les transformeurs approchés renvoient des ensembles d'états mémoire.

[2] Ou un transformeur inverse, mais comme ils sont de même nature, nous ne considérons dans cette section que les transformeurs.

De plus, seule la sur-approximation des transformeurs est disponible, ce qui rend plus difficile la définition de la sous-approximation de la loi de composition.

Soit $\mathcal{T}$ une analyse exacte de transformeurs ; soit $\overline{\mathcal{T}}$ sa sur-approximation :

$$\begin{aligned}\mathcal{T} : \mathbf{\Sigma} &\longrightarrow \mathbf{\Sigma} \\ \overline{\mathcal{T}} : \mathbf{\Sigma} &\longrightarrow \tilde{\wp}(\mathbf{\Sigma}) \\ \sigma &\longrightarrow \{\sigma' : \overline{t}(\sigma,\sigma')\}\end{aligned}$$

$\overline{t}$ est tel que : $\forall \sigma \in \mathbf{\Sigma},\ \mathcal{T}(\sigma) = \sigma' \implies \overline{t}(\sigma,\sigma')$, et est représenté dans PIPS par un polyèdre convexe.

Les lois de composition qui nous intéressent sont $\overline{\bullet}_{\overline{\mathcal{R}\mathcal{T}}}$ and $\underline{\bullet}_{\underline{\mathcal{R}}\overline{\mathcal{T}}}$. Dans les sections 3.1.3 et 4.3.3, nous avons déjà proposé des approximations possibles :

$$\overline{\mathcal{R}}\,\overline{\bullet}_{\overline{\mathcal{R}\mathcal{T}}}\,\overline{\mathcal{T}} = \lambda\sigma.\ \overline{\bigcup_{\sigma' \in \overline{\mathcal{T}}(\sigma)}}\ \overline{\mathcal{R}}(\sigma')$$

$$\underline{\mathcal{R}}\,\underline{\bullet}_{\underline{\mathcal{R}}\overline{\mathcal{T}}}\,\overline{\mathcal{T}} = \lambda\sigma.\ \underline{\bigcap_{\sigma' \in \overline{\mathcal{T}}(\sigma)}}\ \underline{\mathcal{R}}(\sigma')$$

La sur-approximation peut se réécrire :

$$\overline{\bigcup_{\sigma' \in \overline{\mathcal{T}}(\sigma)}}\ \overline{\mathcal{R}}(\sigma') = \overline{\mathrm{proj}}_{\sigma'}\big(\{\Phi : \overline{t}(\sigma,\sigma')\} \cap \{\Phi : \overline{r}(\Phi,\sigma')\}\big)$$

Elle se compose de l'intersection de deux polyèdres, et de la sur-approximation d'une projection, qui est aisément calculable (voir section 7.1.2).

La sous-approximation est beaucoup plus complexe à définir. Nous renvoyons donc le lecteur au théorème 8.2, page 152, pour plus de détails. En réalité, nous ne calculons jamais de sous-approximation, et la définition de cet opérateur nous permet uniquement de disposer d'un critère d'exactitude (voir propriété 8.8). Ce critère donne de bons résultats comme le démontrent les exemples fournis dans la table 7.1.

7.5 Conclusion

Qui ne rêverait pas d'une représentation idéale des régions de tableaux ? Elle permettrait de représenter tous les motifs d'accès trouvés dans les applications réelles, et l'on disposerait donc d'opérateurs internes et de lois de composition externes précises. Elle permettrait de prendre en compte des informations de contexte symboliques, comme certaines études ont montré que cela était nécessaire [29, 89]. Enfin, la complexité des analyses, aussi bien en terme de temps de calcul que d'occupation de la mémoire, serait très faible ...

Cependant, malgré les très nombreuses études [161, 122, 37, 19, 98, 100, 83, 164, 159, 92, 87, 54, 125] sur le sujet, cette représentation idéale n'a toujours pas été trouvée, et tout l'art consiste à effectuer un compromis entre précision, flexibilité et complexité.

$\underline{\mathcal{R}}$ et $\overline{\mathcal{R}}$	$\overline{\mathcal{T}}$ (k modifié)	Cas du Th. 8.2	$\underline{\mathcal{R}} \underline{\bullet} \overline{\mathcal{T}}$	$\overline{\mathcal{R}} \bar{\bullet} \overline{\mathcal{T}}$	$\mathcal{C}_{\overline{\mathcal{R}} \bar{\bullet} \overline{\mathcal{T}} \equiv \underline{\mathcal{R}} \underline{\bullet} \overline{\mathcal{T}}}$
$\Phi = \sigma'(k)$	$\sigma'(k) = \sigma(k) + 1$	1.1.2	$\Phi = \sigma(k) + 1$[a]	$\Phi = \sigma(k) + 1$	*true*
$\Phi = \sigma'(k)$	$\sigma'(k) \geq \sigma(k)$	1.1.1.1	$\varnothing$[b]	$\Phi \geq \sigma(k)$	*false*
$\Phi = \sigma'(k)$	$\sigma'(k) \geq 5$	1.1.1.2	$\Phi = 5$[c]	$\Phi \geq 5$	*false*
$1 \leq \Phi \leq \sigma'(k)$	$\sigma'(k) \geq 5$	1.1.1.2	$1 \leq \Phi \leq 5$	$1 \leq \Phi$	*false*
$\Phi = \sigma'(k)$		1.1.1.1	$\varnothing$		*false*
$\Phi = \sigma'(i)$	$\sigma'(k) \geq \sigma(k)$	1.2	$\Phi = \sigma(i)$	$\Phi = \sigma(i)$	*true*
$2\Phi = 3\sigma'(k)$	$4\sigma'(k) = 5\sigma(j)$	1.1.2	$8\Phi = 15\sigma(j)$	$8\Phi = 15\sigma(j)$	*true*

[a] $\bar{t}(\sigma, \sigma'_1) \wedge \bar{t}(\sigma, \sigma'_2) \wedge \sigma'_{1/v} \neq \sigma'_{2/v} = \{\sigma'_1(k) = \sigma(k) + 1 \wedge \sigma'_2(k) = \sigma(k) + 1 \wedge \sigma'_1(k) < \sigma'_2(k)\}$ non faisable.
[b] $\underline{\mathcal{R}}(\sigma'_1) \cap \underline{\mathcal{R}}(\sigma'_2 \neq \sigma'_1) = \{\Phi = \sigma'_1(k) \wedge \Phi = \sigma'_2(k) \wedge \sigma'_1(k) < \sigma'_2(k) \wedge \sigma'_1(k) \geq \sigma(k) \wedge \sigma'_2(k) \geq \sigma(k)\} = \varnothing$.
[c] $k_{\min} = 5$, $k_{\max} = \infty$; $\underline{\mathcal{R}}(\sigma' : \sigma'(k) = k_{\min}) \cap \underline{\mathcal{R}}(\sigma' : \sigma'(k) = k_{\max}) = \{\Phi = 5\} \cap \{\} = \{\Phi = 5\}$

Table 7.1: Exemples pour la composition des régions et des transformeurs

Notons tout de même la tendance actuelle aux représentations sous forme de listes de régions compactes (polyèdres convexes, RSDs), et le souci de prendre en compte des informations de contexte de plus en plus complexes.

Dans ce cadre, la représentation que nous avons choisie a l'avantage de limiter la complexité de l'analyse, surtout en occupation mémoire, tout en permettant de représenter une grande variété de formes, et de prendre en compte des informations de contexte — comme les transformeurs, les évaluation d'expression, les conditions de continuation, ou les préconditions — aussi bien pour les sur- que pour les sous-approximations. Par contre, on peut lui reprocher de ne pas permettre de représenter certains motifs d'ensembles d'éléments de tableaux assez fréquents dans les applications réelles, comme les *stencils* à neuf points, les accès aux éléments pairs d'un vecteur, ... Nous prévoyons donc d'implanter d'autres représentations des régions, sous forme de listes de polyèdres ou de Z-polyèdres [9].

Chapter 8

Representation and Operators

Obviously, a lot of efforts have been spent over the last ten years to summarize memory effects on array elements. Time and space complexity, as well as accuracy are the usual issues. In PIPS, we have chosen to use convexity to reduce space complexity, while still preserving accuracy: Array regions are represented by *parameterized convex polyhedra*, and summarization[1] occurs at each step of the propagation.

This propagation involves several operators: Over-approximations of usual set operators ($\overline{\cup}$, $\overline{\cap}$ and $\overline{\boxminus}$); but also their under-approximate counterparts; and of course several composition laws, necessary to combine array regions with other analyses. In Chapter 6, some properties were enforced on all these operators to ensure safe approximations. Hence, correctness proofs could be delayed until the choice of the actual representation of array regions, including the choice of the operators. The latter are built using the basic operators on convex polyhedra which will be presented in a first section.

Under-approximate analyses are necessary as soon as difference operators are involved, which is the case for IN and OUT regions. However, we have seen in Chapter 4 the problems which arise because the representation is not a lattice for the under-approximation: In particular, fixed-points are not defined anymore. The recommended solution is to compute over-approximations and test their exactness using an *exactness criterion*: When the over-approximation is exact, then the best possible under-approximation is equal to the over-approximation; otherwise, several under-approximate solutions are possible, and the solution chosen in PIPS is the empty set of array elements.

Instead of defining a specific exactness criterion for each semantic function, we found it more convenient to define an exactness criterion for each operator: Testing the exactness of a semantic function is then done by successively testing the exactness of each operation it involves[2]. This is not strictly equivalent, because, as will be shown later, some under-approximate operators are not associative, due to the lack of an underlying lattice.

The remainder of this chapter is organized as follows. The first section is a quick description of the basic operators on polyhedra. Section 8.2 then gives our definition of a convex array region and presents the notations used throughout this chapter.

[1] That is to say approximation by a convex polyhedron.

[2] Except for `if` statements, because the under-approximation is not directly built from the exact semantics (see Sections 6.3.4 and 6.5.3)

Section 8.3 describes the internal operators (approximations of $\cup$, $\cap$ and $\boxminus$), while the composition laws are presented in Section 8.4. Section 8.5 describes the array region analysis of the program in Figure 5.2. Other types of representation are finally presented in Section 8.7.

8.1 Polyhedra: Definitions and Basic Operators

The first part of this section is devoted to definitions relative to convex polyhedra over $\mathbb{Z}^n$, disjunctive normal forms, and conjuncto-negative forms [121] which are used in PIPS to perform some operations on convex polyhedra. The next subsection describes the main operators, with an eye to exactness checking.

8.1.1 Definitions

Convex polyhedra

Definition 8.1
A *convex polyhedron over* $\mathbb{Z}$ is a set of points of $\mathbb{Z}^n$ defined by

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

where A is a matrix of $\mathbb{Z}^m \times \mathbb{Z}^n$ and b a vector of $\mathbb{Z}^m$.

P can be viewed as the intersection of the half-spaces defined by its inequalities. A polyhedron can also be represented by a *generating system* [9, 59].

Definition 8.2
Let P be a convex polyhedron. Its *generating system* is made of three sets: a set of vertices $V = \{v_1, \ldots, v_\sigma\}$, a set of rays $R = \{r_1, \ldots, r_\rho\}$, and a set of lines $L = \{l_1, \ldots, l_\delta\}$.

- A *vertex* is a point $v \in P$ which is not a linear combination of other points of P:

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

- A *ray* is a vector $r \in \mathbb{R}^n$ such that there exists an half-line parallel to r, and entirely included in P:

$$\forall\, x \in P,\; \forall\, \rho \in \mathbb{Z}^+,\; x + \rho r \in P$$

- A *line* is a vector $l \in \mathbb{R}^n$ such that l and $-l$ are rays of P:

$$\forall\, x \in P \quad \forall\, \lambda \in \mathbb{Z} \quad x + \lambda l \in P$$

P is then defined by:

$$P = \left\{ x : \left| \begin{array}{l} \exists\, \nu_1, \dots, \nu_\alpha \in [0,1] \\ \sum_{i=1}^{\alpha} \nu_i = 1 \\ \exists\, \rho_1, \dots, \rho_\beta \in \mathbb{Z}^+ \\ \exists\, \lambda_1, \dots, \lambda_\delta \in \mathbb{Z} \end{array} \right. \quad \text{and } x = \sum_{i=1}^{\alpha} \nu_i v_i + \sum_{i=1}^{\delta} \rho_i r_i + \sum_{i=1}^{\delta} \lambda_i l_i \right\}$$

These two representations are equivalent, and algorithms [90, 74] have been designed for the conversion between them. This is not a mere mathematical toy: Some operations on convex polyhedra, such as intersection, perform easier on systems of inequalities, while others are less painful on generating systems.

Disjunctive normal forms and conjuncto-negative forms

The choice of keeping convex representations does not forbid to temporarily use more complex structures when necessary. In particular, we shall use PRESBURGER formulae with no universal quantifiers.

A PRESBURGER formula is a first order predicate over $\mathbb{Z}$. LESERVOT [121] has shown that two equivalent representations can be used: Disjunctive normal forms (DNF) and conjuncto-negative forms (CNF).

Property 8.1
Any PRESBURGER *formula F can be rewritten as a* DNF, *that is to say as a disjunction of convex polyhedra:*

$$F = \{x \in \mathbb{Z}^m : \bigvee_i P_i(x)\}$$

where $P_i(x)$ is a convex polyhedra over $\mathbb{Z}^m$.

Property 8.2
Any PRESBURGER *formula F can be rewritten as a* CNF, *which is the conjunction of one convex polyhedron and several negated polyhedra:*

$$F = \{x \in \mathbb{Z}^m : P_o(x) \wedge \ (\bigwedge_i \neg P_i(x)\}$$

where $P_i(x)$ is a convex polyhedron over $\mathbb{Z}^m$.

The difference between the two representations is highlighted in Figure 8.1. LESERVOT has designed an efficient algorithm to generate the *shortest* DNF from a CNF. As a consequence, a non-feasible CNF is converted into an empty DNF: The algorithm also provides an efficient feasibility test. We refer the reader to [121] for more details. Notice that the first work in this area was due to PUGH and WONNACOTT [147, 176]; in particular, Chapter 11 in WONNACOTT's thesis describes their techniques to simplify PRESBURGER formulae containing negations.

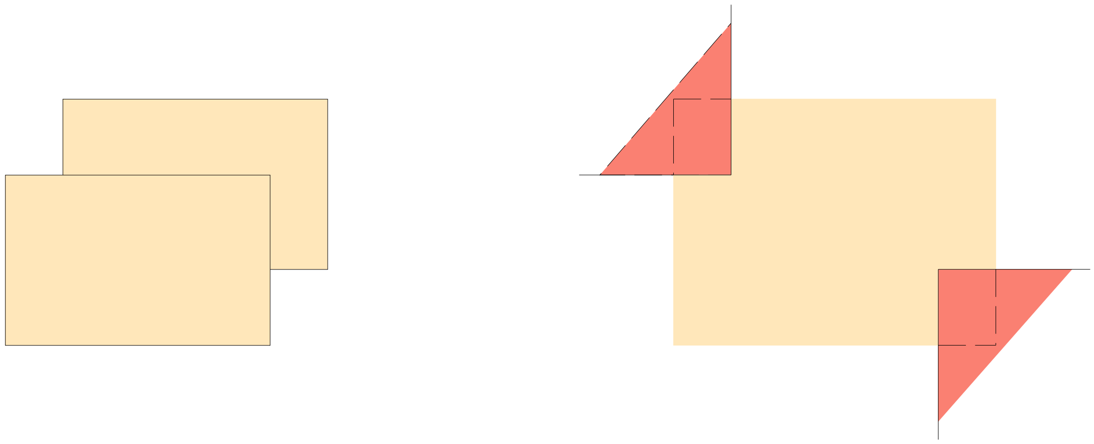


Figure 8.1: DNF vs. CNF representations

8.1.2 Operators on convex polyhedra

Polyhedra are sets of points, and thus support most traditional set operators: Test for emptiness (or feasibility), union, intersection, difference, projection. However, some of them do not preserve convexity, and must be replaced by approximations.

Feasibility

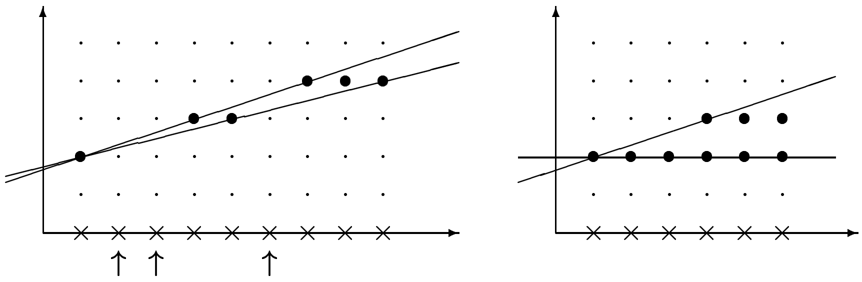
A polyhedron over Z^n is said to be feasible when it contains at least one element in $\mathbb{Z}^n$. There exists a handful of algorithms to perform this test. The first family computes the coordinates of the elements that are solutions of the inequality system of the polyhedron. Exact methods such as the GOMORY's cutting plane method [81] and PIP [71] belong to them, as well as the simplex method which gives solutions in $\mathbb{Q}^n$, and is thus an inexact test. The second family only checks the existence of a solution. The OMEGA test [144] is an exact test in $\mathbb{Z}^n$, unlike the FOURIER-MOTZKIN elimination method [77] which successively eliminates the variables of the system. In PIPS, we only use the FOURIER-MOTZKIN and simplex algorithms.

All these algorithms have an exponential complexity. However, they perform relatively well on usual cases [9]. Moreover, the convex polyhedra used in PIPS contain equalities as well as inequalities, and the feasibility tests we use explicitly deal with equalities, thus reducing the complexity of pair-wise combination of inequalities. In addition, the FOURIER-MOTZKIN algorithm is used only on sufficiently small systems, and a heuristic based on the number of constraints and the average number of variables per constraint is used to switch to the simplex algorithm.

Elimination of a variable

It has been shown in the previous part that array regions are functions of the memory store, and must be converted into the same memory store before being combined. This operation implies the elimination of the variables that belong to the original store. Likewise, the computation of the regions of a loop involves the elimination of the loop index, as described in the next chapter. This is done by projecting the polyhedron representing the region along the variables to eliminate.

The FOURIER-MOTZKIN algorithm [77] performs this elimination. In fact, it computes the convex hull of the projections of all the single integer points that belong to the initial polyhedron. The result is a convex polyhedron that may contain elements that are not the projection of an integer point. This is illustrated in Figure 8.2.



a) non-exact projection *b) exact projection*

— *polyhedron frame.*
• *element of the polyhedron.*
× *element of the projection by Fourier-Motzkin.*
↑ *element that does not belong to the exact projection.*

Figure 8.2: Projection of a polyhedron

ANCOURT [12] and PUGH [144] have proposed a necessary and sufficient condition for testing the exactness of the elimination of a variable between two inequalities.

Theorem 8.1
Let

$$S = \begin{cases} av + A \leq 0 \\ -bv + B \leq 0 \end{cases}$$

be an inequality system in $\mathbb{Z}$, *with* $a \in \mathbb{N}^+$, $b \in \mathbb{N}^+$, $A = c + \sum_{i=1}^{\alpha} a_i v_i$, $B = d + \sum_{i=1}^{\beta} b_i v_i$, *and* $c, d, a_i, b_i \in \mathbb{Z}$.

Let S' *be the system resulting from the elimination of the variable* v *from* S *by* FOURIER-MOTZKIN *algorithm:*

$$S' = \begin{cases} bA + aB \leq 0 \end{cases}$$

S' *has the same solutions as the exact projection of* S *along* v *if and only if the equation:*

$$aB + bA + ab - a - b + 1 \leq 0 \tag{8.1}$$

is redundant with S.

Proof We do not give the proof here, because it is beyond the scope of this thesis. We refer the reader to the previously cited works [12, 144] for more details. □

Note

1. The previous *necessary and sufficient* condition becomes a *sufficient* condition if the redundancy test (which is in fact a feasibility test) is not exact.
2. In practice, we generally test the redundancy of (8.1) with the whole system from which S is extracted. This generally restricts the domain of A and B, and the result is therefore more likely to be exact.
3. A simpler sufficient condition for exactness is $ab - a - b + 1 \leq 0$. It may be useful for quick tests.

Intersection

The intersection of two convex polyhedra is a convex polyhedra. Indeed, a polyhedron is the intersection of half-spaces. Thus, the intersection of two polyhedra is the intersection of all their half-spaces.

Property 8.3

$$\begin{cases} P_1 = \{x \mid A_1.x \leq b_1\} \\ P_2 = \{x \mid A_2.x \leq b_2\} \end{cases} \Longrightarrow \quad P_1 \cap P_2 = \left\{ x \mid \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} .x \leq \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \right\}$$

Union and convex hull

The union of two convex polyhedra is not necessarily a convex polyhedron. Figure 8.3 is an illustration of this property: The points in the initial polyhedra are represented by • and +, and their union is obviously non-convex.

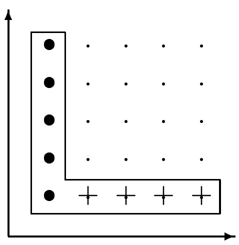


Figure 8.3: Union of two polyhedra

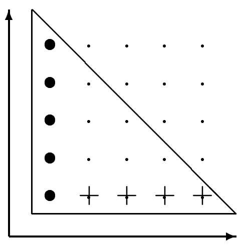


Figure 8.4: Convex hull of two polyhedra

The smallest convex over-approximation of the union of two polyhedra is their *integer hull*. As shown on Figure 8.4, it may contain points that do not belong to the original polyhedra. Notice that the convex hull in $\mathbb{Q}^n$ of two convex polyhedra is easier to perform and contains no additionnal point. This operation can be easily performed on the generating systems.

Property 8.4
Let P_1 and P_2 be two convex polyhedra whose generating systems are:

$$\left\{\begin{array}{l} V_1 = \{v_1^1, \ldots, v_{\alpha_1}^1\} \\ R_1 = \{r_1^1, \ldots, r_{\beta_1}^1\} \\ L_1 = \{l_1^1, \ldots, l_{\delta_1}^1\} \end{array}\right. \quad \textit{and} \quad \left\{\begin{array}{l} V_2 = \{v_1^2, \ldots, v_{\alpha_2}^2\} \\ R_2 = \{r_1^2, \ldots, r_{\beta_2}^2\} \\ L_2 = \{l_1^2, \ldots, l_{\delta_2}^2\} \end{array}\right.$$

then convex_hull(P_1, P_2) *is defined by the following generating system:*

$$\left\{\begin{array}{l} V = \{v_1^1, \ldots, v_{\alpha_1}^1, v_1^2, \ldots, v_{\alpha_2}^2\} \\ R = \{r_1^1, \ldots, r_{\beta_1}^1, r_1^2, \ldots, r_{\beta_2}^2\} \\ L = \{l_1^1, \ldots, l_{\delta_1}^1, l_1^2, \ldots, l_{\delta_2}^2\} \end{array}\right.$$

■

Since we are interested in exactness properties, it will be necessary to know whether convex_hull$(P_1, P_2) = P_1 \bigcup P_2$ or not. This is achieved by using CNFs as shown in the following property.

Property 8.5
Let P_1 and P_2 be two convex polyhedra. Let $P_c =$ convex_hull(P_1, P_2).
Then convex_hull$(P_1, P_2) = P_1 \bigcup P_2$ *if and only if $P_c \vee \neg P_1 \vee \neg P_2$ is not feasible.*

Proof $P_c \vee \neg P_1 \vee \neg P_2 = P_C \vee \neg(P_1 \wedge P_2)$. Then

$$P_c \vee \neg P_1 \vee \neg P_2 \text{ not feasible} \implies P_c \subseteq P_1 \wedge P_2$$

Now, by definition of convex hulls, $P_1 \wedge P_2 \subseteq P_c$. Thus $P_c = P_1 \wedge P_2 = P_1 \bigcup P_2$.
On the contrary,

$$P_c \vee \neg P_1 \vee \neg P_2 \text{ feasible} \implies P_c \not\subseteq P_1 \wedge P_2 \implies P_c \neq P_1 \wedge P_2$$

□

Difference

The difference of two convex polyhedra is not necessarily a convex polyhedron. This operation is not even straightforward. Algorithms [9, 147, 176, 121] have been designed that generate a list of polyhedra, or DNF, that exactly represents the difference. The method used in PIPS relies on CNF and DNF. It is based on the fact that $P_1 - P_2 = P_1 \wedge \neg P_2$, which is a CNF. Converting it into a DNF yields the desired result. We denote the corresponding operator $-_{\text{dnf}}$. An over-approximation is obtained by computing the convex hull of the resulting polyhedra (denoted as convex_hull$(P_1 -_{\text{dnf}} P_2)$), whose exactness can be checked as previously.

8.2 What is a Convex Array Region?

Intuitively, a convex array region is a function which can be represented by a *parameterized convex polyhedron*, the *parameters* being the program variable values in the current memory store. More formally:

Definition 8.3
A convex region $\mathcal{R}$ for an array A of dimension d is a function from the set of memory stores $\mathbf{\Sigma}$ to the set of convex parts of $\mathbb{Z}^d$, $\wp_{\text{convex}}(\mathbb{Z}^d)$, such that:

$$\begin{aligned} \mathcal{R} : \mathbf{\Sigma} &\longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d) \\ \sigma &\longrightarrow \{\Phi : r(\Phi, \sigma)\} \end{aligned}$$

with $\Phi \in \mathbb{Z}^d$, and r being represented by a parameterized convex polyhedron. ■

For instance, the region corresponding to the statement

```
WORK(J,K) = J + K
```

is represented by the polyhedron $\{\phi_1 = \sigma(\mathtt{J}), \phi_2 = \sigma(\mathtt{K})\}$. ϕ_1 and ϕ_2 represent the indices of the first and second dimensions of `WORK`. The polyhedron is parameterized by the values of `J` and `K` in the current memory store.

To simplify the discussions in the remainder of this chapter, we adopt the following conventions:

- In Part II, array region analyses were presented as functions from statements ($\boldsymbol{S}$) or statements and expressions ($\boldsymbol{S} \oplus \boldsymbol{E}$), to the set of functions from stores to the powerset $\wp(Z^n)$ (or one of its subsets). Exact and approximate array regions, functions from $\mathbf{\Sigma}$ to $\wp(Z^n)$, were thus denoted by $\mathcal{R}[\![S]\!]$, $\overline{\mathcal{R}}[\![S]\!]$ or $\underline{\mathcal{R}}[\![S]\!]$ for a statement S. In this chapter, $[\![S]\!]$ is merely omitted to simplify the notations. Missing $[\![S]\!]$ and corresponding universal quantifiers can be trivially added by the careful reader.

- As already implicitly done in Definition 8.3, we consider that array region analyses only have to deal with a single array. The extension to lists of array regions is trivial, as shown in [17].

- Also, operators are independent of the types of region they combine, either READ, WRITE, IN or OUT. Corresponding subscripts are therefore omitted. Consequently, in the remainder of this chapter, $\mathcal{R}$ denotes an exact array region of any type, for any statement, and $\overline{\mathcal{R}}$ and $\underline{\mathcal{R}}$ are the corresponding over- and under-approximations:

$$\begin{aligned} \mathcal{R} &: \mathbf{\Sigma} \longrightarrow \wp(\mathbb{Z}^d) \\ \overline{\mathcal{R}} &: \mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d) \\ \underline{\mathcal{R}} &: \mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d) \end{aligned}$$

- In examples, regions will be denoted as:

<WORK(ϕ_1,ϕ_2)-W-MAY-{ϕ_1==J, ϕ_2==K}>
<WORK(ϕ_1,ϕ_2)-W-EXACT-{ϕ_1==J, ϕ_2==K}>
<WORK(ϕ_1,ϕ_2)-W-MUST-{ϕ_1==J, ϕ_2==K}>

First, the name of the array, and its corresponding ϕ variables. Then, the type of the region (`R`, `W`, `IN` or `OUT`), and its *approximation*, `MAY`, `MUST` or even `EXACT`[3] when the region has been flagged as exact using an exactness criterion. And finally, the polyhedron defining the region, parameterized by the program variable values in the current memory store. From now on, they will improperly be denoted by the corresponding variable names (`J` instead of $\sigma(\texttt{J})$) when no confusion is possible.

8.3 Internal Operators

We have seen in Chapter 6 that several internal operators are necessary to implement array region analyses. They are approximations of usual set operators: $\overline{\cup}$, $\underline{\cup}$, $\overline{\cap}$, $\underline{\cap}$, $\overline{\boxminus}$, $\underline{\boxminus}$, and approximations of finite unions, implemented as projections as suggested in Section 6.3.6. Each operator is described below. Over-approximations are usually straightforwardly defined from basic operators on polyhedra. Whereas under-approximations require more care, because of the lack of an underlying lattice: Most of them are defined from the corresponding over-approximation using an exactness criterion as shown in Chapter 4.

8.3.1 Union

As stated in Section 8.1.2, the union of two convex polyhedra is not necessarily a convex polyhedron. The smallest convex over-approximation of two convex polyhedra is their integer hull; however, their smallest convex hull in $\mathbb{Q}^n$ is easier to compute, and contains no additional integer point; it will be denoted by convex_hull. On the contrary, the greatest convex under-approximation is not defined in the target domain ($\mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d)$). The solution which is used in PIPS implementation of array regions follows the solution proposed in the proof of Theorem 4.3, and uses the exactness criterion of the convex_hull operator[4] to define $\underline{\cup}$ from $\overline{\cup}$:

Definition 8.4

$$
\begin{array}{l}
\overline{\cup} : (\mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d)) \times (\mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d)) \longrightarrow (\mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d)) \\
\quad \mathcal{R}_1 = \lambda\sigma.\{\Phi : r_1(\Phi,\sigma)\}, \mathcal{R}_2 = \lambda\sigma.\{\Phi : r_2(\Phi,\sigma)\} \\
\qquad \longrightarrow \lambda\sigma.\{\Phi : \text{convex\_hull}(r_1(\Phi,\sigma), r_2(\Phi,\sigma))\} \\
\underline{\cup} : (\mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d)) \times (\mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d)) \longrightarrow (\mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d)) \\
\quad \mathcal{R}_1 = \lambda\sigma.\{\Phi : r_1(\Phi,\sigma)\}, \mathcal{R}_2 = \lambda\sigma.\{\Phi : r_2(\Phi,\sigma)\} \\
\qquad \longrightarrow \begin{array}[t]{ll}
\textbf{If} & \forall\sigma, \text{convex\_hull}(r_1(\Phi,\sigma), r_2(\Phi,\sigma)) = r_1(\Phi,\sigma) \cup r_2(\Phi,\sigma) \\
\textbf{Then} & \lambda\sigma.\{\Phi : r_1(\Phi,\sigma) \cup r_2(\Phi,\sigma)\} \\
\textbf{Else} & \lambda\sigma.\varnothing
\end{array}
\end{array}
$$

[3]Even if `EXACT` cannot be considered as an approximation in the strict meaning of the term!
[4]See Property 8.5.

A more precise under-approximation could have been chosen for the cases when the exactness criterion is not true; for instance $\lambda\sigma.\{\Phi : r_1(\Phi,\sigma) \cap r_2(\Phi,\sigma)\}$, since $\cap$ is an internal operator. But the result would very often be the empty set, and at the expense of a greater complexity.

We must now prove that $\overline{\cup}$ and $\underline{\cup}$ are safe approximations of $\cup$. This is the object of the next property.

Property 8.6
With the above notations,

$$(\underline{\mathcal{R}}_1 \subseteq \mathcal{R}_1 \subseteq \overline{\mathcal{R}}_1 \wedge \underline{\mathcal{R}}_2 \subseteq \mathcal{R}_2 \subseteq \overline{\mathcal{R}}_2) \implies \underline{\mathcal{R}}_1 \underline{\cup} \underline{\mathcal{R}}_2 \subseteq \mathcal{R}_1 \cup \mathcal{R}_2 \subseteq \overline{\mathcal{R}}_1 \overline{\cup} \overline{\mathcal{R}}_2$$

Proof $\mathcal{R}_1 \cup \mathcal{R}_2 \subseteq \overline{\mathcal{R}}_1 \overline{\cup} \overline{\mathcal{R}}_2$ comes from the fact that the convex hull of two convex polyhedra is always an over-approximation of their union (by definition). And $\underline{\mathcal{R}}_1 \underline{\cup} \underline{\mathcal{R}}_2 \subseteq \mathcal{R}_1 \cup \mathcal{R}_2$ comes from the fact that either $\underline{\mathcal{R}}_1 \underline{\cup} \underline{\mathcal{R}}_2 = \overline{\mathcal{R}}_1 \overline{\cup} \overline{\mathcal{R}}_2 = \mathcal{R}_1 \cup \mathcal{R}_2$ or $\underline{\mathcal{R}}_1 \underline{\cup} \underline{\mathcal{R}}_2 = \varnothing$. □

8.3.2 Intersection

We have seen in the previous chapter that the intersection of two convex polyhedra always is a convex polyhedron. Thus $\overline{\cap} = \underline{\cap} = \cap$. There is no need for an *exactness criterion*: This operation is always exact, which means that the exactness of the intersection of two approximate regions only depend on the exactness of the operands.

8.3.3 Difference

As seen in Section 8.1.2, the difference of two convex polyhedra may not be a convex polyhedron. But we can provide an exact representation of the difference as a DNF, that is to say a list of convex polyhedra (this operator is denoted by $-_{\text{dnf}}$). We must then merge the components of the list to obtain a convex representation: By computing their convex hull for the over-approximation; and by checking whether the convex hull is equivalent to the DNF for the under-approximation, as for $\underline{\cup}$. The resulting definition therefore is:

Definition 8.5

$$\begin{aligned}
&\overline{\boxminus} : (\mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d)) \times (\mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d)) \longrightarrow (\mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d)) \\
&\quad \mathcal{R}_1 = \lambda\sigma.\{\Phi : r_1(\Phi,\sigma)\}, \mathcal{R}_2 = \lambda\sigma.\{\Phi : r_2(\Phi,\sigma)\} \\
&\qquad \longrightarrow \lambda\sigma.\{\Phi : \text{convex_hull}(r_1(\Phi,\sigma) -_{\text{dnf}} r_2(\Phi,\sigma))\} \\
&\underline{\boxminus} : (\mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d)) \times (\mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d)) \longrightarrow (\mathbf{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d)) \\
&\quad \mathcal{R}_1 = \lambda\sigma.\{\Phi : r_1(\Phi,\sigma)\}, \mathcal{R}_2 = \lambda\sigma.\{\Phi : r_2(\Phi,\sigma)\} \\
&\qquad \longrightarrow \textbf{If} \quad \forall\sigma,\ \text{convex_hull}(r_1(\Phi,\sigma) -_{\text{dnf}} r_2(\Phi,\sigma)) = r_1(\Phi,\sigma) -_{\text{dnf}} r_2(\Phi,\sigma) \\
&\qquad\quad\ \ \textbf{Then} \quad \lambda\sigma.\{\Phi : \text{convex_hull}(r_1(\Phi,\sigma) -_{\text{dnf}} r_2(\Phi,\sigma))\} \\
&\qquad\quad\ \ \textbf{Else} \quad \lambda\sigma.\varnothing
\end{aligned}$$

We must now prove that $\underline{\boxminus}$ and $\overline{\boxminus}$ are safe approximations of $\boxminus$.

Property 8.7
With the above notations,

$$(\underline{\mathcal{R}}_1 \subseteq \mathcal{R}_1 \subseteq \overline{\mathcal{R}}_1 \wedge \underline{\mathcal{R}}_2 \subseteq \mathcal{R}_2 \subseteq \overline{\mathcal{R}}_2) \implies \underline{\mathcal{R}}_1 \,\underline{\boxminus}\, \underline{\mathcal{R}}_2 \subseteq \mathcal{R}_1 \boxminus \mathcal{R}_2 \subseteq \overline{\mathcal{R}}_1 \,\overline{\boxminus}\, \overline{\mathcal{R}}_2$$

Proof $\mathcal{R}_1 \boxminus \mathcal{R}_2 \subseteq \overline{\mathcal{R}}_1 \,\overline{\boxminus}\, \overline{\mathcal{R}}_2$ comes from the fact that the convex hull of a list of convex polyhedra is always an over-approximation of their union (by definition). And $\underline{\mathcal{R}}_1 \,\underline{\boxminus}\, \underline{\mathcal{R}}_2 \subseteq \mathcal{R}_1 \boxminus \mathcal{R}_2$ comes from the fact that either $\underline{\mathcal{R}}_1 \,\underline{\boxminus}\, \underline{\mathcal{R}}_2 = \overline{\mathcal{R}}_1 \,\overline{\boxminus}\, \overline{\mathcal{R}}_2 = \mathcal{R}_1 \boxminus \mathcal{R}_2$ or $\underline{\mathcal{R}}_1 \,\underline{\boxminus}\, \underline{\mathcal{R}}_2 = \varnothing$. □

8.3.4 Union over a range

The exact semantics of array regions for `do` loops involves operations such as:

$$\bigcup_{k=1}^{k=\mathcal{E}[\![exp]\!]} \mathcal{R}_k$$

with $\mathcal{R}_k = \lambda\sigma.\{\Phi : r(\Phi,\sigma,k)\}$. $\mathcal{E}[\![\mathtt{n}]\!]$ may be a great number in practice (or even unknown), and computing the union would be too complex. However, a finite union over k is mathematically equivalent to a projection along k:

$$\bigcup_{k=1}^{k=\mathcal{E}[\![exp]\!]} \mathcal{R}_k = \lambda\sigma.\,\mathrm{proj}_k\big(\{\Phi : r(\Phi,\sigma,k) \wedge (1 \le k \le \mathcal{E}[\![exp]\!])\}\big)$$

Unfortunately, this cannot be readily used to compute regions, because of the several sources of approximations:

- The first source comes from the operands of the finite union: Each $\mathcal{R}_k$ is approximated either by $\overline{\mathcal{R}}_k$ or $\underline{\mathcal{R}}_k$, which verify:

  $$\forall k,\ \underline{\mathcal{R}}_k \sqsubseteq \mathcal{R}_k \sqsubseteq \overline{\mathcal{R}}_k$$

  If $\mathcal{R}_k = \{\Phi : r(\Phi,\sigma,k)\}$, $\overline{\mathcal{R}}_k = \{\Phi : \overline{r}(\Phi,\sigma,k)\}$, and $\underline{\mathcal{R}}_k = \{\Phi : \underline{r}(\Phi,\sigma,k)\}$ then

  $$\forall k,\Phi,\sigma,\ \ \underline{r}(\Phi,\sigma,k) \implies r(\Phi,\sigma,k) \implies \overline{r}(\Phi,\sigma,k)$$

- The evaluation of expression *exp* may also lead to an approximation, either $\underline{\mathcal{E}}[\![exp]\!]$ or $\overline{\mathcal{E}}[\![exp]\!]$. Because of the ordering between the exact evaluation function and its approximations, we have:

  $$1 \le k \le \underline{\mathcal{E}}[\![exp]\!] \implies 1 \le k \le \mathcal{E}[\![exp]\!] \implies 1 \le k \le \overline{\mathcal{E}}[\![exp]\!]$$

  and thus,

  $$\begin{aligned}&\{\Phi : \underline{r}(\Phi,\sigma,k) \wedge (1 \le k \le \underline{\mathcal{E}}[\![exp]\!])\}\text{[5]}\\ &\qquad \sqsubseteq \{\Phi : r(\Phi,\sigma,k) \wedge (1 \le k \le \mathcal{E}[\![exp]\!])\}\\ &\qquad\qquad \sqsubseteq \{\Phi : \overline{r}(\Phi,\sigma,k) \wedge (1 \le k \le \overline{\mathcal{E}}[\![exp]\!])\}\end{aligned}$$

- Lastly, the projection along k of a convex polyhedron is not necessarily a convex polyhedron. We have seen in Section 8.1 that an over-approximation can be computed using FOURIER-MOTZKIN algorithm:

$$\mathrm{proj}_k\big(\{\Phi : r(\Phi,\sigma,k) \wedge (1 \le k \le \mathcal{E}[\![exp]\!])\}\big)$$
$$\sqsubseteq \overline{\mathrm{proj}}_k\big(\{\Phi : \overline{r}(\Phi,\sigma,k) \wedge (1 \le k \le \overline{\mathcal{E}}[\![exp]\!])\}\big)$$

  For the under-approximation, we just check whether the over-approximation is exact.

Our definition of the approximations of the finite union is thus:

Definition 8.6

$$\overline{\bigcup_{k=1}^{k=\mathcal{E}[\![exp]\!]}} \mathcal{R}_k = \lambda\sigma.\, \overline{\mathrm{proj}}_k\big(\{\Phi : r(\Phi,\sigma,k) \wedge (1 \le k \le \overline{\mathcal{E}}[\![exp]\!])\}$$

$$\underline{\bigcup_{k=1}^{k=\mathcal{E}[\![exp]\!]}} \mathcal{R}_k = \begin{array}{lll} \textbf{If} & \forall\sigma, & \underline{\mathcal{E}}[\![exp]\!] = \overline{\mathcal{E}}[\![exp]\!] \\ & & \wedge\, \overline{\mathrm{proj}}_k\big(\{\Phi : r(\Phi,\sigma,k) \wedge (1 \le k \le \underline{\mathcal{E}}[\![exp]\!])\} \\ & & \quad = \mathrm{proj}_k\big(\{\Phi : r(\Phi,\sigma,k) \wedge (1 \le k \le \underline{\mathcal{E}}[\![exp]\!])\} \\ \textbf{Then} & \multicolumn{2}{l}{\lambda\sigma.\, \overline{\mathrm{proj}}_k\big(\{\Phi : r(\Phi,\sigma,k) \wedge (1 \le k \le \underline{\mathcal{E}}[\![exp]\!])\}} \\ \textbf{Else} & \lambda\sigma.\varnothing & \end{array}$$

Notice that the exactness criterion used to define the under-approximation has two components. It can be summarized as such:

1. The loop bounds must be exactly represented.

2. And the projection must be exact.

A more precise, and still interesting, under-approximation could have been defined as the *dark shadow* of the polyhedron, defined in [144] as the set of integer points which are the projection of an actual integer point in the original polyhedron. In this case, the exactness criterion would be exactly the same, but would not be used to *define* the under-approximation, since the *dark shadow* is equal to the actual projection whenever the above exactness criterion is *true*.

8.4 External Composition Laws

The definitions of the exact and approximate semantics of array regions involved composing them with other analyses of two types:

[5] $\{\Phi : \underline{r}(\Phi,\sigma,k) \wedge (1 \le k \le \underline{\mathcal{E}}[\![exp]\!])\} = \{\Phi : \underline{r}(\Phi,\sigma,k)\} \cap \{\Phi : (1 \le k \le \underline{\mathcal{E}}[\![exp]\!])$. Since the under-approximation of $\cap$ is exact, we do not need to use the under-approximation of $\wedge$.

Filters Such semantic functions, either exact or approximate, take as input a memory store, and return it if certain conditions are met, or return the undefined store otherwise. In PIPS, the condition is always represented by a convex polyhedron. The definition of a filter analysis $\mathcal{F}$ is therefore such as:

$$\begin{aligned}\mathcal{F} : \mathbf{\Sigma} &\longrightarrow \mathbf{\Sigma}^6 \\ \sigma &\longrightarrow \textbf{if } f(\sigma) \textbf{ then } \sigma \textbf{ else } \perp\end{aligned}$$

f being represented by a convex polyhedron over the program variable values in the current memory store.

Three analyses presented in Chapter 5 belong to this class: The characteristic function of the restriction to booleans of the expression evaluation function ($\mathcal{E}_c$), preconditions ($\mathcal{P}$) and continuation conditions ($\mathcal{C}$). And of course their approximations.

Transformers Exact transformers are functions from the set of stores $\mathbf{\Sigma}$ to itself, whereas approximate transformers and inverse transformers are functions from the set of memory stores to the *powerset* of memory stores:

$$\begin{aligned}\overline{\mathcal{T}} : \mathbf{\Sigma} &\longrightarrow \tilde{\wp}(\mathbf{\Sigma}) \\ \sigma &\longrightarrow \{\sigma' : t(\sigma, \sigma')\} \\ \overline{\mathcal{T}}^{-1} : \mathbf{\Sigma} &\longrightarrow \tilde{\wp}(\mathbf{\Sigma}) \\ \sigma &\longrightarrow \{\sigma' : t(\sigma', \sigma)\}\end{aligned}$$

In PIPS, t is represented by a convex polyhedron, which gives the relations existing between the values of variables in the input memory store of the statement, and in the output memory store. The list of modified variables is also provided to avoid adding trivial relations in the representations, that is to say constraints such as $\sigma'(v) = \sigma(v)$.

Exact array regions are functions which take as input a single memory store. They can thus be trivially combined with the result of other exact analyses using the usual composition law $\circ$. For approximations, more work has to be done to ensure safe approximations, and, in the case of transformers, to handle the fact that the image of the analysis no more is a single memory store, but a set of memory stores. The basic properties required for approximate composition laws have been given in Section 4.3.3. This section defines them for the chosen representation, and discusses their correctness.

8.4.1 Composing with filter analyses

Let $\mathcal{F}$ be an exact filter analysis:

$$\begin{aligned}\mathcal{F} : \mathbf{\Sigma} &\longrightarrow \mathbf{\Sigma} \\ \sigma &\longrightarrow \textbf{if } f(\sigma) \textbf{ then } \sigma \textbf{ else } \perp\end{aligned}$$

[6] In this section we also omit the text of the program, that is to say the statement or expression, to which the function corresponds.

and $\overline{\mathcal{F}}$ and $\underline{\mathcal{F}}$ be its over- and under-approximations:

$$\begin{aligned}\overline{\mathcal{F}} : & \mathbf{\Sigma} \longrightarrow \mathbf{\Sigma} \\ & \sigma \longrightarrow \textbf{if } \overline{f}(\sigma) \textbf{ then } \sigma \textbf{ else } \perp \\ \underline{\mathcal{F}} : & \mathbf{\Sigma} \longrightarrow \mathbf{\Sigma} \\ & \sigma \longrightarrow \textbf{if } \underline{f}(\sigma) \textbf{ then } \sigma \textbf{ else } \perp\end{aligned}$$

Let $\mathcal{R}$ be an exact region, and $\overline{\mathcal{R}}$ and $\underline{\mathcal{R}}$ its approximations:

$$\begin{aligned}\mathcal{R} : \mathbf{\Sigma} & \longrightarrow \wp(\mathbb{Z}^d) \\ \sigma & \longrightarrow \{\Phi : r(\Phi, \sigma)\} \\ \overline{\mathcal{R}} : \mathbf{\Sigma} & \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d) \\ \sigma & \longrightarrow \{\Phi : \overline{r}(\Phi, \sigma)\} \\ \underline{\mathcal{R}} : \mathbf{\Sigma} & \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^d) \\ \sigma & \longrightarrow \{\Phi : \underline{r}(\Phi, \sigma)\}\end{aligned}$$

To define the approximate composition laws $\overline{\circ}_{\overline{\mathcal{R}\mathcal{F}}}$ and $\underline{\circ}_{\underline{\mathcal{R}\mathcal{F}}}$, let us first look at the nature of the operation involved by the usual composition law.

$$\begin{aligned}\mathcal{R} \circ \mathcal{F}(\sigma) &= \mathcal{R}(\textbf{if } f(\sigma) \textbf{ then } \sigma \textbf{ else } \perp) \\ &= \textbf{if } f(\sigma) \textbf{ then } \mathcal{R}(\sigma) \textbf{ else } \varnothing \\ &= \{\Phi : r(\Phi, \sigma) \wedge f(\sigma)\} \\ &= \{\Phi : r(\Phi, \sigma)\} \cap \{\Phi : f(\sigma)\}\end{aligned}$$

In the case of filter analyses, $\mathcal{R} \circ \mathcal{F}$ can therefore be approximated by:

$$\begin{aligned}\overline{\mathcal{R}}\, \overline{\circ}_{\overline{\mathcal{R}\mathcal{F}}}\, \overline{\mathcal{F}}(\sigma) &= \{\Phi : \overline{r}(\Phi, \sigma)\}\, \overline{\cap}\, \{\Phi : \overline{f}(\sigma)\} \\ \underline{\mathcal{R}}\, \underline{\circ}_{\underline{\mathcal{R}\mathcal{F}}}\, \underline{\mathcal{F}}(\sigma) &= \{\Phi : \underline{r}(\Phi, \sigma)\}\, \underline{\cap}\, \{\Phi : \underline{f}(\sigma)\}\end{aligned}$$

Since $\overline{\cap} = \underline{\cap} = \cap$, we also have $\overline{\circ}_{\overline{\mathcal{R}\mathcal{F}}} = \underline{\circ}_{\underline{\mathcal{R}\mathcal{F}}} = \circ$, and the exactness of the composition only depends on the exactness of the operands ($\overline{\mathcal{R}} \equiv \underline{\mathcal{R}}$ and $\overline{f} \equiv \underline{f}$).

8.4.2 Composing with transformers

Composing regions with transformers is *a priori* more difficult since approximate transformers (or inverse transformers) give results in $\wp(\mathbf{\Sigma})$ instead of $\mathbf{\Sigma}$. Moreover, since only over-approximate transformers are defined, great care must be brought to the definition of under-approximate composition laws.

Since approximate transformers and inverse transformers have the same mathematical and semantic nature, we only consider in this section transformers. The transposition to inverse transformers is trivial. So, let $\mathcal{T}$ be an exact transformer analysis, and $\overline{\mathcal{T}}$ be its over-approximation:

$$\begin{aligned}\mathcal{T} : \mathbf{\Sigma} & \longrightarrow \mathbf{\Sigma} \\ \overline{\mathcal{T}} : \mathbf{\Sigma} & \longrightarrow \tilde{\wp}(\mathbf{\Sigma}) \\ \sigma & \longrightarrow \{\sigma' : \overline{t}(\sigma, \sigma')\}\end{aligned}$$

$\bar{t}$ is such that: $\forall \sigma \in \mathbf{\Sigma},\ \mathcal{T}(\sigma) = \sigma' \implies \bar{t}(\sigma, \sigma')$, and is represented in PIPS by a convex polyhedron. For regions, we use the same notations as in the previous section.

The composition laws we are interested in are $\overline{\bullet}_{\overline{\mathcal{R}\mathcal{T}}}$ and $\underline{\bullet}_{\underline{\mathcal{R}}\overline{\mathcal{T}}}$. In Section 4.3.3, we already proposed a safe definition for each case, that is to say a definition which preserves the approximation ordering:

$$\overline{\mathcal{R}}\,\overline{\bullet}_{\overline{\mathcal{R}\mathcal{T}}}\,\overline{\mathcal{T}} = \lambda\sigma.\ \overline{\bigcup_{\sigma' \in \overline{\mathcal{T}}(\sigma)}}\ \overline{\mathcal{R}}(\sigma')$$

$$\underline{\mathcal{R}}\,\underline{\bullet}_{\underline{\mathcal{R}}\overline{\mathcal{T}}}\,\overline{\mathcal{T}} = \lambda\sigma.\ \underline{\bigcap_{\sigma' \in \overline{\mathcal{T}}(\sigma)}}\ \underline{\mathcal{R}}(\sigma')$$

Let us first consider the definition of the over-approximate composition law, $\overline{\bullet}_{\overline{\mathcal{R}\mathcal{T}}}$. It is defined using an over-approximate union operator. If this (possibly infinite) union was exact, we would successively have:

$$\begin{aligned}
\bigcup_{\sigma' \in \overline{\mathcal{T}}(\sigma)} \overline{\mathcal{R}}(\sigma') &= \bigcup_{\sigma' \in \overline{\mathcal{T}}(\sigma)} \{\Phi : \overline{f}(\sigma')\} \\
&= \{\Phi : \exists \sigma',\ \bar{t}(\sigma,\sigma') \wedge \overline{r}(\Phi,\sigma')\} \\
&= \mathrm{proj}_{\sigma'}\left(\{\Phi : \bar{t}(\sigma,\sigma') \wedge \overline{r}(\Phi,\sigma')\}\right) \\
&= \mathrm{proj}_{\sigma'}\left(\{\Phi : \bar{t}(\sigma,\sigma')\} \cap \{\Phi : \overline{r}(\Phi,\sigma')\}\right)
\end{aligned}$$

Since $\overline{\cap} = \cap$, and since it is easy to compute an over-approximation of the projection, we can define the over-approximation of the union as:

$$\overline{\bigcup_{\sigma' \in \overline{\mathcal{T}}(\sigma)}}\ \overline{\mathcal{R}}(\sigma') = \overline{\mathrm{proj}}_{\sigma'}\left(\{\Phi : \bar{t}(\sigma,\sigma')\} \cap \{\Phi : \overline{r}(\Phi,\sigma')\}\right)$$

Thus $\overline{\bullet}_{\overline{\mathcal{R}\mathcal{T}}}$ is the combination of an intersection, and the over-approximation of a projection.

To define the under-approximation of the composition law, $\underline{\bullet}_{\underline{\mathcal{R}\mathcal{T}}}$, we cannot use an exactness criterion of the over-approximation, because $\underline{\mathcal{R}}$ is not composed with an under-approximation of $\mathcal{T}$, but with its over-approximation. We must therefore use the particular definition of $\underline{\bullet}_{\underline{\mathcal{R}\mathcal{T}}}$ given before:

$$\underline{\mathcal{R}}\,\underline{\bullet}_{\underline{\mathcal{R}}\overline{\mathcal{T}}}\,\overline{\mathcal{T}} = \lambda\sigma.\ \underline{\bigcap_{\sigma' : \bar{t}(\sigma,\sigma')}}\ \{\Phi : \underline{r}(\Phi,\sigma')\}$$

Notice that the exact intersection cannot be computed in the general case, because the domain on which it is performed may be infinite, or we may not be able to find out that it is finite. The next theorem gives an under-approximation of the exact intersection.

It first checks if the domain on which the intersection is performed is not empty. If it is empty (case 2), then the intersection is empty. In the opposite case, It checks if the variables v which appear in the region polyhedron are modified by the transformer. If none of them is modified (case 1.2), then the region is the same everywhere on the domain, and the polyhedron is therefore invariant. On the contrary, if some variables which appear in $\underline{r}$ are modified by the transformer, we then check if the restriction of the transformer to these variables is invertible. If it is invertible (case 1.1.2), then the domain on which the intersection is performed is restricted to a unique point, which is a simple function of σ. The result is therefore the initial region in which the restriction to v of σ' is replaced by the corresponding function of σ. If the transformer is not invertible, and if for any two element of the domain the intersection is empty, then it is also empty on the whole domain (case 1.1.1.1). Finally (case 1.1.1.2), if the transformer is not invertible on the domain restricted to v, and if the intersection is not trivially empty, we must under-approximate the intersection. This is done by further over-approximating the transformer so that the domain is enlarged to a constant parallelepiped. By convexity of the domain and of the region, the intersection on the enlarged domain is equal to the intersection on the vertices (with possibly infinite coordinates) of the parallelepiped.

Theorem 8.2
With the above notations, and for all $\sigma \in \mathbf{\Sigma}$:

Let v be the minimal subset of $\boldsymbol{V}$ such that $\forall \sigma'$, $\underline{\mathcal{R}}(\sigma') = \underline{\mathcal{R}}(\sigma'_{/v})$ where $\sigma'_{/v}$ is the restriction of σ' to the sub-domain v[7]. Let $\overline{t}_{/v}(\sigma, \sigma'_{/v})$ denote the restriction of $\overline{t}(\sigma, \sigma')$ to $\sigma'_{/v}$.

1. ***If*** $\exists \sigma' : \overline{t}(\sigma, \sigma')$
   Then

   1.1. ***If*** $v \neq \varnothing$
   Then

   1.1.1. ***If*** $\exists \sigma'_1, \sigma'_2 : \overline{t}(\sigma, \sigma'_1) \wedge \overline{t}(\sigma, \sigma'_2) \wedge \sigma'_{1/v} \neq \sigma'_{2/v}$
   Then

   1.1.1.1. ***If*** $\forall \sigma'_1 : \overline{t}(\sigma, \sigma'_1),\ \forall \sigma'_2 : \overline{t}(\sigma, \sigma'_2),$
   $$(\sigma'_{1/v} \neq \sigma'_{2/v}) \implies \underline{\mathcal{R}}(\sigma'_1) \cap \underline{\mathcal{R}}(\sigma'_2) = \varnothing$$
   Then $\displaystyle \bigcap_{\sigma' : \overline{\overline{t}(\sigma, \sigma')}} \underline{\mathcal{R}}(\sigma') = \bigcap_{\sigma' : \overline{t}(\sigma, \sigma')} \underline{\mathcal{R}}(\sigma') = \varnothing$

   1.1.1.2. ***Else***
   Let $\vec{w}$ be the vector of the n variables w^i modified by $\overline{\mathcal{T}}$, and which belong to v. Let $\vec{w}_{min}$ and $\vec{w}_{max}$ in $\mathbb{Z}^n$ be the vectors of the minimum and maximum integer values of all the variables in $\vec{w}$, when σ varies:
   $$\vec{w}_{min} = \max\{\vec{w}'_{min} : (\forall \sigma, \sigma' : \overline{t}(\sigma, \sigma'),\ \vec{w}'_{min} \leq \sigma'(\vec{w}))\}$$
   $$\vec{w}_{max} = \min\{\vec{w}'_{min} : (\forall \sigma, \sigma' : \overline{t}(\sigma, \sigma'),\ \sigma'(\vec{w}) \leq \vec{w}'_{max})\}$$
   Then we have,
   $$\bigcap_{\substack{\sigma' : \forall i \in [1..n], \\ \sigma'(w^i) = w^i_{min} \\ \vee\, \sigma'(w^i) = w^i_{max}}} \{\Phi : \underline{r}(\Phi, \sigma')\} \sqsubseteq \bigcap_{\sigma' : \overline{t}(\sigma, \sigma')} \underline{\mathcal{R}}(\sigma')$$

[7] In other words, v is the set of variables which appear in the polyhedron $\underline{r}(\sigma, \sigma')$.

1.1.2. ***Else*** $\exists ! \sigma'_{0/v} : \bar{t}_{/v}(\sigma, \sigma'_{0/v})$ *and* $\sigma'_{0/v} = f(\sigma)$,

$$\bigcap_{\sigma' : \underline{\bar{t}(\sigma,\sigma')}} \underline{\mathcal{R}}(\sigma') = \bigcap_{\sigma' : \bar{t}(\sigma,\sigma')} \underline{\mathcal{R}}(\sigma') = \underline{\mathcal{R}}(\sigma'_{0/v}) = \{\Phi : \underline{r}(\Phi, f(\sigma'_{0/v}))\}$$

1.2. ***Else*** $\displaystyle\bigcap_{\sigma' : \underline{\bar{t}(\sigma,\sigma')}} \underline{\mathcal{R}}(\sigma') = \bigcap_{\sigma' : \bar{t}(\sigma,\sigma')} \underline{\mathcal{R}}(\sigma') = \underline{\mathcal{R}}(\sigma)$

2. ***Else*** $\displaystyle\bigcap_{\sigma' : \underline{\bar{t}(\sigma,\sigma')}} \underline{\mathcal{R}}(\sigma') = \bigcap_{\sigma' : \bar{t}(\sigma,\sigma')} \underline{\mathcal{R}}(\sigma') = \varnothing$

Proof Cases 2, 1.2, 1.1.2, 1.1.1.1 are trivial (see the explanations preceding the theorem). However, Case 1.1.1.2 deserves a proof.

We want to show that:

$$\bigcap_{\substack{\sigma' : \forall i \in [1..n], \\ \sigma'(w^i) = w^i_{\min} \\ \vee\, \sigma'(w^i) = w^i_{\max}}} \{\Phi : \underline{r}(\Phi, \sigma')\} = \bigcap_{\sigma' : \vec{w}_{\min} \leq \sigma'(\vec{w}) \leq \vec{w}_{\max}} \{\Phi : \underline{r}(\Phi, \sigma')\} \sqsubseteq \bigcap_{\sigma' : \bar{t}(\sigma,\sigma')} \{\Phi : \underline{r}(\Phi, \sigma')\}$$

The *inclusion* directly derives from the definitions of $\vec{w}_{\min}$ and $\vec{w}_{\max}$; they imply that:

$$\forall \sigma' : \bar{t}(\sigma, \sigma'),\ \vec{w}_{\min} \leq \sigma'(\vec{w}) \leq \vec{w}_{\max}$$

Thus the initial domain $\{\sigma' : \bar{t}(\sigma, \sigma')\}$ is included in the domain $\{\sigma' : \vec{w}_{\min} \leq \sigma'(\vec{w}) \leq \vec{w}_{\max}\}$; and the intersection can only be smaller if the domain is enlarged.

The *equality* comes from the convexity the region and the fact that the domain on which the intersection is performed is a parallelepiped. Let us first prove this property on a one-dimensional domain:

$$\bigcap_{a \leq x \leq b} \{\Phi : \underline{r}(\Phi, x)\} = \{\Phi : \underline{r}(\Phi, a)\} \cap \{\Phi : \underline{r}(\Phi, b)\}$$

This is equivalent to showing that

$$\forall x_0 : a \leq x_0 \leq b,\ \{\Phi : \underline{r}(\Phi, a) \wedge \underline{r}(\Phi, b)\} \subseteq \{\Phi : \underline{r}(\Phi, x_0)\}$$

Or that any constraint from $\underline{r}(\Phi, x_0)$ is redundant with a constraint of $\underline{r}(\Phi, a)$ or $\underline{r}(\Phi, b)$.

Let us consider the following constraint from $\underline{r}(\Phi, \sigma')$ (which is a convex polyhedron), where $a_0 \neq 0$:

$$A\Phi + a_0 x + c \leq 0$$

We want to show that

$$A\Phi + a_0 x_0 + c \leq 0 \tag{8.2}$$

is redundant with either

$$A\Phi + a_0 a + c \leq 0 \tag{8.3}$$

or

$$A\Phi + a_0 b + c \leq 0 \tag{8.4}$$

for any x_0 such that $a \leq x_0 \leq b$. To prove that (8.2) is redundant with the system made of the two other inequalities, we add the opposite inequality

$$-A\Phi - a_0 x_0 - c + 1 \leq 0 \tag{8.5}$$

to the system, and we show that there is no solution.

$$\begin{aligned}
(8.5) \wedge (8.3) &\implies a_0 \times (a - x_0) \leq 0 && (8.6)\\
(8.5) \wedge (8.4) &\implies a_0 \times (b - x_0) \leq 0 && (8.7)\\
a \leq x_0 &\implies a - x_0 \leq 0 \implies ((8.6) \implies a_0 \geq 0)\\
x_0 \leq b &\implies b - x_0 \geq 0 \implies ((8.7) \implies a_0 \leq 0)
\end{aligned}$$

Thus the system implies that $a_0 = 0$, which is impossible because of the previous assumption that $a_0 \neq 0$. Therefore, the system is not feasible, and (8.2) is redundant with (8.3) and (8.4).

As a consequence the property on a one-dimensional domain holds. It can be extended to multi-dimensional domains by recursion on the number of dimensions. For two dimensions, we would show that:

$$\bigcap_{a \leq x \leq b} \bigcap_{c \leq y \leq d} \{\Phi : \underline{r}(\Phi, x, y)\} =$$
$$\{\Phi : \underline{r}(\Phi, a, c)\} \cap \{\Phi : \underline{r}(\Phi, a, d)\} \cap \{\Phi : \underline{r}(\Phi, b, c)\} \cap \{\Phi : \underline{r}(\Phi, b, d)\}$$

And for n dimensions:

$$\bigcap_{\substack{\sigma' : \forall i \in [1..n], \\ \sigma'(w^i) = w^i_{\text{min}} \\ \vee\, \sigma'(w^i) = w^i_{\text{max}}}} \{\Phi : \underline{r}(\Phi, \sigma')\} = \bigcap_{\sigma' : \vec{w}_{\text{min}} \leq \sigma'(\vec{w}) \leq \vec{w}_{\text{max}}} \{\Phi : \underline{r}(\Phi, \sigma')\}$$

□

In the previous theorem, the result is exact in several cases (when the transformer represents a sequence containing a `stop` statement, or when it is invertible for instance). But in case (1.1.1.2) the result is a potentially coarse approximation. In fact, the transformer is further over-approximated by its smallest parallelepipedic integer hull. This result could certainly be enhanced. However, our purpose is to check the exactness of the corresponding over-approximated operation. And if the transformer is not invertible on the domain of the variables used in the region description, there is very little chance that the under-approximation given by case (1.1.1.2) be equal to the over-approximation. This leads to the following exactness criterion, whose correctness is ensured by Theorem 8.2:

Property 8.8
Let v be the minimal subset of $\boldsymbol{V}$ such that $\forall\sigma'$, $\underline{\mathcal{R}}(\sigma') = \underline{\mathcal{R}}(\sigma'_{/v})$ where $\sigma'_{/v}$ is the restriction of σ' to the sub-domain v. Let $\overline{t}_{/v}(\sigma, \sigma'_{/v})$ denote the restriction of $\overline{t}(\sigma, \sigma')$ to $\sigma'_{/v}$.

$$
\begin{array}{l}
\mathcal{C}_{\overline{\mathcal{R}}\,\bar{\bullet}\,\overline{\mathcal{T}} \equiv \underline{\mathcal{R}}\,\underline{\bullet}\,\overline{\mathcal{T}}} = \\
\quad \mathcal{C}_{\overline{\mathcal{R}} \equiv \underline{\mathcal{R}}} \wedge \begin{array}[t]{ll}
\textbf{If} & \exists\sigma' : \overline{t}(\sigma, \sigma') \\
\textbf{Then} & \\
& \begin{array}{ll}
\textbf{If} & v \neq \varnothing \\
\textbf{Then} & \\
& \begin{array}{ll}
\textbf{If} & \exists\sigma'_1, \sigma'_2 : \overline{t}(\sigma, \sigma'_1) \wedge \overline{t}(\sigma, \sigma'_2) \wedge \sigma'_{1/v} \neq \sigma'_{2/v} \\
\textbf{Then} & (\overline{\mathcal{R}}\,\bar{\bullet}\,\overline{\mathcal{T}} \equiv \lambda\sigma.\varnothing) \\
\textbf{Else} & \textit{let } \sigma'_{0/v} : \overline{t}_{/v}(\sigma, \sigma'_{0/v}) \\
& \textit{let } f : \sigma'_{0/v} = f(\sigma) \\
& (\overline{\mathcal{R}}\,\bar{\bullet}\,\overline{\mathcal{T}} \equiv \{\Phi : \underline{r}(\Phi, f(\sigma'_{0/v}))\})
\end{array} \\
\textbf{Else} & (\overline{\mathcal{R}}\,\bar{\bullet}\,\overline{\mathcal{T}} \equiv \underline{\mathcal{R}})
\end{array} \\
\textbf{Else} & (\overline{\mathcal{R}}\,\bar{\bullet}\,\overline{\mathcal{T}} \equiv \lambda\sigma.\varnothing)
\end{array}
\end{array}
$$

Some examples of the composition of regions with transformers are provided in Table 8.1. The first column gives the initial (exact) region; the second gives the over-approximate transformer; the results are displayed in Columns 4 and 5; The third column gives the step of Theorem 8.2 which gives the solution in Column 5; the last column shows the results of the exactness criterion as provided in Property 8.8. We can see that, on our examples, it never fails to identify exact results.

$\underline{\mathcal{R}}$ and $\overline{\mathcal{R}}$	$\overline{\mathcal{T}}$ (k modified)	case in Th. 8.2	$\underline{\mathcal{R}}\,\underline{\bullet}\,\overline{\mathcal{T}}$	$\overline{\mathcal{R}}\,\bar{\bullet}\,\overline{\mathcal{T}}$	$\mathcal{C}_{\overline{\mathcal{R}}\,\bar{\bullet}\,\overline{\mathcal{T}} \equiv \underline{\mathcal{R}}\,\underline{\bullet}\,\overline{\mathcal{T}}}$
$\Phi = \sigma'(k)$	$\sigma'(k) = \sigma(k) + 1$	1.1.2	$\Phi = \sigma(k) + 1$[a]	$\Phi = \sigma(k) + 1$	*true*
$\Phi = \sigma'(k)$	$\sigma'(k) \geq \sigma(k)$	1.1.1.1	$\varnothing$[b]	$\Phi \geq \sigma(k)$	*false*
$\Phi = \sigma'(k)$	$\sigma'(k) \geq 5$	1.1.1.2	$\Phi = 5$[c]	$\Phi \geq 5$	*false*
$1 \leq \Phi \leq \sigma'(k)$	$\sigma'(k) \geq 5$	1.1.1.2	$1 \leq \Phi \leq 5$	$1 \leq \Phi$	*false*
$\Phi = \sigma'(k)$		1.1.1.1	$\varnothing$		*false*
$\Phi = \sigma'(i)$	$\sigma'(k) \geq \sigma(k)$	1.2	$\Phi = \sigma(i)$	$\Phi = \sigma(i)$	*true*
$2\Phi = 3\sigma'(k)$	$4\sigma'(k) = 5\sigma(j)$	1.1.2	$8\Phi = 15\sigma(j)$	$8\Phi = 15\sigma(j)$	*true*

[a] $\overline{t}(\sigma, \sigma'_1) \wedge \overline{t}(\sigma, \sigma'_2) \wedge \sigma'_{1/v} \neq \sigma'_{2/v} = \{\sigma'_1(k) = \sigma(k)+1 \wedge \sigma'_2(k) = \sigma(k)+1 \wedge \sigma'_1(k) < \sigma'_2(k)\}$ is not feasible.
[b] $\underline{\mathcal{R}}(\sigma'_1) \cap \underline{\mathcal{R}}(\sigma'_2 \neq \sigma'_1) = \{\Phi = \sigma'_1(k) \wedge \Phi = \sigma'_2(k) \wedge \sigma'_1(k) < \sigma'_2(k) \wedge \sigma'_1(k) \geq \sigma(k) \wedge \sigma'_2(k) \geq \sigma(k)\} = \varnothing$.
[c] $k_{\min} = 5$, $k_{\max} = \infty$; $\underline{\mathcal{R}}(\sigma' : \sigma'(k) = k_{\min}) \cap \underline{\mathcal{R}}(\sigma' : \sigma'(k) = k_{\max}) = \{\Phi = 5\} \cap \{\} = \{\Phi = 5\}$

Table 8.1: Examples for the composition of regions with transformers

```
C  T(K) {}
    K = FOO()
C  T(K) {K==K#init+I-1}
    DO I = 1,N
       DO J = 1,N
          WORK(J,K) = J + K
       ENDDO
C  T(K) {K==K#init+1}
       CALL INC1(K)
       DO J = 1,N
          WORK(J,K) = J*J - K*K
          A(I) = A(I)+WORK(J,K)+WORK(J,K-1)
       ENDDO
    ENDDO
```

Figure 8.5: Transformers for the program of Figure 5.2

8.5 A Step-by-Step Example

Let us consider the program of Figure 5.2 on Page 67, and let us compute its READ, WRITE, IN and OUT regions, according to the semantic functions provided in Chapter 6, and to the operators presented in the previous sections.

Unlike the implementation available in PIPS, we do not use preconditions. In this example, it has no effect on the result and its exactness, and it would only affect the readability of regions! We also do not use continuation conditions, because there is no `stop` statement, and they would always be equal to the identity function. But we use transformers, since a scalar variable (`K`) is modified in the main loop body. We provide in Figure 8.5 the transformers which are not equal to the identity function. Finally, we do not compute the regions of scalar variables, even though it is implemented in PIPS.

8.5.1 READ and WRITE regions

Since this is a backward analysis, we begin with the regions of the second `J` loop body, and proceed towards the surrounding constructs.

Second `J` loop body In the two instructions,

```
WORK(J,K) = J*J-K*K
A(I) = A(I)+WORK(J,K)+WORK(J,K-1)
```

the array indices are affine functions of the program variables. Each reference can thus be exactly represented by a convex polyhedron. For instance, for the first instruction, we obtain:

C <WORK(ϕ_1,ϕ_2)-W-EXACT-{ϕ_1==J, ϕ_2==K}>

For the second instruction, we have to merge the regions corresponding to the two references to array `WORK`. The over-approximation is the convex hull of the initial polyhedra; and since the exactness criterion defined in Section 8.3.1 is favorable, this is an exact region:

```
C <WORK(φ1,φ2)-R-EXACT-{φ1==J,K-1<=φ2<=K}>
C <A(φ1)-R-EXACT-{φ1==I}>
C <A(φ1)-W-EXACT-{φ1==I}>
```

For the whole loop body, since no integer scalar variable is modified, the result is the mere union of the regions of each statement:

```
C <WORK(φ1,φ2)-W-EXACT-{φ1==J, φ2==K}>
C <WORK(φ1,φ2)-R-EXACT-{φ1==J,K-1<=φ2<=K}>
C <A(φ1)-R-EXACT-{φ1==I}>
C <A(φ1)-W-EXACT-{φ1==I}>
```

Second `J` loop We obtain over-approximations of the regions of the second `J` loop by first adding the constraints on the loop index due to its bounds (`1<=J<=N`) in the polyhedra describing the regions, and then eliminating the loop index. The loop bounds are affine functions of the loop index, and for the four preceding regions, the elimination is exact. The resulting regions are thus exact regions (see Section 8.3.4):

```
C <A(φ1)-R-EXACT-{φ1==I}>
C <A(φ1)-W-EXACT-{φ1==I}>
C <WORK(φ1,φ2)-R-EXACT-{1<=φ1<=N,K-1<=φ2<=K}>
C <WORK(φ1,φ2)-W-EXACT-{1<=φ1<=N,φ2==K}>
```

We similarly obtain the regions for the first loop, and for the first loop body (see Figure 8.6).

Outermost loop body The next step computes the regions for the body of the outermost loop (index `I`). It first computes the regions for the sub-sequence made of the call to `INC1` and the second `J` loop. We know that for READ and WRITE regions:

$$\mathcal{R}[\![S_1;S_2]\!] = \mathcal{R}[\![S_1]\!] \cup \mathcal{R}[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]$$

Here the regions of S_1 are the empty set. But the over-approximation of $\mathcal{T}[\![S_1]\!]$ is not the identity function. We must therefore compose the regions previously obtained for the second `J` loop by the transformer modeling the effects of the call on the value of `K`:

```
T(K) {K==K#init+1}
```

Here `K#init` represents the value of `K` before the call.

`K` does not appear in the polyhedron describing the regions of the array `A`: Hence, the composition does not affect them. But the regions for array `WORK` depend on the value of `K`. We can compute the over-approximations of their composition by the transformer by adding its constraint to their polyhedra, eliminating `K`, and then renaming `K#init` into `K`. This operation is exact according to the exactness criterion presented on Page 154 (in fact, we can prove that the transformer is invertible). We thus obtain:

C <A(ϕ_1)-R-EXACT-{ϕ_1==I}>
C <A(ϕ_1)-W-EXACT-{ϕ_1==I}>
C <WORK(ϕ_1,ϕ_2)-R-EXACT-{1<=ϕ_1<=N,K<=ϕ_2<=K+1}>
C <WORK(ϕ_1,ϕ_2)-W-EXACT-{1<=ϕ_1<=N,ϕ_2==K+1}>

We can now compute the regions for the whole sequence by adding to the previous regions the regions of the first J loop. We finally obtain:

C <A(ϕ_1)-R-EXACT-{ϕ_1==I}>
C <A(ϕ_1)-W-EXACT-{ϕ_1==I}>
C <WORK(ϕ_1,ϕ_2)-R-EXACT-{1<=ϕ_1<=N,K<=ϕ_2==K+1}>
C <WORK(ϕ_1,ϕ_2)-W-EXACT-{1<=ϕ_1<=N,K<=ϕ_2<=K+1}>

Outermost loop The previous regions depend on the value of K at the current iteration. To get rid of it, we use the transformer of the loop, which provides its invariant, as proposed in Section 6.3.6:

T(K) {K==K#init+I-1}

We compose the previous regions with this transformer as was done for the call statement. This operation is exact, because we can prove that the transformer is invertible. This gives:

C <A(ϕ_1)-R-EXACT-{ϕ_1==I}>
C <A(ϕ_1)-W-EXACT-{ϕ_1==I}>
C <WORK(ϕ_1,ϕ_2)-R-EXACT-{1<=ϕ_1<=N,K+I-1<=ϕ_2==K+I}>
C <WORK(ϕ_1,ϕ_2)-W-EXACT-{1<=ϕ_1<=N,K+I-1<=ϕ_2<=K+I}>

Here K represents the value of K before the execution of the loop.

And after adding the constraints on the loop index I due to its bounds, and eliminating I, we get the regions of the outermost loop, which are exact according to the exactness criterion.

C <A(ϕ_1)-R-EXACT-{1<=ϕ_1<=N}>
C <A(ϕ_1)-W-EXACT-{1<=ϕ_1<=N}>
C <WORK(ϕ_1,ϕ_2)-R-EXACT-{1<=ϕ_1<=N,K<=ϕ_2<=K+N}>
C <WORK(ϕ_1,ϕ_2)-W-EXACT-{1<=ϕ_1<=N,K<=ϕ_2<=K+N}>

8.5.2 IN regions

The propagation of IN regions follows the same principle as for READ and WRITE regions: It begins from the innermost statements, and proceeds towards the surrounding constructs. We therefore begin with the body of the second J loop.

Second J loop body The first instruction does not read any array element, and its IN regions are represented by the empty list. The second instruction is an assignment, and its IN regions are equal to its READ regions:

C <WORK(ϕ_1,ϕ_2)-IN-EXACT-{ϕ_1==J, K-1<=ϕ_2<=K}>
C <A(ϕ_1)-IN-EXACT-{ϕ_1==I}>

```
      K = FOO()

C  <A(φ1)-R-EXACT-{1<=φ1<=N}>
C  <A(φ1)-W-EXACT-{1<=φ1<=N}>
C  <WORK(φ1,φ2)-R-EXACT-{1<=φ1<=N,K<=φ2<=K+N}>
C  <WORK(φ1,φ2)-W-EXACT-{1<=φ1<=N,K<=φ2<=K+N}>
      DO I = 1, N

C  loop body:
C  <A(φ1)-R-EXACT-{φ1==I}>
C  <A(φ1)-W-EXACT-{φ1==I}>
C  <WORK(φ1,φ2)-R-EXACT-{1<=φ1<=N,K<=φ2==K+1}>
C  <WORK(φ1,φ2)-W-EXACT-{1<=φ1<=N,K<=φ2<=K+1}>

C  <WORK(φ1,φ2)-W-EXACT-{1<=φ1<=N,φ2==K}>
         DO J = 1, N
C  <WORK(φ1,φ2)-W-EXACT-{φ1==J, φ2==K}>
            WORK(J,K) = J+K
         ENDDO
         CALL INC1(K)

C  <A(φ1)-R-EXACT-{φ1==I}>
C  <A(φ1)-W-EXACT-{φ1==I}>
C  <WORK(φ1,φ2)-R-EXACT-{1<=φ1<=N,K-1<=φ2<=K}>
C  <WORK(φ1,φ2)-W-EXACT-{1<=φ1<=N,φ2==K}>
         DO J = 1, N

C  <WORK(φ1,φ2)-W-EXACT-{φ1==J, φ2==K}>
            WORK(J,K) = J*J-K*K
C  <WORK(φ1,φ2)-R-EXACT-{φ1==J,K-1<=φ2<=K}>
C  <A(φ1)-R-EXACT-{φ1==I}>
C  <A(φ1)-W-EXACT-{φ1==I}>
            A(I) = A(I)+WORK(J,K)+WORK(J,K-1)
         ENDDO
      ENDDO
```

Figure 8.6: READ and WRITE regions for the program of Figure 5.2

To obtain the IN regions of the loop body, we must remove from this set the regions written by the first instruction, which are:

C <WORK(ϕ_1,ϕ_2)-W-EXACT-{ϕ_1==J, ϕ_2==K}>

The result can be represented by a list of convex polyhedra, and we get:

C <WORK(ϕ_1,ϕ_2)-IN-EXACT-{ϕ_1==J, ϕ_2==K-1}>
C <A(ϕ_1)-IN-EXACT-{ϕ_1==I}>

Second J loop We must now compute the IN regions for the whole loop, according to the formula provided in Section 6.5.5. Since no scalar variable is modified within the loop nest, we do not have to take into account the transformer of the loop, and the function we use therefore is:

$$\mathcal{R}_i[\![\texttt{do i=1,n } S \texttt{ enddo}]\!] = \bigcup_{k=1}^{k=\mathcal{E}[\![\texttt{n}]\!]} \left\{ \mathcal{R}_i[\![S]\!] \circ \mathcal{E}[\![\texttt{i=}k]\!] \boxminus \bigcup_{k'=1}^{k'=k-1} \mathcal{R}_w[\![S]\!] \circ \mathcal{E}[\![\texttt{i=}k']\!] \right\}$$

We first compute the sets of array elements written by all the preceding iterations $(\bigcup_{k'=1}^{k'=k-1} \mathcal{R}_w[\![S]\!])$. $\mathcal{R}_w[\![S]\!]$ is here

C <WORK(ϕ_1,ϕ_2)-W-EXACT-{ϕ_1==J$'$, ϕ_2==K}>
C <A(ϕ_1)-W-EXACT-{ϕ_1==I}>

The current iteration is iteration j. We thus obtain $\bigcup_{j'=1}^{j'=j-1} \mathcal{R}_w[\![S]\!]$ by adding the constraint 1<=J$'$<=J-1 to the polyhedra of the previous regions, and by eliminating J$'$. This is an exact operation, and we get:

C <WORK(ϕ_1,ϕ_2)-W-EXACT-{1<=ϕ_1<=J-1, ϕ_2==K}>
C <A(ϕ_1)-W-EXACT-{ϕ_1==I,2<=J}>

We then remove these sets from the IN regions of the loop body, and get:

C <WORK(ϕ_1,ϕ_2)-IN-EXACT-{ϕ_1==J, ϕ_2==K-1}>
C <A(ϕ_1)-IN-EXACT-{ϕ_1==I,J==1}>

These regions represent the contribution of the current iteration to the IN regions of the whole loop. We must compute their union over the range of the loop index to obtain the IN regions of the whole loop. This operation, performed as usual, is exact, and gives:

C <WORK(ϕ_1,ϕ_2)-IN-EXACT-{1<=ϕ_1<=N,ϕ_2==K-1}>
C <A(ϕ_1)-IN-EXACT-{ϕ_1==I}>

Outermost loop body The IN regions of the sub-sequence made of the call to INC1 and the second J loop are the IN regions of the second J loop, composed with the transformer of the call to INC1:

C <WORK(ϕ_1,ϕ_2)-IN-EXACT-{1<=ϕ_1<=N,ϕ_2==K}>
C <A(ϕ_1)-IN-EXACT-{ϕ_1==I}>

The IN regions of the outermost loop body are then computed by removing from these regions those written by the first J loop, which are:

C <WORK(ϕ_1,ϕ_2)-W-EXACT-{1<=ϕ_1<=N,ϕ_2==K}>

This operation is exact, and gives:

C <A(ϕ_1)-IN-EXACT-{ϕ_1==I}>

```
      K = FOO()

C  <A(φ1)-IN-EXACT-{1<=φ1<=N}>
      DO I = 1, N

C  loop body:
C  <A(φ1)-IN-EXACT-{φ1==I}>

         DO J = 1, N
            WORK(J,K) = J+K
         ENDDO
         CALL INC1(K)

C  <A(φ1)-IN-EXACT-{φ1==I}>
C  <WORK(φ1,φ2)-IN-EXACT-{1<=φ1<=N,φ2==K-1}>
         DO J = 1, N
            WORK(J,K) = J*J-K*K
C  <WORK(φ1,φ2)-IN-EXACT-{φ1==J, K-1<=φ2<=K}>
C  <A(φ1)-IN-EXACT-{φ1==I}>
            A(I) = A(I)+WORK(J,K)+WORK(J,K-1)
         ENDDO
      ENDDO
```

Figure 8.7: IN regions for the program of Figure 5.2

Outermost loop The IN regions for the loop are then obtained in the same way as detailed for the second J loop:

```
C <A(φ1)-IN-EXACT-{1<=φ1<=N}>
```

Figure 8.7 summarizes all these results.

8.5.3 OUT regions

Lastly, we want to compute the OUT regions. We assume that WORK is not reused afterwards, and that the elements of A described by A(1:N) are reused. The OUT regions for the outermost loop are thus:

```
C <A(φ1)-OUT-EXACT-{1<=φ1<=N}>
```

Outermost loop body The OUT regions of the loop body at iteration I are the intersection of its WRITE regions with 1) the OUT regions of the whole loop, minus the regions written by following iterations, and 2) the regions imported by the following iterations.

The loop does not export any element of the array WORK, and no iteration imports any of them. Thus the OUT regions of the body of the loop for the array WORK are the empty set.

More work is necessary for the array A. We first compute the regions written by the following iterations (union over the range I+1<=I′<=N):

```
C <A(φ1)-W-EXACT-{I+1<=φ1<=N}>
```

and remove them from the OUT regions of the loop:

```
C <A(φ1)-OUT-EXACT-{1<=φ1<=I}>
```

We then take the intersection with the regions written by the current iteration, and get:

```
C <A(φ1)-OUT-EXACT-{φ1==I}>
```

This represents the array elements exported by the loop body towards the instructions executed after the whole loop. We must now compute the set of elements exported towards the other iterations of the loop.

The array elements imported by an iteration I' following the iteration I are represented by the region

```
C <A(φ1)-IN-EXACT-{φ1==I',I+1<=I'<=N}>
```

From this set, we must remove the elements which are written by the iterations prior to iteration I' (`I''<=I'-1`) but ulterior to iteration I (`I+1<=I''`). These elements are obtained by computing the union over the range `I+1<=I''<=I'-1` of the WRITE regions of the loop:

```
C <A(φ1)-W-EXACT-{I+1<=φ1<=I'-1,I+1<=I'<=N}>
```

The subtraction from the previous IN region gives:

```
C <A(φ1)-IN-EXACT-{φ1==I',I+1<=I'<=N}>
```

To obtain the elements imported by all the following iterations, we take the union over the range `I+1<=I'<=N` of the previous region:

```
C <A(φ1)-IN-EXACT-{I+1<=φ1<=N}>
```

The intersection with the region written by the current iteration I therefore gives the empty set: It exports no element towards the following iterations.

Finally, the OUT regions for the whole loop body are:

```
C <A(φ1)-OUT-EXACT-{φ1==I}>
```

Statements of the body of loop I We go through these statements in reverse order. We thus begin with the second J loop. To obtain its OUT regions, we simply take the intersection of its WRITE regions with the OUT regions of the whole sequence of instructions constituting the loop body. Since these regions do not depend on variables modified by the loop (except the loop index), we do not have to take care of the transformer of the loop. We therefore obtain:

```
C <A(φ1)-OUT-EXACT-{φ1==I}>
```

For the first J loop, the result is the intersection of its WRITE regions with 1) the OUT regions of the whole sequence (here we get the empty set), union 2) the regions imported by the following instructions in the order of the sequence.

The second J loop imports the elements described by:

```
C <A(φ1)-IN-EXACT-{φ1==I}>
C <WORK(φ1,φ2)-IN-EXACT-{1<=φ1<=N,φ2==K-1}>
```

```
      K = FOO()
C  <A(φ1)-OUT-EXACT-{1<=φ1<=N}>
      DO I = 1, N

C  loop body:
C  <A(φ1)-OUT-EXACT-{φ1==I}>

C  <WORK(φ1,φ2)-OUT-EXACT-{1<=φ1<=N,φ2==K}>
         DO J = 1, N
C  <WORK(φ1,φ2)-OUT-EXACT-{φ1==J, φ2==K, 1<=J<=N}>
            WORK(J,K) = J+K
         ENDDO
         CALL INC1(K)

C  <A(φ1)-OUT-EXACT-{φ1==I}>
         DO J = 1, N

C  <WORK(φ1,φ2)-OUT-EXACT-{φ1==J, φ2==K}>
            WORK(J,K) = J*J-K*K
C  <A(φ1)-OUT-EXACT-{φ1==I}>
            A(I) = A(I)+WORK(J,K)+WORK(J,K-1)
         ENDDO
      ENDDO
```

Figure 8.8: OUT regions for the program of Figure 5.2

We must first compose them by the transformer of the call:

```
C <A(φ1)-IN-EXACT-{φ1==I}>
C <WORK(φ1,φ2)-IN-EXACT-{1<=φ1<=N,φ2==K}>
```

before computing their intersection with the WRITE regions of the first `J` loop:

```
C <WORK(φ1,φ2)-OUT-EXACT-{1<=φ1<=N,φ2==K}>
```

From these regions, we can similarly compute the regions of the two `J` loop bodies, and their instructions. They are provided in Figure 8.8.

8.6 Where does inexactness come from?

In the previous example, all regions were exact regions. However, in real applications, a significant portion of array regions are approximations. A major source of inexactness often is the lack of structure of the programs (`goto`s are used instead of structured tests and loops). An aggressive restructuring phase is then necessary. For example, in many of our qualitative experiments (for example on `DYNA` [167]), *dead code elimination* based on *interprocedural preconditions* was necessary to clean up the code and enable the retrieval of interesting information. The interprocedural structure of the program may also be a limiting factor. This point will be discussed in Part IV.

Another limitation is due to nonlinear expressions. Some of them can sometimes be transformed into affine expressions by a partial evaluation phase. But in many cases, they must be handled as such. When they appear in test conditions, the resulting array regions are approximations, but may still carry sufficient information for the next program analyses or transformations.

> For example, for the following piece of code:
>
> ```
> do i = 1,n
>    if (R.lt.0.D0) then
>       A(i) = ...
>    endif
> enddo
> ```
>
> the convex region corresponding to the loop body is:
>
> ```
> <A(φ1)-W-MAY-{φ1==i}>
> ```
>
> This abstraction is sufficient to show that there exist no loop-carried dependences, hence that the loop is parallel.

When negations appear in test conditions, these drawbacks can be lessened or even eliminated by using other types of representations, as discussed in the next section.

However, this is not the case when array subscripts or assignments to variables used in array subscripts are nonlinear. Sophisticated techniques, which are not available in PIPS, are then necessary. For example, when nonlinear subscripts are due to the hand linearization of a multidimensional array, as in `TRFD` [25], the data structure may be recovered from the flat array [42].

Finally, even in pieces of code where all expressions are perfectly linear, convex polyhedra are not always sufficient to exactly represent array regions. The first reason is that some access patterns intrinsically are non-convex.

> An example is provided in Figure 8.9, where the loop nest references nine-point stencils. The over-approximate region of array `A` computed by PIPS for the loop body is:
>
> ```
> <A(φ1,φ2)-R-MAY-{i-2<=φ1<=i+2, j-2<=φ2<=j+2}>
> ```
>
> It contains four elements which do not belong to the exact region.

Another limiting factor is the non-associativity of the under-approximate union of two convex regions.

> Let us consider the following piece of code:
>
> do i =1,n A(i) = B(i-1) + B(i+1)
> + B(i) enddo
>
> And let us compute the READ regions of array `B` for the loop body. The polyhedron describing the first reference `B(i-1)` is $\{\Phi_1 : \Phi_1 = \mathtt{i} - 1\}$. This is an exact region since the subscript expression is linear:
>
> ```
> <B(φ1)-R-EXACT-{φ1==i-1}>
> ```
>
> Then we compute its union with the polyhedron for the second reference ($\{\Phi_1 : \Phi_1 = \mathtt{i} + 1\}$). With our exactness criteria, we can prove that this operation is not exact, and leads to an over-approximation. We get:

```
<B(ϕ1)-R-MAY-{i-1<=ϕ1<=i+1}>
```

Finally, we add the third reference descriptor $\{\Phi_1 : \Phi_1 = \mathtt{i}\}$, and the summary region becomes:

```
<B(ϕ1)-R-MAY-{i-1<=ϕ1<=i+1}>
```

This is an exact representation, but because of the non-associativity of our union operator, we cannot prove it. In fact, this comes from the fact that the under-approximate region for the first two references implicitly was the empty set. When adding the polyhedron for the third reference, we would have had $\varnothing \cup \{\Phi_1 : \Phi_1 = i\} = \{\Phi_1 : \Phi_1 = i\}$. This is different from the over-approximation we already computed, and which therefore is not exact.

In practice, we do not compute the under-approximation, but use a series of criteria the test whether the combination (union, intersection or difference) of an over-approximate region and an exact one can be exact. These criteria were given in [18, 65].

```
 do i = 1,n
   do j = 1, m
     B(i,j) = A(i,j) + A(i,j-1) + A(i,j-2)
1           + A(i,j+1) + A(i,j+2)
2           + A(i+1,j) + A(i+2,j)
3           + A(i-1,j) + A(i-2,j)
   enddo
 enddo
```

Figure 8.9: Nine-point stencil loop

8.7 Related Work

Who would not dream of an ideal representation? It would allow an exact representation of all access patterns found in real life applications, and so be supported by precise internal operators and external composition laws. It would allow to take symbolic context information into account, since it has been shown to be necessary by several studies [29, 89]. And finally, the complexity of the whole framework both in terms of memory requirements and computation time should be very low ...

However, except for specific applications, this ideal representation has not yet been found, and all studies provide a compromise between accuracy and complexity: Some frameworks are based on exact representations, but only when some requirements are met; another approach is to compute compact summaries to avoid memory space complexity; lastly, a recent trend is to use infinite representation based on lists of compact sets of array elements. The advantages and drawbacks of each class of representation are detailed below, with an eye to the ideal representation!

8.7.1 Exact representations

In fact this phrase collectively denotes the methods which try to retain as much information as inline expansion (see Chapter 10), by keeping loop indices in the representation, and by avoiding summarization. The idea was introduced by LI and YEW [122] and the representation was called *Atom Image*. An atom image is built for each reference; it includes the subscript expressions on each dimension of the array, as well as the loop bounds of each surrounding loop. Atom images are not merged, thus avoiding this source of time complexity, but at the expense of memory requirements.

The obvious advantage of this method is its accuracy. However, atom images can only be built when subscript expressions are linear functions of enclosing loop indices and of constant symbolic variables whose values are known at compile time. This restriction has since been partially removed by HIND [100, 101]: His access descriptors can contain constant symbolic values whose values are unknown; and they also include array bounds. Another drawback is that the time complexity burden is merely delayed to the subsequent analyses (usually dependence analysis).

8.7.2 Compact representations

To avoid time complexity in subsequent analyses and huge memory requirements, another approach is to use compact representations, by merging array regions relative to individual references, into a single array region. Several representations have been proposed:

Convex polyhedra were first used by TRIOLET [161]. They can represent various shapes of array regions, provided that they be convex. They do not include stride information, thus loosing accuracy in some applications. But they are very useful when symbolic information is required, because any linear constraint can be included in polyhedra. However, the complexity of the operators is theoretically exponential, and is generally high, even though it has been shown to be polynomial in practical cases [9].

Regular Section Descriptors There are several definitions for RSDs. The initial ones [37] can represent elements, rows, columns, diagonals and their higher dimensional analogs, but cannot be used to represent triangular regions or discontinuous portions of rows.

HAVLAK also gives a definition of RSDs with bounds and strides [98, 99], but it cannot represent diagonals. And combinations of both representations are also used, for instance in [164].

The advantage of RSDs is that the complexity of merging and comparison operators is very low. However, they cannot include symbolic information, such as `if` conditions for instance.

Data Access Descriptors DADs have been defined by BALASUNDARAM [19] as a compromise between the low complexity provided by RSDs and the accuracy of convex polyhedra. They are based on *simple sections*, which are convex polyhedra in which constraints are either parallel to the axes, or at a 45 degree angle to a pair of axes. They can represent triangular regions, but, as convex polyhedra, not discontinuous rows. And as RSDs, they do not include context information on their own.

Integer programming projection TANG [159] exactly summarizes loop effects on array variables (even in case of multiple array references) in the form of an integer programming projection. However, the source language is very restricted.

Also notice that convex polyhedron is the only representation which allows powerful translation algorithms at procedure boundaries (see Chapter 11).

8.7.3 Lists of regions

As accuracy nowadays appears as a necessity to achieve better results in parallelizing compilers, lists of regions are more and more used. They allow to precisely represent unions of regions, and to avoid loosing too much accuracy when performing differences. It is unclear whether such representations raise or lower the complexity of the whole parallelization process: On one hand, comparing two lists of regions for dependence tests is more expensive than combining two individual regions; on the other hand, the cost of merging is avoided and reduced to appending the region at the end of the current list. And very often, cheap tests are performed to decide whether actually merging regions would not result in less precision, thus resulting in shorter lists whenever possible.

The proposed representations are based on several compact representations of array element sets:

Convex polyhedra are used in FIAT/SUIF [92]. As said in the previous section, their main interest is their ability to include symbolic context information. However, this is not fully exploited in FIAT/SUIF, where only loop bounds are taken into account.

Regular section descriptors An extended version of RSDs is used in POLARIS [164]. But since it could not include context information, an extension to *gated* RSDs has also been informally proposed to take `if` conditions into account.

Data access descriptors Lists of DADs have been proposed to study inter-loop parallelism in [54]. However, they cannot contain context information.

Complementary sections have been defined by MANJUNATHAIAH [125]: They contain a simple section to describes the hull of the region, and several simple sections which represent the complementary of the exact region in the hull. Efficient operators have been defined to study coarse grain parallelism. This representation has the same drawbacks as DADs.

Guarded regular section descriptors are used in the PANORAMA compiler [135, 87]. They are merely RSDs guarded by a predicate, thus allowing to include symbolic context information such as `if` conditions. Theses sets are exact sets, unless they contain an unknown component (dimension or context constraint). Convex polyhedra should be more accurate, because they can keep approximate information about array descriptors (Φ variables). But the implementation of guarded RSDs in PANORAMA seems very efficient.

Symbolic data descriptors have been proposed in [83]. They contain a guard, which is not necessarily a linear expression, and an access pattern which provides a possibly guarded expression for each dimension. This representation is very general, but it cannot represent coupled dimensions of arrays (`A(I,I)`). Moreover, the operators are not described, and the complexity of the analysis is thus not available.

Presburger formula have also been proposed for array region analyses by WONNACOTT in [176]. They are not, strictly speaking, lists of compact sets, but can also be viewed as combinations of compact sets. In fact, they are very similar to lists of convex polyhedra, except that they are better suited to represent differences, and are better suited to take into account the negative constraints which appear as test conditions.

8.8 Conclusion

The purpose of this part was to present our implementation of the intraprocedural propagation of READ, WRITE, IN and OUT array regions in the PIPS parallelizer. And thus to demonstrate the feasibility of the theoretical framework introduced in Part II, especially for the computation of under-approximations from the over-approximations using a computable exactness criterion.

The chosen representation of array regions as parameterized convex polyhedra appears as one of the most practical way to exploit linear symbolic information, by composing regions with other analyses such as transformers and filters. We will also see in the next part that this representation is well adapted to the translation of array regions across procedure boundaries.

The patterns that can be represented by convex polyhedra are very various. But some very common patterns, such as discontinuous rows or nine-point stencils, are nevertheless missed. One of our intentions for the future is therefore to implement other types of representations, in particular lists of polyhedra or Z-polyhedra [9]. The accuracy would be increased, without sacrificing the ability to retain context information. Moreover, it would allow comparisons between the representations, independently of the program representation, and of other factors such as the degree of symbolic information available.

IV

ANALYSES INTERPROCÉDURALES

INTERPROCEDURAL ARRAY REGION ANALYSES

Chapitre 9

Analyses Interprocédurales de Régions

(Résumé des chapitres 10, et 11)

Les deux parties précédentes décrivaient la propagation intraprocédurale des régions de tableaux. Cependant, les applications réelles sont généralement divisées en plusieurs procédures, de façon à permettre la réutilisation de code, et pour augmenter la lisibilité des programmes. Pour la plupart des compilateurs, les appels de procédures sont des barrières infranchissables, et empêchent la plupart des analyses et des optimisations, comme la propagation de constantes, ou l'analyse des dépendances, qui sont pourtant de la plus grande utilité dans les paralléliseurs pour espérer des gains de performances [29]. La solution de l'expansion de procédure qui consiste à remplacer chaque appel par le code de la procédure appelée n'est pas toujours possible, ni souhaitable à cause de la complexité qu'elle engendre. Les analyses interprocédurales sont donc de plus en plus répandues.

Cette partie est consacrée à l'analyse interprocédurale des régions de tableaux dans PIPS. Celle-ci est constituée de deux composantes : la *propagation* sur le graphe de contrôle interprocédural, et la *traduction* des régions de l'espace de nommage de la procédure source dans celui de la procédure cible. Dans une première section, nous décrirons donc la propagation interprocédurale des régions dans PIPS. Et dans la section suivante, nous présenterons un nouvel algorithme de traduction des régions.

9.1 Propagation Interprocédurale

Dans PIPS, les analyses interprocédurales sont effectuées sur le graphe des appels du programme. C'est un graphe acyclique car la norme FORTRAN [8] ne permet pas la récursivité.

Les régions READ, WRITE, et IN sont propagées en arrière sur cette représentation du programme. A chaque site d'appel, les régions résumées de la procédure appelée sont traduites de l'espace de nommage de la procédure appelée dans celui de la procédure appelante, en utilisant les relations qui existent entre les paramètres formels et les paramètres réels, et entre les déclarations de variables globales. Ces régions traduites sont alors utilisées lors de l'analyse intraprocédurale des procédures appelantes, et ainsi de suite. Ceci est illustré dans la figure 9.1.

Au contraire, la propagation des régions OUT s'effectue en avançant dans le graphe des appels. Les régions de chaque site d'appel sont tout d'abord traduites de l'espace de nommage de l'appelant dans celui de l'appelé, puis sont fusionnées pour former un résumé unique[1], comme le montre la figure 9.2. Ce résumé est alors utilisé pour effectuer l'analyse intraprocédurale des régions OUT de la procédure appelée.

Ces façons de procéder permettent de prendre en compte le flot de contrôle interne à chaque procédure, mais pas le flot de contrôle interprocédural. De plus, l'utilisation systématique de résumés ne permet pas de distinguer les informations spécifiques à tel ou tel chemin d'accès, et conduit donc à considérer des combinaisons ne pouvant pas se produire, problème connu sous le nom d'*unrealizable paths* [119]. L'utilisation du clônage sélectif [53, 91], ou l'utilisation de représentations du programme plus générales [134], pourraient en réduire les effets.

9.2 Traduction des Régions

Cette section décrit la partie *traduction* de la propagation interprocédurale. Comme les variables source et cible peuvent ne pas avoir la même déclaration (*array reshaping*), cette opération n'est pas aisée à réaliser.

En examinant les programmes du Perfect Club [25], nous avons identifié un certain nombre de cas non-exclusifs qu'il est nécessaire de pouvoir traiter :

1. l'*array reshaping* dû à des déclarations de dimensions différentes ;
2. les décalages (*offsets*) entre les premiers éléments des tableaux source et cible, dûs au passage de paramètres (`CALL F(A(1,J))` par exemple) ;
3. les décalages dûs à des déclarations de `COMMON` différentes dans l'appelant et l'appelé (par exemple, dans le programme `TRFD`, le `COMMON` `TR2PRT` n'est pas déclaré de la même manière dans les procédures `TRFPRT` et `TRFOUT`) ;
4. les variables de types différents (comme dans le programme `OCEAN`).

La méthode décrite dans cette section permet de résoudre ces quatre problèmes. Elle est basée sur le fait que deux éléments correspondants des tableaux source et cible doivent avoir la même *valeur d'indice*[2], au décalage entre leur premier élément près. Ceci est présenté dans la section 9.2.2.

Cependant, le *système de traduction* qui en résulte peut contenir des termes non-linéaires ; et même si ce n'est pas le cas, il cache les relations triviales qui existent entre les descripteurs des deux tableaux (variables Φ). Nous proposons donc un algorithme qui recherche tout d'abord ces relations triviales, avant d'utiliser les *valeurs d'indice* pour les dimensions restantes. Le résultat est un *système de traduction simplifié*.

9.2.1 Notations

Dans la suite de cette section, nous utiliserons les notations suivantes :

[1] Rappelons que les régions OUT de la procédure principale sont initialisées à l'ensemble vide.

[2] La valeur d'indice d'un élément de tableau est son *rang* dans le tableau à partir du premier élément, les éléments étant rangés par colonnes [8].

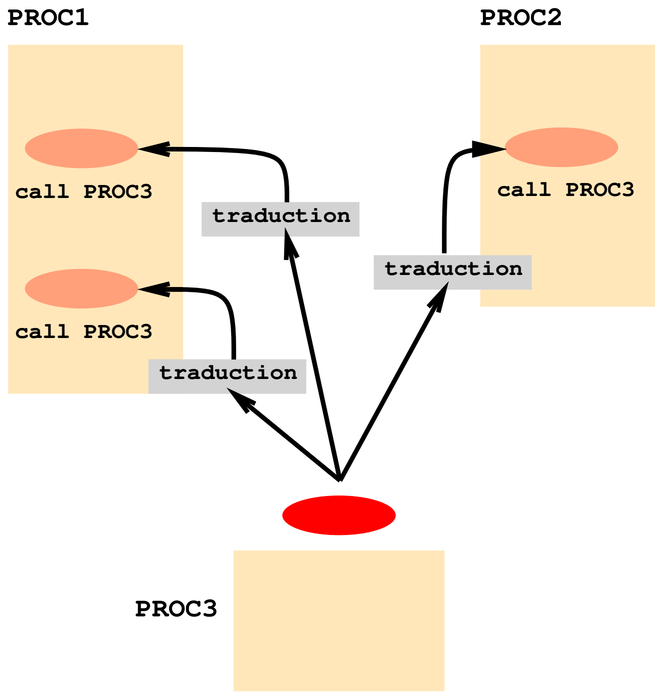


Figure 9.1: Propagation interprocédurale en arrière

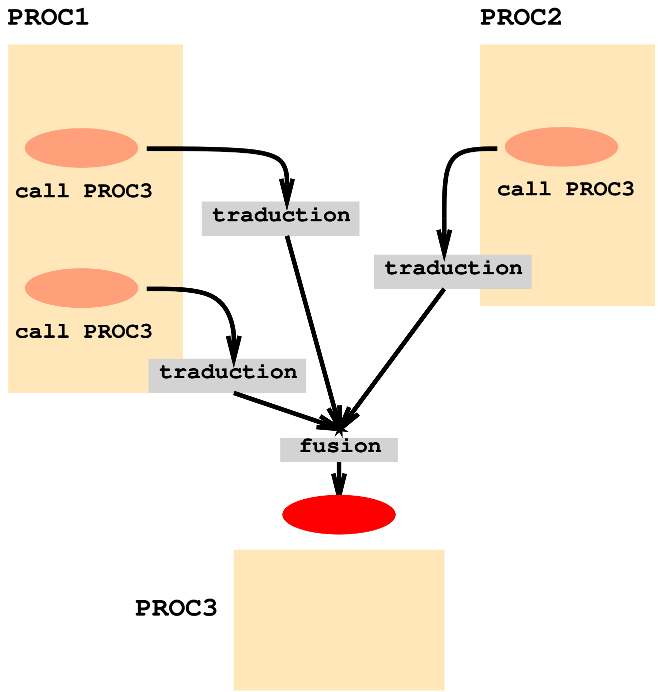


Figure 9.2: Propagation interprocédurale en avant

```
subroutine FOO(C,n)
complex C(n,10,20),D
common D(5,10)
call BAR(C,2n,100)
end
```

```
C  <D2(φ1,φ2)-W-EXACT-{1<=φ1<=10, 1<=φ2<=9}>
C  <D1(φ1)-W-EXACT-{1<=φ1<=10}>
C  <R(φ1,φ2)-W-EXACT-{1<=φ1<=N1, 1<=φ2<=N2}>
   subroutine BAR(R,n1,n2)
   real R(n1,n2)
   common D1(10), D2(10,9)
   ...
   end
```

Figure 9.3: Traduction interprocédurale : exemple

	source	$\longmapsto$	*cible*
tableau	A		B
dimension	α		β
bornes inférieures	$l_{a_1},\ldots,l_{a_\alpha}$		$l_{b_1},\ldots,l_{b_\beta}$
bornes supérieures	$u_{a_1},\ldots,u_{a_\alpha}$		$u_{b_1},\ldots,u_{b_\beta}$
taille des éléments[3]	s_a		s_b
descripteurs	$\phi_1,\ldots,\phi_\alpha$		$\psi_1,\ldots,\psi_\beta$

Les valeurs d'indice de $A(\phi_1,\ldots,\phi_\alpha)$ et $B(\psi_1,\ldots,\psi_\beta)$ sont alors[4]:

$$subscript\_value(A(\phi_1,\ldots,\phi_\alpha)) = \sum_{i=1}^{\alpha}[(\phi_i - l_{a_i})\prod_{j=1}^{i-1}(u_{a_j} - l_{a_j} + 1)]$$

$$subscript\_value(B(\psi_1,\ldots,\psi_\beta)) = \sum_{i=1}^{\beta}[(\psi_i - l_{b_i})\prod_{j=1}^{i-1}(u_{b_j} - l_{b_j} + 1)]$$

Une autre information nécessaire est le décalage du premier élément de A à partir du premier élément de B dans la mémoire. Cette information est fournie différemment selon le type d'association entre A and B :

tableau source	$\mapsto$	*tableau cible*	*décalage*
formel	$\mapsto$	*réel*	référence au site d'appel : $B(o_{b_1},\ldots,o_{b_\beta})$ *offset* $= s_b \times subscript\_value(B(o_{b_1},\ldots,o_{b_\beta}))$
réel	$\mapsto$	*formel*	référence au site d'appel : $A(o_{a_1},\ldots,o_{a_\alpha})$ *offset* $= -s_a \times subscript\_value(A(o_{a_1},\ldots,o_{a_\alpha}))$
global	$\mapsto$	*global*	décalage numérique différence entre le décalage de A par rapport au début du common dans la procédure source, et le décalage de B par rapport au début du common dans la procédure cible.

Considérons par exemple le programme imaginaire de la figure 9.3, qui contient toutes les difficultés rencontrées dans les programmes réels. Le but est de trouver les régions READ et WRITE au site d'appel, connaissant les régions résumées de la procédure BAR. Les valeurs utilisées pour la traduction sont :

[3]Unité: la taille de la quantité minimale de mémoire accessible, généralement un octet.

[4]Avec la convention que $\prod_{k=k_1}^{k_2} = 1$ quand $k_2 < k_1$.

$\texttt{R} \mapsto \texttt{C}$: $A = \texttt{R}$, $B = \texttt{C}$; $\alpha = 2$, $\beta = 3$; $l_{a_1} = l_{a_2} = 1$, $l_{b_1} = l_{b_2} = lb_3 = 1$; $u_{a_1} = \texttt{n1}$, $u_{a_2} = \texttt{n2}$; $u_{b_1} = \texttt{n}$, $u_{b_2} = 10$, $u_{b_3} = 20$; $s_a = 4$, $s_b = 8$; $\mathit{offset} = 0$;

$\texttt{D1} \mapsto \texttt{D}$: $A = \texttt{D1}$, $B = \texttt{D}$; $\alpha = 1$, $\beta = 2$; $l_{a_1} = 1$, $l_{b_1} = l_{b_2} = 1$; $u_{a_1} = 10$; $u_{b_1} = 5$, $u_{b_2} = 10$; $s_a = 4$, $s_b = 8$; $\mathit{offset} = 0$;

$\texttt{D2} \mapsto \texttt{D}$: $A = \texttt{D2}$, $B = \texttt{D}$; $\alpha = 2$, $\beta = 2$; $l_{a_1} = l_{a_2} = 1$, $l_{b_1} = l_{b_2} = 1$; $u_{a_1} = 10$, $u_{a_2} = 9$; $u_{b_1} = 5$, $u_{b_2} = 10$; $s_a = 4$, $s_b = 8$; $\mathit{offset} = 40$.

9.2.2 Système de traduction général

Avec les notation précédentes, les descripteurs de l'élément $B(\psi_1, \ldots, \psi_\beta)$ correspondant à l'élément source$A(\phi_1, \ldots, \phi_\alpha)$ doivent vérifier le système suivant :

$$\exists\, \delta_a, \delta_b / \left\{ \begin{array}{l} s_a \times \mathit{subscript\_value}(A(\phi_1, \ldots, \phi_\alpha)) + \delta_a + \mathit{offset} \\ \ = s_b \times \mathit{subscript\_value}(B(\psi_1, \ldots, \psi_\beta)) + \delta_b \\ 0 \leq \delta_a < s_a \\ 0 \leq \delta_b < s_b \end{array} \right. \qquad (S)$$

Les variables δ servent à parcourir les cellules mémoire élémentaires des éléments de tableaux associés, comme on peut le voir sur la figure 9.4.

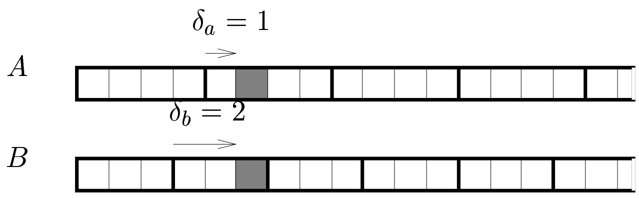


Figure 9.4: Signification des variables δ

Dans notre exemple, les systèmes suivants seraient construits :

$\texttt{R} \mapsto \texttt{C}$:

$$\left\{ \begin{array}{l} 4[(\phi_1 - 1) + \texttt{n1}(\phi_2 - 1)] + \delta_a = \\ \quad 8[(\psi_1 - 1) + \texttt{n}(\psi_2 - 1) + 10\texttt{n}(\psi_3 - 1)] + \delta_b \\ 0 \leq \delta_a < 4,\ 0 \leq \delta_b < 8,\ \texttt{n1} = 2\texttt{n} \end{array} \right.$$

$\texttt{D1} \mapsto \texttt{D}$:

$$\left\{ \begin{array}{l} 4(\phi_1 - 1) + \delta_a = 8[(\psi_1 - 1) + 5(\psi_2 - 1)] + \delta_b \\ 0 \leq \delta_a < 4,\ 0 \leq \delta_b < 8 \end{array} \right.$$

$\texttt{D2} \mapsto \texttt{D}$:

$$\left\{ \begin{array}{l} 4[(\phi_1 - 1) + 10(\phi_2 - 1)] + \delta_a + 40 = 8[(\psi_1 - 1) + 5(\psi_2 - 1)] + \delta_b \\ 0 \leq \delta_a < 4,\ 0 \leq \delta_b < 8 \end{array} \right.$$

L'utilisation de S pour la traduction a plusieurs inconvénients :

1. dans le cas de la traduction d'un paramètre formel en un paramètre réel (ou vice-versa), S est généralement non-linéaire (Comme dans notre premier exemple) ;

2. de façon à conserver une représentation convexe, les variables δ doivent être éliminées, ce qui conduit à une approximation ;

3. et même dans les cas favorables, l'équation apparaissant dans le système S est assez complexe, et cache les relations triviales existant entre les variables ϕ et ψ, comme $\phi_1 = \psi_1$; ceci rend les analyses suivantes inutilement complexes, et n'est pas acceptable dans un environnement interactif, où les régions sont affichées pour l'utilisateur.

Dans la section suivante, nous montrons comment éviter ces trois problèmes.

9.2.3 Système de traduction simplifié

Élimination des variables δ inutiles

Théorème 9.1
Si s_b divise s_a et offset, alors S est équivalent au système suivant[5]*:*

$$\exists\, \delta'_a / \begin{cases} s'_a \times subscript\_value(A(\phi_1,\ldots,\phi_\alpha)) + \delta'_a + \dfrac{offset}{s_b} \\ \quad = subscript\_value(B(\psi_1,\ldots,\psi_\beta)) \\ 0 \leq \delta'_a < s'_a \\ s'_a = \dfrac{s_a}{s_b} \end{cases}$$

Note

1. Dans le cas de la traduction d'un paramètre réel en un paramètre formel (ou vice-versa), s_b divise $s_a \Rightarrow s_b$ divise *offset* ;

2. en réalité, nous remplaçons juste s_a par $\frac{s_a}{s_b}$, s_b par 1, *offset* par $\frac{offset}{s_b}$ et nous utilisons S sans δ_b.

   Dans notre exemple, comme s_a divise s_b et *offset* dans les trois cas, les systèmes de traduction deviennent :

   `R` $\mapsto$ `C`:
   $$\begin{cases} (\phi_1 - 1) + \mathtt{n1}(\phi_2 - 1) = \\ \quad 2[(\psi_1 - 1) + \mathtt{n}(\psi_2 - 1) + 10\mathtt{n}(\psi_3 - 1)] + \delta_b \\ 0 \leq \delta_b < 2,\ \mathtt{n1} = 2\mathtt{n} \end{cases}$$
   `D1` $\mapsto$ `D`:
   $$\begin{cases} \phi_1 - 1 = 2[(\psi_1 - 1) + 5(\psi_2 - 1)] + \delta_b \\ 0 \leq \delta_b < 2 \end{cases}$$
   `D2` $\mapsto$ `D`:
   $$\begin{cases} (\phi_1 - 1) + 10(\phi_2 - 1) + 10 = 2[(\psi_1 - 1) + 5(\psi_2 - 1)] + \delta_b \\ 0 \leq \delta_b < 2 \end{cases}$$

Diminuer le degré de (S)

Définition 9.1
Une dimension d ($d \leq min(\alpha,\beta)$) est dite *similaire* pour les tableaux A et B si les trois conditions suivantes sont respectées :

[5] Il y a bien sûr un système similaire si s_a divise s_b et *offset*.

1. *Condition pour le décalage :*
Il ne doit pas y avoir de décalage entre le premier élément de B et celui de A sur la dimension d :

formel	$\mapsto$	*réel*	$\forall\, i/1 \leq i \leq d,\ o_{b_i} = l_{b_i}$
réel	$\mapsto$	*formel*	$\forall\, i/1 \leq i \leq d,\ o_{a_i} = l_{a_i}$
global	$\mapsto$	*global*	$\lvert \mathit{offset}\rvert \bmod s_a \prod_{i=1}^{d}(u_{a_i} - l_{a_i} + 1) = 0$ $\wedge \lvert \mathit{offset}\rvert \bmod s_b \prod_{i=1}^{d}(u_{b_i} - l_{b_i} + 1) = 0$

2. *Condition pour la première dimension :*
Les longueurs en octets des premières dimensions de A et B sont égales :
$$s_a(u_{a_d} - l_{a_d} + 1) = s_b(u_{b_d} - l_{b_d} + 1)$$
Ceci signifie que la première dimension compense entièrement la différence entre s_a et s_b. C'est pourquoi s_a et s_b n'apparaissent plus dans la condition suivante.

3. *Condition pour les dimensions suivantes ($2 \leq d \leq min(\alpha, \beta)$) :*
Les dimensions précédentes doivent être similaires, et la longueur de la dimension d de A doit être la même que celle de B :
$$u_{a_d} - l_{a_d} = u_{b_d} - l_{b_d}$$
Ceci n'est pas nécessaire si $d = \alpha = \beta$.

Notations 9.1
Nous utilisons maintenant les notations suivantes, pour $k \in [2..min(\alpha, \beta)]$:

k_subscript_value:
$$\mathit{k\_subscript\_value}(A(\phi_1, \ldots, \phi_\alpha)) = \sum_{i=k}^{\alpha}[(\phi_i - l_{a_i})\prod_{j=k}^{i-1}(u_{a_j} - l_{a_j} + 1)]$$
C'est le rang de l'élément de tableau $A(\phi_1, \ldots, \phi_\alpha)$ à partir de l'élément $A(\phi_1, \ldots, \phi_{k-1}, l_{a_k}, \ldots, l_{a_\alpha})$, c'est-à-dire le premier élément de la k-ième dimension.

k_offset:
C'est le décalage relatif à la k-ième dimension :

formel	$\mapsto$	*réel*	$\mathit{k\_subscript\_value}(B(o_{b_1}, \ldots, o_{b_\beta}))$
réel	$\mapsto$	*formel*	$-\mathit{k\_subscript\_value}(A(o_{a_1}, \ldots, o_{a_\alpha}))$
global	$\mapsto$	*global*	$\left\lfloor \frac{\mathit{offset}}{sa \prod_{i=1}^{k}(u_{a_i} - l_{a_i} + 1)} \right\rfloor$

Théorème 9.2
Si les dimensions 1 à $d-1$ ($1 \leq d-1 \leq min(\alpha, \beta)$) sont similaires, alors S est équivalent à :
$$\exists\, \delta_a, \delta_b / \begin{cases} s_a(\phi_1 - l_{a_1}) + \delta_a = s_b(\psi_1 - l_{b_1}) + \delta_b \\ \forall\, i \in [2..d-1],\ \ \phi_i - l_{a_i} = \psi_i - l_{b_i} \\ \mathit{d\_subscript\_value}(A(\phi_1, \ldots, \phi_\alpha)) + \mathit{d\_offset} = \\ \quad \mathit{d\_subscript\_value}(B(\psi_1, \ldots, \psi_\beta))^6 \\ 0 \leq \delta_a < s_a \\ 0 \leq \delta_b < s_b \end{cases} \qquad (S_d)$$

Dans notre exemple, les systèmes de traduction deviennent finalement :

`R` $\mapsto$ `C`:

$$\begin{cases} \phi_1 - 1 = 2(\psi_1 - 1) + \delta_b \\ \phi_2 - 1 = (\psi_2 - 1) + 10(\psi_3 - 1) \\ 0 \leq \delta_b < 2 \end{cases}$$

Ce système ne contient plus que des contraintes linéaires.

`D1` $\mapsto$ `D`:

$$\begin{cases} \phi_1 - 1 = 2(\psi_1 - 1) + \delta_b \\ (\psi_2 - 1) = 0 \\ 0 \leq \delta_b < 2 \end{cases}$$

Il n'y a plus que des relations très simples entre les variables ϕ et ψ. En particulier, il est maintenant évident que $\psi_2 = 1$, ce qui était caché dans le système initial.

`D2` $\mapsto$ `D`:

$$\begin{cases} \phi_1 - 1 = 2(\psi_1 - 1) + \delta_b \\ (\phi_2 - 1) + 1 = (\psi_2 - 1) \\ 0 \leq \delta_b < 2 \end{cases}$$

Le décalage pour le problème global a été transformé ici en un décalage n'affectant que la deuxième dimension (le terme $+1$ dans la seconde équation).

Enfin, l'algorithme de traduction est le suivant :

Algorithme

1. `entrée : une région` R_A `du tableau` A
2. $R_B \;=\; R_A$
3. $d = \; nombre\_de\_dimensions\_similaires(A, B) + 1$
4. `si` $d = 1$ `alors`
5. $\quad$ *système_de_traduction* $= S$
6. `sinon`
7. $\quad$ *système_de_traduction* $= S_d$
8. `finsi`
9. `ajouter` *système_de_traduction* `à` R_B
10. `éliminer les variables` δ
11. `éliminer les variables` ϕ
12. `renommer les variables` ψ `en variables` ϕ
13. `traduire le polyèdre de` R_B `dans`
    `l'espace de noms de la procédure cible`
14. `pour tout` $i \in [1..\beta]$ `ajouter` $l_{b_i} \leq \phi_i \leq u_{b_i}$ `à` R_B
15. `sortie :` R_B

[6]Dans le cas formel $\mapsto$ réel, si $d = min(\alpha, \beta) = \alpha$, cette équation peut être remplacée par $\forall\, i \in [d..\beta]$, $\psi_i = o_{b_i}$.

À chaque pas, l'exactitude de l'opération courante est vérifiée, selon les techniques présentées dans la partie III. En particulier, l'étape 13 est effectuée en utilisant un polyèdre convexe modélisant le passage des paramètres réels aux paramètres formels, et pouvant donc être assimilé à un transformeur ; c'est donc l'opérateur de combinaison avec un transformeur qui est utilisé à ce niveau, ainsi que son critère d'exactitude.

La dernière étape est particulièrement utile lorsqu'il n'y a qu'association partielle entre A et B, ce qui se produit lorsqu'ils appartiennent à un `COMMON` déclaré différemment dans les procédures source et cible.

Dans l'exemple de la figure 11.5, les régions résultantes sont toutes des régions exactes :

$$\texttt{<C(}\phi_1\texttt{,}\phi_2\texttt{,}\phi_3\texttt{)-W-EXACT-}\{\texttt{1<=}\phi_1\texttt{<=N,1<=}\phi_2\texttt{<=10,1<=}\phi_3\texttt{<=20,}\phi_2\texttt{+10}\phi_3\texttt{<=110}\}\texttt{>}$$
$$\texttt{<D(}\phi_1\texttt{,}\phi_2\texttt{)-W-EXACT-}\{\texttt{1<=}\phi_1\texttt{<=5, 2<=}\phi_2\texttt{<=10}\}\texttt{>}$$
$$\texttt{<D(}\phi_1\texttt{,}\phi_2\texttt{)-W-EXACT-}\{\texttt{1<=}\phi_1\texttt{<=5, }\phi_2\texttt{==1}\}\texttt{>}$$

9.3 Conclusion

Il est difficile de comparer la seule analyse interprocédurale des régions de tableaux de différents paralléliseurs, car la qualité des résultats dépend de nombreux facteurs, comme la prise en compte d'informations provenant d'analyses préliminaires, la représentation des ensembles d'éléments de tableaux, ou la qualité des analyses intraprocédurales. Indépendamment donc de ces facteurs, les comparaisons peuvent se faire à deux niveaux : celui de la propagation, et celui de la traduction.

Nous avons vu que les mécanismes de propagation disponibles dans PIPS permettent de prendre en compte le flot de contrôle intraprocédural des procédures, mais pas le flot de contrôle interprocédural, ce qui se traduit par une moindre précision lors des analyses sensibles au flot de contrôle, comme les régions IN et OUT. Ces analyses bénéficieraient de l'utilisation de techniques comme le clônage sélectif [53, 91].

Nous avons également présenté un nouvel algorithme de traduction des régions de tableaux au niveau des appels de procédure. Il peut être vu comme la synthèse et l'extension des techniques existantes. En effet, celles-ci utilisaient soit uniquement la détection des cas triviaux, soit uniquement la linéarisation [35] ; ou encore l'une et l'autre des deux méthodes [92] selon la réussite ou l'échec de la première.

Cependant, notre algorithme ne permet toujours pas de traiter certains cas de figure, comme la traduction entre des tableaux déclarés comme `MAT(N,M)` et `V(N*M)`. Pour résoudre ces problèmes de manière exacte, des techniques de reconnaissance de cas (*pattern matching*) ou des méthodes non-linéaires seraient nécessaires. Mais les analyses ultérieures devraient aussi être étendues à ces domaines, avec l'inévitable complexité qu'ils induisent.

Chapter 10

Interprocedural Propagation

The two previous parts were devoted to the intraprocedural propagation of array regions. However, real life applications are usually split into several procedures, to factorize code as much as possible, and to provide a readable code. In traditional compilers, procedure boundaries prevent most analyses and optimizations, such as constant propagation or dependence analyses for instance, which have been shown to be of paramount importance in parallelizing compilers [29] to achieve reasonable performances.

The present part is thus devoted to the interprocedural analysis of array regions. It can be decomposed into two components: The propagation along the interprocedural control flow graph, and the translation of regions from the source procedure name space into the target procedure name space. This chapter is devoted to the interprocedural propagation, while the next chapter provides an original translation algorithm. In a first section, we review the main approaches which have been proposed to solve problems due to procedure calls. Section 10.2 more particularly presents array region propagation in PIPS. And the last section quickly reviews the related work.

10.1 Handling Procedure Calls

Since procedure boundaries are handled very conservatively by traditional compilers, the first and most intuitive approach to handle procedure calls is to expand them *inline*, that is to say to replace each call by the corresponding code, in a recursive top-down or bottom-up manner. The main advantage is that usual intraprocedural analyses and transformations can be applied on the resulting code, with the same precision. However, there are several counter-arguments:

- First, inlining is not always possible [40], due to array reshaping, or recursive procedures in languages other than FORTRAN.

- Second, the resulting code is usually much larger than the original. If a procedure `a` calls procedure `b` t_{ab} times, and procedure b calls procedure c t_{bc} times, then c is expanded $t_{ab} \times t_{bc}$! The growth of the code is thus exponential[1].

[1] I have mistakenly printed the interprocedural control flow graph of program `SPICE` from the Perfect Club benchmarks. It is 1308 pages long! A counter-argument of weight ...

- Finally, even in the cases where the resulting code is not overwhelmingly large, the complexity of subsequent analyses may grow dramatically. For instance, if a loop contains n_1 references to array A, and a procedure which contains n_2 such references is called within, then $n_1 \times n_2$ additional dependence tests are necessary to analyze the loop.

Automatic inlining is already provided as an option by several compilers, for instance `gcc` (option `-finline-functions`, which tries to expand only sufficiently small functions), `acc` or `f77` (option `-O4`).

Another less widely spread approach, at least in commercial compilers, is to compute summaries of the behavior of procedures, translate them as accurately as possible into the callers' or callees' name space depending on the direction of the analysis, and use the translated summaries as abstractions of procedures. These methods have the inverse advantages and consequences of inlining. They do not have its complexity limitations, since no code is duplicated, and since summaries are usually more compact than the code they abstract. Moreover, there is no restriction on the input code. However, much precision may be lost, due to summarization techniques, and translation processes.

In between, a recent techniques tries to lessen the drawbacks of interprocedural analysis, while preserving a reasonable complexity. *Selective cloning* [91, 53] duplicates procedures on the basis of calling contexts. A heuristic based on integer variable values has proved useful in subsequent array region analyses, without increasing the program size too dramatically [91]. However, even if this approach alleviates some problem, it still requires interprocedural analysis techniques.

In fact, there exists plenty of interprocedural analysis frameworks, which differ on several factors: The degree of summarization, which has been studied in the previous part for array region analyses, and the accuracy of the translation which will be dealt with in the next chapter, are among the most obvious issues. But the type of graph on which the information is conveyed, the ability to perform flow-sensitive[2] or context sensitive[3] analyses, are also of importance. These problems are not unrelated [96, 95]:

- Interprocedural analyses can be performed on the program *call graph*, where nodes represent individual procedures, and edges represent call sites. Each edge is labeled with the actual parameters associated to the corresponding call site.

  Call graphs can be constructed very efficiently [152], and can provide sufficient information for many program optimizations, and for parallelization. However, they do not allow flow-sensitive interprocedural analyses, because they do not take into account the intraprocedural control flow of individual procedures.

- To perform flow sensitive analyses, such as array region analyses, interprocedural representations of the program must therefore provide information about the internal control flow of procedures. The most precise approach is to take into account the whole internal control flow graph of every procedure. However, resulting data flow analyses may be as complex as on a fully inlined program.

[2]By interprocedural flow-sensitive analyses, we mean analyses whose results depend on the flow of control inside individual procedures; this is the case of our array regions analyses. See [126] for a more general definition.

[3]By interprocedural context sensitive analyses, we mean analyses whose result depend on the execution path taken to reach a point in the program.

To reduce this complexity, several sparse representations have been designed for particular classes of problems, such as the *program summary graph* [36], the SSA form [67, 158], the *sparse data flow evaluation graph* [41], the *system dependence graph* [75, 26], ... The *unified interprocedural graph* [96] provides a demand-driven unified representation which combines the advantages of the sparse representations without restricting the scope of the possible analyses.

- Another issue is the ability to avoid taking into account *unrealizable* paths [119]. In the graph of Figure 10.1, there are only two possible paths through procedure `foo`. However, most analyzers consider two additional paths: One from the first call to the return node for the second call; and another one from the second call to the return node for the first call. Several solutions have been proposed

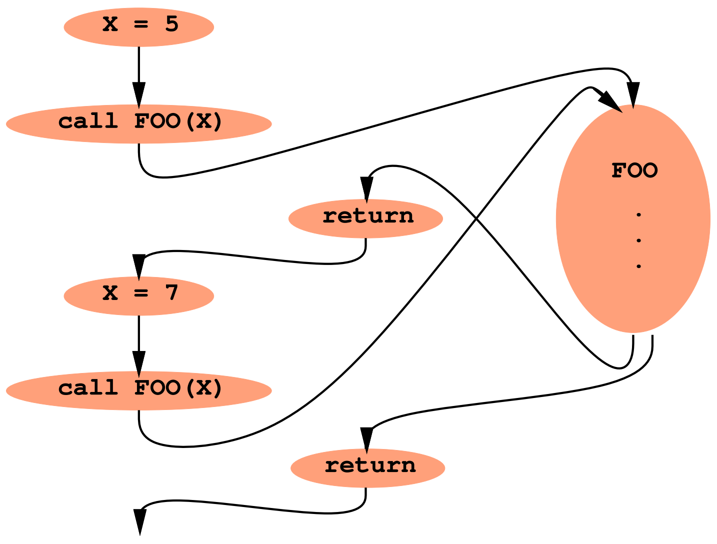


Figure 10.1: Unrealizable paths

for this problem, the most common approach being to tag solutions with path history or path specific information [134, 156, 119]. However, this may result in an exponential growth of the number of solutions at each node, and thus in overwhelming execution times and memory requirements.

10.2 Interprocedural Propagation of Regions in PIPS

PIPS interprocedural analysis mechanism allows flow sensitive analyses, since the program representation contains the individual control flow graphs of all the procedures. However, due to the choice of extensive summarization, it does not allow to avoid unrealizable paths, and to take into account path-specific information: Even if translation across procedure boundaries is performed at each call site using the correspondences between formal and actual parameters, and between `common` declarations, all information about paths to and from a procedure is merged into a single representation.

For forward analyses (see Figure 10.2), the callers are analyzed first; the information at each call site is propagated to the corresponding subroutine to form a summary;

when several call sites correspond to a single procedure, the summaries corresponding to the call sites are merged to form a unique summary. This summary is then used to perform the intraprocedural analysis of the called procedure.

For backward analyses (see Figure 10.3), the leaves of the tree are analyzed first. The procedure summaries are then used at each call site, after the translation process, during the intraprocedural analysis of the callers.

In both cases the scheduling is performed by *pipsmake* (see Appendix A). This is a fully interprocedural framework: Cloning is not implemented. In the above taxonomy, PIPS also ranges in the compilers which do not take into account path specific information, even if the context of each call is taken into account during the translation process.

The interprocedural propagation of READ, WRITE, and IN regions is a outward analysis. At each call site, the summary regions of the called subroutine are translated from the callee's name space into the caller's name space, using the relations between actual and formal parameters, and between the declarations of global variables in both routines.

On the contrary, the interprocedural propagation of OUT regions is a inward analysis. The regions of all the call sites are first translated from the callers' name space into the callee's name space, and are then merged to form a unique summary. The OUT regions of the main routine are initialized to the empty set, since a program is supposed to have no side effect, viewed from the outside (I/Os are performed from within the program itself).

10.3 Other Approaches

The purpose of earliest implementations of array region analyses was to support data dependence analysis across procedure boundaries [161, 163, 35, 37, 122, 98, 99] or between different tasks [19]. Hence, only over-approximations of effects on array element sets were required. Moreover, the necessity of symbolic analyses had not yet been brought to light, and usually, context information such as `if` conditions were not taken into account. The desired result was therefore inherently flow-insensitive.

However, because of the need for more advanced program transformations such as array privatization, flow sensitive array region analyses are now commonplace in research parallelizers [154, 100, 101, 83, 93, 92, 135, 87, 65]. In particular, the computation of upward exposed used regions, which is performed in all the previously cited parallelizers, requires a backward propagation through the internal control flow graph of individual procedures. But if sequence ordering is generally taken into account, conditional instructions are not homogeneously handled. In particular, FIDA summaries [100, 101] do not include context information other than loop and array bounds; the analysis is thus not fully flow sensitive.

Unrealizable paths are seldom avoided during array region analyses. FIAT/SUIF [92] is the only parallelizer which actually performs such path-specific analyses, through selective procedure cloning. Preliminary analyses provide the necessary context information to perform the cloning, and array region propagation is performed on the cloned call graph, using traditional methods. This method is also used in [83]. The *Hierarchical Super Graph* of PANORAMA [135, 87], which is an extension of MYERS' *supergraph* [134], also provides the necessary framework to perform path-specific anal-

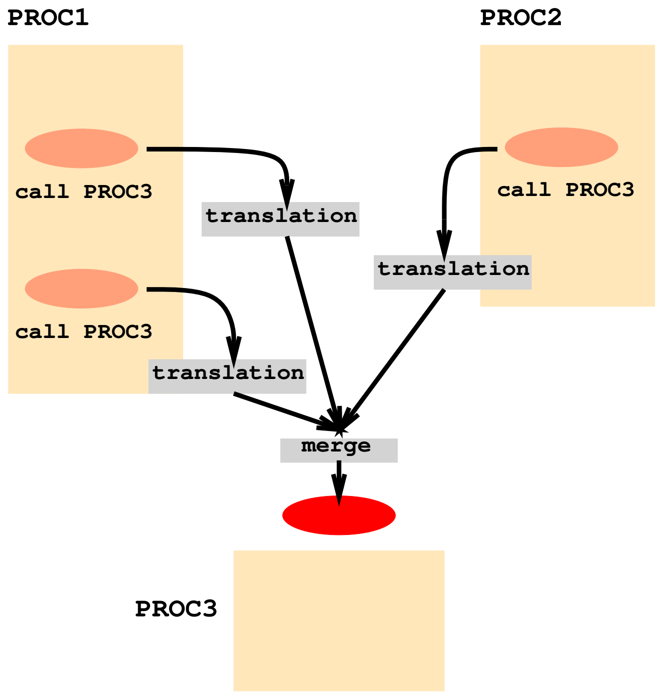


Figure 10.2: Interprocedural inward propagation

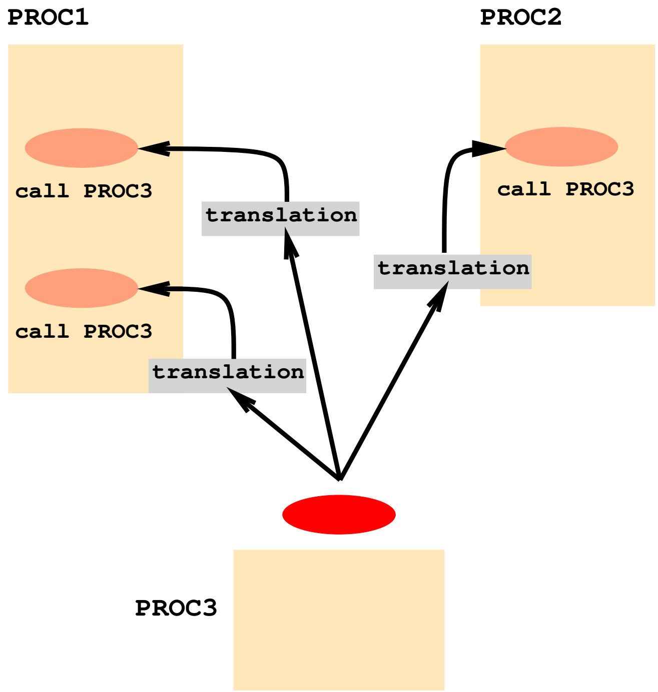


Figure 10.3: Interprocedural outward propagation

yses. And the proposed algorithm actually goes along each possible path. However, since no symbolic information is propagated downward, they do not take advantage of this possibility.

Finally, notice that despite the arguments against full inlining, it is still used in one of the most advanced parallelizers, POLARIS [28], which successfully parallelizes medium sized applications. However, they plan to implement interprocedural propagation of array regions in a near future [164].

10.4 Conclusion

It is rather difficult to compare the sole interprocedural propagation of array regions in different parallelizers, because the precision of the results also depend on several independent factors, such as the quality of preliminary analyses, the characteristics of the array region representation (whether it can include symbolic context information or not, for instance), or the precision of the translation of array region across procedure boundaries.

In this context, PIPS array region analyses rank well among the other approaches. These analyses are sensible to the internal control flow of individual procedures, and powerful preliminary analyses provide precise symbolic information. However, the chosen approach of aggressive summarization combined with a full interprocedural framework (no cloning) prevents interprocedural path differentiation, whereas other parallelizers [83, 92] achieve this goal through automatic selective procedure cloning. Since PIPS is an interactive tool, automatic cloning would not be in harmony with its design policy. However, demand-driven cloning is planned for a near future. More accurate results are expected from this transformation, especially for OUT regions which are propagated forward in the call graph. Finally, our interprocedural propagation of array regions is supported by a powerful translation algorithm which is presented in the next chapter.

Chapter 11

Interprocedural Translation

Part of this chapter is reproduced from [65]
with the kind permission of IJPP.

This chapter focuses on the translation part of array region interprocedural analyses. Because the source and target arrays may not have the same declaration (*array reshaping*), this operation is not straightforward. In particular, by examining the Perfect Club benchmarks [25], we found it necessary to handle several non-exclusive cases:

1. Array reshaping due to different dimension declarations (see Figure 11.1).

2. Offsets between the first elements of the source and target arrays due to parameter passing (see an example from the program `OCEAN` in Figure 11.2).

3. Offsets due to different `COMMON` declarations in the caller and the callee (e.g. in the program `TRFD`, the common `TR2PRT` is not similarly declared in routines `TRFPRT` and `TRFOUT`, see Figure 11.3).

4. Target and source variables of different types (e.g. in the program `OCEAN`, see Figure 11.4).

The method described in this section tackles these four points. It is based on the fact that two elements of the source and target arrays are (partially) associated if they have the same subscript values[1], up to the offset between their first element. This is described in section 11.2.

However, the resulting *translation system* may contain non-linear terms, and it hides the trivial relations existing between the ϕ variables of both arrays. Hence, we propose in section 11.3 an algorithm that first tries to discover these trivial relations before using the subscript values. It results in a *simplified translation system.*

> The contrived program in Figure 11.5 will serve as an illustration all along this chapter. It contains the main difficulties encountered in real life programs, as previously stated. Here, the purpose is to find the READ and WRITE regions of the call site, from the summary regions of procedure `BAR`.

[1] The subscript value of an array elements is its *rank* in the array, array elements being stored in column order (see [8], Section 5.4.3).

```
SUBROUTINE AMHMTM(DEL,RPI,SV,P1,P2,AM,HM,TM)

DIMENSION TM(12,12)
...
CALL IMINV(TM,12,DET,LLL,MMM)
...
END
```

```
SUBROUTINE IMINV(A,N,D,L,M)
DIMENSION A(1),L(1),M(1)
...
END
```

Figure 11.1: Different dimensions (SPC77)

```
PROGRAM OCEAN
...
REAL A(NA1,NA2)
...
CALL IN(A(1,K2Q),KZN,NWH)
...
END
```

```
SUROUTINE IN(ARRAY,LOC,NW)
DIMENSION ARRAY(1)
...
END
```

Figure 11.2: Offset due to parameter passing (OCEAN)

```
SUBROUTINE TRFPRT(NORB,NIJ,IA,TX,JFLAG)
COMMON/TR2PRT/VALINT(3),JC,JCINT(15)
...
CALL TRFOUT(I,J,K,L,NX,VAL,0)
...
END
```

```
  SUBROUTINE TRFOUT(I,J,K,L,N,VAL,NCALL)
  COMMON/TR2PRT/V(3),JC,J1(3),J2(3),J3(3),
+ J4(3),N1(3)
  ...
   V(JC), J1(JC), J2(JC), J3(JC), J4(JC)
   and N1(JC) written here
  ...
  END
```

Figure 11.3: Different COMMON declaration (TRFD)

```
PROGRAM OCEAN
...
COMPLEX CA(NW1,NA2)
...
CALL IN(CALPN,NW)
...
END
```

```
SUROUTINE IN(ARRAY,LOC,NW)
DIMENSION ARRAY(1)
...
END
```

Figure 11.4: Different types of variables (OCEAN)

```
subroutine FOO(C,n)
complex C(n,10,20),D
common D(5,10)
call BAR(C,2n,100)
end
```

```
C  <D2(φ1,φ2)-W-EXACT-{1<=φ1<=10, 1<=φ2<=9}>
C  <D1(φ1)-W-EXACT-{1<=φ1<=10}>
C  <R(φ1,φ2)-W-EXACT-{1<=φ1<=n1, 1<=φ2<=n2}>
   subroutine BAR(R,n1,n2)
   real R(n1,n2)
   common D1(10), D2(10,9)
   ...
   end
```

Figure 11.5: Interprocedural propagation: example

11.1 Notations

11.1.1 From the program

This section describes the notations used throughout Chapter 11. The purpose is to translate a region of the array A from the source procedure into a region of the array B of the target procedure. The following table summarizes the characteristics of both arrays:

	source	$\longmapsto$	*target*
array	A		B
dimension	α		β
lower bounds	$l_{a_1}, \ldots, l_{a_\alpha}$		$l_{b_1}, \ldots, l_{b_\beta}$
upper bounds	$u_{a_1}, \ldots, u_{a_\alpha}$		$u_{b_1}, \ldots, u_{b_\beta}$
size of elements[2]	s_a		s_b
region descriptors	$\phi_1, \ldots, \phi_\alpha$		$\psi_1, \ldots, \psi_\beta$

The subscript values of $A(\phi_1, \ldots, \phi_\alpha)$ and $B(\psi_1, \ldots, \psi_\beta)$ are thus[3]:

$$\mathit{subscript\_value}(A(\phi_1, \ldots, \phi_\alpha)) = \sum_{i=1}^{\alpha} [(\phi_i - l_{a_i}) \prod_{j=1}^{i-1} (u_{a_j} - l_{a_j} + 1)]$$

$$\mathit{subscript\_value}(B(\psi_1, \ldots, \psi_\beta)) = \sum_{i=1}^{\beta} [(\psi_i - l_{b_i}) \prod_{j=1}^{i-1} (u_{b_j} - l_{b_j} + 1)]$$

Another necessary information is the offset of the first element of A from the first element of B in the memory layout. This information is provided differently, depending on the type of aliasing between A and B:

[2]Unit: *numerical storage unit* (see [8], Section 4).

[3]With the convention that $\prod_{k=k_1}^{k_2} = 1$ when $k_2 < k_1$.

source parameter	$\mapsto$	*target parameter*	*offset*
formal	$\mapsto$	*actual*	reference at call site: $B(o_{b_1}, \ldots, o_{b_\beta})$ $\mathit{offset} = s_b \times \mathit{subscript\_value}(B(o_{b_1}, \ldots, o_{b_\beta}))$
actual	$\mapsto$	*formal*	reference at call site: $A(o_{a_1}, \ldots, o_{a_\alpha})$ $\mathit{offset} = -s_a \times \mathit{subscript\_value}(A(o_{a_1}, \ldots, o_{a_\alpha}))$
global	$\mapsto$	*global*	numerical offset difference between the offset of A in the declaration of the common in the source subroutine, and the offset of B in the declaration of the common in the target subroutine.

As an illustration, let us consider the contrived program in Figure 11.5. The translation coefficients are:

`R` $\mapsto$ `C`: $A =$ `R`, $B =$ `C`; $\alpha = 2$, $\beta = 3$; $l_{a_1} = l_{a_2} = 1$, $l_{b_1} = l_{b_2} = lb_3 = 1$; $u_{a_1} =$ `n1`, $u_{a_2} =$ `n2`; $u_{b_1} =$ `n`, $u_{b_2} = 10$, $u_{b_3} = 20$; $s_a = 4$, $s_b = 8$; $\mathit{offset} = 0$.

`D1` $\mapsto$ `D`: $A =$ `D1`, $B =$ `D`; $\alpha = 1$, $\beta = 2$; $l_{a_1} = 1$, $l_{b_1} = l_{b_2} = 1$; $u_{a_1} = 10$; $u_{b_1} = 5$, $u_{b_2} = 10$; $s_a = 4$, $s_b = 8$; $\mathit{offset} = 0$.

`D2` $\mapsto$ `D`: $A =$ `D2`, $B =$ `D`; $\alpha = 2$, $\beta = 2$; $l_{a_1} = l_{a_2} = 1$, $l_{b_1} = l_{b_2} = 1$; $u_{a_1} = 10$, $u_{a_2} = 9$; $u_{b_1} = 5$, $u_{b_2} = 10$; $s_a = 4$, $s_b = 8$; $\mathit{offset} = 40$.

11.1.2 Equivalent memory unit arrays

If s_a and s_b are different, or if A and B belong to a `COMMON` which does not have the same layout in the target and source procedures, the array elements that are associated are said to be *partially associated* (see [8], Section 17.1.3). This means that two corresponding elements may have non-matching storage units as shown in Figure 11.6. Identifying partially associated elements thus involves refering to individual memory

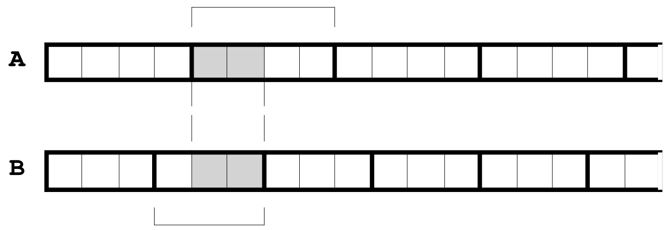


Figure 11.6: Partial association

units. In order to simplify the proofs and explanations of this chapter, we introduce the *memory unit array* A' associated to A as the array which has the same number of dimensions as A, and the same number of memory units per dimension (hence the same memory layout), but with memory unit elements instead of elements of size s_a (see Figure 11.7). We also define B' as the *memory unit array* associated to B. The characteristics of A' and B' are summarized in the following table:

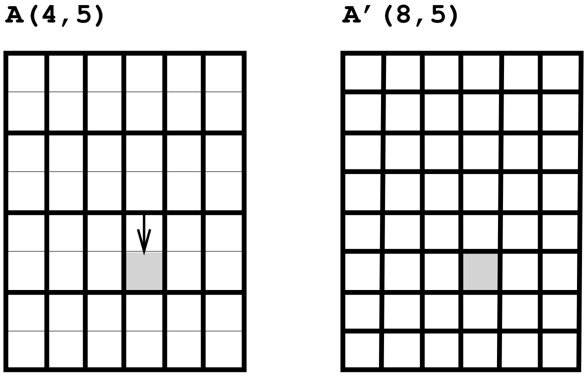


Figure 11.7: Equivalent memory unit array

	source	$\mapsto$	*target*
array	A'		B'
dimension	α		β
lower bounds	$0, l_{a_2}, \ldots, l_{a_\alpha}$		$0, l_{b_2}, \ldots, l_{b_\beta}$
upper bounds	$s_a(u_{a_1} - l_{a_1} + 1), u_{a_2}, \ldots, u_{a_\alpha}$		$s_b(u_{b_1} - l_{b_1} + 1), u_{b_2}, \ldots, u_{b_\beta}$
size of elements	1		1
descriptors	$\phi_1', \phi_2, \ldots, \phi_\alpha$		$\psi_1', \psi_2, \ldots, \psi_\beta$

Property 11.1
$A'(\phi_1', \phi_2, \ldots \phi_\alpha)$ corresponds to $A(\phi_1, \phi_2, \ldots \phi_\alpha)$ if and only if:

$$\exists \delta_a : 0 \leq \delta_a < s_a \wedge \phi_1' = s_a(\phi_1 - l_{a_1}) + \delta_a$$

Similarly $B'(\psi_1', \psi_2, \ldots, \psi_\beta)$ corresponds to $B(\psi_1, \psi_2, \ldots, \psi_\beta)$ if and only if:

$$\exists \delta_b : 0 \leq \delta_b < s_b \wedge \psi_1' = s_b(\psi_1 - l_{b_1}) + \delta_b$$

The meaning of δ variables is shown by the arrow on Figure 11.7.

Property 11.2
The subscript values of two corresponding elements of A and A' or B and B' are related by:

$$\exists \delta_a / \begin{cases} 0 \leq \delta_a < s_a \\ \phi_1' = s_a(\phi_1 - l_{a_1}) + \delta_a \\ subscript\_value(A'(\phi_1', \phi_2, \ldots \phi_\alpha)) \\ \ = s_a \times subscript\_value(A(\phi_1, \phi_2, \ldots \phi_\alpha)) + \delta_a \end{cases}$$

and,

$$\exists \delta_b / \begin{cases} 0 \leq \delta_b < s_b \\ \psi_1' = s_b(\phi_1 - l_{b_1}) + \delta_b \\ subscript\_value(B'(\psi_1', \psi_2, \ldots, \psi_\beta)) \\ \ = s_b \times subscript\_value(B(\psi_1, \psi_2, \ldots, \psi_\beta)) + \delta_b \end{cases}$$

Proof Let us assume that $A'(\phi'_1, \phi_2, \ldots, \phi_\alpha)$ corresponds to $A(\phi_1, \phi_2, \ldots, \phi_\alpha)$. Property 11.1 implies that $\exists \delta_a : 0 \leq \delta_a < s_a \wedge \phi'_1 = s_a(\phi_1 - l_{a_1}) + \delta_a$. Moreover the subscript value of $A'(\phi'_1, \phi_2, \ldots, \phi_\alpha)$ is:

$$\phi'_1 + \sum_{i=2}^{\alpha} [\ (\phi_i - l_{a_i}) \times (s_a \times (u_{a_1} - l_{a_1} + 1)) \times \prod_{j=2}^{i-1} (u_{a_j} - l_{a_j} + 1)\ ]$$

By first replacing ϕ'_1 by its value, and then factorizing s_b, the last expression is successively equal to:

$$\begin{aligned}
& s_a \times (\phi_1 - l_{a_1}) + \delta_a + \sum_{i=2}^{\alpha} [\ (\phi_i - l_{a_i}) \times (s_a \times (u_{a_1} - l_{a_1} + 1)) \times \prod_{j=2}^{i-1} (u_{a_j} - l_{a_j} + 1)\ ] \\
&= s_a \times \{ \sum_{i=1}^{\alpha} [\ (\phi_i - l_{a_i}) \times \prod_{j=1}^{i-1} (u_{a_j} - l_{a_j} + 1)\ ] \} + \delta_a \\
&= s_a \times \mathit{subscript_value}(A(\phi_1, \ldots \phi_\alpha)) + \delta_a
\end{aligned}$$

The proof is similar for B and B'. □

And finally, the offset between the first element of A' and the first element of B' is identical to *offset*, whose unit is already the numerical memory unit.

11.2 General Translation System

Theorem 11.1
With the previous notations, $B(\psi_1, \ldots, \psi_\beta)$ is partially associated to the source element $A(\phi_1, \ldots, \phi_\alpha)$ if and only if:

$$\exists\ \delta_a, \delta_b / \begin{cases} s_a \times \mathit{subscript\_value}(A(\phi_1, \ldots, \phi_\alpha)) + \delta_a + \mathit{offset} \\ \quad = s_b \times \mathit{subscript\_value}(B(\psi_1, \ldots, \psi_\beta)) + \delta_b \\ 0 \leq \delta_a < s_a \\ 0 \leq \delta_b < s_b \end{cases} \qquad (S)$$

Proof $B(\psi_1, \ldots, \psi_\beta)$ is partially associated to $A(\phi_1, \ldots, \phi_\alpha)$ if and only if there exists a memory unit in $B(\psi_1, \ldots, \psi_\beta)$ that also belongs to $A(\phi_1, \ldots, \phi_\alpha)$. With the notations of Section 11.1.2, this is strictly equivalent to:

$$\exists \delta_a, \delta_b / \begin{cases} 0 \leq \delta_a < s_a \\ 0 \leq \delta_b < s_b \\ \phi'_1 = s_a(\phi_1 - l_{a_1}) + \delta_a \\ \psi'_1 = s_b(\psi_1 - l_{b_1}) + \delta_b \\ B'(\psi'_1, \psi_2, \ldots, \psi_\beta) \text{ is totally associated to } A'(\phi'_1, \phi_2, \ldots, \phi_\alpha) \end{cases}$$

$B'(\psi'_1, \psi_2, \ldots, \psi_\beta)$ is totally associated to $A'(\phi'_1, \phi_2, \ldots, \phi_\alpha)$ if and only if their subscript values are identical (modulo the offset between their first elements):

$$\textit{subscript\_value}(A'(\phi'_1, \phi_2, \ldots \phi_\alpha)) + \textit{offset} = \textit{subscript\_value}(B'(\psi'_1, \psi_2, \ldots, \psi_\beta))$$

From Property 11.2, this can be rewritten as:

$$\begin{aligned} s_a \times \textit{subscript\_value}&(A(\phi_1, \ldots \phi_\alpha)) + \delta_a + \textit{offset} \\ &= s_b \times \textit{subscript\_value}(B(\psi_1, \ldots, \psi_\beta)) + \delta_b \end{aligned}$$

Putting all together, we finally obtain (S). $\square$

For our example, the following systems would be built:

$\mathtt{R} \mapsto \mathtt{C}$:
$$\begin{cases} 4[(\phi_1 - 1) + \mathtt{n1}(\phi_2 - 1)] + \delta_a = \\ \quad 8[(\psi_1 - 1) + \mathtt{n}(\psi_2 - 1) + 10\mathtt{n}(\psi_3 - 1)] + \delta_b \\ 0 \leq \delta_a < 4,\ 0 \leq \delta_b < 8,\ \mathtt{n1} = 2\mathtt{n} \end{cases}$$
$\mathtt{D1} \mapsto \mathtt{D}$:
$$\begin{cases} 4(\phi_1 - 1) + \delta_a = 8[(\psi_1 - 1) + 5(\psi_2 - 1)] + \delta_b \\ 0 \leq \delta_a < 4,\ 0 \leq \delta_b < 8 \end{cases}$$
$\mathtt{D2} \mapsto \mathtt{D}$:
$$\begin{cases} 4[(\phi_1 - 1) + 10(\phi_2 - 1)] + \delta_a + 40 = 8[(\psi_1 - 1) + 5(\psi_2 - 1)] + \delta_b \\ 0 \leq \delta_a < 4,\ 0 \leq \delta_b < 8 \end{cases}$$

Using (S) as the translation system has several drawbacks:

1. In the *formal* $\leftrightarrow$ *actual* cases, S is generally non-linear (it is the case in our first example).

2. In order to keep a convex representation, δ variables must be eliminated; this operation may be inexact, leading to an over-approximation.

3. Even in favorable cases, the equation in (S) is rather complex, and hides the trivial relations existing between ϕ and ψ variables, such as $\phi_1 = \psi_1$; this makes the subsequent analyses unnecessarily complex, and is not acceptable in an interactive environment where regions are displayed for the user.

The following section describes a method that alleviates these three problems.

11.3 Simplified Translation System

11.3.1 Elimination of unnecessary δ variables

Theorem 11.2
If s_b divides both s_a and offset, then (S) is equivalent to the following system[4]*:*

$$\exists\, \delta'_a / \begin{cases} s'_a \times subscript\_value(A(\phi_1,\ldots,\phi_\alpha)) + \delta'_a + \dfrac{offset}{s_b} \\ = subscript\_value(B(\psi_1,\ldots,\psi_\beta)) \\ 0 \leq \delta'_a < s'_a \\ s'_a = \dfrac{s_a}{s_b} \end{cases}$$

Note

1. In the *formal* $\leftrightarrow$ *actual* cases, (s_b divides s_a) $\Longrightarrow$ (s_b divides *offset*).
2. In fact, we just replace s_a by $\frac{s_a}{s_b}$, s_b by 1, *offset* by $\frac{offset}{s_b}$ and use (S) without δ_b.
3. An interesting (and very frequent) case is when $s_a = s_b$. (S) is then equivalent to:

$$subscript\_value(A(\phi_1,\ldots,\phi_\alpha)) + \frac{offset}{s_b} = subscript\_value(B(\psi_1,\ldots,\psi_\beta))$$

Proof

$$s_b \text{ divides } s_a \text{ and } offset \Longrightarrow \begin{cases} s_a \times subscript\_value(A(\phi_1,\ldots,\phi_\alpha)) + offset \\ = s_b \times \dfrac{s_a \times subscript\_value(A(\phi_1,\ldots,\phi_\alpha)) + offset}{sb} \end{cases}$$

Let δ'_a be $\frac{\delta_a}{s_b} + r_a$ such that $\delta_a = s_b \times \delta'_a$. Let δ'_b be $\frac{\delta_b}{s_b} + r_b$ such that $\delta_b = s_b \times \delta'_b$.

$$\begin{cases} 0 \leq \delta_a < s_a \\ 0 \leq \delta_b < s_b \end{cases} \Longrightarrow \begin{cases} 0 \leq \delta'_a < \dfrac{s_a}{s_b} \\ 0 \leq \delta'_b < \dfrac{s_b}{s_b} \end{cases} \Longrightarrow \begin{cases} 0 \leq \delta'_a < \dfrac{s_a}{s_b} \\ \delta'_b = 0 \end{cases}$$

Then, (S) is equivalent to:

$$\exists\, \delta'_a, \delta'_b / \begin{cases} s_b \times \dfrac{s_a \times subscript\_value(A(\phi_1,\ldots,\phi_\alpha)) + offset}{sb} + s_b \times \delta'_a \\ = s_b \times subscript\_value(B(\psi_1,\ldots,\psi_\beta)) + s_b \times \delta'_b \\ 0 \leq \delta'_a < \dfrac{s_a}{s_b} \\ \delta'_b = 0 \end{cases}$$

[4] Of course, there is a similar system if s_a divides s_b and *offset*.

And finally, with $s'_a = \frac{s_a}{s_b}$, and after substitution of δ'_b by its value and division of the main equation by s_b, (S) is equivalent to:

$$\exists\, \delta'_a / \begin{cases} s'_a \times subscript\_value(A(\phi_1, \ldots, \phi_\alpha)) + \dfrac{offset}{sb} + \delta'_a \\ = subscript\_value(B(\psi_1, \ldots, \psi_\beta)) \\ 0 \le \delta'_a < s'_a \\ s'_a = \dfrac{s_a}{s_b} \end{cases}$$

□

In our working example, since s_a divides s_b and *offset* in all three cases, the translation systems become:

`R` $\mapsto$ `C`*:*

$$\begin{cases} (\phi_1 - 1) + \mathtt{n1}(\phi_2 - 1) = \\ \quad 2[(\psi_1 - 1) + \mathtt{n}(\psi_2 - 1) + 10\mathtt{n}(\psi_3 - 1)] + \delta_b \\ 0 \le \delta_b < 2,\ \mathtt{n1} = 2\mathtt{n} \end{cases}$$

`D1` $\mapsto$ `D`*:*

$$\begin{cases} \phi_1 - 1 = 2[(\psi_1 - 1) + 5(\psi_2 - 1)] + \delta_b \\ 0 \le \delta_b < 2 \end{cases}$$

`D2` $\mapsto$ `D`*:*

$$\begin{cases} (\phi_1 - 1) + 10(\phi_2 - 1) + 10 = 2[(\psi_1 - 1) + 5(\psi_2 - 1)] + \delta_b \\ 0 \le \delta_b < 2 \end{cases}$$

11.3.2 Decreasing the degree of (S)

We have shown how to eliminate δ variables. But the main equation of the translation system may still be non-linear. We first give some definitions and notations, and then show how to decrease the degree of (S) without losing information.

Definition 11.1
A dimension d ($d \le min(\alpha, \beta)$) is said to be similar for arrays A and B if the following three conditions are met:

1. *Condition for the offset:*

   There must be no offset between the first element of B and the first element of A on dimension d:

formal	$\mapsto$	*actual*	$\forall\, i/1 \le i \le d,\ o_{b_i} = l_{b_i}$
actual	$\mapsto$	*formal*	$\forall\, i/1 \le i \le d,\ o_{a_i} = l_{a_i}$
global	$\mapsto$	*global*	$\lvert offset \rvert \bmod s_a \prod_{i=1}^{d}(u_{a_i} - l_{a_i} + 1) = 0$ $\wedge \lvert offset \rvert \bmod s_b \prod_{i=1}^{d}(u_{b_i} - l_{b_i} + 1) = 0$

2. *Condition for the first dimension:*
   The lengths in bytes of the first dimensions of A and B are equal:
   $$s_a(u_{a_d} - l_{a_d} + 1) = s_b(u_{b_d} - l_{b_d} + 1)$$
   This means that the first dimension entirely compensates the difference between s_a and s_b. This is why s_a and s_b are not used in the next condition.
3. *Condition for the next dimensions ($2 \leq d \leq min(\alpha, \beta)$):*
   Assuming that the previous dimensions are similar, the lengths of the d-th dimensions of A and B must be equal:
   $$u_{a_d} - l_{a_d} = u_{b_d} - l_{b_d}$$
   This is not necessary if $d = \alpha = \beta$.

This definition only takes into account dimensions of identical ranks. The general case would try to discover minimal sets of globally similar dimensions.

> For instance if the declarations of A and B are $A(l, m, n)$ and $B(m, l, n)$, the global lengths of dimensions 1 and 2 are similar (dimensions 1 and 2 are globally similar); as a consequence, the third dimension is similar.

But the complexity of the algorithm for discovering these sets would be too high compared to the expected gain, especially in real life programs.

Notations 11.1
We now use the following notations for $k \in [2..min(\alpha, \beta)]$:

k_subscript_value:

$$k\_subscript\_value(A(\phi_1, \ldots, \phi_\alpha)) = \sum_{i=k}^{\alpha}[(\phi_i - l_{a_i})\prod_{j=k}^{i-1}(u_{a_j} - l_{a_j} + 1)]$$

It is the rank of the array element $A(\phi_1, \ldots, \phi_\alpha)$ from the element $A(\phi_1, \ldots, \phi_{k-1}, l_{a_k}, \ldots, l_{a_\alpha})$, i.e. from the first element of the k-th dimension.

k_offset:

It is the offset relative to the k-th dimension:

formal	$\mapsto$	*actual*	$k\_subscript\_value(B(o_{b_1}, \ldots, o_{b_\beta}))$
actual	$\mapsto$	*formal*	$-k\_subscript\_value(A(o_{a_1}, \ldots, o_{a_\alpha}))$
global	$\mapsto$	*global*	$\left\lfloor \frac{offset}{s_a \prod_{i=1}^{k}(u_{a_i} - l_{a_i} + 1)} \right\rfloor$

> The number of grey elements in the bidimensionnal array of Figure 11.8 represents the 2_*subscript_value* of the black element $(2, 3)$. The arrow shows the element $(2, 1)$ from which the 2_*subscript_value* is computed.

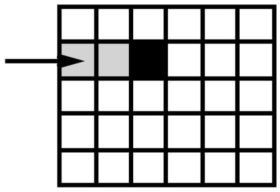


Figure 11.8: $k\_subscript\_value$

Note We trivially have:

$$k\_subscript\_value(A'(\phi'_1, \phi_2, \ldots, \phi_\alpha)) = k\_subscript\_value(A(\phi_1, \phi_2, \ldots, \phi_\alpha))$$

and,

$$k\_subscript\_value(B(\psi'_1, \psi_2, \ldots, \psi_\beta)) = k\_subscript\_value(B(\psi_1, \psi_2, \ldots, \psi_\beta))$$

Theorem 11.3
If dimensions 1 to $d-1$ $(1 \leq d-1 \leq min(\alpha, \beta))$ are similar, then (S) is equivalent to:

$$\exists\, \delta_a, \delta_b / \begin{cases} s_a(\phi_1 - l_{a_1}) + \delta_a = s_b(\psi_1 - l_{b_1}) + \delta_b \\ \forall\, i \in [2..d-1], \quad \phi_i - l_{a_i} = \psi_i - l_{b_i} \\ d\_subscript\_value(A(\phi_1, \ldots, \phi_\alpha)) + d\_offset = \\ \quad d\_subscript\_value(B(\psi_1, \ldots, \psi_\beta))^5 \\ 0 \leq \delta_a < s_a \\ 0 \leq \delta_b < s_b \end{cases} \qquad (S_d)$$

In order to prove this theorem, we first consider two lemma.

Lemma 11.3.1
If dimensions 1 to $d-1$ $(1 \leq d-1 \leq min(\alpha, \beta))$ are similar, then:

$$\forall\, k \in [1..d-1],\ s_a \prod_{j=1}^{j=k} (u_{a_j} - l_{a_j} + 1) = s_b \prod_{j=1}^{j=k} (u_{b_j} - l_{b_j} + 1)$$

Proof The third condition in Definition 11.1 is:

$$\forall\, j \in [2..d-1],\ u_{a_j} - l_{a_j} = u_{b_j} - l_{b_j}$$

This implies:

$$\forall\, k \in [2..d-1],\ \prod_{j=2}^{k} (u_{a_j} - l_{a_j} + 1) = \prod_{j=2}^{k} (u_{b_j} - l_{b_j} + 1)$$

[5] In the formal $\mapsto$ actual case, if $d = min(\alpha, \beta) = \alpha$, this equation can be replaced by $\forall\, i \in [d..\beta],\ \psi_i = o_{b_i}$.

The second condition in Definition 11.1 gives:

$$s_a \times (u_{a_1} - l_{a_1} + 1) = s_b \times (u_{b_1} - l_{b_1} + 1)$$

From the two previous relations, we deduce that, $\forall\, k \in [2..d-1]$,

$$s_a \times (u_{a_1} - l_{a_1} + 1) \times \prod_{j=2}^{k} (u_{a_j} - l_{a_j} + 1)$$
$$= s_b \times (u_{b_1} - l_{b_1} + 1) \times \prod_{j=2}^{k} (u_{b_j} - l_{b_j} + 1)$$

And finally:

$$\forall\, k \in [1..d-1],\ s_a \times \prod_{j=1}^{k} (u_{a_j} - l_{a_j} + 1) = s_b \times \prod_{j=1}^{k} (u_{b_j} - l_{b_j} + 1)$$

$\square$

Lemma 11.3.2
If dimensions 1 to $d-1$ ($1 \leq d-1 \leq min(\alpha, \beta)$) are similar, then

$$offset = s_a \times \prod_{j=1}^{d-1} (u_{a_j} - l_{a_j} + 1) \times d\_offset = s_b \times \prod_{j=1}^{d-1} (u_{b_j} - l_{b_j} + 1) \times d\_offset$$

Proof There are three cases:

1. formal $\mapsto$ actual:

$$\begin{aligned} offset &= s_b \times subscript\_value(B(o_{b_1}, \ldots, o_{b_\beta})) \\ &= s_b \times \{ \sum_{i=1}^{d-1} [(o_{b_i} - l_{b_i}) \textstyle\prod_{j=1}^{i-1} (u_{b_j} - l_{b_j} + 1)] \\ &\quad + \sum_{i=d}^{\beta} [(o_{b_i} - l_{b_i}) \textstyle\prod_{j=1}^{i-1} (u_{b_j} - l_{b_j} + 1)] \} \end{aligned}$$

From definition 11.1, we know that $\forall\, i \in [1..d-1],\ o_{b_i} = l_{b_i}$. Thus,

$$\begin{aligned}
\mathit{offset} &= s_b \times \{\sum_{i=d}^{\beta}[(o_{b_i} - l_{b_i})\textstyle\prod_{j=1}^{i-1}(u_{b_j} - l_{b_j} + 1)]\} \\
&= s_b \times \{\sum_{i=d}^{\beta}[(o_{b_i} - l_{b_i})\textstyle\prod_{j=1}^{d-1}(u_{b_j} - l_{b_j} + 1) \times \prod_{j=d}^{i-1}(u_{b_j} - l_{b_j} + 1)]\} \\
&= s_b \times \prod_{j=1}^{d-1}(u_{b_j} - l_{b_j} + 1) \times \{\sum_{i=d}^{\beta}[(o_{b_i} - l_{b_i})\textstyle\prod_{j=d}^{i-1}(u_{b_j} - l_{b_j} + 1)]\} \\
&= s_b \times \prod_{j=1}^{d-1}(u_{b_j} - l_{b_j} + 1) \times \mathit{d_offset}
\end{aligned}$$

From Lemma 11.3.1, $s_b \times \prod_{j=1}^{d-1}(u_{b_j} - l_{b_j} + 1) = s_a \times \prod_{j=1}^{d-1}(u_{a_j} - l_{a_j} + 1)$; thus:

$$\mathit{offset} = s_b \times \prod_{j=1}^{d-1}(u_{b_j} - l_{b_j} + 1) \times \mathit{d\_offset} = s_a \times \prod_{j=1}^{d-1}(u_{a_j} - l_{a_j} + 1) \times \mathit{d\_offset}$$

2. actual $\mapsto$ formal: the proof is similar to the previous case.

3. global $\mapsto$ global:

   We know from Definition 11.1 that

$$\mathit{offset} \bmod s_a \prod_{j=1}^{d-1}(u_{a_j} - l_{a_j} + 1) = \mathit{offset} \bmod s_b \prod_{j=1}^{d-1}(u_{b_j} - l_{b_j} + 1) = 0$$

   This implies:

$$\begin{aligned}
\mathit{offset} &= s_a \times \prod_{j=1}^{d-1}(u_{a_j} - l_{a_j} + 1) \times \frac{\mathit{offset}}{s_a \prod_{j=1}^{d-1}(u_{a_j} - l_{a_j} + 1)} \\
&= s_b \times \prod_{j=1}^{d-1}(u_{b_j} - l_{b_j} + 1) \times \frac{\mathit{offset}}{s_b \prod_{j=1}^{d-1}(u_{b_j} - l_{b_j} + 1)}
\end{aligned}$$

   Finally, from the definition of *d_offset*:

$$\mathit{offset} = s_a \times \prod_{j=1}^{d-1}(u_{a_j} - l_{a_j} + 1) \times \mathit{d\_offset} = s_b \times \prod_{j=1}^{d-1}(u_{b_j} - l_{b_j} + 1) \times \mathit{d\_offset}$$

□

Proof (Theorem 11.3)

Let us assume that dimensions 1 to $d-1$ ($1 \leq d-1 \leq min(\alpha, \beta)$) are similar for arrays A and B (and thus for A' and B').

$B(\psi_1, \ldots, \psi_\beta)$ is partially associated to $A(\phi_1, \ldots, \phi_\alpha)$ if and only if[6]:

$$\exists \delta_a, \delta_b / \begin{cases} 0 \leq \delta_a < s_a \\ 0 \leq \delta_b < s_b \\ \phi'_1 = s_a(\phi_1 - l_{a_1}) + \delta_a \\ \psi'_1 = s_b(\psi_1 - l_{b_1}) + \delta_b \\ B'(\psi'_1, \psi_2, \ldots, \psi_\beta) \text{ is totally associated to } A'(\phi'_1, \phi_2, \ldots, \phi_\alpha) \end{cases}$$

A' and B' have the same first dimension. This implies that two corresponding elements have the same rank:

$$\phi'_1 = \psi'_1 \tag{11.4}$$

And thus, from the relations between ϕ'_1 and ϕ_1, and ψ'_1 and ψ_1:

$$\exists \delta_a, \delta_b / \begin{array}{l} 0 \leq \delta_a < s_a \\ 0 \leq \delta_b < s_b \\ s_a(\phi_1 - l_{a_1}) + \delta_a = s_b(\psi_1 - l_{b_1}) + \delta_b \end{array} \tag{11.5}$$

A' and B' have the same i-th dimension, whenever $i \in [2..d-1]$. This similarly implies that:

$$\forall\, i \in [2..d-1],\ \phi_i - l_{a_i} = \psi_i - l_{b_i}$$

$B'(\psi'_1, \psi_2, \ldots, \psi_\beta)$ is totally associated to $A'(\phi'_1, \phi_2, \ldots, \phi_\alpha)$ if and only if their subscript values are identical (modulo the offset between their first elements):

$$subscript\_value(A'(\phi'_1, \phi_2, \ldots \phi_\alpha)) + offset = subscript\_value(B'(\psi'_1, \psi_2, \ldots, \psi_\beta))$$

This can be rewritten as:

$$\begin{aligned} &\phi'_1 + \sum_{i=2}^{\alpha}[\ (\phi_i - l_{a_i}) \times (s_a \times (u_{a_1} - l_{a_1} + 1)) \times \textstyle\prod_{j=2}^{i-1}(u_{a_j} - l_{a_j} + 1)\ ] + offset \\ &\qquad = \psi'_1 + \sum_{i=2}^{\beta}[\ (\psi_i - l_{b_i}) \times (s_b \times (u_{b_1} - l_{b_1} + 1)) \times \textstyle\prod_{j=2}^{i-1}(u_{b_j} - l_{b_j} + 1)\ ] \end{aligned} \tag{11.6}$$

With Equations (11.4) and (11.5), and from Lemma 11.3.1 we know that:

$$\begin{aligned} &\phi'_1 + \sum_{i=2}^{d-1}[(\phi_i - l_{a_i}) \times (s_a \times (u_{a_1} - l_{a_1} + 1)) \times \textstyle\prod_{j=2}^{i-1}(u_{a_j} - l_{a_j} + 1)] = \\ &\qquad \psi'_1 + \sum_{i=2}^{d-1}[(\psi_i - l_{b_i}) \times (s_b \times (u_{b_1} - l_{b_1} + 1)) \times \textstyle\prod_{j=2}^{i-1}(u_{b_j} - l_{b_j} + 1)] \end{aligned}$$

[6]See the proof of Theorem 11.1.

Equation 11.6 then becomes:

$$s_a \times \sum_{i=d}^{\alpha}[\ (\phi_i - l_{a_i}) \times \prod_{j=1}^{i-1}(u_{a_j} - l_{a_j} + 1)\ ] + \mathit{offset}$$
$$= s_b \times \sum_{i=d}^{\beta}[\ (\psi_i - l_{b_i}) \times \prod_{j=2}^{i-1}(u_{b_j} - l_{b_j} + 1)\ ]$$

We factorize $\prod_{j=1}^{d-1}(u_{a_j} - l_{a_j} + 1)$ in the first part of the previous equation, and $\prod_{j=1}^{d-1}(u_{b_j} - l_{b_j} + 1)$ in the second part:

$$s_a \times \prod_{j=1}^{d-1}(u_{a_j} - l_{a_j} + 1) \times \sum_{i=d}^{\alpha}[(\phi_i - l_{a_i})\prod_{j=d}^{i-1}(u_{a_j} - l_{a_j} + 1)] + \mathit{offset}$$
$$= s_b \times \prod_{j=1}^{d-1}(u_{b_j} - l_{b_j} + 1) \times \sum_{i=d}^{\beta}[(\phi_i - l_{b_i})\prod_{j=d}^{i-1}(u_{b_j} - l_{b_j} + 1)]$$

From Lemma 11.3.2, we know that

$$\mathit{offset} = s_a \times \prod_{j=1}^{d-1}(u_{a_j} - l_{a_j} + 1) \times \mathit{d\_offset} = s_b \times \prod_{j=1}^{d-1}(u_{b_j} - l_{b_j} + 1) \times \mathit{d\_offset}$$

Thus, Equation 11.6 is equivalent to:

$$s_a \times \prod_{j=1}^{d-1}(u_{a_j} - l_{a_j} + 1) \times [\ \mathit{d\_subscript\_value}(A(\phi_1, \ldots, \phi_\alpha) + \mathit{d\_offset}\,]$$
$$= s_b \times \prod_{j=1}^{d-1}(u_{b_j} - l_{b_j} + 1) \times \mathit{d\_subscript\_value}(B(\psi_1, \ldots, \psi_\beta)) \quad (11.10)$$

Finally, since $s_a \times \prod_{j=1}^{d-1}(u_{a_j} - l_{a_j} + 1) = s_b \times \prod_{j=1}^{d-1}(u_{b_j} - l_{b_j} + 1)$ (Lemma 11.3.1), we can divide on both sides by $s_a \times \prod_{j=1}^{d-1}(u_{a_j} - l_{a_j} + 1)$:

$$\mathit{d\_subscript\_value}(A(\phi_1, \ldots, \phi_\alpha) + \mathit{d\_offset} = \mathit{d\_subscript\_value}(B(\psi_1, \ldots, \psi_\beta)) \quad (11.11)$$

In conclusion, putting together (11.4), (11.5) and (11.11), (S) is equivalent to (S_d).
□

In our working example, the translation systems finally become:

R $\mapsto$ C:

$$\begin{cases} \phi_1 - 1 = 2(\psi_1 - 1) + \delta_b \\ \phi_2 - 1 = (\psi_2 - 1) + 10(\psi_3 - 1) \\ 0 \le \delta_b < 2 \end{cases}$$

Notice that the system now only contains linear equations.

`D1` $\mapsto$ `D`:

$$\begin{cases} \phi_1 - 1 = 2(\psi_1 - 1) + \delta_b \\ (\psi_2 - 1) = 0 \\ 0 \le \delta_b < 2 \end{cases}$$

There are now only very simple relations between ϕ and ψ variables. In particular, it becomes obvious that $\psi_2 = 1$, which was hidden in the original system.

`D2` $\mapsto$ `D`:

$$\begin{cases} \phi_1 - 1 = 2(\psi_1 - 1) + \delta_b \\ (\phi_2 - 1) + 1 = (\psi_2 - 1) \\ 0 \le \delta_b < 2 \end{cases}$$

Notice how the offset for the whole problem has been turned into an offset for the sole second dimension (the term 1 in the second equation).

Algorithm 11.1

1. `input: a region` R_A `corresponding to the array` A
2. $R_B \;=\; R_A$
3. $d = \mathit{number\_of\_similar\_dimensions}(A, B) + 1$
4. `if` $d = 1$ `then`
5. $\mathit{translation\_system} = S$
6. `else`
7. $\mathit{translation\_system} = S_d$
8. `endif`
9. `add` $\mathit{translation\_system}$ `to` R_B
10. `eliminate` δ `variables`
11. `eliminate` ϕ `variables`
12. `rename` ψ `variables into` ϕ `variables`
13. `translate` R_B`'s polyhedron into`
    `the target routine's name space`
14. `for all` $i \in [1..\beta]$ `add` $l_{b_i} \le \phi_i \le u_{b_i}$ `to` R_B
15. `output:` R_B

At each step, the exactness of the current operation is checked. At Step 3, if an intermediate expression used to check the similarity is not linear, the current dimension is declared as non-similar, and the next dimensions are not considered. At Steps 5 and 7, if a constraint cannot be built because of a non-linear term, it is not used (this leads to an over-approximation of the solution set), and the translation is declared inexact. At Steps 10 and 11, the exactness of the variable elimination is verified with the usual conditions [12, 144].

Step 13 is performed using the relations between formal and actual parameters, and between the declarations of global variables in the source and target routines (this gives a *translation context system*). The variables belonging to the name space of the source routine are then eliminated. These relations are modeled by a convex polyhedron which can be seen as a transformer between the source and target name spaces. The operation which is performed has therefore the same nature as the composition of a region by a transformer, and its exactness is checked in the same way.

The last step is particularly useful in case of a partial matching between A and B, which is the case when A and B belong to a `COMMON` that is not similarly declared in the source and target routine.

11.4 Related Work

The previous work closest to ours for the interprocedural translation are those of Triolet [161], Tang [159] Hall [93], Leservot [121], Burke and Cytron [35] and Maslov [128].

Many other less recent works [37, 99, 19] have addressed the problem of the interprocedural propagation of array element sets. But they did not provide sufficient symbolic analyses, and did not tackle array reshaping.

Triolet **[161]**

In his thesis, Triolet addressed the problem of interprocedural translation in a very limited way: No array reshaping, except when due to an offset in the actual parameter to represent a column in a matrix for instance; and the `COMMON`s in the caller and callee had to be similarly declared.

Tang **[159]**

Tang summarizes multiple array references in the form of an integer programming problem. It provides exact solutions, but the source language is very restricted, and array reshaping is only handled in very simple cases (sub-arrays, as Triolet [161]).

Hall ***et al.*** **[93]**

Fiat/Suif includes an intra- and inter-procedural framework for the analysis of array variables. Approximations of array elements sets are represented by lists of polyhedra.

For the interprocedural translation, they have adopted a method basically similar to ours. However, similar dimensions are taken into account only when the system is not linear; and in this case, they do not try to build a system similar to S_d (see Page 197), possibly missing a linear translation system. Moreover, they do not handle global $\mapsto$ global translation when the `COMMON` to which the source and target arrays belong, does not have the same data layout in the caller and callee. We think it may be because their formulae do not separately handle the subscript values of individual array elements and the offset between the first elements of the source and target arrays.

LESERVOT [**121**]

LESERVOT has extended FEAUTRIER's array data flow analysis [72] to handle procedure calls. However, the constraint of the static control program still holds. This implies that no partial association is allowed at procedure boundaries (i.e. the source and target arrays have the same type); furthermore, only very simple array reshapes are handled (the same cases as in [161] and [159]).

BURKE **and** CYTRON [**35**]

They alleviate the memory disambiguation problem by linearizing all array accesses when possible. This is equivalent to using (S) in our method. However, we have seen that this may lead to non linear expressions, that may prevent further dependence testing for instance. On the contrary, our method avoids linearization whenever possible by detecting similar dimensions, and partially linearizing the remaining dimensions if possible and necessary. This approach eliminates the linearization versus subscript-by-subscript problem as formulated by BURKE and CYTRON.

DUMAY [**68**]

DUMAY proposes a method (named *contextual linearization*) to solve systems of non linear constraints. It basically replaces non-linear terms by single dummy variables, which are further constrained by the context provided by other constraints of the system.

This method could not be used as such for exact array regions because it does not preserve the exactness of the representation. It could however be used after all else failed, to preserve some information about region parameters.

MASLOV [**128**]

MASLOV describes a very general method for simplifying systems containing polynomial constraints. This is the case of the general translation system presented in Section 11.3. The advantage of MASLOV's method when it succeeds, is that it generates a linear system which has exactly the same solutions as the original non linear system.

However, we think that most cases that arise in real life programs and that can be solved using MASLOV's method can also be solved by our algorithm, thus avoiding the cost of a more general method; for instance, the translation from `A(N,M,L)` to `B(N,M,L)` yields the equation $\psi_1 + \mathtt{N}\psi_2 + \mathtt{NM}\psi_3 = \phi_1 + \mathtt{N}\phi_2 + \mathtt{NM}\phi_3$ which he gives as an example; we solve it by simply verifying that all three dimensions are similar.

11.5 Conclusion

One of the primary interests of array region analyses remains their ability to deal with programs made of several procedures. This requires translation algorithms between formal and actual parameter regions powerful enough to handle the frequent array reshapes which appear in real life FORTRAN programs.

In this context, the algorithm presented in this chapter appears as a synthesis and extension of previously existing linear methods, such as the detection of trivial cases — which do not cover all the types of array reshape — or full linearization — which often results in non-linear expressions:

- *Synthesis* because it provides a unified linear framework to handle all the cases also solved by these two techniques, which can be viewed as the extreme cases of our algorithm.
- *Extension* because it broadens their application field: The detection of trivial relations between the region descriptors of the source and target arrays combined with partial linearization alleviates the drawbacks of full linearization and preserves the visual form of the region in the case of an interactive compiler; `COMMON`s declared differently in the source and target procedures are handled, as well as arrays of different types. In these last two cases, linearization is avoided as much as possible.

However, our algorithm does not handle all possible practical cases, such as the translation between arrays declared as `MAT(N,M)` and `V(N*M)`. To settle such problems, pattern matching or non-linear techniques are necessary, ideally polynomial methods. Unfortunately, the resulting regions would contain polynomial integer constraints, and all array region analyses, as well as subsequent analyses and program transformations, would have to be extended to this domain, with the unavoidable complexity it generates.

V

APPLICATIONS

Chapitre 12

Applications dans PIPS

(Résumé des chapitres 13, et 14)

Comme nous l'avons déjà indiqué dans l'introduction, les applications possibles des analyses de régions de tableaux sont nombreuses. Mais leur domaine d'utilisation privilégié reste celui de la parallélisation automatique. Les deux applications qui génèrent les gains de performance les plus importants sont l'analyse des dépendances interprocédurales et la privatisation de tableaux [29].

TRIOLET proposait d'utiliser les régions READ et WRITE sur-estimées pour effectuer des analyses de dépendances interprocédurales. Sa méthode fut d'abord implantée dans PARAFRASE [162], puis dans PIPS [105, 139, 177]. Nous présentons dans une première section l'implantation existant actuellement dans PIPS. En particulier, nous proposons une solutions aux problèmes dus à l'utilisation de résumés, et discutons les apports éventuels des régions IN et OUT. Dans le chapitre 13, nous proposons aussi, mais de manière plus anecdotique, d'utiliser les régions de tableaux pour améliorer le traitement des variables d'induction lors des tests de dépendance de PIPS.

Dans une deuxième section, nous proposons un nouvel algorithme de privatisation de tableaux utilisant les régions IN et OUT, et prenant en compte les problèmes d'initialisation du tableau privatisé et de mise à jour du tableau initial lorsque cela est nécessaire.

12.1 Analyse des Dépendances

Dans PIPS, le graphe de dépendances utilisé pour la génération de code est construit en deux étapes (voir figure 16.3 à la page 253) :

1. Tout d'abord, le graphe des *chaînes* [1, 133] est construit à partir des effets des instructions, ou des régions de tableaux READ et WRITE, qui expriment aussi ces effets, mais de manière plus précise ; ceci est permis par l'utilisation d'une même structure de donnée.

2. Puis ce graphe est raffiné par l'application successive de tests de dépendance de complexité graduelle. Les tests les plus complexes sont basés essentiellement sur des tests de faisabilité de polyèdres convexes, comme nous le verrons ultérieurement sur des exemples.

Nous montrons dans la section 12.1.1 comment les régions READ et WRITE peuvent être utilisées pour effectuer ces tests de dépendance. Puis nous proposons une méthode pour éviter les effets de l'utilisation systématique de régions résumées. Enfin, nous envisagerons dans la section 12.2 l'utilisation des régions IN et OUT pour effectuer ces tests de dépendance.

12.1.1 Utilisation des régions READ et WRITE

Les sur-approximations des régions READ et WRITE fournissent une information de même nature que les effets READ et WRITE utilisés dans PIPS pour l'étude des dépendances intraprocédurales [177]. Il peuvent donc également être utilisés pour représenter les dépendances. Le test de dépendance consiste alors à étudier si deux régions $\mathcal{R}_1$ et $\mathcal{R}_2$ correspondant à deux vecteurs d'itération différents $\vec{I}+\vec{I_1}$ et $\vec{I}+d\vec{I}+\vec{I_2}$ ($\vec{I}$ étant le vecteur d'itération des boucles communes), peuvent contenir des éléments de tableaux identiques :

$$\mathcal{R}_1(\vec{I}+\vec{I_1}) \cap \mathcal{R}_2(\vec{I}+d\vec{I}+\vec{I_2}) \stackrel{?}{=} \varnothing$$

Des informations de contexte (les *préconditions*) peuvent éventuellement être prises en compte ; dans l'implantation actuelle, elles sont déjà incluses dans les polyèdres décrivant les régions, et n'ont donc pas besoin d'être considérées au niveau du test de dépendance.

Une fois ce système de dépendance construit, il ne reste plus qu'à tester sa faisabilité.

Considérons l'exemple de la procédure EXTR dans le programme de la figure 2.1 à la page 14. Il existe une dépendance potentielle entre l'ensemble des éléments de tableaux lus lors de la première invocation de la fonction D, représentée par la région (avec $\vec{I}=j$) :

$$\mathcal{R}_1(j) = \left\{ \begin{array}{l} \phi_1 = j \\ \mathtt{K} \leq \phi_2 \leq 1+\mathtt{K} \\ \mathtt{L} \leq \phi_3 \leq 2+\mathtt{L} \\ \phi_1 \leq 52 \\ \phi_2 \leq 21 \\ 1 \leq \phi_3 \leq 60 \\ \mathtt{J2} = 2\mathtt{JA}-1 \\ \mathtt{K} = \mathtt{K1} \\ \mathtt{L} = \mathtt{NI} \\ \mathtt{J1} \leq j \\ j \leq \mathtt{JA} \\ 2 \leq \mathtt{J1} \\ 2 \leq \mathtt{K1} \end{array} \right.$$

et la référence en écriture à T(J,1,NC+3), représentée par :

$$\mathcal{R}_2(j+dj) = \begin{cases} \phi_1 = j + dj \\ \phi_2 = 1 \\ \phi_3 = \mathtt{NC+3} \\ \mathtt{J2} = \mathtt{2JA-1} \\ \mathtt{K} = \mathtt{K1} \\ \mathtt{L=NI} \\ \mathtt{J1} \leq j + dj \\ j + dj \leq \mathtt{JA} \\ 2 \leq \mathtt{J1} \\ 2 \leq \mathtt{K1} \end{cases}$$

Un test de faisabilité sur l'intersection des régions donne alors l'ensemble vide :

$$\forall j, \mathcal{R}_1(j) \cap \mathcal{R}_2(j+dj) = \varnothing$$

Ceci provient simplement du fait que dans la première région $\phi_2 \geq \mathtt{K} \geq \mathtt{K1} \geq 2$, alors que dans la seconde $\phi_2 = 1$. La dépendance considérée n'existe donc pas réellement, et peut être enlevée du graphe des dépendances.

L'avantage d'utiliser les régions de tableaux au lieu des effets (qui donnent les références aux éléments de tableaux), est que les appels de procédure et les instructions intraprocédurale sont traités de manière uniforme. Par contre, des tests simples (et rapides) sur les expressions d'indice, comme le test du GCD, ne peuvent pas être utilisés, alors que cela est possible avec les références. Une solution serait de garder dans la représentation des régions la référence d'origine lorsque cela est possible (région non résumée), afin d'effectuer les tests rapides sur ces références et de ne construire le système de dépendance qu'en cas d'échec.

12.1.2 Résumés ou régions propres ?

L'implantation des régions présentée jusqu'à présent dans cette thèse calcule systématiquement des régions résumées, même au niveau des instructions simples. Ceci peut avoir des conséquences désastreuses.

Considérons par exemple le programme de la figure 12.1. Dans chacune des boucles l'élément de tableau A(i) est écrit et les éléments A(i-10) et A(i+10) sont lus. Si les régions sont fusionnées systématiquement, la région obtenue pour les deux éléments lus est représentée par $i - 10 \leq \phi_1 \leq i + 10$; son intersection avec la région représentant A(i), avec $\mathtt{i} = i + di$, n'est pas vide. La dépendance doit donc être conservée, alors qu'elle n'existe pas réellement, et qu'elle pourrait être éliminée en utilisant les effets au lieu des régions.

La solution que nous proposons est de ne pas fusionner les régions au niveau des instructions simples, et de réserver ces technique à la propagation des régions. Nous appelons les régions correspondantes les *régions propres* car elles ne contiennent que les éléments de tableaux référencés par l'instruction elle-même. Ceci fait une différence au niveau des instructions conditionnelles et des boucles do : les régions propres d'un

```
program proper
integer i,A(30)

do i = 11,20
   A(i) = A(i-10)+A(i+10)
enddo
do i = 11,20
   call dum1(A(i), A(i-10), A(i+10))
enddo
do i = 11,20
   call dum2(A(i), A(i-10)+A(i+10))
enddo
end
```

```
subroutine dum1(i,j,k)
integer i,j,k
i = j + k
end

subroutine dum2(i,j)
integer i,j
i = j
end
```

Figure 12.1: Un exemple pour montrer la nécessité des régions propres

`if` sont les régions de sa condition avant fusion ; et celles d'une boucle `do` sont les régions de son en-tête ; au contraire de leurs régions *normales* qui sont des résumés pour l'ensemble de la structure.

Nous proposons aussi d'éviter la fusion au niveau des sites d'appel de procédure. Nous ne remettons pas en question l'utilisation de résumés pour représenter les effets de la procédure appelée, mais uniquement lors de la traduction, lorsque les arguments formels sont remplacés par les arguments réels correspondants. L'avantage de cette approche apparaît mieux sur un exemple.

> Le programme `proper` contient deux appels de procédure. Dans le premier, une seule référence à un élément de tableau correspond à chaque argument formel. Les régions résumées de `dum1`, qui sont :
>
> ```
> <i-w-{}>
> <j-r-{}>
> <k-r-{}>
> ```
>
> sont traduites en :
>
> $<A(\phi_1)-W-\{\phi_1==i,\ 11<=i<=20\}>$
> $<A(\phi_1)-R-\{\phi_1==i-10,\ 11<=i<=20\}>$
> $<A(\phi_1)-R-\{\phi_1==i+10,\ 11<=i<=20\}>$
> $<i-R-\{\}>$[1]
>
> Ici, une approche naïve consisterait à fusionner les deux régions READ du tableau `A` et, comme précédemment, la dépendance ne pourrait pas être éliminée.

La fusion doit donc être évitée à l'issue du processus de traduction.

[1] Cette région provient de la référence à `i` dans `A(i-10)` et `A(i+10)`.

> Dans le deuxième appel, deux références à un élément de tableau correspondent à un même argument de `dum2`. Ceci n'est permis par la norme FORTRAN [8] que si l'argument formel n'est pas modifié par la procédure. Dans ce cas, la région de l'argument formel scalaire est simplement remplacée par la région READ de l'expression passée en argument. Et l'on obtient les mêmes régions que dans le cas précédent.

La fusion doit donc ici aussi être évitée. Les régions propres sont implantées dans PIPS comme un intermédiaire de calcul des régions READ et WRITE. Le coût induit est donc négligeable. Et ceci nous permet de paralléliser les trois boucles de l'exemple précédent.

12.1.3 Utilisation des régions IN et OUT

Le graphe de dépendance construit à l'aide des régions READ et WRITE représente les *conflits de mémoire*. Comme les régions IN et OUT ont été introduites pour modéliser le flot des valeurs, la tentation est grande de les utiliser pour construire un graphe de dépendance basé sur les *conflits de valeurs* ; et ceci même si les régions IN et OUT fournissent une information moins précise que les fonctions sources [34, 72, 147]. La question est donc : *que peut-on espérer de l'utilisation des régions* IN *et* OUT *pour l'analyse des dépendances ?*

Les régions IN d'une instruction représentent les éléments de tableau qu'elle importe, tandis que ces régions OUT contiennent les éléments qu'elle exporte. Ainsi, si S_1 et S_2 sont deux instructions consécutives, $\overline{\mathcal{R}_i}[\![S_2]\!] \,\overline{\cap}\, \overline{\mathcal{R}_o}[\![S_1]\!]$ donne une sur-approximation de l'ensemble des éléments définis par S_1 réutilisés directement par S_2. C'est donc bien une sur-approximation d'une dépendance de valeurs. Cependant, si il existe une instruction S_1' entre S_1 et S_2, alors les éléments importés par S_2 peuvent provenir de S_1' au lieu de S_1 ; ce qui n'est pas exprimé par $\overline{\mathcal{R}_i}[\![S_2]\!] \,\overline{\cap}\, \overline{\mathcal{R}_o}[\![S_1]\!]$, opération qui serait exécutée si l'on remplaçait purement et simplement les régions READ et WRITE par les régions IN et OUT lors de la construction du graphe des *chaînes*. Ainsi certaines dépendances indirectes seraient éliminées, mais pas toutes.

En réalité, effectuer les tests de dépendance avec les régions IN et OUT à ce niveau de granularité n'apporterait pas un gain significatif, sauf en cas d'appel de procédure — ce qui revient à privatiser certains tableaux utilisés par celle-ci. Par contre, comme nous le verrons dans la section suivante, ces régions peuvent être utilisées à un niveau de granularité plus élevé pour détecter les régions de tableau privatisables, et donc éliminer des dépendances entre itérations.

12.2 Privatisation de Tableaux

Notre but est ici de déterminer les ensembles d'éléments de tableaux dont le calcul est local à une itération d'une boucle. Plusieurs algorithmes de privatisation de scalaires existent déjà [66, 138]. Mais il serait trop complexe de les utiliser tels quels pour les tableaux, car cela nécessiterait de pouvoir comparer ou combiner des régions de tableaux d'instructions différentes quelconques, tout en tenant compte des valeurs de variables entre les états mémoire les précédant.

Pour éviter ces problèmes, nous proposons donc dans cette section d'utiliser les régions IN et OUT. Bien sûr, comme ces régions sont des résumés, la précision ne sera pas la même qu'avec des extensions des méthodes de privatisation de scalaires. Mais nous pensons que les résultats obtenus seront suffisants pour traiter les applications usuelles. Dans PIPS, les régions sont représentées par des polyèdres convexes. Cependant, l'algorithme que nous présentons ici est indépendant de toute représentation.

12.2.1 Un peu de vocabulaire

Il y a à peu près autant de définitions de la privatisabilité d'un tableau que de papiers à ce propos [123, 166, 92, 17]. Nous présentons donc ici les différents critères que nous prenons en compte.

Portée L'idée de base est de détecter les régions de tableaux utilisées comme temporaires ; ceci peut s'appliquer en théorie à n'importe quelle portion de programme, pourvu que l'on puisse ensuite déclarer les nouveaux tableaux locaux. Nous nous restreignons donc ici aux boucles, pour lesquelles les langages parallèles (ou extensions parallèles de langages séquentiels) fournissent généralement les moyens de déclarer les tableaux privés. Mais nous pourrions effectuer le même travail pour les procédures.

Élimination des *fausses* dépendances Le but de la privatisation de tableaux est d'éliminer les dépendances dues à la réutilisation de parties de tableaux qui servent en réalité de structures temporaires. Toutefois, il peut aussi être intéressant de détecter les calculs locaux qui ne génèrent pas de dépendances pour pouvoir agir sur l'allocation de la mémoire. Notre algorithme permettra donc les deux approches.

Tableaux ou régions de tableaux Dans la plupart des cas, ce sont les tableaux entiers qui sont utilisés comme temporaires ; mais ce n'est pas systématique, notamment pour des tableaux déclarés comme des variables globales. L'algorithme présenté en section 12.2.3 détecte donc les *régions de tableaux privatisables.*

Copy-in et Copy-out Dans certains cas de figure, le tableau est utilisé comme temporaire, sauf dans certaines itérations, qui *importent* des valeurs du tableau avant l'exécution de la boucle (*copy-in*), ou qui *exportent* certaines valeurs vers les instructions suivant l'exécution de la boucle (*copy-out*). Notre algorithme permet de traiter les deux cas à la demande, selon l'architecture considérée par exemple.

12.2.2 Notations

Dans cette section, nous considérons tous les types de boucles, boucles `do` et boucles `do while`. Pour traiter ces dernières plus facilement, nous introduisons un compteur d'itérations i, dont la valeur est 0 avant l'exécution de la boucle, et k après la k-ième itération. De plus, nous noterons $\mathcal{R}_{type}(i)$ la région de type *type* correspondant à la i-ème itération. Nous ne considérons pas ici les problèmes dûs aux modifications d'états mémoire ; nous renvoyons pour cela le lecteur au chapitre 14.

Nous utiliserons dans notre algorithme les types de régions suivants :

$\mathcal{R}_{priv}$	régions privatisables
$\mathcal{R}_{copy\text{-}out}$	régions pour le copy-out
$\mathcal{R}_{copy\text{-}in}$	régions pour le copy-in
$\mathcal{R}_{out\_ncopy}$	régions non candidates pour le copy-out
$\mathcal{R}_{in\_ncopy}$	régions non candidates pour le copy-in

En réalité, seule une sous-approximation des régions privatisables ($\underline{\mathcal{R}}_{priv}$) nous intéresse, car cela n'aurait pas de sens de privatiser des éléments de tableau dont le calcul n'est pas effectivement local. De même, il vaut mieux importer ou exporter trop d'éléments de tableaux, que pas assez. Nous calculerons donc les régions $\overline{\mathcal{R}}_{copy\text{-}in}$ et $\overline{\mathcal{R}}_{copy\text{-}out}$.

12.2.3 Algorithme

L'algorithme de privatisation des régions de tableaux de la figure 12.2 prend en entrée trois régions différentes (IN, WRITE, et OUT) correspondant à l'itération i, et trois valeurs booléennes indiquant les problèmes à résoudre : *copy_in_p* et *copy_out_p* prennent la valeur *vrai* pour résoudre les problèmes de copy-in et de copy-out ; et *spurious_dep_p* prend la valeur *vrai* si le but est d'éliminer des fausses dépendances, et *faux* si une simple analyse de localité est souhaitée. La sortie consiste en trois régions, déjà présentées ci-dessus : $\underline{\mathcal{R}}_{priv}$, $\overline{\mathcal{R}}_{copy\text{-}out}$, et $\overline{\mathcal{R}}_{copy\text{-}in}$.

$\overline{\mathcal{R}}_{in\_ncopy}(i)$ contient les éléments de tableaux que l'on ne doit pas importer si le tableau est privatisé. Si le copy-in n'est pas requis (étape 5), alors aucun élément de tableau importé par l'itération ne peut faire partie de l'ensemble des éléments privatisés et importés ; dans le cas contraire (étape 3), les éléments de tableaux privatisés ne peuvent être importés que des instructions précédant la boucle, et pas des autres itérations.

De même, $\overline{\mathcal{R}}_{out\_ncopy}(i)$ contient les éléments de tableaux que l'on ne doit pas exporter s'ils sont privatisés. Si le copy-out n'est pas requis (étape 10), alors les éléments exportés par la boucle ne peuvent faire partie des éléments privatisés et exportés ; dans le cas contraire (étape 8), les éléments de tableaux privatisés ne peuvent être exportés que vers les instructions suivant l'exécution de la boucle, et pas vers d'autres itérations.

L'ensemble des éléments privatisables est alors égal à l'ensemble des éléments calculés dans l'itération courante ($\underline{\mathcal{R}}_w(i)$), et qui ne sont pas interdits de privatisation (étape 12) ! Si la privatisation n'est effectuée que pour éliminer des dépendances (étape 14), alors seuls les éléments de tableaux qui sont aussi privatisables dans d'autres itérations doivent être privatisés.

Enfin, nous générons les ensembles d'éléments de tableaux privatisables à importer du tableau initial ou à exporter vers celui-ci. Quand le copy-in et le copy-out ne sont pas permis, ces ensembles sont vides. Sinon, les éléments à importer sont ceux qui sont privatisables, et dont la valeur est importée par l'itération courante (étape 17) ;

`entrée :` $\overline{\mathcal{R}}_i(i)$, $\overline{\mathcal{R}}_w(i)$, $\overline{\mathcal{R}}_o(i)$, $copy\_in\_p$, $copy\_out\_p$, $spurious\_dep\_p$
`si` $copy\_in\_p$ `alors`
    $\overline{\mathcal{R}}_{in\_ncopy}(i) = \overline{\mathcal{R}}_i(i) \ \overline{\cap} \ \overline{\bigcup}_{i'<i} \ \overline{\mathcal{R}}_w(i)$
`sinon`
    $\overline{\mathcal{R}}_{in\_ncopy}(i) = \overline{\mathcal{R}}_i(i)$
`finsi`
`si` $copy\_out\_p$ `alors`
    $\overline{\mathcal{R}}_{out\_ncopy}(i) = \overline{\mathcal{R}}_o(i) \ \overline{\cap} \ \overline{\bigcup}_{i'>i} \ \overline{\mathcal{R}}_i(i)$
`sinon`
    $\overline{\mathcal{R}}_{out\_ncopy}(i) = \overline{\mathcal{R}}_o(i)$
`finsi`
$\underline{\mathcal{R}}_{priv}(i) = (\underline{\mathcal{R}}_w(i) \ \underline{\boxminus} \ \overline{\mathcal{R}}_{in\_ncopy}(i)) \ \underline{\boxminus} \ \overline{\mathcal{R}}_{out\_ncopy}(i)$
`si` $spurious\_dep\_p$ `alors`
    $\underline{\mathcal{R}}_{priv}(i) = (\underline{\mathcal{R}}_{priv}(i) \ \underline{\cap} \ \overline{\bigcup}_{i'<i} \ \underline{\mathcal{R}}_{priv}(i'))$
        $\underline{\cup} \ (\underline{\mathcal{R}}_{priv}(i) \ \underline{\cap} \ \overline{\bigcup}_{i'>i} \ \underline{\mathcal{R}}_{priv}(i'))$
`finsi`
`si` $copy\_in\_p$ `alors`
    $\overline{\mathcal{R}}_{copy\text{-}in}(i) = \overline{\mathcal{R}}_i(i) \ \overline{\cap} \ \underline{\mathcal{R}}_{priv}(i)$
`sinon`
    $\overline{\mathcal{R}}_{copy\text{-}in}(i) = \lambda\sigma.\varnothing$
`finsi`
`si` $copy\_out\_p$ `alors`
    $\overline{\mathcal{R}}_{copy\text{-}out}(i) = \overline{\mathcal{R}}_o(i) \ \overline{\cap} \ \underline{\mathcal{R}}_{priv}(i)$
`sinon`
    $\overline{\mathcal{R}}_{copy\text{-}out}(i) = \lambda\sigma.\varnothing$
`finsi`
`sortie :` $\underline{\mathcal{R}}_{priv}(i)$, $\overline{\mathcal{R}}_{copy\text{-}out}(i)$, $\overline{\mathcal{R}}_{copy\text{-}in}(i)$

Figure 12.2: Algorithme de détection des régions de tableaux privatisables

et ceux à exporter sont ceux qui sont privatisables, et dont les valeurs sont exportées par l'itération courante.

12.3 Autres travaux et Conclusion

Nous avons montré dans cette partie comment les analyses de régions de tableaux présentées précédemment sont utilisées dans PIPS pour analyser les dépendances. Nous avons notamment montré la nécessité d'utiliser les régions READ et WRITE *propres* au lieu des régions *résumées*, et de disposer d'algorithmes spécifiques pour utiliser les informations supplémentaires fournies par les régions IN et OUT.

Nous avons à cet effet montré comment elles pouvaient être utilisées pour détecter les régions de tableau privatisables. L'algorithme proposé permet entre autres de résoudre les problèmes d'initialisation du tableau privatisé (*copy-in*) et de mise à jour (*copy-out*) du tableau initial si certaines valeurs doivent être importées ou exportées, et ceci de manière uniforme.

Les travaux sur l'analyse de dépendance à l'aide de régions de tableaux sont très nombreux [162, 163, 122, 37, 19, 98, 99, 35, 92]. Les différences portent essentiellement sur le type de représentation choisi pour les ensembles d'éléments de tableaux, et les tests de dépendance utilisés, de manière à obtenir un compromis entre précision et complexité.

Une autre tendance actuelle est d'utiliser les dépendances directes [34, 72, 147] pour générer du code, au lieu des dépendances sur la mémoire, de manière à détecter le parallélisme caché par la réutilisation des variables. Ces analyses sont très précises, mais assez coûteuses, et imposent souvent des restrictions sur le langage source.

Une approche intermédiaire est de faire précéder l'analyse des dépendances standard par une phase de privatisation de tableaux, et là encore plusieurs algorithmes basés sur l'utilisation des régions de tableaux ont été proposés [164, 92, 123], qui soit sont moins généraux que le nôtre, soit utilisent des analyses de natures différentes, et sont donc moins uniformes.

Chapter 13

Dependence Analysis

As stated in the introduction, array regions have numerous potential applications. But even if they could be used in specific areas, their main interest remains in the domain of automatic parallelization. The purpose of this part is to describe two applications in this field, which are implemented in PIPS: This chapter is devoted to dependence analysis, and the next one to array region privatization, which has been shown to be of paramount importance to achieve good performances [29].

TRIOLET originally proposed to use over-approximate READ and WRITE array regions to enhance dependence tests across procedure boundaries, which hide actual array references. His approach was first implemented in PARAPHRASE [162], work later resumed within the framework of PIPS [105, 139, 177]. In this chapter, we more thoroughly, though still informally, describe the use of array regions in PIPS to disprove dependences. In particular, we propose solutions to problems due to summarization and induction variables. And we discuss the use of IN and OUT regions to further enhance dependence analysis.

The organization of the chapter is the following. Section 13.1 describes PIPS dependence analysis framework. The next section is devoted to the extension of this framework using array regions. The related work is finally quickly reviewed in Section 13.3.

13.1 PIPS Dependence Analysis Framework

In PIPS the ultimate dependence graph used for code generation is built in several phases (see Figure 16.3 on Page 253):

- **Effects** (see Appendix A, Section A.2.1) gather information about read and written scalar and array references. For assignments, each effect merely contains the actual reference (that is to say the name of variable, and its subscripts in case of an array) and the nature of the memory operation (read or write). To handle procedure calls, effects are propagated backwards interprocedurally. During the translation process, effects on array variables are enlarged to the whole variable, thus loosing accuracy. The translated effects are used at call sites in the same manner as intraprocedural effects.

- **Use-def**, **def-use** and **def-def chains** [1, 133] are built from effects or array regions; the result is an initial dependence graph: Each node represents a statement; edges correspond to dependences and are annotated with lists of *conflicts*. A conflict is a pair of effects, the first one being the source of the dependence, and the second one its target. The nature of the conflict (flow- anti- or output-dependence) is given by the nature (write or read) of the source and target effects. There is no dependence test for arrays at this level: Edges are added in the dependence graph on the sole basis of variable names given by effects.

- A **dependence test** is then applied to each conflict. Dependences on scalars are assumed, but more work is performed on arrays. The result is a dependence graph with hopefully less edges than in the original graph. In addition, each remaining conflict is annotated with the level of the dependence.

PIPS dependence test is a gradual test: It first applies inexpensive techniques such as the *constant consistency* test or the GCD test (which tests whether the gcd of the coefficients in a dependence equation divides the constant). If the dependence still holds after these quick tests, a *dependence system* is built, and more expensive tests are applied: *Consistency* test, *feasibility* test, and *lexicopositivity* tests. When all these successive tests fail, the dependence is assumed and the corresponding dependence level is computed from the dependence system representation.

Quick tests are directly applied on reference subscripts. But for the more complicated tests, a *dependence system* is built, represented by a convex polyhedron. Two types of constraints are used:

Dependence constraints are directly derived from the references. Let $\vec{I}$ denote the iteration vector of the common enclosing loops of the two references, $\vec{I_1}$ and $\vec{I_2}$ the iteration vector of the remaining loops for respectively the first and second references. Let $r_1(\vec{I}+\vec{I_1})$ and $r_2(\vec{I}+d\vec{I}+\vec{I_2})$ represent the access vectors of the two references respectively at iterations $\vec{I}$ and $\vec{I}+d\vec{I}$ of the common enclosing loops. If there exists $\vec{I}$, $\vec{I_1}$, $\vec{I_2}$ and $d\vec{I}$ such that $r_1(\vec{I}+\vec{I_1}) = r_2(\vec{I}+d\vec{I}+\vec{I_2})$, then the dependence must be assumed.

For instance, in the following loop nest,

```
do I = 1,N
  do J = 1,N
    A(I,I) = A(I,J+K)
  enddo
enddo
```

there is a potential dependence between references `A(I,I)` and `A(I,J+K)`, respectively at iterations $\vec{I} = (\ i \quad j\ )^T$ and $\vec{I}+d\vec{I} = (\ i+di \quad j+dj\ )^T$. We have:

$$r_1(\vec{I}) = \begin{pmatrix} i \\ i \end{pmatrix}$$

$$r_2(\vec{I}+d\vec{I}) = \begin{pmatrix} i+di \\ j+dj+\mathtt{K} \end{pmatrix}$$

The dependence constraints are thus:

$$\begin{cases} i = i + di \\ i = j + dj + \mathtt{K} \end{cases}$$

Here the dependence cannot be disproved.

Another dependence constraint, the *lexicopositivity* constraint $d\vec{I} > 0$, is also added in the most advanced test, to eliminate inner level dependences. It is not added in the dependence system for the first tests, because inner level dependences may be useful for some program transformations, such as code distribution [4].

Context constraints Depending on PIPS compilation options, some context constraints may be taken into account when building the dependence system. They can merely be loop bounds (in the above example $1 \le i \le \mathtt{N}$ and $1 \le j \le \mathtt{N}$), but also *preconditions*, that is to say general constraints about the program variable values. We denote $\mathcal{P}_1(\vec{I} + \vec{I_1})$ and $\mathcal{P}_2(\vec{I} + d\vec{I} + \vec{I_2})$ the context information associated with the first reference at iteration $\vec{I} + \vec{I_1}$ and the second at iteration $\vec{I} + d\vec{I} + \vec{I_2}$. If there is a dependence between the two reference, then it must verify the context condition. The dependence system is built by merely appending the dependence equations, $\mathcal{P}_1(\vec{I} + \vec{I_1})$ and $\mathcal{P}_2(\vec{I} + d\vec{I} + \vec{I_2})$, to the initial system.

In the previous example, the dependence cannot be resolved if no information about the value of K is available. If the preconditions are:

$$\mathcal{P}_1(\vec{I}) = \{1 \le i \le \mathtt{N}, 1 \le j \le \mathtt{N}, \mathtt{N} \le \mathtt{K}\}$$
$$\mathcal{P}_2(\vec{I}) = \{1 \le i + di \le \mathtt{N}, 1 \le j + dj \le \mathtt{N}, \mathtt{N} \le \mathtt{K}\}$$

the dependence system becomes,

$$\begin{cases} i = i + di \\ i = j + dj + \mathtt{K} \\ \mathtt{N} \le \mathtt{K} \\ 1 \le i \le \mathtt{N} \\ 1 \le j \le \mathtt{N} \\ 1 \le i + di \le \mathtt{N} \\ 1 \le j + dj \le \mathtt{N} \end{cases}$$

and the dependence can be disproved by a feasibility test.

Up to now, we have not considered loop nests in which variables other than the loop index vary. Automatic induction variable substitution is not implemented in PIPS[1]. Nevertheless, the dependence test safely handles variables modified within enclosing loops. Let $\vec{K}$ denote the vector of such variables (excluding loop indices). Then, exactly as $\vec{I}$ is replaced by $\vec{I} + d\vec{I}$ in the terms or constraints corresponding to the second reference, $\vec{K}$ is similarly replaced by $\vec{K} + d\vec{K}$.

In the following program, taken from PIPS non-regression tests,

[1] But some information is already available from the preconditions.

```
j = 1
do i = 1, n
   A(j) = A(j) + A(j-1)
   j = j + 2
enddo
```

there is a potential dependence between the write reference to `A(j)` and the read reference to `A(j-1)`. However, `j` is modified in the loop body. The dependence equation is thus

$$j = j + dj - 1$$

PIPS gives a few additional context constraints, and the dependence system becomes:

$$\begin{cases} j = j + dj - 1 \\ j = 2 \times i - 1 \\ j + dj = 2 \times i + 2 \times di - 1 \\ 1 \leq i \leq \mathtt{n} \\ 1 \leq i + di \leq \mathtt{n} \end{cases}$$

The first constraint simplifies to $dj = 1$, and the third one to $dj = 2 \times di$, which are incompatible. The solution set is therefore empty, and the dependence is disproved.

However, this technique fails to disprove some dependences whenever the relations between the values of the loop index and the loop variants is not available, as in the following contrived example:

```
j = foo()
do i = 1, n
   a(j) = 0.0
   j = j + 1
   b(i) = a(j-1)
enddo
```

Using the preconditions discovered by PIPS, the dependence system built for the dependence between `A(j)` and `A(j-1)` is:

$$\begin{cases} j = j + dj - 1 \\ 1 \leq i \leq \mathtt{n} \\ 1 \leq i + di \leq \mathtt{n} \end{cases}$$

The information available from preconditions is not sufficient to disprove the dependence, whereas it does not actually exist.

Using effects to perform a dependence analysis can be done very effectively [177] whenever there is no procedure calls, and relations between loop indices and induction variables are perfectly known, through preconditions. However, if array elements are referenced by a procedure call, or if induction variables are unknown, the test may fail.

We show in the next section how to partially alleviate these problems using array regions.

13.2 Using Regions to Disprove Dependences

In PIPS, array regions are of the same type as effects: They contain a reference giving the variable name and its descriptors instead of the actual subscripts (eg. `A(`ϕ_1`,`ϕ_2`)` instead of `A(I,I)`); in addition, they include a polyhedron which describes the region, polyhedron which is undefined in the case of effects. Array regions can thereby be used in the same way as effects to build *chains*. But for the dependence test, a different type of information is available: Two polyhedra are provided instead of two references. We show in the first section how the dependence test is applied. Then, we discuss the impact of summarization techniques on the accuracy of the dependence test. The problem of variables modified within the loop nest is then considered. Finally, we investigate the possible consequences of using IN and OUT regions instead of more traditional READ and WRITE regions for the dependence test.

13.2.1 Using READ and WRITE regions

Over-approximate READ and WRITE regions provide an information of the same nature as READ and WRITE effects. They can thus be used in the same way to model dependences. However, the representation is not similar.

For instance, a written reference `A(I,I)` is represented by the effect

`<A(I,I)-W-{}>`

and the region

`<A(`ϕ_1`,`ϕ_2`)-W-{`ϕ_1`==I,`ϕ_2`==I}>`

The dependence test suite must therefore be slightly modified. With effects, the question is whether a reference may be the same as another one, across iterations. The new problem now is to find whether two array regions, corresponding to iterations $\vec{I}+\vec{I_1}$ and $\vec{I}+d\vec{I}+\vec{I_2}$, $\mathcal{R}_1(\vec{I}+\vec{I_1})$ and $\mathcal{R}_2(\vec{I}+d\vec{I}+\vec{I_2})$, may contain the same array elements:

$$\mathcal{R}_1(\vec{I}+\vec{I_1}) \cap \mathcal{R}_2(\vec{I}+d\vec{I}+\vec{I_2}) \stackrel{?}{=} \varnothing$$

Here, the dependence system is $\mathcal{R}_1(\vec{I}+\vec{I_1}) \cap \mathcal{R}_2(\vec{I}+d\vec{I}+\vec{I_2})$ instead of

$$\big(r_1(\vec{I}+\vec{I_1}) = r_2(\vec{I}+d\vec{I}+\vec{I_2})\big) \cap \mathcal{P}_1(\vec{I}+\vec{I_1}) \cap \mathcal{P}_2(\vec{I}+d\vec{I}+\vec{I_2})$$

in the case of effects. Notice that preconditions are not used anymore because in the present implementation they are included in regions. Once the dependence system has been built, the usual suite of dependence tests can be performed.

Let us take as an example the subroutine `EXTR` in Figure 2.1 on Page 14. There is a potential dependence between the array elements read by the first invocation of function D, represented by the region (with $\vec{I} = j$):

$$\mathcal{R}_1(j) = \begin{cases} \phi_1 = j \\ \mathtt{K} \leq \phi_2 \leq 1 + \mathtt{K} \\ \mathtt{L} \leq \phi_3 \leq 2 + \mathtt{L} \\ \phi_1 \leq 52 \\ \phi_2 \leq 21 \\ 1 \leq \phi_3 \leq 60 \\ \mathtt{J2} = 2\mathtt{JA} - 1 \\ \mathtt{K} = \mathtt{K1} \\ \mathtt{L} = \mathtt{NI} \\ \mathtt{J1} \leq j \\ j \leq \mathtt{JA} \\ 2 \leq \mathtt{J1} \\ 2 \leq \mathtt{K1} \end{cases}$$

and the write reference to `T(J,1,NC+3)`, represented by:

$$\mathcal{R}_2(j + dj) = \begin{cases} \phi_1 = j + dj \\ \phi_2 = 1 \\ \phi_3 = \mathtt{NC+3} \\ \mathtt{J2} = \mathtt{2JA-1} \\ \mathtt{K} = \mathtt{K1} \\ \mathtt{L=NI} \\ \mathtt{J1} \leq j + dj \\ j + dj \leq \mathtt{JA} \\ 2 \leq \mathtt{J1} \\ 2 \leq \mathtt{K1} \end{cases}$$

And a feasibility test easily finds that

$$\forall j, \mathcal{R}_1(j) \cap \mathcal{R}_2(j + dj) = \varnothing$$

It merely comes from the fact that in the first region $\phi_2 \geq \mathtt{K} \geq \mathtt{K1} \geq 2$, while in the second $\phi_2 = 1$.

The advantage of using array regions instead of effects (or references) is that procedure calls and intraprocedural statements can be uniformly and accurately handled. The counterpart is that cheap tests cannot be performed anymore because the subscripts are not available in array regions. A solution would be to keep actual references in the regions corresponding to intraprocedural statements, perform the quick dependence tests whenever the initial regions contain actual references, and switch to the region polyhedra only if the first tests fail, or if at least one region corresponds to a procedure call (ie. does not include an actual reference).

13.2.2 Summarization *versus* proper regions

The choices made in the previous part of this thesis imply that summarization is intensively used for the computation of array regions, even at the level of mere assignments. However, this approach may have dramatic consequences.

Consider for instance the program of Figure 13.1. In each loop the array element `A(i)` is written, and elements `A(i-10)` and `A(i+10)` are read. If regions are systematically merged, the region for the two read elements is represented by $i-10 \leq \phi_1 \leq i+10$, whose intersection with the region for `A(i)`, with `i` $= i + di$, is not empty. Thus a dependence must be assumed, whereas it does not actually exists, and could be removed using effects instead of regions!

```
program proper
integer i,A(30)

do i = 11,20
   A(i) = A(i-10)+A(i+10)
enddo
do i = 11,20
   call dum1(A(i), A(i-10), A(i+10))
enddo
do i = 11,20
   call dum2(A(i), A(i-10)+A(i+10))
enddo
end
```

```
subroutine dum1(i,j,k)
integer i,j,k
i = j + k
end

subroutine dum2(i,j)
integer i,j
i = j
end
```

Figure 13.1: A sample program to advocate proper regions

The solution we propose is to avoid merging at the level of simple instructions, or expressions, and to reserve summarization techniques to the propagation phase of array regions. We call the resulting regions *proper regions*, because they contain the regions accessed by the current statement itself. This makes a difference for `if` conditionals and `do` loops: The proper regions of an `if` are the regions of the condition, and for the `do` loop, they are the regions of the range; whereas their *normal* regions are summaries for the whole construct.

Avoiding summarization does not only apply to assignments, but also to procedure calls, and more precisely to actual arguments. We do not question summarization for the procedure called, but during the translation process, when formal arguments are replaced by actual ones. Let us take an example to illustrate this.

Program `proper` contains two procedure calls. In the first one, exactly one array reference corresponds to each formal argument. The summary regions of `dum1`, which are:

```
<i-w-{}>
<j-r-{}>
<k-r-{}>
```

are translated into

```
<A(φ1)-W-{φ1==i, 11<=i<=20}>
<A(φ1)-R-{φ1==i-10, 11<=i<=20}>
<A(φ1)-R-{φ1==i+10, 11<=i<=20}>
<i-R-{}>
```
[2]

Here a naive approach would merge the two READ regions for array `A` and the dependence would not be disproved.

Summarization must thus be avoided *after* the translation process.

In the second call, two array references correspond to the second formal argument of `dum2`. This can only appear when the formal argument is a read scalar variable. In this case, the region of the scalar argument is merely replaced by the regions of the expression passed as an argument.

So, summarization must also be avoided when computing the regions of an expression. Using these techniques, PIPS was able to successfully parallelize the three loops in `proper`.

Computing *proper regions* can be done while computing normal READ and WRITE regions, with no significant overhead: For assignments, proper regions are first computed; a copy is made, to be later stored in the `PROPER_REGIONS` resource handled by *pipsdbm* (see Appendix A for more details); then, a summary is computed to be stored in the `REGIONS` resource, and to be propagated along the program representation for further computations.

13.2.3 Handling variable modifications

We have seen in Section 13.1 that variables modified within loop nests may not be accurately handled. Besides implementing induction variable substitution, another solution is to use array regions. Our purpose is not to formally describe the method, but to merely illustrate it with an example.

We consider the contrived example already presented in Section 13.1, along with its READ and WRITE regions:

```
  j = foo()
  do i = 1, n
c ----- loop body ----
c <a(φ1)-W-{φ1==j,1<=i<=n}>
c <a(φ1)-R-{φ1==j,1<=i<=n}>
c --------------------
c <a(φ1)-W-{φ1==j,1<=i<=n}>
     a(j) = 0.0
     j = j + 1
c <a(φ1)-R-{φ1==j-1,1<=i<=n}>
c <b(φ1)-W-{φ1==i,1<=i<=n}>
     b(i) = a(j-1)
  enddo
```

[2]This region comes from the reference to `i` in `A(i-10)` and `A(i+10)`.

The regions of the loop body refer to the same memory store. Hence, the letter `j` which appears in the two polyhedra refers to a unique value for a given iteration. However, both regions still depend on `j`, whose value is unknown. The idea is to use the *transformer* of the loop: As shown in Section 8.5, it gives an over-approximation of the loop invariant, and can be used to get rid of variables modified within the loop.

In our example, this transformer is `T(j) {j==j#init+i-1}`, `j#init` denoting the value of `j` before the execution of the loop. Composing regions of the loop body with this transformer, we get:

```
<a(φ1)-W-{φ1==j#init+i-1,1<=i<=n}>
<a(φ1)-R-{φ1==j#init+i-1,1<=i<=n}>
```

Both regions do not depend anymore on the value of `j`, and solely rely on the loop index and loop invariants. The usual dependence test can now be applied. The dependence system is (with $j' =$ `j#init`):

$$\begin{cases} \phi_1 = j' + i - 1 \\ \phi_1 = j' + i + di - 1 \\ 1 \leq i \leq \mathtt{n} \\ 1 \leq i + di \leq \mathtt{n} \end{cases}$$

The first two equations lead to $di = 0$, which does not correspond to a loop carried dependence.

Preconditions solely gives the values of variables against each other, and against the input memory store of the routine. However, they cannot track values as soon as non-linear expressions are involved. On the contrary using transformers leads to regions which depend on the loop index, and the values of variables in the store just preceding the loop. The main difference is that preconditions give a *static* information, while transformers give a *dynamic* information.

However, this method has a major drawback. Using the region summaries of the loop body amounts to coalescing the inner statements into a single one. In particular, actual references are lost. This may be sufficient to disprove loop carried dependences, but either the inner level dependences must be kept or another analysis must be performed. Another drawback is due to the choice in PIPS to summarize as soon as possible. Hence array regions of the loop body may sometimes be coarse approximations of the actual references.

13.2.4 Using IN and OUT regions

Up to now, the dependence graph is built using READ and WRITE regions, hence represents *memory based* dependences. Since IN and OUT regions have been introduced to model the flow of values, the temptation is high to use them to build *value based* dependence graphs. However, IN and OUT regions carry less information than source functions [34, 72, 147] for instance, because they do not keep trace of the precise references responsible for the accesses they collectively represent. The question therefore is: *What can be achieved using* IN *and* OUT *regions?*

IN regions of a statement represent the array elements it *imports*, while its OUT regions contain the elements it *exports*. Hence, if S_1 and S_2 are two consecutive statements, $\overline{\mathcal{R}}_i[\![S_2]\!] \overline{\cap} \overline{\mathcal{R}}_o[\![S_1]\!]$ gives an over-approximation of the set of elements which are defined by S_1 and directly reused in S_2. This is an over-approximation of the *value based* dependences. However, if S_1 and S_2 are not consecutive, for instance if there exists S_1' in between, then the elements imported by S_2 may come from S_1' instead of S_1; this is not expressed in $\overline{\mathcal{R}}_i[\![S_2]\!] \overline{\cap} \overline{\mathcal{R}}_o[\![S_1]\!]$, operation which would be performed if we merely replaced READ by IN regions and WRITE by OUT regions when building *chains*. Hence some spurious memory based dependences are removed, but not all of them. Let us take an example to show the advantages and consequences of using IN and OUT regions.

Let us consider the sample program of Figure 13.2. A temporary array is defined and immediately reused within the call to `bar`. The same pattern is later found inside the same loop, but expanded in the calling procedure `foo`.

If we merely use the IN and OUT regions corresponding to simple instructions (assignments and calls), we must assume a loop carried dependence between `s2` and `s3`, because `s2` exports elements which are imported by `s3`. However, no dependence is found between `s1` and the other statements, because the IN and OUT regions of `bar` are empty sets, `work` being used as a temporary .

Let us now examine dependences at a higher granularity. The IN and OUT regions corresponding to the call to `bar` for the array `work` are empty sets; it is the same for the `j` loop, considered as a single compound statement. Thus, there is no value-based dependence between these two parts of the code: The `i` loop can be parallelized, provided that there is no dependence in the hidden parts of code, that `work` is renamed in `bar`, and that a local array replaces `work` in the `j` loop.

```
      subroutine foo
      common/tmp/work(1000)
      ...
      do i = 1,n
s1       call bar
         do j = 1,n
            do k = 1, 1000
s2             work(k) = ...
            enddo
            ....
            do k = 1, 1000
s3             ... = work(k)
            enddo
         enddo
      enddo
```

```
      subroutine bar
      common/tmp/work(1000)
      do k = 1, 1000
         work(k) = ...
      enddo
      ....
      do k = 1, 1000
         ... = work(k)
      enddo
      end
```

Figure 13.2: Example: How to use IN and OUT regions to eliminate dependences

To summarize, IN and OUT regions could be used exactly as READ and WRITE regions to build the dependence graph. However, there is little room for any gain at this granularity. The sole interest is when there are procedure calls. But we have also seen that some gain can be expected at a higher granularity, especially when arrays are used as temporaries. Unfortunately, this is not handled by the usual dependence analysis scheme, and a particular analysis is devoted in PIPS to the detection of privatizable array sections (see Chapter 14).

13.3 Related Work

There has been a lot of work over the years about dependence analysis techniques, the main issues being complexity and accuracy: Cheap tests, such as the GCD test and BANERJEE's test [20] perform well on many cases, but more complex algorithms based on the FOURIER-MOTZKIN pair-wise elimination method, or on the simplex, are necessary to achieve a good accuracy. Therefore, dependence tests combining successive individual tests have been suggested [35, 105, 129, 175, 143]. If an individual test concludes to the existence or non-existence of a solution, then the analysis is complete, otherwise a more complicated test is applied. PIPS dependence test ranks among these approaches [105, 177].

However, most of the above studies only deal with individual references, and not with compound instructions, such as procedures. To analyze dependences across procedure boundaries, TRIOLET introduced a novel approach, based on array regions. Section 13.2.1 shows how this technique is applied in PIPS. Related works are numerous [162, 163, 122, 37, 19, 98, 99, 35, 92], the main issues being again complexity and accuracy, but with respect to the representation of array regions. More details on these studies can be found in Chapters 8 and 11.

In most parallelizing compilers (PETIT [176], PARAPHRASE-2 [142], FIAT/SUIF [92], PANORAMA [135], ...), induction variables are recognized and substituted [4, 3, 174, 180, 69], and dependence tests do not have to deal with variables modified in loop bodies. While this may sometimes allow to disprove dependences, this may also lead to non-linear subscripts, which often prevent subsequent dependence tests. Several studies have been dedicated to solving such problems in the general case [68, 97, 30, 89, 176]. By using regions, some dependences may be disproved, whereas other techniques based on linear algebra may fail. Consider for instance the program of Figure 13.3. After induction variable substitution, array subscripts are non-linear, and an advanced dependence test is required. On the contrary, using array regions of the inner-loop body, we can deduce that there is no loop-carried dependence between `a(k)` and `a(k-1)` at the innermost loop level.

All the previously cited works deal with *memory based* dependences. However, these techniques fail to discover parallelism hidden by variable reuse. Several approaches have therefore been designed to study *value based* dependences [34, 72, 130, 147]. They chiefly differ on the algorithms used to compute the dependences, which has an effect on complexity, and the restrictions on the source language (see Sections 6.10 and 14.5). IN and OUT regions were not designed to achieve the same accuracy as these methods. However, we have seen that they can be used to disprove interprocedural *value based* dependences, which is not done in the previously cited works. This is less accurate than the method proposed by LESERVOT [121], but the source language is much less restricted.

```
k = foo()                                  k0 = foo()
do i = 1, n            induction           do i = 1, n
  do j = 1,n           variable              do j = 1,n
     a(k) = 0.0        substitution            a(k0+n*(i-1)+j-1) = 0.0
     k = k + 1            ==>                  b(k0+n*(i-1)+j) =
     b(k) = a(k-1)                                    a(k0+n*(i-1)+j-1)
  enddo                                      enddo
enddo                                      enddo
```

Figure 13.3: Drawbacks of induction variable substitution

13.4 Conclusion

We have seen how easily convex array regions can be integrated in PIPS dependence analysis framework. Some precautions are nevertheless necessary to avoid losing accuracy where more classical dependence tests perform well. In particular, merging regions must be avoided at the level of *simple* statements, basically assignments and procedure calls, that is to say before any intraprocedural propagation. We call this type of information *proper regions* to recall that they exactly correspond to the statement they are attached to. *Proper regions* are implemented in PIPS as an intermediary of READ and WRITE region propagation: No additional analysis is thus needed.

Array regions could also be used to more efficiently disprove loop-carried dependences when there are induction variables. This is mainly due to the fact that for regions the reference memory store is dynamic, hence can be the closest one; whereas for preconditions, it is the entry store of the procedure.

IN and OUT regions can readily be used to disprove some value based dependences. However, apart from specific cases, no gain can be expected at a fine granularity, such as in PIPS dependence test: A coarser grain, such as procedure calls and whole loops or sequences of instructions, is necessary to be able to efficiently take them into account. The result would be a dependence graph in which some value based dependences have been removed. However, many memory based dependences still exist, and variables must be at least renamed to avoid memory clashes during execution. Since using IN and OUT regions is equivalent to declare arrays or array sections as local variables, a specific analysis has been preferred in PIPS. This is the object of the next chapter.

Chapter 14

Array Privatization

Recent studies [79, 29] have highlighted the need for advanced program optimizations to deal with memory management issues when compiling programs for massively parallel machines or hierarchical memory systems. For instance, BLUME and EIGENMANN [29] have shown that array privatization could greatly enhance the amount of potential parallelism in sequential programs. This technique basically aims at discovering array regions that are used as temporaries in loops, and can thus be replaced by local copies on each processor. Basically, an array region is said to be privatizable in a loop if each read of an array element is preceded by a write in the same iteration, and several different iterations may access each privatized array element [123, 166]. Solving such problems requires a precise intra- and inter-procedural analysis of array data flow, that is to say how individual array element values are defined and used (or *flow*) during program execution. In this chapter, we show how IN and OUT regions can be used to detect privatizable array regions.

The first section reviews several definitions of array privatization and related issues. Our algorithm is then presented in Section 14.2. Section 14.4 discusses code generation issues. The related work is then presented in Section 14.5.

14.1 Several Definitions

There are almost as many definitions of array privatizability as papers about it [123, 166, 92, 17]. In this section, we try to distinguish the different criteria for array privatizability which are used in these definitions; we discuss their usefulness, and give our own definition.

Scope The basic idea is to detect arrays or array regions that are used as temporaries in a particular fragment of code or lexical scope. When parallelizing scientific programs, the purpose is to eliminate spurious dependences between different iterations of a loop in order to parallelize it, as for scalar privatization [66, 138]. However, such techniques could also be used to restructure dusty desks, for instance by detecting arrays which are only used as local variables in several subroutines, but are declared as global variables to limit memory usage on a particular architecture, thus preventing further transformations. IN and OUT regions would provide the necessary informations to perform this type of privatization. However, we will only focus in this chapter on array privatization for loops.

Eliminating false dependences As just stated, our purpose is to eliminate spurious dependences (*i.e.* anti- and output-dependences) across iterations of a loop in order to expose parallelism. Such dependences appear when common parts of an array are used as temporaries by several iterations. However, detecting arrays used as temporaries even when it generates no dependence is also of interest for memory allocation. An application is the compilation of signal processing applications based on dynamic single assignment. Although we primarily focus on eliminating dependences, we will present in the next section the changes necessary when dependence removal is not the main concern.

Arrays or array sections In most cases, whole arrays are used as temporaries. However, it might not always be the case, for instance on array boundaries, or for global variables, which are declared once and used in several different ways along the program. In these cases, it is necessary to detect which part of the array is actually used as a temporary to allocate the minimum amount of memory for the new local array. This is what we call *array region privatization.*

Copy-in and copy-out In some cases, the array is used as a temporary, except in some iterations, which import some of its values from the instructions preceding the loop, or which exports some of its values towards the instructions following the loop. The first problem is often called the *copy-in* problem, and the second one, the *copy-out* or *finalization* problem.

Indeed, when these problems occur, the array is not an actual temporary data structure. However, it can be handled as such by declaring it as a local array, and importing (resp. exporting) the necessary elements from (resp. to) the original array. This is very similar to generating communications for a distributed memory machine. In our algorithm, both problems are handled, but this can be toggled off depending on circumstances such as the target architecture for instance. In a previous work [17], the copy-in problem was not considered.

To summarize, the algorithm presented in the next section detects privatizable array regions; when it is required, it handles the *copy-in* and *copy-out* problems; and though our primary goal is to remove spurious dependences, this feature can be toggled off for specific applications.

14.2 Array Region Privatizability Detection

The purpose is to find the set of array elements whose computation is local to a particular iteration of a loop. Several algorithms for scalar privatization have already been designed [66, 138]. However, they cannot be used as such because they would involve comparing or combining regions from any different statements, and it would be too expensive to compute the relations between the values of variables in the memory stores preceding these statements ($\frac{(n-1)(n-2)}{2}$ transformers for n statements, plus region combinations).

To avoid these problems, we propose in this chapter to use IN and OUT regions. Of course, since IN and OUT regions are summaries, some accuracy may be lost, in comparison with an extension of traditional scalar privatization algorithms. However, we expect that the accuracy will be sufficient for usual applications. In PIPS, regions are represented as convex polyhedra. However, the algorithm presented here is very general and does not rely on the chosen representation. To improve the precision of the results, in particular when computing the difference of two regions, or when eliminating variables, more powerful representations can thus be used.

14.2.1 Notations

In this chapter, we consider all types of loops, `DO` and `WHILE` loops. To handle the latter more easily, we introduce an iteration counter i, whose value is 0 before the loop, and k after k iterations.

We are interested in properties of one particular iteration, for instance the i-th iteration. Referring to these properties with the notations used in Chapter 6 would yield very long and unreadable equations. We thus adopt the following notations, assuming that $\mathcal{R}_{type}$ is an exact region of type *type*.

$$\mathcal{R}_{type}(i) = \mathcal{R}_{type}[\![\texttt{do while(}C\texttt{)}\ \ S]\!] \circ \mathcal{T}[\![\{S\}^{i-1}]\!]$$

and,

$$\bigcup_{i'<i} \mathcal{R}_{type}(i') = \mathcal{R}_{type}[\![i'=0;\texttt{do while(}C\texttt{.and.}i'<i\texttt{)}\ \ S;i'=i'+1]\!]$$

The corresponding approximations are similarly defined.

Finally, we will use the following new types of regions, only defined for loop bodies, and equal to the empty set otherwise:

$\mathcal{R}_{priv}$	private array regions
$\mathcal{R}_{copy\text{-}out}$	copy-out array regions
$\mathcal{R}_{copy\text{-}in}$	copy-in array regions
$\mathcal{R}_{out\_ncopy}$	array regions not candidate for copy-out
$\mathcal{R}_{in\_ncopy}$	array regions not candidate for copy-in

In fact, we only need an under-approximation of private array regions, that is to say, array elements which are certainly private. It would have no meaning to privatize array elements which may not actually be private. Similarly, if we do not import or export enough array elements when handling copy-in and copy-out problems, referring to them afterwards may yield unknown values. We thus compute over-approximations of $\mathcal{R}_{copy\text{-}in}$ and $\mathcal{R}_{copy\text{-}out}$.

14.2.2 Algorithm

This section presents the algorithm used in PIPS to detect privatizable array regions. It takes as input three different regions corresponding to the current loop body (iteration i): IN, WRITE, and OUT regions; And also three boolean values, to indicate which problem has to be solved: *copy_in_p* and *copy_out_p* are set to *true* to handle copy-in and copy-out problems, and *spurious_dep_p* is set to *true* when the purpose is to remove anti- and output-dependences, and set to *false* when a mere locality analysis is desired. The ouput consists in three regions, which were presented above: $\underline{\mathcal{R}}_{priv}$, $\overline{\mathcal{R}}_{copy\text{-}out}$, and $\overline{\mathcal{R}}_{copy\text{-}in}$, whatever the input options are.

1. `input:` $\overline{\mathcal{R}}_i(i)$, $\overline{\mathcal{R}}_w(i)$, $\overline{\mathcal{R}}_o(i)$, *copy_in_p*, *copy_out_p*, *spurious_dep_p*
2. `if` *copy_in_p* `then`
3. $\quad\overline{\mathcal{R}}_{in\_ncopy}(i) \;=\; \overline{\mathcal{R}}_i(i) \;\overline{\cap}\; \overline{\bigcup}_{i'<i}\; \overline{\mathcal{R}}_w(i)$
4. `else`
5. $\quad\overline{\mathcal{R}}_{in\_ncopy}(i) \;=\; \overline{\mathcal{R}}_i(i)$
6. `endif`
7. `if` *copy_out_p* `then`
8. $\quad\overline{\mathcal{R}}_{out\_ncopy}(i) \;=\; \overline{\mathcal{R}}_o(i) \;\overline{\cap}\; \overline{\bigcup}_{i'>i}\; \overline{\mathcal{R}}_i(i)$
9. `else`
10. $\quad\overline{\mathcal{R}}_{out\_ncopy}(i) \;=\; \overline{\mathcal{R}}_o(i)$
11. `endif`
12. $\underline{\mathcal{R}}_{priv}(i) \;=\; (\;\underline{\mathcal{R}}_w(i) \;\underline{\boxminus}\; \overline{\mathcal{R}}_{in\_ncopy}(i)) \;\underline{\boxminus}\; \overline{\mathcal{R}}_{out\_ncopy}(i)$
13. `if` *spurious_dep_p* `then`
14. $\quad\underline{\mathcal{R}}_{priv}(i) \;=\; (\underline{\mathcal{R}}_{priv}(i) \;\underline{\cap}\; \overline{\bigcup}_{i'<i}\; \underline{\mathcal{R}}_{priv}(i'))$
    $\qquad\qquad\qquad\underline{\cup}\; (\underline{\mathcal{R}}_{priv}(i) \;\underline{\cap}\; \overline{\bigcup}_{i'>i}\; \underline{\mathcal{R}}_{priv}(i'))$
15. `endif`
16. `if` *copy_in_p* `then`
17. $\quad\overline{\mathcal{R}}_{copy\text{-}in}(i) \;=\; \overline{\mathcal{R}}_i(i) \;\overline{\cap}\; \underline{\mathcal{R}}_{priv}(i)$
18. `else`
19. $\quad\overline{\mathcal{R}}_{copy\text{-}in}(i) \;=\; \lambda\sigma.\varnothing$
20. `endif`
21. `if` *copy_out_p* `then`
22. $\quad\overline{\mathcal{R}}_{copy\text{-}out}(i) \;=\; \overline{\mathcal{R}}_o(i) \;\overline{\cap}\; \underline{\mathcal{R}}_{priv}(i)$
23. `else`
24. $\quad\overline{\mathcal{R}}_{copy\text{-}out}(i) \;=\; \lambda\sigma.\varnothing$
25. `endif`
26. `output:` $\underline{\mathcal{R}}_{priv}(i)$, $\overline{\mathcal{R}}_{copy\text{-}out}(i)$, $\overline{\mathcal{R}}_{copy\text{-}in}(i)$

$\overline{\mathcal{R}}_{in\_ncopy}(i)$ contains the array elements which are not allowed to be imported if the array is privatized. If copy-in is not required (Step 5), then no array element which is imported by the iteration can belong to the set of element which will be imported and privatized. If copy-in is required (Step 3), privatized array elements can be imported, but solely from the statements preceding the loop, and not from the other iterations.

Similarly, $\overline{\mathcal{R}}_{out\_ncopy}(i)$ contains the array elements which are not allowed to be exported if the array is privatized. If copy-out is not required (Step 10), then no array element which is exported by the iteration can belong to the set of elements which will be exported and privatized. If copy-out is required (Step 8), array elements can be exported, but solely to the statements following the loop, and not to the other iterations.

Then, the set of privatizable elements is equal to the set of elements which are computed in the current iteration ($\underline{\mathcal{R}}_w(i)$) and which are allowed to be imported or exported before and after the loop (Step 12). If privatization is performed in order to remove spurious dependences (Step 14), then only array elements which may also be private in other iterations are to be taken into account.

At last, we generate over-approximations of the copy-in and copy-out regions. When copy-in or copy-out are not allowed, empty regions are returned. When they are allowed, copy-in regions are regions which are private and whose value is imported (Step 17). And copy-out regions are regions which are private and whose values are exported (Step 22).

14.3 Example

Let us compute the private regions of the outermost loop in the program of Figure 5.2. We only consider the array `WORK`, since the array `A` does not generate any inter-loop dependence. According to Section 8.5, the input to the privatization algorithm is the following:

$$\begin{aligned}
&\overline{\mathcal{R}}_w(i) = \underline{\mathcal{R}}_w(i) = \{\Phi_1, \Phi_2 : 1 \leq \Phi_1 \leq n \wedge k+i-1 \leq \Phi_2 \leq k+i\}\\
&\overline{\mathcal{R}}_i(i) = \varnothing\\
&\overline{\mathcal{R}}_o(i) = \varnothing
\end{aligned}$$

k and n being the values of `k` and `n` before the execution of the loop; and,

$$\begin{aligned}
&\mathit{copy_in_p} = \mathit{true}\\
&\mathit{copy_out_p} = \mathit{true}\\
&\mathit{spurious_dep_p} = \mathit{true}
\end{aligned}$$

Since no element of `WORK` is neither imported nor exported by any iteration of the loop, Steps 3 and 8 first give:

$$\begin{aligned}
&\overline{\mathcal{R}}_{in_ncopy}(i) = \varnothing\\
&\overline{\mathcal{R}}_{out_ncopy}(i) = \varnothing
\end{aligned}$$

The candidate privatizable region at Step 12 therefore is:

$$\underline{\mathcal{R}}_{priv}(i) = \underline{\mathcal{R}}_w(i) = \{\Phi_1, \Phi_2 : 1 \leq \Phi_1 \leq n \wedge k+i-1 \leq \Phi_2 \leq k+i\}$$

Since $\mathit{spurious\_dep\_p} = \mathit{true}$, we must then update this result according to Step 14. We successively have:

$$\begin{aligned}
&\overline{\bigcup_{i'<i}} \underline{\mathcal{R}}_{priv}(i') = \{\Phi_1, \Phi_2 : 1 \leq \Phi_1 \leq n \wedge k \leq \Phi_2 \leq k+i-1\}\\
&\overline{\bigcup_{i'>i}} \underline{\mathcal{R}}_{priv}(i') = \{\Phi_1, \Phi_2 : 1 \leq \Phi_1 \leq n \wedge k+i \leq \Phi_2 \leq n\}\\
&\underline{\mathcal{R}}_{priv}(i) = \{\Phi_1, \Phi_2 : 1 \leq \Phi_1 \leq n \wedge k+i-1 \leq \Phi_2 \leq k+i\}
\end{aligned}$$

```
KO = FOO()
DOALL I = 1, N
PRIVATE WORK,J,K
   K = KO+I-1
   DOALL J = 1, N
     WORK(J,K) = J+K
   ENDDO
   CALL INC1(K)
   DOALL J = 1, N
     WORK(J,K) = J*J-K*K
   ENDDO
   DO J = 1, N
      A(I) = A(I)+WORK(J,K)+WORK(J,K-1)
   ENDDO
ENDDO
```

Figure 14.1: Parallel version of the program of Figure 5.2

Steps 17 and 22 then give:

$$\overline{\mathcal{R}}_{copy\text{-}in}(i) = \varnothing$$
$$\overline{\mathcal{R}}_{copy\text{-}out}(i) = \varnothing$$

Therefore, the computations concerning the array WORK are entirely local to each iteration, and it can be privatized. Since the current implementation in PIPS can only privatize whole arrays, the code generation phase does not take advantage of the fact that only a small part of the array is actually private. After performing induction variable substitution by hand, PIPS generates the parallel code of Figure 14.1.

We can now assume that part of the array WORK is reused after the execution of the loop, and that, for instance,

$$\overline{\mathcal{R}}_o(i) = \{\Phi_1, \Phi_2 : 1 \leq \Phi_1 \leq n \wedge \Phi_2 = k + i - 1\}$$

We then successively have:

$$\overline{\mathcal{R}}_{in\_ncopy}(i) = \varnothing$$
$$\overline{\mathcal{R}}_{out\_ncopy}(i) = \varnothing$$
$$\underline{\mathcal{R}}_{priv}(i) = \{\Phi_1, \Phi_2 : 1 \leq \Phi_1 \leq n \wedge k + i - 1 \leq \Phi_2 \leq k + i\}$$
$$\overline{\mathcal{R}}_{copy\text{-}in}(i) = \varnothing$$
$$\overline{\mathcal{R}}_{copy\text{-}out}(i) = \{\Phi_1, \Phi_2 : 1 \leq \Phi_1 \leq n \wedge \Phi_2 = k + i - 1\}$$

Here, the privatizable region is the same as previously, but each iteration must update a different column of the initial array. The code generated in this case would certainly depend on the target architecture.

14.4 Code Generation

The previous section was devoted to array privatizability detection, under several possible options. Given the resulting regions, the problem is then to generate code using this information. This has already been handled by others [6]. Though it is not our main concern in this study, we give here a few hints about the problems which arise.

Once privatizable array regions have been determined, the dependence graph must be refined. In fact, READ and WRITE regions are recomputed, private regions being taken into account. The dependence analysis is then performed with the new regions. For loops which still cannot be parallelized, no particular work has to be done as far as privatizable arrays are concerned. However, for parallelizable loops, private, copy-in and copy-out regions must be considered.

The generated code will of course depend on the target architecture. On a distributed memory machine, private copies of private regions must be allocated on each processor; while on shared memory systems, private copies with different names must be allocated in the global memory for each processor. Generating these private copies from array regions represented by convex polyhedra, and converting former accesses to the global array into references to the local array, are themselves not trivial issues (see [12, 145, 44, 43, 6]).

Handling copy-in and copy-out regions also depends on the target parallel model. In a message passing paradigm, copy-in sets must be converted into messages to fetch values from the global arrays to initialize local copies; and copy-out sets must be converted into messages to update global arrays from the local copies. When compiling for a shared memory machine, or virtual shared memory system, the local array must be initialized with values from the global copy and conversely.

Another issue is whether privatizing is worth doing on a particular architecture, especially when dealing with copy-in and copy-out regions. Whether it is more profitable to parallelize a loop, allocate copies of private arrays, initialize them from the global arrays, perform the computation and finally export some local values to the global array, or to sequentially execute the loop, is still an open question. In the case of a loop nest, the level at which each array is privatized can also influence the performances.

14.5 Related Work

Many work has already been done to optimize locality on distributed or hierarchical memory machines, in order to expose parallelism, or to enhance performances degraded by communications or cache misses. The first approach is based on loop transformations (loop reversal, fusion and interchange in [155], compound loop transformation with loop interchange, reversal, skewing, and tiling in [172], loop permutation in [114] and loop interchange and blocking in [80, 31]), basically to concentrate near array references in innermost loops, so that processors access compact sets of data (*array contraction* [155]). The second approach is to transform the array element layout in shared memory so that all the data accessed by one processor are contiguous in the shared address space [14].

However, these techniques reorganize the computation or the data sets so that accesses are contiguous, but they do not remove dependences when an array region is used as a temporary by several iterations. It was shown that, in several benchmarks, many loops are not parallelizable because of this problem [29]. The last approach for enhancing the locality of references is then to privatize or expand array regions, that is to say to replicate data onto several processors when they are used as temporaries. Many work has already been done is this direction [123, 165, 166, 164, 92, 70, 130, 17], and the differences with our approach are described below. Notice also that run-time techniques have been recently introduced to privatize arrays in cases which cannot be discovered using static analyses [148, 149]. We do not further present them because they are beyond the scope of our study.

Li [123]

Li was the first to propose an algorithm for array privatization. His framework includes the handling of the *copy-out* problem, but not *copy-in*. However, his approach is intraprocedural and has not been extended to handle procedure calls, at least to our knowledge. The algorithm is very general, and not bound to any representation of array regions.

Tu and Padua [165, 166, 164]

Tu and Padua then presented an interprocedural algorithm, using *MustDefine* and *PossiblyExposedUse* sets, represented by RSDs with bounds, strides and unit slopes. They treat the *copy-out* problem by performing an additional liveness analysis. The *copy-in* problem is not handled.

Hall et al. [92]

FIAT/Suif uses an extension of Tu and Padua algorithm which handles the *copy-in* problem. However, no interprocedural liveness analysis is performed to handle the *copy-out* problem. Instead, when the last iteration of the loop is the last defining iteration for the privatized section, the last iteration of the loop is set apart, and sequentially executed after the loop nest with the privatized array. This alleviates costly analyses, but is less general than our approach.

Feautrier [70]

For shared memory machines, Feautrier proposes to perform another transformation called *array expansion* which extends arrays to higher dimensions to transform the program into single assignment form. The purpose is the same as *array privatization*, that is to say to remove spurious dependences. *Copy-in* and *Copy-out* problems do not appear anymore, because of single assignment form: Elements imported by the loop cannot be over-written by the loop; and elements exported by some iterations cannot be over-written by other iterations. The elements to prefetch in local or cache memory can nevertheless be obtained from the source functions of all the references in the current loop; and the elements to keep in local memories correspond to the sources in the program continuation. However, array expansion is a memory consuming techniques, and requires additional program transformations such as *array contraction* [155] to reduce the amount of allocated memory.

Finally, this approach is restricted to *static control* programs, that is to say mono-procedural programs with linear loop bounds, test conditions and array subscripts.

MAYDAN **et al. [130]**

MAYDAN proposes a whole array privatization algorithm based on an efficient implementation of FEAUTRIER's array data flow analysis [70, 72]. The input language is very restricted (monoprocedural static control programs), and whole arrays are privatized instead of array sections. This last constraint is lessened by the fact that copy-in and copy-out are handled. This method has two drawbacks. The first is that too many array elements are declared as local. In another work from the same team [6], this problem is tackled when generating the code. The second problem lies in the fact that the copy-out set is equal to the set of outward-exposed definitions, and not to the outward exposed definitions actually used in the continuation of the program, as OUT regions.

BRANDES **[34]**

BRANDES also proposes an algorithm for array expansion based on direct dependence analyses, for monoprocedural general control flow programs.

CALLAND **et al. [38, 39]**

.

Another array expansion method is proposed by CALLAND. It is a generalization of two program transformations originally introduced by PADUA and WOLFE in [136] to remove anti and output dependences. The transformed loops can then be parallelized using ALLEN and KENNEDY algorithm [4].

CREUSILLET **[17]**

In a previous work, we presented another algorithm also based on IN and OUT regions. This algorithm did not cope with the copy-in problem, and the approximations of the many intermediate regions used, as well as the resulting regions, were not clearly given. The final version of the algorithm presented above directly derives from the former algorithm, and benefits from our experience on approximations and exactness.

14.6 Conclusion

The main goal of this chapter was to show how IN and OUT regions can be used to privatize array regions. The resulting algorithm is very general: it can be used to detect computations local to a particular scope (no copy-in and copy-out, no consideration of false dependences), but also to detect privatizable sections when importing or exporting values is allowed, or when the purpose is to remove spurious dependences to enhance the amount of potential parallelism. To our knowledge, this is the first time each problem is isolated in such a way while presented in a uniform framework. Moreover, the approximation of each type of region is clearly given, and is ensured by the use of safe operators.

The associated code generation is still to be defined. More work is necessary to take several architectural or computation paradigms into account. Also, array privatization was presented for loops. But the same idea could be used to restructure array declarations in subroutines, when a global array is in fact used as a temporary array. The algorithm trivially derives from the algorithm given above; but code generation is again the remaining issue.

VI

CONCLUSION

Chapitre 15

Contributions et Perspectives

Les analyses de régions de tableaux ont suscité de nombreuses études lors de la dernière décennie. Leur utilité n'est plus à démontrer [163, 29, 101, 92, 65], et les débats actuels concernent le choix de la représentation des ensembles d'éléments de tableaux, ainsi que le type et la quantité d'information symbolique à prendre en compte pour analyser de manière suffisamment précise des applications réelles, sans pour autant dégrader la complexité en temps et en espace mémoire.

Cependant, alors que les analyses sur-estimées ont été étudiées en profondeur sur le plan formel et sont maintenant des techniques bien maîtrisées, les analyses sous-estimées ont été utilisées sans autre forme de procès, de manière empirique. Nous avons donc cherché dans cette thèse à proposer un cadre formel unifié pour toutes les analyses de régions de tableaux.

Dans ce domaine, nos principales contributions sont les suivantes:

- Nous avons rappelé que les problèmes que posent les analyses de régions sous-estimées lorsque le domaine choisi n'est pas clos pour l'union ensembliste : dans ce cas, il n'existe pas de meilleure définition pour les sous-approximations. Une solution est de changer le domaine d'étude ; par exemple, le domaine des formules de PRESBURGER est clos pour l'union. WONNACOTT [176] obtient des résultats encourageants en terme de complexité pour des analyses intraprocédurales. Mais d'autres seraient nécessaires pour valider l'utilisation des formules de PRESBURGER dans un cadre interprocédural.

  Les autres solutions existantes ne donnant pas de résultats satisfaisants, nous proposons de définir les régions sous-estimées à partir des régions sur-estimées à l'aide d'un critère d'exactitude calculable. Dans la pratique, nous suggérons même de n'effectuer que l'analyse sur-estimée, et de la marquer comme *exacte* lorsque le critère d'exactitude est vrai; en cas d'exactitude, la régions sous-estimée est alors égale à la régions sur-estimée; dans le cas contraire, c'est l'ensemble vide.

- Nous avons défini précisément les régions comme étant des fonctions de l'ensemble des états mémoire ($\mathbf{\Sigma}$) vers les parties de $\mathbb{Z}^n$. La prise en compte de l'état mémoire est nécessaire car il permet de ne pas manipuler directement les valeurs de variables, et facilite donc les analyses symboliques. Des études précoces [37, 19] proposaient des représentation des régions ne prenant pas en compte cette dimension, et ne permettaient donc pas des analyses précises lorsque des variables étaient modifiées dans le corps du programme, ou en présence d'instructions conditionnelles.

- Nous avons défini les sémantiques exactes et approchées de plusieurs types de régions: READ, WRITE, IN et OUT, les deux premières abstrayant les effets des instructions sur les ensembles d'éléments de tableaux, les deux dernières pouvant être utilisées pour effectuer des analyses du flot des données. Le formalisme utilisé a l'avantage de ne pas reposer sur une représentation particulière du programme. Ceci nous a permis de comparer sur un même plan plusieurs approches issues de la littérature.

  Nous avons montré la nécessité d'analyses préliminaires : évaluation des expressions, *transformeurs*, et *conditions de continuation*. Les transformeurs permettent de prendre en compte les modifications des valeurs de variables. Les conditions de continuation sont utilisées pour prendre en compte les instructions d'arrêt (`stop`).

  Par contre nous avons montré que, même si elles sont utiles lors des combinaisons ou comparaisons de régions, la prise en compte des *préconditions* n'affecte pas la sémantique de ces analyses.

Ce cadre formel étant indépendant de toute représentation particulière des régions, nous avons ensuite décrit l'implantation effectuée dans PIPS, où les régions sont représentées par des polyèdres convexes paramétrés par les valeurs des variables scalaires entières du programme. Les régions ainsi définies sont donc bien des fonctions de l'état mémoire courant, ce qui permet d'effectuer des analyses symboliques tant que les contraintes de linéarité sont bien respectées. Les opérateurs abstraits approchés d'union, d'intersection, de différence, et les lois de composition externes ayant été utilisés pour décrire les sémantiques des analyses de régions ont été définis, ainsi que leurs critères d'exactitude.

Une partie de notre étude a également été consacrée à l'analyse interprocédurale des régions de tableaux. Nous avons notamment proposé un nouvel algorithme de traduction des régions de tableaux entre procédures, utilisant une approche exclusivement linéaire, mais plus puissant que les algorithmes existants. Son originalité réside dans sa capacité à préserver autant que possible les relations qui existent entre les dimensions des tableaux considérés. De plus, il permet de traiter les cas où les tableaux en jeu appartiennent à des `COMMON`s déclarés différemment dans les procédures source et cible, ainsi que les cas où les tableaux sont de types différents, cas qui apparaissent effectivement dans des applications réelles.

Enfin, nous avons présenté deux applications possibles des analyses de régions de tableaux : l'analyse des dépendances en présence d'appels de procédures, et la privatisation de tableaux. Cette dernière transformation a fait l'objet d'un algorithme original de détection des régions de tableaux privatisables, prenant en compte de manière uniforme les problèmes d'initialisation des tableaux privés en entrée de boucle, et de mise à jour des tableaux globaux en sortie de boucle. Cet algorithme repose sur l'utilisation des régions WRITE, IN et OUT définies précédemment.

L'implantation actuelle dans PIPS comporte les régions READ, WRITE, IN et OUT, ainsi que l'utilisation des régions pour l'analyse des dépendances[1], et pour la détection des sections de tableaux privatisables. Ceci représente plus de 12000 lignes de

[1] Une implantation existait déjà dans PIPS, mais elle utilisait des régions *résumées* et insensibles au

code C, dont 1500 pour la traduction interprocédurale, 3100 pour les opérateurs sur les régions, et 800 pour la privatisation de tableaux; les opérateurs et la traduction interprocédurale reposent sur la bibliothèque d'algèbre linéaire C3, dans laquelle sont intégrés la bibliothèque *Polyhedron Library* de l'IRISA [169], et une extension aux listes de polyèdres et leurs complémentaires par LESERVOT [120, 121]. La figure 15.3 montre comment nos réalisations s'enchaînent avec les autres phases et transformations de PIPS. Les parties les plus foncées montrent nos réalisations, et les parties hachurées les autres phases auxquelles nous avons contribué.

Des expériences qualitatives ont été effectuées pour valider cette implantation. Des tests de non régression sont effectués chaque jour. Ils comportent une application complète (`ONDE24`, de l'Institut français du pétrole), et plusieurs extraits d'applications réelles, ainsi que des tests systématiques portant sur certains points particuliers de l'implantation (comme la traduction interprocédurale par exemple). De plus, plusieurs applications réelles ont été analysées pour tester la robustesse de l'implantation: Le benchmark du Perfect Club `OCEAN` 2900 lignes); `DYNA` [167], un programme d'astrophysique de plus de 4000 lignes; un programme de l'entreprise RENAULT d'environ 50000 lignes; et trois programmes de l'ONERA, `AILE` (3000 lignes), `CR2CNF` (1020 lignes), et `OA118` (1450 lignes). Nous nous sommes plus particulièrement intéressés aux trois dernières applications, car des résultats d'expériences antérieures étaient disponibles. Pour ces codes, l'utilisation des régions et de la privatisation de tableaux dans les boucles et au niveau des procédures ont permis de paralléliser avec succès plusieurs boucles comportant des appels de procédure.

Cette étude concernant plusieurs domaines, les futurs travaux qui en découlent et les perspectives qu'elle ouvre sont de plusieurs natures:

Applications des régions Nous avons présenté deux applications possibles des régions de tableaux : l'analyse des dépendance et la détection des régions de tableaux privatisables. Mais nous avons vu dans l'introduction que bien d'autres applications étaient envisageables. Il conviendrait de vérifier l'adéquation effective et la faisabilité de ces propositions. Notamment, une étude comparative serait nécessaire pour démontrer l'intérêt particulier des régions OUT par rapport à d'autre techniques utilisées dans d'autres paralléliseurs.

Génération de code Nous avons volontairement limité cette étude aux analyses de programme. Tout un travail reste à réaliser pour exploiter les résultats de la détection des régions de tableau privatisables et générer du code pour machine à mémoire distribuée.

D'autres représentations La représentation choisie pour les régions, les polyèdres convexes en nombre entiers, a l'avantage de limiter la complexité de l'analyse tout en permettant de représenter une grande variété de formes d'ensembles d'éléments de tableaux, et de prendre en compte des informations sur le contexte courant. Mais elle ne permet pas de représenter certains *motifs* d'accès qui apparaissent dans des applications réelles, comme les *stencils* à neuf points utilisés dans des calculs sismiques (voir figure 15.1), ou des schémas d'accès utilisées

flot de contrôle au lieu des régions *propres* (voir chapitre 12), et donnait donc des résultats manquant de précision.

dans les calculs dits *red-black* (voir figure 15.2). Nous prévoyons donc d'implanter d'autres représentations des régions, sous forme de listes de polyèdres ou de Z-polyèdres [9].

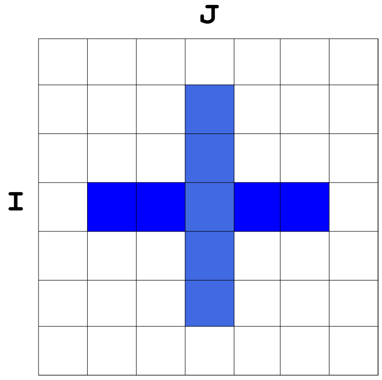


Figure 15.1: *Stencil* à neuf points

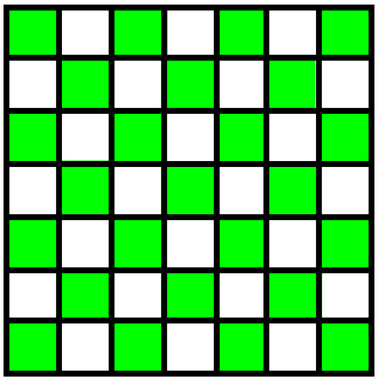


Figure 15.2: Schéma d'accès dans les calculs dits *red-blacks*

Expériences Nous avons déjà effectué avec succès plusieurs expériences sur les régions de tableaux et la privatisation, démontrant ainsi la robustesse de l'implantation. Mais d'autres expériences mesurant le gain obtenu après analyse des dépendances avec les régions et privatisation des tableaux seraient nécessaires. C'est l'un de nos projets.

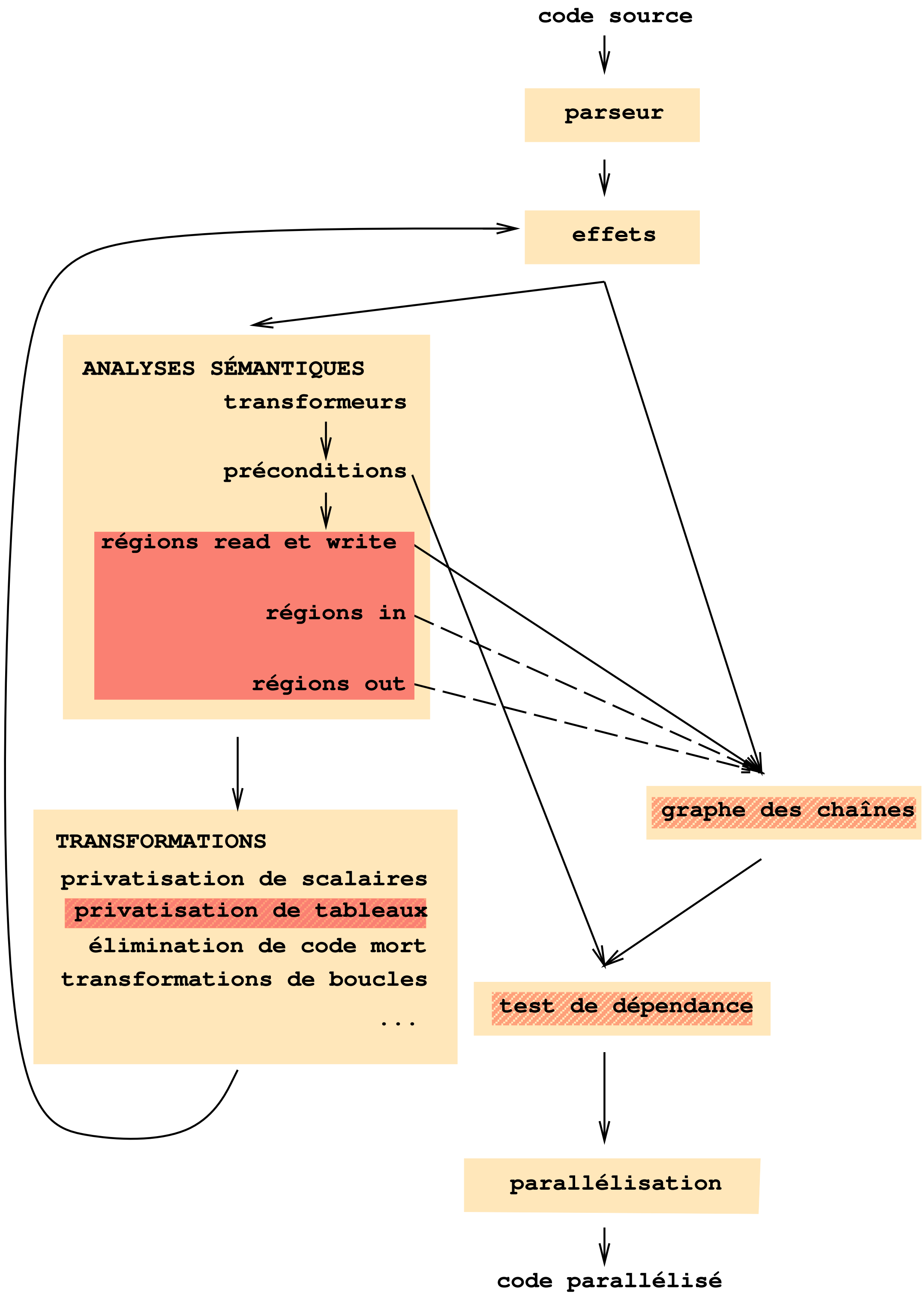


Figure 15.3: Phases et transformations de PIPS

Chapter 16

Contributions and Perspectives

Array region analyses have raised a considerable amount of effort over the last decade. Their usefulness is now widely acknowledged [163, 29, 101, 92, 65], and current debates are concerned with the representation of array element sets, and the type and amount of symbolic information required to handle real life programs, time and space complexity *versus* accuracy being the main issue.

However, while over-approximate array region analyses have been extensively studied on a formal ground and are now well-known techniques, under-approximate analyses have been empirically used without further ceremony! One of the purposes of this thesis was therefore to provide a unified formal framework for all types of array region analyses.

In this domain, our main contributions are the following:

- We have recalled the problems which arise for under-approximate array region analyses when the chosen domain is not closed under set union. In this case, under-approximations are not uniquely defined. One solution is to switch to another domain which is closed under union, such as PRESBURGER formulae. WONNACOTT [176] describes encouraging results at the procedure level as far as complexity is concerned, but more experiments would be useful to validate the use of PRESBURGER formulae in an interprocedural setting. Other existing solutions do not yield satisfying results.

  Hence we propose to compute under-approximate regions from over-approximate ones, using a computable exactness criterion. In practice, we even suggest to only perform the over-approximate analysis, and to flag the results as exact according to the exactness criterion; in case of exactness, the under-approximation is implicitly equal to the over-approximation; in the opposite case, it is the empty set.

- We have precisely defined array regions as functions from the set of memory stores ($\mathbf{\Sigma}$) to the powerset of $\mathbb{Z}^n$. Taking stores into account is necessary to perform symbolic analyses where variable values are represented by variable names. Some early studies were based on representations which cannot be used to integrate this dimension, and thus do not allow precise analyses when variable values are modified, or when there are conditional instructions.

- We have defined the exact an approximate semantics of several types of array regions: READ, WRITE, IN and OUT. The first two analyses abstract the effects of instructions upon array element sets. The other analyses can be used to perform array data flow analyses. The advantage of the chosen formalism is the independence it provides from any array region and program representation. We could therefore compare several related works on the same basis.

We have shown that several preliminary analyses are necessary to ensure accurate and safe approximations: Expression evaluation, transformers and continuation conditions. Transformers are used to evaluate the modification of variable values; and continuation conditions to model the effect of `stop` statements.

On the contrary, we have shown that using preconditions does not affect the semantics of array region analyses, even if they are useful when combining or comparing regions.

This formal framework being independent of any particular representation of array regions, we have then described their implementation in PIPS as convex polyhedra, parameterized by the values of the program integer scalar variables. Our regions are therefore functions of the current memory store, and allow some symbolic analyses, as long as linearity constraints are met. Abstract operators (union, intersection, difference and external composition laws), used to specify the semantics of array region analyses, have been instantiated for the chosen representation, and their exactness criteria have been defined.

One part of our study was also devoted to the interprocedural analysis of array regions. In particular, we have proposed a new algorithm to translate array regions across procedure boundaries. This algorithm is exclusively based on linear techniques, but is nevertheless more powerful that existing methods relying on the same mathematical framework. Its originality lies in its ability to preserve relationships between the dimensions of the source and target arrays as much as possible. In addition, it handles the cases in which arrays belong to `common` blocks declared differently in the two procedures, and the cases where the arrays have different types. These situations actually occur in real life programs.

Finally, we have presented two possible applications of array region analyses: Interprocedural dependence analyses, and array privatization. For this last transformation, we have designed an original algorithm for the detection of privatizable array regions, which provides the necessary information to handle initialization and updating problems of privatized or initial arrays before and after the actual loop execution (the so-call copy-in and copy-out problems). This algorithm uses the WRITE, IN and OUT regions previously defined.

The actual implementation in the PIPS parallelizer includes the intra- and interprocedural propagation of READ, WRITE, IN and OUT regions, as well as their use to enhance dependence analysis[1], and for the detection of privatizable array regions. This implementation represents more than 12000 lines of C code: 1500 for the interprocedural translation, 3100 for the operators on array regions, and 800 for array privatization. The operators and the interprocedural translation rely on a linear algebra library (C3), in which are included a *Polyhedra Library* from IRISA [169], and an extension to lists

[1] An implementation had already been done in PIPS, but it was based on summary flow insensitive array regions instead of proper regions. And the results were sometimes less precise than with proper regions.

of polyhedra and their complements due to LESERVOT [120, 121]. Figure 16.3 shows how our realizations (in dark rectangles) interact with the other phases of PIPS. The hatched rectangles show the phases to which we also contributed.

Qualitative experiments have been performed to validate this implementation. Non-regression tests are run every day. They include one complete application (`ONDE24`, from the French petroleum institute), and several excerpts of real applications, as well as systematic tests for specific parts of the implementation (e.g. interprocedural translation, or loops). In addition, several real applications have been analyzed to test the robustness of the implementation: The Perfect Club benchmark `OCEAN` (2900 lines); an astrophysical program (`DYNA` [167], more than 4000 lines); a code from the French company RENAULT (around 50000 lines); and three programs from the French institute ONERA, `AILE` (3000 lines), `CR2CNF` (1020 lines), and `OA118` (1450 lines). We focused more particularly on the last three applications, for which results of early experiments were available. And we found out that several loops containing procedure calls were successfully parallelized, thanks to array regions and array privatization (both at the level of loops and at the level of procedures).

Since this study approaches several subjects, the future works which can be derived, and the perspectives it opens are of several natures:

Applications of array regions We have presented in this thesis two possible applications of array region analyses: Dependence analysis, and detection of privatizable array regions. But, as already noted in the introduction, there are several other potential applications. It would be interesting to verify that such applications are adequate and feasible. In particular, a comparative study would be necessary to demonstrate the unique interest of using OUT regions compared to other techniques used in other parallelizers.

Code generation The framework of this study was purposely restricted to program analyses. Some more work is however necessary to exploit all the results of the detection of privatizable array regions, and generate code for distributed memory machines.

Other representations The chosen representation for array regions, parameterized integer convex polyhedra, allows various patterns and provides the ability to exploit context information at a reasonable expense. However, it cannot represent some very common patterns such as nine-points stencils used in seismic computations (see Figure 16.1) or such as red-black computation patterns (see Figure 16.2). We therefore intend to implement other types of representations such as lists of polyhedra or Z-polyhedra [9].

Experiments We have already performed successful experiments with array region analyses and array privatization, thus assessing the robustness of our implementation. However, more experiments measuring the gain obtained by using regions for the dependence analysis and by applying array privatization would be necessary. This is one of our projects for a near future.

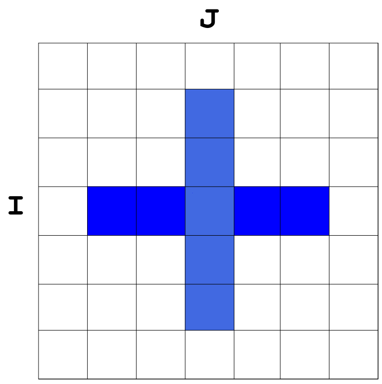


Figure 16.1: Nine-points stencil

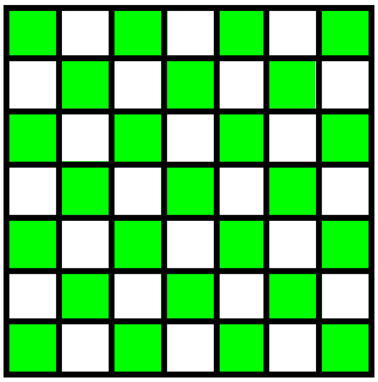


Figure 16.2: Red-black computation pattern

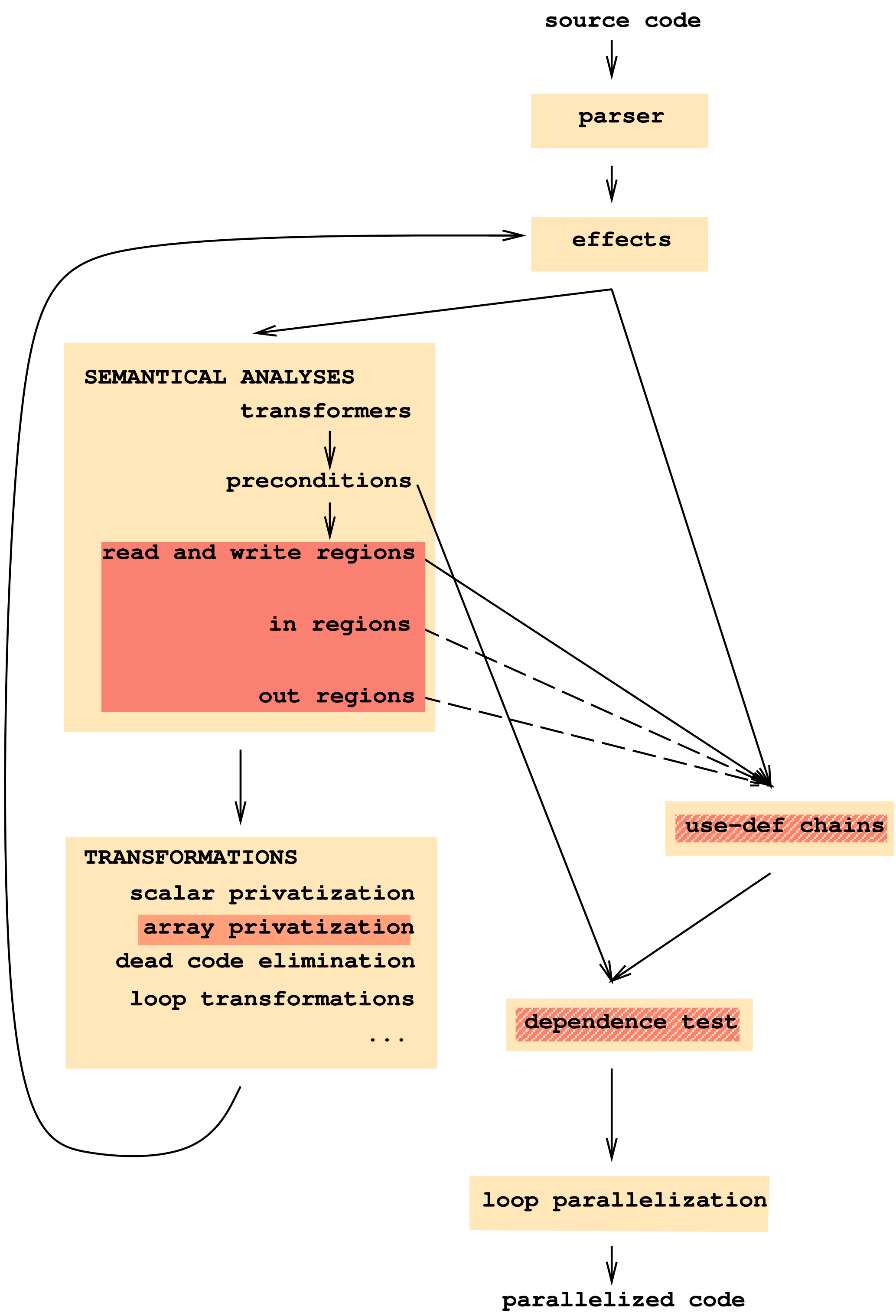


Figure 16.3: PIPS phases

VII

ANNEXE

APPENDIX

Appendix A

PIPS Overview

We reproduce here a technical report describing PIPS internals, with the permission of the main author, Ronan KERYELL.

A.1 Introduction

Detecting the maximum level of parallelism in sequential programs has been a major source of interest during the last decade. This process is of utmost importance when trying to cope with the increasing number of vector and parallel supercomputers and, perhaps unfortunately, the huge number of already existing "dusty deck" sequential programs. Nowadays, since the development of high-end parallel computers has not fulfilled the wildest hopes of entrepreneurial companies in the parallel supercomputing business, we can expect more down-to-earth developments of work-group parallel servers and workstations with few but powerful processors.

Together with this architectural evolution, three main directions in compiler developments have been pursued: compilers for sequential machines (with superscalar processors), parallelizers of sequential programs and compilers for explicit parallel programs. All these approaches benefit from deep global program analyses, such as interprocedural and semantical analyses, to perform optimizations, vectorization, parallelization, transformations, restructuring and reverse-engineering of programs. Since these approaches often require the same or share a common ground of analyses, it is interesting to factorize them out in a common development tool to get the benefit of code re-use and modular programming.

PIPS is such a highly modular workbench for implementing and assessing various interprocedural compilers, optimizers, parallelizers, vectorizers, restructurers, etc. without having to build a new compiler from scratch. It has been used in various compilation areas since its inception in 1988 as the PIPS project (Interprocedural Parallelization of Scientific Programs). PIPS has been developed through several research projects funded by DRET (French ARPA), CNRS (French NSF) and the European Union (ESPRIT programs). The initial design was made by Rémi TRIOLET, François IRIGOIN and Pierre JOUVELOT. It has proved good enough since then not to require any major change.

Our project aims at combining advanced interprocedural and semantical analyses [163] with a requirement for compilation speed. The mathematical foundations of the semantical analysis implemented in PIPS are linear programming and polyhedra theory. Despite such advanced techniques, PIPS is able to deal with real-life programs such as benchmarks provided by ONERA (French research institute in aeronautics), or the Perfect Club in quite reasonable time. An excerpt of a fluid dynamics code for a wing analysis (Figure A.2) will be used as a running example in this paper.

PIPS is a multi-target parallelizer. It incorporates many code transformations and options to tune its features and phases. PIPS offers interesting solutions to solve programming design problems such as generating high-level data structures from specifications, co-generating their documentation, defining user interface description and configuration files, dealing with object persistence and resource dependences, being able to write parts of PIPS in various languages, etc.

This paper details how PIPS achieves these goals. We first present a general overview of PIPS in the next section. In Section A.3 we describe the mechanisms that deal with PIPS data structures, their dependences and their interprocedural relationships. The general design is discussed in Section A.3. The programming environment, including the data structures generator and the linear library, and the documentation process is presented in Section A.4. At last, the different user interfaces in Section A.5 are presented before the related work in Section A.6.1.

A.2 Overview

PIPS is a source-to-source Fortran translator (Figure A.1), Fortran still being the language of choice for most scientific applications. Beside standard Fortran 77, various dialects can be used as input (HPF) or output (CMF, Fortran 77 with Cray directives, Fortran 77 with HPF parallelism directives, etc.) to express parallelism or code/data distributions. Fortran can be seen here as a kind of portable language to run on various computers without having to deal with an assembly code back-end, although the PIPS infrastructure could be used as a basis to generate optimized assembly code. Other parsers or prettyprinters could be added and the internal representation could be extended if need be to cope with other imperative languages such as C.

Following the field research history, PIPS has first been used to improve the vectorization of Fortran code for parallel vector computers with shared memory (Fortran 77 with `DOALL`, Fortran 77 with Cray micro-tasking directives, Fortran 90). It is now mainly targeted at generating code for distributed memory machines, using different methods (processor and memory bank code for control distribution [13], CMF, CRAFT polyhedral method [140] or message-passing code from HPF [45]).

As we see in Section A.3, the translation process is broken into smaller modular operations called phases in PIPS. For the user, these phases can be classified into various categories according to their usage. Since PIPS is an interprocedural environment, procedures and functions are very important in PIPS and are referred to as *module* in the sequel.

A.2.1 Analyses

An analysis phase computes some internal information that can be later used to parallelize (or generate) some code, some user information (such as a program complexity measure) to be later displayed by a prettyprinter, etc. The most interesting analyses available in PIPS are the semantical analyses described below.

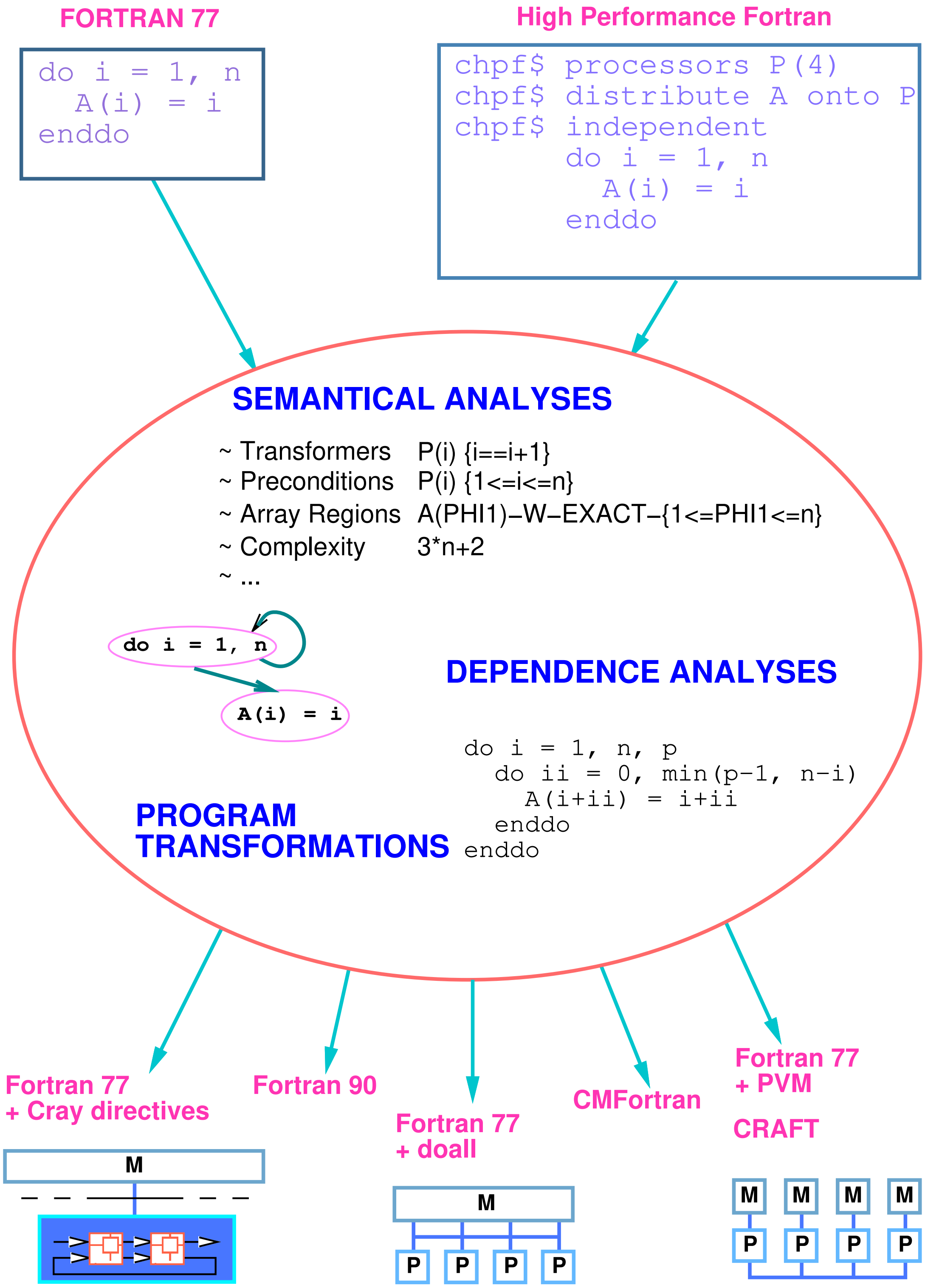


Figure A.1: User view of the PIPS system.

```
c
      PROGRAM EXTRMAIN
c
      DIMENSION T(52,21,60)
      COMMON/CT/T
      COMMON/CI/I1,I2,...
      COMMON/CJ/J1,J2,JA,...
      COMMON/CK/K1,K2,...
      COMMON/CNI/L
      DATA N1,N4,N7,N17 /1,4,7,17/

      READ(NXYZ) I1,I2,J1,JA,K1,K2
      REWIND NXYZ
c
      IF(J1.GE.1.AND.K1.GE.1) THEN
         N4=4
         J1=J1+1
         J2=2*JA+1
         JA=JA+1
         K1=K1+1
         CALL EXTR(N7,N17)
         CALL EXTR(N4,N17)
         CALL EXTR(N1,N17)
      ENDIF
      END
c
c
c Compute the extrapolation coefficients
c for all the wing area (K=K1)
c
      SUBROUTINE EXTR(NI,NC)
c
      DIMENSION T(52,21,60)
      COMMON/CT/T
      COMMON/CI/I1,I2,...
      COMMON/CJ/J1,J2,JA,...
      COMMON/CK/K1,K2,...
      COMMON/CNI/L
      L=NI
      K=K1
      DO 300 J=J1,JA
         S1=D(J,K  ,J,K+1)
         S2=D(J,K+1,J,K+2)+S1
         S3=D(J,K+2,J,K+3)+S2
         T(J,1,NC+3)=S2*S3/((S1-S2)*(S1-S3))
         T(J,1,NC+4)=S3*S1/((S2-S3)*(S2-S1))
         T(J,1,NC+5)=S1*S2/((S3-S1)*(S3-S2))
         JH=J1+J2-J
         T(JH,1,NC+3)=T(J,1,NC+3)
         T(JH,1,NC+4)=T(J,1,NC+4)
         T(JH,1,NC+5)=T(J,1,NC+5)
300   CONTINUE
      END
c
c
c Compute D=DISTANCE
c
      REAL FUNCTION D(J,K,JP,KP)
c
      DIMENSION T(52,21,60)
      COMMON/CT/T
      COMMON/CNI/L
c
      D=SQRT((T(J,K,L  )-T(JP,KP,L  ))**2
     1     +(T(J,K,L+1)-T(JP,KP,L+1))**2
     2     +(T(J,K,L+2)-T(JP,KP,L+2))**2)
      END
```

Figure A.2: Code excerpt from an ONERA benchmark.

Effects

Since an imperative language such as Fortran deals with memory locations, the basic semantical analyses in PIPS deal with the effects of instructions on memory. First, the *proper* effects are computed for each sub-statement of a module: a read or write operation on an array element, the update of an index variable in a DO loop, etc. These effects can be displayed to the user *via* a prettyprinter like:

```
C     <must be read   >: J NC S1 S2 S3
C     <must be written>: T(J,1,NC+5)
         T(J,1,NC+5) = S1*S2/((S3-S1)*(S3-S2))
```

Then, since the abstract syntax tree is recursively defined in PIPS, the *cumulated* effects are recursively aggregated from all the proper effects of the sub-statements in a statement. For the loop 300 on Figure A.2 this leads to:

```
C     <may be read    >: J J2 JH K L NC S1 S2 S3 T(*,*,*)
C     <may be written >: JH S1 S2 S3 T(*,*,*)
C     <must be read   >: J1 JA
C     <must be written>: J
      DO 300 J = J1, JA
         ...
```

Since having information about effects on internal variables in a function is irrelevant to calling procedures, the *summary* effects of a procedure are constructed by removing these local effects from the cumulated effects of the procedure. Interprocedural propagation will be commented in Section A.3.4.

Transformers

To improve the accuracy of the effect information, to provide more information for dependence testing and to select better program transformations, statements are labeled with preconditions that express constraints on integer scalar variables. Since such variables are often used in array references and loop bounds, preconditions can often precisely define which subpart of an array is referenced by a memory reference expression, thus refining cumulated effects into Regions (Section A.2.1).

Mathematically speaking, PIPS preconditions are computed via transformers that are abstract commands mapping a store to a set of stores [105, 103]. They are relations between the values of integer variables in an initial store and those in a final store. In PIPS, the considered relations are polyhedra, represented by systems of linear equalities and inequalities, because they provide a sound and general mathematical framework for reasoning about array accesses which are mostly linear in practice. Below, two statement transformers are given by the prettyprinter.

```
C  T(J1) {J1==J1#init+1}
         J1 = J1+1
C  T(J2) {J2==2JA+1}
         J2 = 2*JA+1
```

Constant propagation, inductive variable detection, linear equality detection or general linear constraints computation can be performed in this workbench using different operators to deal with PIPS basic control structures. To deal with unstructured control graphs, some fancier operators requiring a fix point computation are needed, although they may slow down the analysis. To provide a better trade-off between time and precision to the user, options are available to tune the precondition construction algorithms.

Unlike most analysis in the field, we are dealing with abstract commands instead of abstract stores for two reasons. First, variables appearing in references cannot be aggregated to build cumulated and summary effects unless they denote the same values, that is they refer to the same store. The module's initial store is a natural candidate to be the unique reference store and, as a result, a relationship between this store and any statement's initial store is needed. This relationship is called a precondition [103] in PIPS and is constructed for a statement by aggregating all the transformers of the statements on the paths from the beginning of the module up to that statement. On the same example than the transformers we have:

```
C  P(J1,N) {N==1, 1<=J1, ...}
         J1 = J1+1
C  P(J1,N) {N==1, 2<=J1, ...}
         J2 = 2*JA+1
C  P(J1,J2,JA,N) {J2==2JA+1, N==1, 2<=J1, ...}
```

Second, dependence tests are performed between two statements, Although a relationship between each statement store would be more useful than two stores, each relative to one statement, this would not be realistic because too many predicates would have to be computed. Thus, a common reference to the same initial store seems to be a good trade-off between accuracy and complexity.

Dependence test

Since the dependence graph [178] is used within many parts of PIPS, such as the parallelizers, the validity check for some transformations, the use-def elimination phase, etc., a multi-precision test has been implemented to provide various degrees of precision and speed. First, a crude by fast algorithm is used and if no positive or negative conclusion is found, a more precise but more compute-intensive test is used, and so on [177].

PIPS dependence tests compute both dependence levels and dependence cones. Dependence cones can be retrieved to implement advanced tiling and scheduling methods.

Regions

Array Regions [63, 65] are used in PIPS to summarize accesses to array elements. READ and WRITE Regions represent the effects of statements and procedures on sets of array elements. They are used by the dependence test to disprove interprocedural dependences. For instance, in our example, the READ regions for the sole statement of function D are:

```
C       <T(φ1 , φ2 , φ3 )-R-EXACT-{ φ1 ==J,K<=φ2 <=K+1,L<=φ3 <=L+2}>

             D=SQRT((T(J,K,L  )-T(JP,KP,L  ))**2
     1      +        (T(J,K,L+1)-T(JP,KP,L+1))**2
     2      +        (T(J,K,L+2)-T(JP,KP,L+2))**2)
```

where the ϕ variables represent the three dimensions of A. To compute this Region, interprocedural preconditions must be used to discover the relations between the values of J and JP, and K and KP.

However, since READ and WRITE Regions do not represent array data flow, they are insufficient for advanced optimizations such as array privatization. IN and OUT regions have thus been introduced for that purpose: for any statement or procedure, IN regions contain its imported array elements, and OUT regions contain its exported array elements. The possible applications are numerous. Among others, IN and OUT regions are already used in PIPS to privatize array sections [16, 61], and we intend to use them for memory allocation when compiling signal processing specifications based on dynamic single assignments.

Another unique feature of PIPS array Regions lies in the fact that, although they are over-approximations of the element sets actually referenced, they are flagged as exact whenever possible. Beside a theoretical interest, there are specific applications such as the compilation of HPF in PIPS.

Array Data Flow Graph

This sophisticated data flow graph extends the basic DFG with some kind of array Regions information with linear predicates. Thus, it is able to finely track array elements accesses according to different parameters and loop indices in programs. However, this is only valid when the whole control can be represented in a linear framework [141, 140].

Complexities

When optimizing some code, or assessing whether to parallelize some program fragment or apply a transformation, it is quite useful to have some estimation of the program time complexity. A static complexity analysis phase has been added to PIPS to give polynomial complexity information about statements and modules [179] from preconditions and other semantical information.

A.2.2 Code generation phases

The result of program analyses and transformations is a new code. These phases generate (1) parallel code from data such as sequential code, semantical informations and the dependence graph, or (2) a message passing code for distributed memory architectures.

HPF Compiler

HPFC is a prototype HPF compiler implemented as a set of PIPS phases. From Fortran 77, HPF static (`align distribute processors template`), dynamic (`realign redistribute dynamic`) and parallelism (`independent new`) directives, it generates portable PVM-based SPMD codes which have run on various parallel architectures (SUN NOW [45], DEC alpha farm, TMC CM5, IBM SP2).

Implementing such a prototype within the PIPS framework has proven to be a fine advantage: First, the integration of the various phases (Figure A.3) within the PipsMake system allows to reuse without new developments all PIPS analyses, and to benefit from the results of these analyses for better code generation and optimizations; Second, the Linear C^3 library has provided the mathematical framework [10, 11] and its implementation to develop new optimizations techniques targeted at generating efficient communication codes. It has been applied to I/O communications [46] and general remappings [48].

Parallelization

The primary parallelization method used for shared-memory machines is based on Allen&Kennedy algorithm and on dependence levels. This algorithm was slightly modified to avoid loop distribution between statements using the same private variable(s). The resulting parallel code can be displayed in two different formats based on Fortran 77 and Fortran 90. Parallel loops are distinguished by the `DOALL` keyword.

Another parallelization algorithm was derived from Allen&Kennedy's to take into account the Cray specificities, the vector units and the micro-tasking library. Two levels of parallelism are selected at most. The inner parallelism is used to generate vector instructions and the outer one is used for task parallelism.

Phase	Function
`hpfc_filter`	preprocessing of the file
`hpfc_parser`	Fortran and HPF directive parser
`hpfc_init`	initialization of the compiler status
`hpfc_directives`	directive analysis
`hpfc_compile`	actual compilation of a module
`hpfc_close`	generation of global runtime parameters
`hpfc_install`	installation of the generated codes
`hpfc_make`	generation of executables
`hpfc_run`	execution of the program under PVM

Figure A.3: Hpfc phases in Pips

Code distribution

This parallelization method emulates a shared memory machine on a distributed architecture with programmable memory banks. It splits the program in two parts: a computational part and a memory part. More precisely, two programs are generated by using a method based on linear algebra: one SPMD program that does the computations in a block distributed control way and another SPMD program that manages the memory accesses.

The development run-time is implemented over PVM but the target is Inmos T9000 with C104 transputer-powered parallel machines with an hight speed interconnection network. This prototype was built as part of the European Puma project [13].

Parallelization with a polyhedral method

By using the Array Data Flow Graph, it is possible to track the movement of each value in a program where control flow can be statically represented in a linear way. Using this information, a schedule is generated to exploit the maximum parallelism and a placement is selected to reduce the number of communications according to their nature without discarding all the parallelism [141, 140].

A.2.3 Transformations

A transformation in Pips takes an object, such as the internal representation of the code, and generates a new version of this object. Transformations currently implemented in Pips are:

- loop distribution;
- scalar and array privatization based on Regions;
- atomizer to split statements in simpler ones such as `A=B op C`;
- loop unrolling;
- loop strip-mining;

- loop interchange using an unimodular transformation;
- loop normalization;
- partial evaluation;
- dead-code elimination based on preconditions;
- use-def elimination;
- control restructurer;
- reduction detection.

A.2.4 Pretty-printers

Pretty-printers are used to transform some PIPS internal representation into a human readable file that can be displayed by the user interface. There are pretty-printers to display the original parsed code, the sequential code, the parallel code in some parallel dialect, the dependence graph, the call graph, etc. Other specialized pretty-printers exist to add some informative decoration to the previous ones, such as displaying the code with regions, preconditions, complexities, etc. There are implemented by using hooks in the more classical pretty-printers.

A.3 PIPS general design

For a PIPS developer, PIPS can be viewed as in Figure A.4. Of course the user interfaces control PIPS behavior but, internally, all the compiling, analyzing and transformation operations are split into basic independent *phases*. They are scheduled by PipsMake, an *à la* `make` interprocedural mechanism that deals with *resources* through the PipsDBM data base manager. An input program is first split into modules that correspond to its subroutines and functions. PipsMake can then apply the transformation phases on these modules according to some predefined rules. The result is a streamlined programming interface with as few side effects as possible between phases. This allows a highly modular programming style with several developers at the same time. To add a phase to PIPS, only a C function (which may be a wrapper if the execution part of the phase is written in another language) is added (which may of course call other functions) that is called by PipsMake with a module name as argument.

A.3.1 Resources

All the data structures that are used, produced or transformed by a phase in PIPS are called resources. A phase can request some data structures in PipsDBM, use or modify them, create some other resources, and then store with PipsDBM the modified or created resources to be used later. Examples of resources are the input code, abstract syntax tree, control flow graph, use-def chains, output code, semantic information such as *effects*, *preconditions*, *regions*, etc. These resources are described with the NEWGEN description language (Section A.4.1) that allows transparent access to persistent data in PipsDBM from different languages.

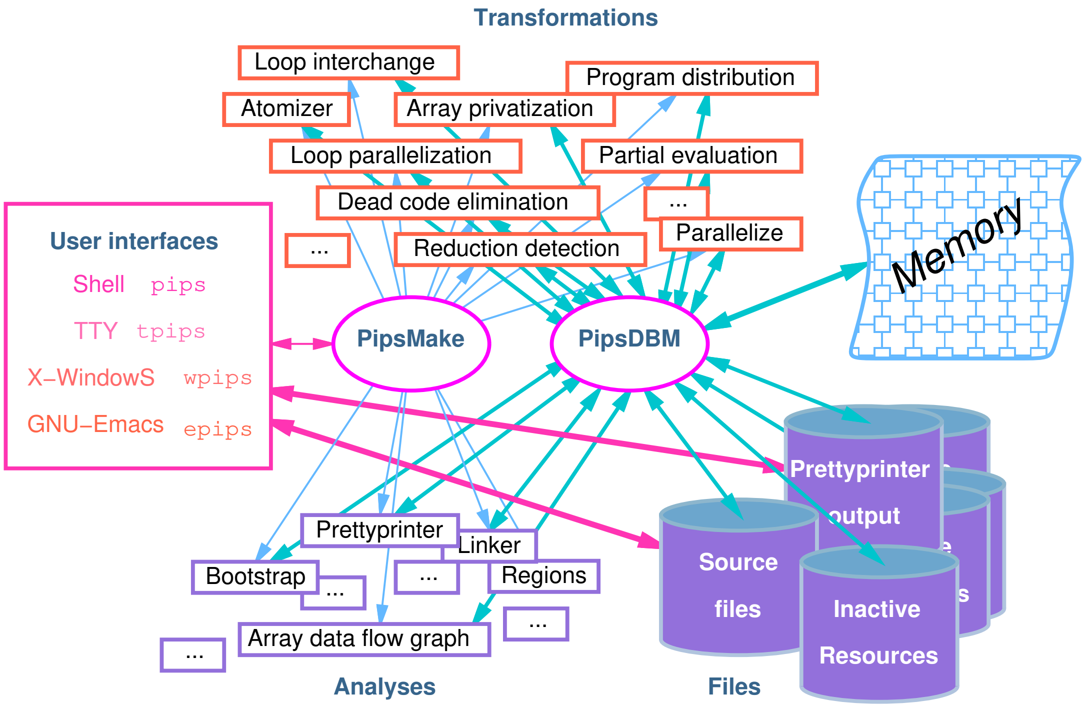


Figure A.4: Overview of the PIPS architecture.

A.3.2 Workspace

Since all the resources about a user code may be stored on disk (Section A.3.5), resources are located in a workspace directory. This workspace is created from the source file given by the user and also contains a log file and a PipsMake data structure (Section A.3.4 that encodes a coherent description of its content).

A.3.3 Phases

Each phase in PIPS use the same interface and simply is a C function taking a module name as argument and returning a boolean describing its completion status (success or failure). Figure A.5 shows the example of the dead code elimination phase.

Since PIPS core is written in C, a wrapper function is needed to program PIPS in other programming languages. Currently, the only phase written in Common-Lisp is the reduction detection phase: it calls a Common-Lisp process to perform the job. Communications are handle through PipsDBM files.

A.3.4 PipsMake consistency manager

From a theoretical point of view, the object types and functions available in PIPS define an heterogeneous algebra with constructors (*e.g.* parsers), extractors (*e.g.* prettyprinters) and operators (*e.g.* loop unrolling). Very few combinations of functions make sense, but many functions and object types are available. This abundance is confusing for casual and even experienced users and it was deemed necessary to assist them by providing default computation rules and automatic consistency management.

```
bool suppress_dead_code(string module_name)

  /* get the resources */
  statement module_stmt = (statement)
      db_get_memory_resource(DBR_CODE, module_name, TRUE);
  set_proper_effects_map((statement_mapping)
      db_get_memory_resource(DBR_PROPER_EFFECTS, module_name, TRUE));
  set_precondition_map((statement_mapping)
      db_get_memory_resource(DBR_PRECONDITIONS, module_name, TRUE));

  set_current_module_statement(module_stmt);
  set_current_module_entity(local_name_to_top_level_entity(module_name));

  debug_on("DEAD_CODE_DEBUG_LEVEL");
  /* really do the job here: */
  suppress_dead_code_statement(module_stmt);
  debug_off();

  /* returns the updated code to Pips DBM */
  DB_PUT_MEMORY_RESOURCE(DBR_CODE, module_name, module_stmt);
  reset_current_module_statement();
  reset_current_module_entity();
  reset_proper_effects_map();
  reset_precondition_map();

  return TRUE;
```

Figure A.5: Excerpt of the function of the dead code elimination phase.

```
proper_effects    > MODULE.proper_effects
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.summary_effects

cumulated_effects > MODULE.cumulated_effects
        < PROGRAM.entities
        < MODULE.code MODULE.proper_effects

summary_effects   > MODULE.summary_effects
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects

hpfc_close          > PROGRAM.hpfc_commons
    ! SELECT.hpfc_parser
    ! SELECT.must_regions
    ! ALL.hpfc_static_directives
    ! ALL.hpfc_dynamic_directives
    < PROGRAM.entities
    < PROGRAM.hpfc_status
    < MAIN.hpfc_host

use_def_elimination > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.chains
```

Figure A.6: Example of `pipsmake-rc` rules.

The PipsMake library – not so far from the Unix make utility – is intended for interprocedural use. The objects it manages are resources stored in memory or/and on disk. The phases are described in a `pipsmake-rc` file by generic rules that use and produce resources. Some examples are shown on Figure A.6. The ordering of the computation is demand-driven, dynamically and automatically deduced from the `pipsmake-rc` specification file. Since it is a quite original part of PIPS, it is interesting to describe a little more the features of PipsMake.

Rule format

For each rule, that is in fact the textual name of a phase C function, it is possible to add a list of constraints composed with a qualifier, an owner name and a resource name. Each constraint has the following syntax:

$$< \textit{qualifyer} > < \textit{owner-name} > . < \textit{resource-name} >$$

The qualifier describing the resource can be for each resource:

`<:` needed;

`>:` generated;

`#:` destroyed by the rule;

`=:` asserts that the resource remains valid even if other dependence rules would suggest to invalidate the resource;

`!:` asks for applying an other rule before executing the rule. Thus, this constraint has a special syntax since it composed with a rule instead of a resource name. In Figure A.6 for instance `hpfc_close` needs an alternate parser for HPF, the must Regions and the directives of all the modules of the program.

This qualifier is thus more complex than an Unix `make` but allows finer tuning to avoid useless computation (for example with the =) and be able to expand the rule dependence behavior with !.

An owner name specifies the owner of the needed resource. For example for the code resource, only the code of this module, or the code of all the modules, etc:

MODULE: the current module itself;

MAIN: the main module if any (that is the Fortran `PROGRAM`);

ALL: all the modules

CALLEES: the set of the called modules by the current one; it allows to describe bottom-up analyses on the callgraph.

CALLERS: the set of the calling modules of the current one; it expresses top-down analyses.

PROGRAM: a resource that is attached to the whole program, such as the entities of a program. This concept is orthogonal to the module concept;

SELECT: select a rule by default.

A rule (that is a phase for a PIPS user) is conventionally called a transformation when a resource name appears in the rule with both $<$ and $>$ qualifiers. For example dead code elimination which transforms the code both needs the code and writes it.

It is possible to chose between several rules to make a resource. By default the first one found in `pipsmake-rc` is selected but the programmer can choose to activate an other default rule through the API (see Section A.3.4 or with the **SELECT** owner). This feature is used for example in the user interfaces to select the kind of decorations in the prettyprinters. For instance the sequential code can be built by several phases, one for the plain code, another one for the plain code with Regions, etc.

Up to date resources & API

PipsMake is responsible for dynamically activating rules to build up to date resources and iterates through the transitive closure of the resource dependences (something Unix's `make` is unfortunately unable to do). A small programming interface is available to drive PipsMake from user interfaces (Section A.5) or other phases:

- `make(string resource_name, string owner_name)` builds the resources `resource_name` for the set of modules or the program described by `owner_name`;
- `apply(string phase_name, string owner_name)` applies the phase `phase_name` to the set of modules or the program described by `owner_name`;
- `activate(string phase_name)` activates this rule as the new default one when more than one rule could be used to build some resources. Since with this new default rule these generated resources are no longer up to date, the resources that can generated by the new default rule and that existed before are recursively deleted.

Interprocedurality

Interprocedurality is dealt with by the PipsMake **CALLEES** and **CALLERS** owner features: A bottom-up algorithm on the interprocedural call graph such as effects or interprocedural transformers computation will have some rules using **CALLEES**, while a top-down algorithm such as interprocedural preconditions will have some rules using **CALLERS** so as to properly order the phase execution and propagate resources interprocedurally. The correct ordering of the computation is deduced from the `pipsmake-rc` file.

For instance, effects computation relies on the three dependence rules shown on Figure A.6: First the proper effects are computed from the code and the summary effects of the **CALLEES** to get the summary effects of the modules called in the current module. Second these proper effects, that only relate to each simple statement, are transformed into the summary effects that take into account the hierarchical structure of the abstract syntax tree. Third, the summary effects are computed to describe the effects that are only non local to the current module and that may later be used to compute the effects of a caller.

Persistence and interruption

Since PipsMake itself is implemented with NewGen, the PipsMake data structure describing the state of Pips is saved on disk when Pips exits. Thus, a Pips operation can continue from a previous saved state.

Furthermore, at a phase boundary, all Pips data structures are managed with PipsDBM. Thus the Pips process can be stopped cleanly at a phase boundary when the control is under PipsMake. Of course the `make()` operation asked to PipsMake fails, but next time the user asks for a `make()`, PipsMake will only launch the phases required to build the lacking resources. This mechanism is heavily used in the user interfaces to insure interactivity.

A.3.5 PipsDBM resource manager

All the data structures used by the phases to seamlessly communicate with each other are under the responsibility of the PipsDBM resource manager.

PipsDBM can decide to store a resource in memory or on disk. Typically, resources that have been modified are written on disk when Pips exits but the programmer can ask to immediately create a file resource to be viewed by a user interface. The programmer can also declare some resources as unloadable or as memory-only.

PipsDBM also manages a logical date for each resource for the dependence mechanism of PipsMake. Logical dates have been introduced to overcome the lack of Unix time stamp precision (1 second under SunOS4) and the difficulty to have very well synchronized client-server operations in an NFS environment. Of course, the Unix date is also used in order to detect resource files modified by the user (typically an edit through a user interface) where the 1 second precision is good enough.

The programming interface is reduced to

```
string db_get_resource(string ressource_name, string module_name, bool pure)
void db_put_resource(string ressource_name, string module_name, void* value)
```

`pure` is used to specify if the programmer wants the genuine resource or only a copy (for example to modify it and give back another resource).

A.3.6 PIPS properties

Global variables to modify or finely tune PIPS behavior are quite useful but unfortunately are often dangerous. Thus these variables are wrapped in PIPS as properties. A property can be a boolean, an integer or a string. Using the fact that properties are centralized, they are initialized from a default property file and possibly from a local `property.rc`. Since the behavior is modified by the properties, they are stored in the workspace in order to restart a PIPS session later in the same way even if the global property file has been modified.

A.4 Programming environment

The PIPS workbench is implemented to form a programming environment. Typically, each phase has its own directory and its own library so that it is easy to add a new phase or modify a phase without too much trouble-shooting. Furthermore, almost all the directory structure is duplicated to have both a production and a development version at the same time.

A.4.1 NewGen: data structure and method generation

In oder to ease data structure portability between different languages and also to allow data persistence, a tool and a language to declare data structures has been developed: NEWGEN [109]. Once a data description of the *domains* (that are the data structures that can be defined in NEWGEN) is written, NEWGEN constructs several methods to create initialized or not data values, access or modify certain parts of constructed values, write or read data from files, or recursively deallocate data values.

A domain can be a sum (`+` like a C-union) or a product (`x` like a C-struct) of other domains, list (`*`), set (`{}`), array (`[]`) or a map (`->`) of domains. Some domains can be declared as "persistent" to avoid being deleted and the "tabulated" attribute associates a unique identifier to each domain element, allowing unique naming through files and different program runs. The mapping of domains is used to attach semantic information to objects: `alignmap = entity->align`.

Of course, some external types can be imported (such as for using the linear C^3 library in PIPS). The user must provide a set of functions for these domains to write, read, free and copy methods. The NEWGEN environment includes also various low-level classes such as lists, hash-tables, sets, stacks etc. with their various methods (iterators, etc.) that ease the writing of new phases.

```
expression = reference + range + call ;
reference = variable:entity x indices:expression* ;
range = lower:expression x upper:expression x increment:expression ;
call = function:entity x arguments:expression* ;
statement = label:entity x number:int x ordering:int x
            comments:string x instruction ;
instruction = block:statement* + test + loop + goto:statement +
              call + unstructured ;
test = condition:expression x true:statement x false:statement ;
```

Figure A.7: Excerpt of the PIPS abstract syntax tree.

An excerpt of the internal representation for Fortran in PIPS is shown on Figure A.7. For example, a `call` is a `function entity` with a list of `arguments expressions`. An `expression` can be a `reference`, a `range` or a `call`, and so on. In fact an assignment is represented itself by a call to a pseudo-intrinsic function "`=`" with 2 arguments, the left hand side and the right hand side argument. And so on for all the intrinsics of the Fortran language. All the syntactic sugar of Fortran has been removed from the internal representation and in this way it is expected not to stick on Fortran idiosyncrasies. The representation is quite simple: there are only 5 cases of statements and 3 cases of expressions. This simplicity also benefits all the algorithms present in PIPS since there are very few different cases to test.

An important feature of NEWGEN is the availability of a general multi domain iterator function (`gen_multi_recurse`). From any NEWGEN domain instance (e.g., `statement`, `instruction`, `expression`) one can start an iterator by providing the list of domains to be visited, and for each domain the function to be applied top-down, which returns a boolean telling whether to go on with the recursion or not, and a function to be applied bottom-up. The iterator is optimized so as to visit only the needed nodes, and not to follow paths on which no nodes of the required types might be encountered.

In the Figure A.8 example, some entities are replaced with others. Entities may appear as variable references, as loop indices, as called functions and in program codes. The recursion starts from `statement stat`: on each `loop`, `reference`, `call` or `code` encountered top-down, the function `gen_true` is applied. This function does nothing but returning true, hence the recursion won't be stopped before having processed all the required domains that can be reached from `stat`. While going back bottom-up, the `replace` functions are applied and perform the required substitution. From the user point of view, only replace functions are to be written.

An example of how NEWGEN defined types are used is shown on Figure A.9. It is

```
static void replace(entity * e)
{ /* replace entity (*e) if required... */}

static void replace_call(call c)
{ replace(&call_function(c));}

static void replace_loop(loop l)
{ replace(&loop_index(l));}

static void replace_reference(reference r)
{ replace(&reference_variable(r)); }

static replace_code(code c) // code_declarations() is a list of entity
{ MAPL(ce, replace(&ENTITY(CAR(ce))), code_declarations(c));}

// args: (obj, [domain, filter, rewrite,]* NULL);
   gen_multi_recurse(stat,
        reference_domain, gen_true, replace_reference,
        loop_domain,      gen_true, replace_loop,
        call_domain,      gen_true, replace_call,
        code_domain,      gen_true, replace_code,
                       NULL);
```

Figure A.8: `gen_multi_recurse` example

a follow-up of Figure A.5. `suppress_dead_code_statement()` begins with a NewGen generic iterator to walk down the module statements applying `dead_statement_filter` on them and walk up applying `dead_statement_rewrite` on all the `statement` domains encountered. In `dead_statement_filter()` one can see that `statement_instruction()` is the accessor to the `instruction` of a `statement` generated by NewGen from the internal representation description. `instruction_tag()` describes the content of an `instruction` (which is a "union"). For loops the result is `is_instruction_loop`. NewGen also generates predicates such as `instruction_loop_p()` to test directly the content of an `instruction`.

NewGen can output data structure declarations and methods for C and Common-Lisp languages since they are the languages used to build Pips. The translation environment is described in Section A.4.4. Note that at the beginning of the Pips project, C++ was not considered stable enough so that all the object methods have been written in C, wrapped and hidden in macros and functions.

A.4.2 Linear C^3 library

An important tool in Pips is the Linear C^3 library that handles vectors, matrices, linear constraints and other structures based on these such as polyhedrons. The algorithms used are designed for integer and/or rational coefficients. This library is extensively used for analyses such as the dependence test, precondition and region computation, and for transformations, such as tiling, and code generation, such as send and receive code in HPF compilation or code control-distribution in wp65. The linear C^3 library is a joint project with irisa and prism laboratories, partially funded

```
static bool dead_statement_filter(statement s)

   instruction i = statement_instruction(s);
   bool ret = TRUE;

   if (!statement_weakly_feasible_p(s))       // empty precondition
      ret = remove_dead_statement(s, i);
   else

      switch (instruction_tag(i))
      case is_instruction_loop:

         loop l = instruction_loop(i);
         if (dead_loop_p(l))                    // DO I=1,0
            ret = remove_dead_loop(s, i, l);
         else if (loop_executed_once_p(s, l))  // DO I=N,N
            remove_loop_statement(s, i, l);
            suppress_dead_code_statement(body);
            ret = FALSE;

         break;

      case is_instruction_test:
         ret = dead_deal_with_test(s, instruction_test(i));
         break;
      case is_instruction_unstructured:
         dead_recurse_unstructured(instruction_unstructured(i));
         ret = FALSE;
         break;



   if (!ret) // Try to rewrite the code underneath
      dead_statement_rewrite(s);

   return ret;


void suppress_dead_code_statement(statement module_stmt)

    gen_recurse(module_stmt,            // recursion from...
                statement_domain,       // on statements
                dead_statement_filter,  // entry function (top-down)
                dead_statement_rewrite);// exit function (bottom-up)
```

Figure A.9: Excerpt of the function of the dead code elimination phase.

by CNRS. IRISA contributed an implementation of CHERNIKOVA algorithm and PRISM a C implementation of PIP (Parametric Integer Programming).

A.4.3 Debugging support

Developing and maintaining such a project requires a lot of debugging support from the programming environment at many levels. At the higher level, it is nice to be able to replay a PIPS session that failed since a bug can appear after quite a long time. For this purpose there is a tool that transforms a WPIPS log file into a TPIPS command file to be re-executed later. Every night, a validation suite is run on hundreds of test cases to do some basic non-regression testing. It is not enough to do intensive debugging but is generally sufficient to ensure that a modification is acceptable.

At the programmer level there are some macros to enable or disable the debug mode in some parts of PIPS according to some environment variables that indicate the debug level. Often, there is such a variable per phase at least and the debug modes can be nested to avoid debugging all the functions used in a phase to debug only this phase for example. The debug levels can also be modified according to the property of the same names.

At the NEWGEN level, the recursive type coherence of a data structure can be verified with `gen_consistent_p()` or even at each assignment or at creation time by setting the variable `gen_debug` to an adequate value.

At last, some Emacs macros and key bindings have been defined for the `gdb` Emacs-mode to display the type of a NEWGEN object, the type of a PIPS entity, an expression of the PIPS abstract syntax tree, a PIPS statement, etc. to alleviate the development effort.

A.4.4 Documentation and configuration files

Such a big project (over 200,000 lines of code for the compilers, mathematical libraries and tools) with developers on different sites) needs to manage a strong documentation support of course, but also as much automatically generated and non-redundant as possible to keep the incoherence level low.

The main idea is to write all the configuration files as technical reports and automatically extract the configuration files from them. Figure A.10 is a simplified synopsis of the documentation flow. C headers are generated from `.newgen` files that come from some document describing the usage of the data structure, such as `ri.tex` for the internal representation . The menu files for the user interfaces are generated from `pipsmake-rc` and the property file. All the information needed to feed PipsMake and PipsDBM comes from the `pipsmake-rc` file. At last, LaTeX2HTML is used to generate most of `http://www.cri.ensmp.fr/pips` from all the LaTeX files.

A.5 User interfaces

Typically, a user interface will use the `make()` command to build a resource to display, use `apply()` to apply a transformation on the code and select default rules, that are options, with `activate()` (Section A.3.4). There are as many as four interfaces to PIPS:

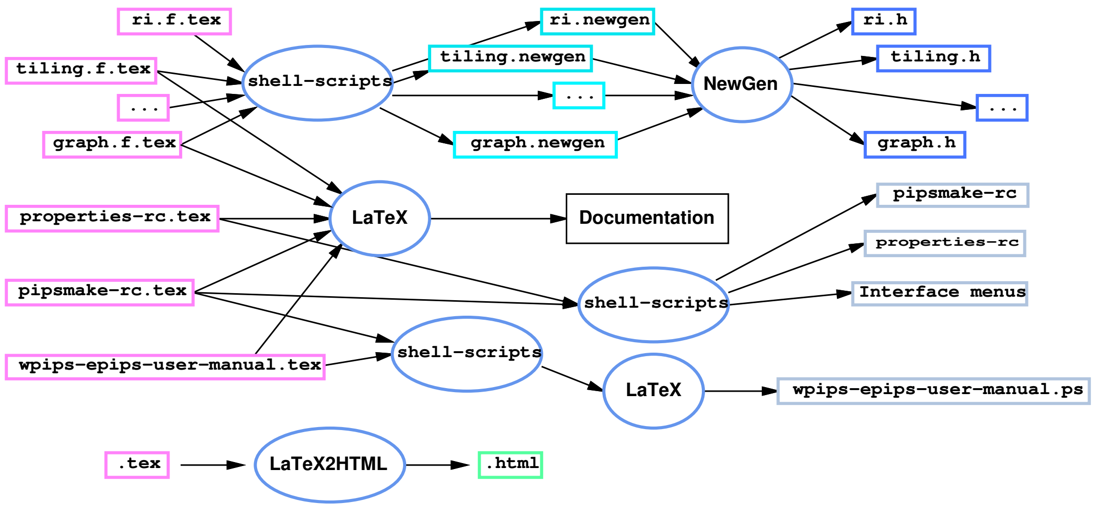


Figure A.10: Documentation and configuration system.

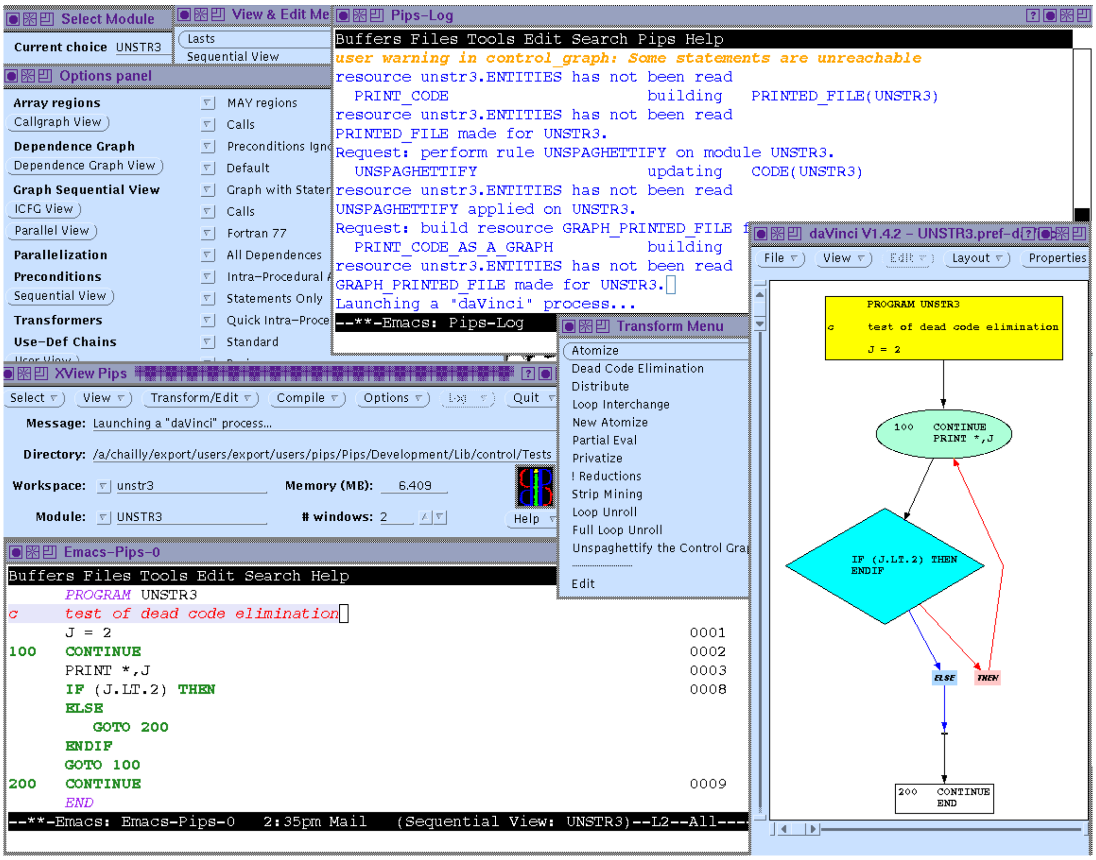


Figure A.11: Emacs-PIPS interface snapshot.

- `pips` is a shell interface, can do only one operation on a workspace and relies a lot on PipsDBM persistence since the `pips` process exits after each operation;

- `tpips` is a line interface with automatic completion. It's much faster to use for experimentation and debugging than a full graphical interface;

- `wpips` is an XView based easy interface interface;

- `epips` is an extended version of the previous one for GNU-Emacs users. The interface inherits the Fortran mode and color highlight features. Further more graphs are displayed with `xtree` and `daVinci`.

All these interfaces are configured according to the documentation A.4.4: when a programmer adds a transformation, it automatically appears in all the interfaces after reconfiguration of the `pipsmake-rc` file.

When dealing with big programs, it may be necessary to interrupt PIPS if a user chooses too costly an option or applies an heavy analysis by mistake. To solve this issue, the user interfaces can interrupt PipsMake at the phase interface level, without compromising the coherence through PipsMake and PipsDBM (Section A.3.4).

A.6 Conclusion

A.6.1 Related Work

Many compiler prototypes have been implemented to test analysis, transformation or code generation techniques.

Parafrase 2 (UIUC): This compiler is a source to source code restructurer for Fortran and C performing analysis, transformation and code generation. It allows automatic vectorization and parallelization.

Polaris (UIUC): This compiler is also a source to source restructurer developed for Fortran 77. It performs interprocedural analysis for automatic parallelization of programs, targeting shared-memory architectures.

SUIF (Stanford): The *Stanford University Intermediate Form* is a low level representation plus some high level additions that keep track of the program structure such as loops, conditions and array references. The input language is C and Fortran through the `f2c` translator. The compiler outputs machine code for different architectures. The compiler initial design and the current distribution of SUIF does not incorporate interprocedural analysis. Such an addition is being worked on [95, 93]. Another problem is that the consistency of analyses is not automatically inforced (for instance invalidation after code transformations).

The originality of PIPS with respect to these other prototypes is the conjunction of:

- Interprocedurality and coherency automatic management in a demand driven system combining database (PipsDBM) and dependence (PipsMake) functions.

- Advanced interprocedural semantic and data-flow analyses already implemented, and their mathematical foundation upon linear algebra (Linear C^3 library).
- Software foundation upon an high level description system (NewGen) which automatically generates functions (access, constructor, destructor) and can iterate on arbitrary structures. An additional by-product is that the code look and feel is homogeneous over the 150,000 lines of the compiler.
- Multi-target:
  - Automatic parallelization for shared-memory
  - Compilation for distributed-memory models (Hpfc, wp65)
  - Automatic data mapping
- The availability of different interfaces (shell, terminal Tpips, graphical Wpips).

Compiler development is a hard task. Here are examples of many research prototypes. They would benefit a lot from already available infrastructures, and of the synergy of other analyses and transformations, especially interprocedural ones.

- (small) program analyzers and parallelizers: Bouclette (LIP, ENS-Lyon), LooPo (U. Passau), Nascent (OGI), PAF (prism, UVSQ), Petit (U. of Maryland), Tiny.
- Parallel Fortran (HPF spirit) compilers: Adaptor (GMD), Annai (CSCS-NEC), D system (Rice U.), Fx (Carnegie Melon U.), Paradigme (UIUC, based on Parafrase 2), SHPF (Southhampton), VFCS (Vienna)
- Parallel C compilers: Pandore (Irisa), EPPP (CRI Montréal)

The html version of this paper should be used to obtain many links to related work and the Pips project.

A.6.2 Remarks

Eight years after its inception, Pips as a workbench is still alive and well. Pips provides a robust infrastructure for new experiments in compilation, program analysis, optimization, transformation and parallelization. The Pips developer environment is described in [106], but it also is possible to develop new phases on top of but *outside* of Pips since all (in fact, most...) Pips data structures can be reloaded using NewGen primitives.

Pips can also be used as a reverse engineering tool. Region analyses provide useful summaries of procedure effects, while precondition-based partial evaluation and dead code elimination reduce code size.

Pips may be less robust than other publicly available source-to-source compilers but the stress is put on the ability to quickly add new phases such as program transformations, semantic analyses, parsers, parallelizers, etc. or new user interfaces without spending time developing a cumbersome infrastructure.

All the on-line reports of the cri can be found at http://www.cri.ensmp.fr/rapports.html.

The Pips software and documentation is available on `http://www.cri.ensmp.fr/pips`.

A.6.3 Acknowledgment

We would like to thank many people that have contributed to the PIPS project: Bruno BARON, Denis BARTHOU, Pierre BERTHOMIER, Arnauld LESERVOT, Guillaume OGET, Alexis PLATONOFF, Rémi TRIOLET, Yi-Qing YANG, Lei ZHOU.

Appendix B

Over- and Under-Representations

This appendix presents the generalization of GALOIS connections proposed by BOURDONCLE in his thesis [33], in the case of over- and under-approximations. An example then shows the limitations of this approach for under-approximate array region analyses.

Definition B.1 (Over-representation)
Let $(\boldsymbol{A}, \perp, \sqsubseteq)$ be a cpo, $\boldsymbol{A}'$ a set, and $\gamma : \boldsymbol{A}' \longrightarrow \boldsymbol{A}$ a *meaning* function. $(\boldsymbol{A}', \preceq, \gamma, \overline{\nabla} = (\overline{\nabla}_i)_{i\in\mathbb{N}})$ is an *over-representation* of $\boldsymbol{A}$ if:

1. $(\boldsymbol{A}', \perp, \preceq)$ is a partial order.

2. γ, the *meaning* or *concretization* function, is monotone.

3. Each element $a \in \boldsymbol{A}$ can be safely over-approximated by $\alpha(a) \in \boldsymbol{A}'$, that is to say $a \sqsubseteq \gamma(\alpha(a))$.

4. $\overline{\nabla}$ is a sequence of *widening* operators $\overline{\nabla}_i : \boldsymbol{A}' \times \boldsymbol{A}' \longrightarrow \boldsymbol{A}'$, such that:

   - $\forall\, \overline{\nabla}_i \in \overline{\nabla},\ \forall\, a, a' \in \boldsymbol{A}',\ \begin{cases} a & \preceq & a\,\overline{\nabla}_i\,a' \\ \gamma(a') & \sqsubseteq & \gamma(a\,\overline{\nabla}_i\,a') \end{cases}$
   - $\forall\, (a_i)_{i\in\mathbb{N}} \in \boldsymbol{A}'$, the sequence $(a'_i)_{i\in\mathbb{N}}$ defined by:

$$\begin{cases} a'_0 & = & a_0 \\ a'_{i+1} & = & a'_i\,\overline{\nabla}_i\,a_{i+1} \end{cases}$$

   has an upper bound.

Definition B.2 (Under-representation)
Let $(\boldsymbol{A}, \perp, \sqsubseteq)$ be a cpo, $\boldsymbol{A}''$ a set, and $\gamma : \boldsymbol{A}'' \longrightarrow \boldsymbol{A}$ a *meaning* function. $(\boldsymbol{A}'', \preceq, \gamma, \underline{\Delta} = (\underline{\Delta}_i)_{i\in\mathbb{N}})$ is an *under-representation* of $\boldsymbol{A}$ if:

1. $(\boldsymbol{A}'', \perp, \preceq)$ is a partial order.

2. γ, the *meaning* or *concretization* function, is monotone.

3. Each element $a \in \boldsymbol{A}$ can be safely under-approximated by $\alpha(a) \in \boldsymbol{A}''$, that is to say $\gamma(\alpha(a)) \sqsubseteq a$.

4. $\underline{\Delta}$ is a sequence of *narrowing* operators $\underline{\Delta}_i : \boldsymbol{A}'' \times \boldsymbol{A}'' \longrightarrow \boldsymbol{A}''$, such that:

   - $\forall\, \underline{\Delta}_i \in \underline{\Delta},\ \forall\, a, a' \in \boldsymbol{A}'',\ \left\{\begin{array}{ccc} a \,\underline{\Delta}_i\, a' & \preceq & a \\ \gamma(a \,\underline{\Delta}_i\, a') & \sqsubseteq & \gamma(a') \end{array}\right.$
   - $\forall\, (a_i)_{i\in\mathbb{N}} \in \boldsymbol{A}''$, the sequence $(a'_i)_{i\in\mathbb{N}}$ defined by:

   $$\left\{\begin{array}{ccl} a'_0 & = & a_0 \\ a'_{i+1} & = & a'_i \,\underline{\Delta}_i\, a_{i+1} \end{array}\right.$$

   has a lower bound.

Bourdoncle then shows how to define safe approximations of least fixed points from the properties of representations. These solutions are presented in the next two theorems.

Theorem B.1 (Over-approximation of least fixed points)
If $\Phi : \boldsymbol{A} \longrightarrow \boldsymbol{A}$ is a continuous recursive function, and if $\overline{\Phi} : \boldsymbol{A}' \longrightarrow \boldsymbol{A}'$ is a safe over-approximation of Φ ($\Phi \circ \gamma \sqsubseteq \gamma \circ \underline{\Phi}$), then the sequence

$$\left\{\begin{array}{ccl} a_0 & = & \perp \\ a_{i+1} & = & a_i \,\overline{\nabla}_i\, \overline{\Phi}(a_i) \end{array}\right.$$

is an increasing sequence whose limit is a safe over-approximation of the least fixed point of Φ.

Theorem B.2 (Under-approximation of least fixed points)
If $\Phi : \boldsymbol{A} \longrightarrow \boldsymbol{A}$ is a continuous recursive function, and if $\underline{\Phi} : \boldsymbol{A}'' \longrightarrow \boldsymbol{A}''$ is a safe under-approximation of Φ ($\gamma \circ \underline{\Phi} \sqsubseteq \Phi \circ \gamma$), then the sequence

$$\left\{\begin{array}{ccl} a_0 & = & \perp \\ a_{i+1} & = & a_i \,\underline{\Delta}_i\, \underline{\Phi}(a_i) \end{array}\right.$$

is a decreasing sequence whose limit is a safe under-approximation of the least fixed point of Φ.

This method does not give the *best solution* but *one* possible solution. The following example shows its limitations. Our purpose is to show that the solution built using $\underline{\Phi}$ as the under-approximation of the least fixed point of Φ can lead to disappointing approximations, whereas the exact solution is known to be computable.

Example We consider array region analysis, and approximate array element sets by convex polyhedra. With the notations of Definition B.2, we have:

$$(\boldsymbol{A}, \perp, \sqsubseteq) = (\boldsymbol{\Sigma} \longrightarrow \wp(\mathbb{Z}^n), \lambda x\sigma.\varnothing, \subseteq)$$

and

$$(\boldsymbol{A}'', \perp, \preceq) = (\boldsymbol{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^n), \lambda x\sigma.\varnothing, \subseteq)$$

γ is trivially defined by $\gamma = \lambda x.x$. And $\underline{\Delta}$ by:

$$\forall\, a, a' \in \boldsymbol{A}'', \quad \left\{ \begin{array}{lcl} a \,\underline{\Delta}_0\, a' & = & a' \\ a \,\underline{\Delta}_i\, a' & = & a',\ \forall\, i > 0 \end{array} \right.$$

Thus the under-approximation of the least fixed point of any continuous recursive function is reached in exactly one iteration and is equal to $\underline{\Phi}(\varnothing)$.

For `do while` loops the semantics of write regions is defined by:

$$\begin{aligned} \mathcal{R}[\![\texttt{do while(}C\texttt{)}\ S]\!] &= (\mathcal{R}[\![\texttt{do while(}C\texttt{)}\ S]\!] \circ \mathcal{T}[\![S]\!] \cup \mathcal{R}[\![S]\!]) \circ \mathcal{E}_c[\![C]\!] \\ &= \Phi(\mathcal{R}[\![\texttt{do while(}C\texttt{)}\ S]\!]) \end{aligned}$$

which must be interpreted as: If the evaluation of the condition $\mathcal{E}_c[\![C]\!]$ is true, then the element written by S ($\mathcal{R}[\![S]\!]$) belong to the result, as well as the elements written by the next iteration (including the evaluation of the condition), but in the memory store modified by S, $\mathcal{T}[\![S]\!]$ (see Section 6.3.5 for more details).

Φ can be under-approximated by:

$$\underline{\Phi} = \lambda f \sigma.(f \circ \mathcal{T}[\![S]\!] \,\underline{\cup}\, \underline{\mathcal{R}}[\![S]\!]) \circ \mathcal{E}_c[\![C]\!]\sigma$$

where $\underline{\cup}$ is an under-approximation of $\cup$ defined by:

$$x \,\underline{\cup}\, y = \textbf{if } x \cup y \in \boldsymbol{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^n) \textbf{ then } x \cup y \textbf{ else } \lambda\sigma.\varnothing$$

For the sake of simplicity, we assume that $\mathcal{E}_c$, $\mathcal{T}$, and $\circ$ do not need to be approximated. Given the previous definition of $\underline{\Delta}$, an under-approximation of $\text{lfp}(\Phi)$ is:

$$\underline{\Phi}(\lambda\sigma.\varnothing) = \underline{\mathcal{R}}[\![S]\!] \circ \mathcal{E}_c[\![C]\!]$$

This is the under-approximation of the region corresponding to the first iteration of the loop, after the evaluation of the condition, which is always true if the loop has at least one iteration.

Now let us consider the loop nest of Figure 4.3. In the innermost loop body, the sole element `A(I)` is referenced. It can be exactly represented, and thus under-approximated, by the convex polyhedron $\{\phi_1 = \texttt{I}\}$. The under-approximation obtained using $\underline{\Phi}$ for the innermost loop is the set of elements read during the first iteration, and is thus $\{\phi_1 = \texttt{I}\}$. It exactly describes the set of array elements read by the five iterations of the `J` loop. Repeating the process for the outermost loop, we then obtain $\{\phi_1 = 1\}$, which is far from the set of array elements actually referenced by the loop nest: $\{1 \leq \phi_1 \leq 5\}$.

Appendix C

Notations

C.1 Sets and Domains

Throughout this thesis, sets names are displayed in capital bold letters, such as $\boldsymbol{A}$. The following sets are used:

$\mathbb{Z}$	relative integers
$\mathbb{Q}$	rationals
$\boldsymbol{B}$	booleans
$\boldsymbol{C}$	constants
$\boldsymbol{N}$	variable, procedure, ... names
$\boldsymbol{R}$	references
$\boldsymbol{V}$	values (8-bit integer, ...)
$\boldsymbol{S}$	statements
$\boldsymbol{O}_1$	unary operators
$\boldsymbol{O}_2$	binary operators
$\boldsymbol{E}$	expressions
$\boldsymbol{E}_B$	boolean expressions ($\boldsymbol{E}_B \subseteq \boldsymbol{E}$)
$\boldsymbol{\Sigma}$	memory stores

Table C.1: Semantic domains

C.2 Semantic Functions

Semantic functions are denoted by calligraphic capital letters, such as $\mathcal{A}$, and the corresponding over- and under-approximations by $\overline{\mathcal{A}}$ and $\underline{\mathcal{A}}$. Analyses are defined by their denotational semantics, using the usual notations:

$$\begin{aligned}\mathcal{A} : \boldsymbol{L} &\longrightarrow \boldsymbol{B} \\ l &\longrightarrow \mathcal{A}[\![l]\!]\end{aligned}$$

The semantic functions defined throughout this thesis are presented in Table C.2.

$\mathcal{O}_1$	$: \boldsymbol{O}_1 \longrightarrow (\boldsymbol{V} \longrightarrow \boldsymbol{V})$	unary operators
$\mathcal{O}_2$	$: \boldsymbol{O}_2 \longrightarrow (\boldsymbol{V} \times \boldsymbol{V} \longrightarrow \boldsymbol{V})$	binary operators
$\mathcal{E}$	$: \boldsymbol{E} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \boldsymbol{V})$	expression evaluation
$\mathcal{T}$	$: \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \boldsymbol{\Sigma})$	transformers
$\mathcal{P}$	$: \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \boldsymbol{\Sigma})$	preconditions
$\mathcal{C}$	$: \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \boldsymbol{\Sigma})$	continuation conditions
$\mathcal{R}_r$	$: \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \wp(\mathbb{Z}^{d_i}))$	array regions of read references
$\mathcal{R}_w$	$: \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \wp(\mathbb{Z}^{d_i}))$	array regions of written references
$\mathcal{R}_i$	$: \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \wp(\mathbb{Z}^{d_i}))$	IN array regions
$\mathcal{R}_o$	$: \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \wp(\mathbb{Z}^{d_i}))$	OUT array regions

Table C.2: Semantic functions

We will also use the characteristic function of the restriction of $\mathcal{E}$ to boolean values, denoted and defined by:

$$\begin{aligned} \mathcal{E}_c : & \boldsymbol{E}_B \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \boldsymbol{\Sigma}) \\ & exp \longrightarrow \mathcal{E}_c[\![exp]\!] = \lambda\sigma.\textbf{if } \mathcal{E}[\![exp]\!]\sigma \textbf{ then } \sigma \textbf{ else } \bot \end{aligned}$$

Appendix D

Preliminary Analyses

This appendix gives the exact and approximate semantics of transformers (Tables D.1 and D.2), preconditions (Table D.4 and D.5) and continuation conditions (Tables D.6 and D.7). All these analyses rely on the evaluation of expressions, which is denoted as $\mathcal{E}$. The reader is refered to Chapter 5 for more explanations.

$$\mathcal{T} : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \boldsymbol{\Sigma})$$

$$\begin{array}{lcl}
\mathcal{T}[\![\texttt{continue}]\!] & = & \lambda\sigma.\sigma \\
\mathcal{T}[\![\texttt{stop}]\!] & = & \lambda\sigma.\bot \\
\mathcal{T}[\![var = exp]\!] & = & \lambda\sigma.\sigma_{[var \leftarrow \mathcal{E}[\![exp]\!]\sigma]} \\
\mathcal{T}[\![var(exp_1, \dots, exp_k) = exp]\!] & = & \lambda\sigma.\sigma_{[var(\mathcal{E}[\![exp_1]\!]\sigma, \dots, \mathcal{E}[\![exp_k]\!]\sigma) \leftarrow Ee[\![exp]\!]\sigma]} \\
\mathcal{T}[\![\texttt{read}\ var]\!] & = & \lambda\sigma.\sigma_{[var \leftarrow \texttt{stdin}]} \\
\mathcal{T}[\![\texttt{read}\ var(exp_1, \dots, exp_k)]\!] & = & \lambda\sigma.\sigma_{[var(\mathcal{E}[\![exp_1]\!]\sigma, \dots, \mathcal{E}[\![exp_k]\!]\sigma) \leftarrow \texttt{stdin}]} \\
\mathcal{T}[\![\texttt{write}\ exp]\!] & = & \lambda\sigma.\sigma \\
\mathcal{T}[\![S_1; S_2]\!] & = & \mathcal{T}[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!] \\
\mathcal{T}[\![\texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] & = & (\mathcal{T}[\![S_1]\!] \circ \mathcal{E}_c[\![C]\!]) \cup (\mathcal{T}[\![S_2]\!] \circ \mathcal{E}_c[\![\texttt{.not.}C]\!]) \\
\mathcal{T}[\![\texttt{do while}(C)\ S]\!] & = & \mathrm{lfp}(\lambda f.f \circ \mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!] \cup \mathcal{E}_c[\![\texttt{.not.}C]\!])
\end{array}$$

Table D.1: Exact transformers

$$\overline{\mathcal{T}} : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \tilde{\wp}(\boldsymbol{\Sigma}))$$

$$\begin{array}{lcl}
\overline{\mathcal{T}}[\![\texttt{continue}]\!] & = & \lambda\sigma.\overline{\{\sigma\}} \\
\overline{\mathcal{T}}[\![\texttt{stop}]\!] & = & \lambda\sigma.\overline{\{\bot\}} \\
\overline{\mathcal{T}}[\![var = exp]\!] & = & \lambda\sigma.\overline{\{\sigma_{[var \leftarrow \mathcal{E}[\![exp]\!]\sigma]}\}} \\
\overline{\mathcal{T}}[\![var(exp_1, \ldots, exp_k) = exp]\!] & = & \lambda\sigma.\overline{\{\sigma_{[var(\mathcal{E}[\![exp_1]\!]\sigma, \ldots, \mathcal{E}[\![exp_k]\!]\sigma) \leftarrow \mathcal{E}[\![exp]\!]\sigma]}\}} \\
\overline{\mathcal{T}}[\![\texttt{read}\ var]\!] & = & \lambda\sigma.\overline{\{\sigma_{[var \leftarrow \texttt{stdin}]}\}} \\
\overline{\mathcal{T}}[\![\texttt{read}\ var(exp_1, \ldots, exp_k)]\!] & = & \lambda\sigma.\overline{\{\sigma_{[var(\mathcal{E}[\![exp_1]\!]\sigma, \ldots, \mathcal{E}[\![exp_k]\!]\sigma) \leftarrow \texttt{stdin}]}\}} \\
\overline{\mathcal{T}}[\![\texttt{write}\ exp]\!] & = & \lambda\sigma.\overline{\{\sigma\}} \\
\overline{\mathcal{T}}[\![S_1; S_2]\!] & = & \overline{\mathcal{T}}[\![S_2]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}[\![S_1]\!] \\
\overline{\mathcal{T}}[\![\texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] & = & (\overline{\mathcal{T}}[\![S_1]\!] \,\overline{\circ}\, \overline{\mathcal{E}}_c[\![C]\!]) \,\overline{\cup}\, (\overline{\mathcal{T}}[\![S_2]\!] \,\overline{\circ}\, \overline{\mathcal{E}}_c[\![\texttt{.not.}C]\!]) \\
\overline{\mathcal{T}}[\![\texttt{do while(}C\texttt{)}\ S]\!] & = & \overline{\mathrm{lfp}}(\lambda f. f \,\overline{\bullet}\, \overline{\mathcal{T}}[\![S]\!] \,\overline{\circ}\, \overline{\mathcal{E}}_c[\![C]\!] \,\overline{\cup}\, \overline{\mathcal{E}}_c[\![\texttt{.not.}C]\!])
\end{array}$$

Table D.2: Approximate transformers

$$\overline{\mathcal{T}}^{-1} : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \tilde{\wp}(\boldsymbol{\Sigma}))$$

$$\begin{array}{lcl}
\overline{\mathcal{T}}^{-1}[\![\texttt{continue}]\!] & = & \lambda\sigma.\overline{\{\sigma\}} \\
\overline{\mathcal{T}}^{-1}[\![\texttt{stop}]\!] & = & \lambda\sigma.\top \\
\overline{\mathcal{T}}^{-1}[\![var = exp]\!] & = & \lambda\sigma.\overline{\{\sigma' : \sigma = \mathcal{T}[\![var = exp]\!]\sigma'\}} \\
\overline{\mathcal{T}}^{-1}[\![var(exp_1, \ldots, exp_k) = exp]\!] & = & \lambda\sigma.\overline{\{\sigma' : \sigma = \mathcal{T}[\![var(exp_1, \ldots, exp_k) = exp]\!]\sigma'\}} \\
\overline{\mathcal{T}}^{-1}[\![\texttt{read}\ var]\!] & = & \lambda\sigma.\overline{\{\sigma' : \sigma = \mathcal{T}[\![\texttt{read}\ var]\!]\sigma'\}} \\
\overline{\mathcal{T}}^{-1}[\![\texttt{read}\ var(exp_1, \ldots, exp_k)]\!] & = & \lambda\sigma.\overline{\{\sigma' : \sigma = \mathcal{T}[\![\texttt{read}\ var(exp_1, \ldots, exp_k)]\!]\sigma'\}} \\
\overline{\mathcal{T}}^{-1}[\![\texttt{write}\ exp]\!] & = & \lambda\sigma.\overline{\{\sigma\}} \\
\overline{\mathcal{T}}^{-1}[\![S_1; S_2]\!] & = & \overline{\mathcal{T}}^{-1}[\![S_1]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}^{-1}[\![S_2]\!] \\
\overline{\mathcal{T}}^{-1}[\![\texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] & = & (\overline{\mathcal{E}}_c[\![C]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}^{-1}[\![S_1]\!]) \,\overline{\cup}\, (\overline{\mathcal{E}}_c[\![\texttt{.not.}C]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}^{-1}[\![S_2]\!]) \\
\overline{\mathcal{T}}^{-1}[\![\texttt{do while(}C\texttt{)}\ S]\!] & = & \\
 & & \overline{\mathrm{lfp}}(\lambda f.(\overline{\mathcal{E}}[\![C]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}^{-1}[\![S]\!] \,\overline{\bullet}\, f) \,\overline{\cup}\, (\overline{\mathcal{E}}_c[\![\texttt{.not.}C]\!]) \,\overline{\bullet}\, \lambda\sigma.\overline{\{\sigma\}}\}))
\end{array}$$

Table D.3: Approximate inverse transformers

$$\mathcal{P} : \boldsymbol{S} \longrightarrow \boldsymbol{\Sigma} \longrightarrow \boldsymbol{\Sigma}$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{S_1; S_2} \\
\quad \mathcal{P}[\![S_1]\!] & = & \mathcal{P}[\![S_1; S_2]\!] \\
\quad \mathcal{P}[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!] & = & \mathcal{T}[\![S_1]\!] \circ \mathcal{P}[\![S_1]\!] \\
\multicolumn{3}{l}{\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2} \\
\quad \mathcal{P}[\![S_1]\!] & = & \mathcal{E}_c[\![C]\!] \circ \mathcal{P}[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] \\
\quad \mathcal{P}[\![S_2]\!] & = & \mathcal{E}_c[\![\texttt{.not.}C]\!] \circ \mathcal{P}[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] \\
\multicolumn{3}{l}{\texttt{do while(}C\texttt{) } S} \\
\quad \mathcal{P}_1[\![\texttt{do while(}C\texttt{) } S]\!] & = & \mathcal{P}[\![\texttt{do while(}C\texttt{) } S]\!] \\
\quad \mathcal{P}_i[\![\texttt{do while(}C\texttt{) } S]\!] \circ \mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!] = & & \\
 & \multicolumn{2}{l}{\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!] \circ \mathcal{P}_{i-1}[\![\texttt{do while(}C\texttt{) } S]\!],\ \forall i > 1} \\
\quad \mathcal{P}_i[\![S]\!] & = & \mathcal{E}_c[\![C]\!] \circ \mathcal{P}_i[\![\texttt{do while(}C\texttt{) } S]\!]
\end{array}
$$

Table D.4: Exact preconditions

$$\overline{\mathcal{P}} : \boldsymbol{S} \longrightarrow \boldsymbol{\Sigma} \longrightarrow \boldsymbol{\Sigma}$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{S_1; S_2} \\
\quad \overline{\mathcal{P}}[\![S_1]\!] & = & \overline{\mathcal{P}}[\![S_1; S_2]\!] \\
\quad \overline{\mathcal{P}}[\![S_2]\!] & = & \overline{\mathcal{T}}[\![S_1]\!] \,\overline{\bullet}\, \overline{\mathcal{P}}[\![S_1]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}^{-1}[\![S_1]\!] \\
\multicolumn{3}{l}{\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2} \\
\quad \overline{\mathcal{P}}[\![S_1]\!] & = & \overline{\mathcal{E}}_c[\![C]\!] \,\overline{\circ}\, \overline{\mathcal{P}}[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] \\
\quad \overline{\mathcal{P}}[\![S_2]\!] & = & \overline{\mathcal{E}}_c[\![\texttt{.not.}C]\!] \,\overline{\circ}\, \overline{\mathcal{P}}[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] \\
\multicolumn{3}{l}{\texttt{do while(}C\texttt{) } S} \\
\quad \overline{\mathcal{P}}_{\text{inv}}[\![\texttt{do while(}C\texttt{) } S]\!] & = & \\
\multicolumn{3}{r}{\overline{\text{lfp}}(\lambda f. \overline{\mathcal{T}}[\![S]\!] \,\overline{\bullet}\, \overline{\mathcal{E}}_c[\![C]\!] \,\overline{\bullet}\, f \,\overline{\bullet}\, \overline{\mathcal{E}}_c[\![C]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}^{-1}[\![S]\!]) \,\overline{\cup}\, \overline{\mathcal{P}}[\![\texttt{do while(}C\texttt{) } S]\!]} \\
\quad \overline{\mathcal{P}}[\![S]\!] & = & \overline{\mathcal{E}}_c[\![C]\!] \,\overline{\circ}\, \overline{\mathcal{P}}_{\text{inv}}[\![\texttt{do while(}C\texttt{) } S]\!]
\end{array}
$$

Table D.5: Over-approximate preconditions

$$\mathcal{C} : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \boldsymbol{\Sigma})$$

$$\begin{array}{lcl}
\mathcal{C}[\![\texttt{continue}]\!] & = & \lambda\sigma.\sigma \\
\mathcal{C}[\![\texttt{stop}]\!] & = & \lambda\sigma.\bot \\
\mathcal{C}[\![\mathit{ref} = \mathit{exp}]\!] & = & \lambda\sigma.\sigma \\
\mathcal{C}[\![\texttt{read}\ \mathit{ref}]\!] & = & \lambda\sigma.\sigma \\
\mathcal{C}[\![\texttt{write}\ \mathit{exp}]\!] & = & \lambda\sigma.\sigma \\
\mathcal{C}[\![S_1; S_2]\!] & = & \mathcal{C}[\![S_1]\!] \cap \mathcal{C}[\![S_2]\!] \circ \mathcal{T}[\![S1]\!] \\
\mathcal{C}[\![\texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] & = & (\mathcal{C}[\![S_1]\!] \circ \mathcal{E}_c[\![C]\!]) \cup (\mathcal{C}[\![S_2]\!] \circ \mathcal{E}_c[\![\texttt{.not.}C]\!]) \\
\mathcal{C}[\![\texttt{do while(}C\texttt{)}\ S]\!] & = & \mathrm{lfp}(\lambda f.(\mathcal{C}[\![S]\!] \cap f \circ \mathcal{T}[\![S]\!]) \circ \mathcal{E}_c[\![C]\!] \cup \mathcal{E}_c[\![\texttt{.not.}C]\!])
\end{array}$$

Table D.6: Exact continuation conditions

$$\overline{\mathcal{C}} : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \boldsymbol{\Sigma})$$

$$\begin{array}{lcl}
\overline{\mathcal{C}}[\![\texttt{continue}]\!]\sigma & = & \lambda\sigma.\sigma \\
\overline{\mathcal{C}}[\![\texttt{stop}]\!] & = & \lambda\sigma.\bot \\
\overline{\mathcal{C}}[\![\mathit{ref} = \mathit{exp}]\!] & = & \lambda\sigma.\sigma \\
\overline{\mathcal{C}}[\![\texttt{read}\ \mathit{ref}]\!] & = & \lambda\sigma.\sigma \\
\overline{\mathcal{C}}[\![\texttt{write}\ \mathit{exp}]\!] & = & \lambda\sigma.\sigma \\
\overline{\mathcal{C}}[\![S_1; S_2]\!] & = & \overline{\mathcal{C}}[\![S_1]\!] \mathbin{\overline{\cap}} \overline{\mathcal{C}}[\![S_2]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{T}}[\![S1]\!] \\
\overline{\mathcal{C}}[\![\texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] & = & (\overline{\mathcal{C}}[\![S_1]\!] \mathbin{\overline{\circ}} \overline{\mathcal{E}}_c[\![C]\!]) \mathbin{\overline{\cup}} (\overline{\mathcal{C}}[\![S_2]\!] \mathbin{\overline{\circ}} \overline{\mathcal{E}}_c[\![\texttt{.not.}C]\!]) \\
\overline{\mathcal{C}}[\![\texttt{do while(}C\texttt{)}\ S]\!] & = & \overline{\mathrm{lfp}}(\lambda f.(\overline{\mathcal{C}}[\![S]\!] \mathbin{\overline{\cap}} f \mathbin{\overline{\bullet}} \overline{\mathcal{T}}[\![S]\!]) \mathbin{\overline{\circ}} \overline{\mathcal{E}}_c[\![C]\!] \mathbin{\overline{\cup}} \overline{\mathcal{E}}_c[\![\texttt{.not.}C]\!])
\end{array}$$

$$\underline{\mathcal{C}} : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \boldsymbol{\Sigma})$$

$$\begin{array}{lcl}
\underline{\mathcal{C}}[\![\texttt{continue}]\!]\sigma & = & \lambda\sigma.\sigma \\
\underline{\mathcal{C}}[\![\texttt{stop}]\!] & = & \lambda\sigma.\bot \\
\underline{\mathcal{C}}[\![\mathit{ref} = \mathit{exp}]\!] & = & \lambda\sigma.\sigma \\
\underline{\mathcal{C}}[\![\texttt{read}\ \mathit{ref}]\!] & = & \lambda\sigma.\sigma \\
\underline{\mathcal{C}}[\![\texttt{write}\ \mathit{exp}]\!] & = & \lambda\sigma.\sigma \\
\underline{\mathcal{C}}[\![S_1; S_2]\!] & = & \underline{\mathcal{C}}[\![S_1]\!] \mathbin{\underline{\cap}} \underline{\mathcal{C}}[\![S_2]\!] \mathbin{\underline{\bullet}} \overline{\mathcal{T}}[\![S1]\!] \\
\underline{\mathcal{C}}[\![\texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] & = & (\underline{\mathcal{C}}[\![S_1]\!] \mathbin{\underline{\circ}} \underline{\mathcal{E}}_c[\![C]\!]) \mathbin{\underline{\cup}} (\underline{\mathcal{C}}[\![S_2]\!] \mathbin{\underline{\circ}} \underline{\mathcal{E}}_c[\![\texttt{.not.}C]\!]) \\
\underline{\mathcal{C}}[\![\texttt{do while(}C\texttt{)}\ S]\!] & = & \overline{\mathrm{lfp}}(\lambda f.(\underline{\mathcal{C}}[\![S]\!] \mathbin{\underline{\cap}} f \mathbin{\underline{\bullet}} \overline{\mathcal{T}}[\![S]\!]) \mathbin{\underline{\circ}} \underline{\mathcal{E}}_c[\![C]\!] \mathbin{\underline{\cup}} \underline{\mathcal{E}}_c[\![\texttt{.not.}C]\!])
\end{array}$$

Table D.7: Approximate continuation conditions

Appendix E

Array Regions: Semantics

This appendix summarizes the exact and approximate semantic functions of READ, WRITE, IN and OUT regions.

$$\mathcal{R}_r : \boldsymbol{S} \oplus \boldsymbol{E} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \wp(\mathbb{Z}^{d_i}))$$

$$\begin{array}{lcl}
\mathcal{R}_r[\![c]\!] & = & \lambda\sigma.\varnothing \\
\mathcal{R}_r[\![var]\!] & = & \lambda\sigma.(\{var\}) \\
\mathcal{R}_r[\![var(exp_1,\ldots,exp_k)]\!] & = & \lambda\sigma.(\{var(\mathcal{E}[\![exp_1,\ldots,exp_k]\!]\sigma)\}) \\
\mathcal{R}_r[\![\mathrm{op}_1\ exp]\!] & = & \mathcal{R}_r[\![exp]\!] \\
\mathcal{R}_r[\![exp_1\ \mathrm{op}_2\ exp_1]\!] & = & \mathcal{R}_r[\![exp_1]\!] \cup \mathcal{R}_r[\![exp_2]\!] \\
\mathcal{R}_r[\![exp_1,\ldots,exp_k]\!] & = & \bigcup_{i=1}^{k} \mathcal{R}_r[\![exp_i]\!]
\end{array}$$

$$\begin{array}{lcl}
\mathcal{R}_r[\![\texttt{continue}]\!] & = & \lambda\sigma.\varnothing \\
\mathcal{R}_r[\![\texttt{stop}]\!] & = & \lambda\sigma.\varnothing \\
\mathcal{R}_r[\![var(exp_1,\ldots,exp_k) = exp]\!] & = & \mathcal{R}_r[\![exp_1,\ldots,exp_k]\!] \cup \mathcal{R}_r[\![exp]\!] \\
\mathcal{R}_r[\![var = exp]\!] & = & \mathcal{R}_r[\![exp]\!] \\
\mathcal{R}_r[\![\texttt{read}\ var]\!] & = & \lambda\sigma.\varnothing \\
\mathcal{R}_r[\![\texttt{read}\ var(exp_1,\ldots,exp_k)]\!] & = & \mathcal{R}_r[\![exp_1,\ldots,exp_k]\!] \\
\mathcal{R}_r[\![\texttt{write}\ exp]\!] & = & \mathcal{R}_r[\![exp]\!] \\
\mathcal{R}_r[\![S_1;S_2]\!] & = & \mathcal{R}_r[\![S_1]\!] \cup \mathcal{R}_r[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!] \\
\mathcal{R}_r[\![\texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] & = & \\
\multicolumn{3}{r}{\mathcal{R}_r[\![C]\!] \cup (\mathcal{R}_r[\![S_1]\!] \circ \mathcal{E}_c[\![C]\!]) \cup (\mathcal{R}_r[\![S_2]\!] \circ \mathcal{E}_c[\![\texttt{.not.}C]\!]\ )} \\
\mathcal{R}_r[\![\texttt{do while}(C)\ S]\!] & = & \\
\multicolumn{3}{r}{\mathrm{lfp}\big(\lambda f.\mathcal{R}_r[\![C]\!] \cup (\mathcal{R}_r[\![S]\!] \cup f \circ \mathcal{T}[\![S]\!]) \circ \mathcal{E}_c[\![C]\!]\big)}
\end{array}$$

Table E.1: Exact READ regions

$$\overline{\mathcal{R}}_r : \boldsymbol{S} \oplus \boldsymbol{E} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \tilde{\wp}(\mathbb{Z}^{d_i}))$$

$$\begin{array}{lcl}
\overline{\mathcal{R}}_r[\![c]\!] & = & \lambda\sigma.\varnothing \\
\overline{\mathcal{R}}_r[\![var]\!] & = & \lambda\sigma.\overline{\{var\}} \\
\overline{\mathcal{R}}_r[\![var(exp_1,\ldots,exp_k)]\!] & = & \lambda\sigma.\overline{\{var(\overline{\mathcal{E}}[\![exp_1,\ldots,exp_k]\!]\sigma)\}} \\
\overline{\mathcal{R}}_r[\![\mathrm{op}_1\ exp]\!] & = & \overline{\mathcal{R}}_r[\![exp]\!] \\
\overline{\mathcal{R}}_r[\![exp_1\ \mathrm{op}_2\ exp_1]\!] & = & \overline{\mathcal{R}}_r[\![exp_1]\!] \,\overline{\cup}\, \overline{\mathcal{R}}_r[\![exp_2]\!] \\
\overline{\mathcal{R}}_r[\![exp_1,\ldots,exp_k]\!] & = & \overline{\bigcup}_{i=1}^{k} \overline{\mathcal{R}}_r[\![exp_i]\!]
\end{array}$$

$$\begin{array}{lcl}
\overline{\mathcal{R}}_r[\![\texttt{continue}]\!] & = & \lambda\sigma.\varnothing \\
\overline{\mathcal{R}}_r[\![\texttt{stop}]\!] & = & \lambda\sigma.\varnothing \\
\overline{\mathcal{R}}_r[\![var(exp_1,\ldots,exp_k) = exp]\!] & = & \overline{\mathcal{R}}_r[\![exp_1,\ldots,exp_k]\!] \,\overline{\cup}\, \overline{\mathcal{R}}_r[\![exp]\!] \\
\overline{\mathcal{R}}_r[\![var = exp]\!] & = & \overline{\mathcal{R}}_r[\![exp]\!] \\
\overline{\mathcal{R}}_r[\![\texttt{read}\ var]\!] & = & \lambda\sigma.\varnothing \\
\overline{\mathcal{R}}_r[\![\texttt{read}\ var(exp_1,\ldots,exp_k)]\!] & = & \overline{\mathcal{R}}_r[\![exp_1,\ldots,exp_k]\!] \\
\overline{\mathcal{R}}_r[\![\texttt{write}\ exp]\!] & = & \overline{\mathcal{R}}_r[\![exp]\!] \\
\overline{\mathcal{R}}_r[\![S_1;S_2]\!] & = & \overline{\mathcal{R}}_r[\![S_1]\!] \,\overline{\cup}\, \overline{\mathcal{R}}_r[\![S_2]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}[\![S_1]\!] \,\overline{\circ}\, \overline{\mathcal{C}}[\![S_1]\!] \\
\overline{\mathcal{R}}_r[\![\texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] & = & \\
\multicolumn{3}{r}{\overline{\mathcal{R}}_r[\![C]\!] \,\overline{\cup}\, (\overline{\mathcal{R}}_r[\![S_1]\!] \,\overline{\circ}\, \overline{\mathcal{E}}_c[\![C]\!]) \,\overline{\cup}\, (\overline{\mathcal{R}}_r[\![S_2]\!] \,\overline{\circ}\, \overline{\mathcal{E}}_c[\![\texttt{.not.}C]\!])} \\
\overline{\mathcal{R}}_r[\![\texttt{do while(}C\texttt{)}\ S]\!] & = & \\
\multicolumn{3}{r}{\overline{\mathrm{lfp}}\big(\lambda f.\overline{\mathcal{R}}_r[\![C]\!] \,\overline{\cup}\, (\overline{\mathcal{R}}_r[\![S]\!] \,\overline{\cup}\, f \,\overline{\bullet}\, \overline{\mathcal{T}}[\![S]\!] \,\overline{\circ}\, \overline{\mathcal{C}}[\![S]\!]) \,\overline{\circ}\, \overline{\mathcal{E}}_c[\![C]\!]\big)}
\end{array}$$

Table E.2: Over-approximate READ regions

$$\underline{\mathcal{R}_r} : \boldsymbol{S} \oplus \boldsymbol{E} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \tilde{\wp}(\mathbb{Z}^{d_i}))$$

$$\begin{array}{lcl}
\underline{\mathcal{R}_r}[\![c]\!] & = & \lambda\sigma.\varnothing \\
\underline{\mathcal{R}_r}[\![var]\!] & = & \lambda\sigma.\underline{\{var\}} \\
\underline{\mathcal{R}_r}[\![var(exp_1,\ldots,exp_k)]\!] & = & \lambda\sigma.\underline{\{var(\underline{\mathcal{E}}[\![exp_1,\ldots,exp_k]\!]\sigma)\}} \\
\underline{\mathcal{R}_r}[\![\mathrm{op}_1\ exp]\!] & = & \underline{\mathcal{R}_r}[\![exp]\!] \\
\underline{\mathcal{R}_r}[\![exp_1\ \mathrm{op}_2\ exp_1]\!] & = & \underline{\mathcal{R}_r}[\![exp_1]\!] \,\underline{\cup}\, \underline{\mathcal{R}_r}[\![exp_2]\!] \\
\underline{\mathcal{R}_r}[\![exp_1,\ldots,exp_k]\!] & = & \underline{\bigcup}_{i=1}^{k} \underline{\mathcal{R}_r}[\![exp_i]\!]
\end{array}$$

$$\begin{array}{lcl}
\underline{\mathcal{R}_r}[\![\texttt{continue}]\!] & = & \lambda\sigma.\varnothing \\
\underline{\mathcal{R}_r}[\![\texttt{stop}]\!] & = & \lambda\sigma.\varnothing \\
\underline{\mathcal{R}_r}[\![var(exp_1,\ldots,exp_k) = exp]\!] & = & \underline{\mathcal{R}_r}[\![exp_1,\ldots,exp_k]\!] \,\underline{\cup}\, \underline{\mathcal{R}_r}[\![exp]\!] \\
\underline{\mathcal{R}_r}[\![var = exp]\!] & = & \underline{\mathcal{R}_r}[\![exp]\!] \\
\underline{\mathcal{R}_r}[\![\texttt{read}\ var]\!] & = & \lambda\sigma.\varnothing \\
\underline{\mathcal{R}_r}[\![\texttt{read}\ var(exp_1,\ldots,exp_k)]\!] & = & \underline{\mathcal{R}_r}[\![exp_1,\ldots,exp_k]\!] \\
\underline{\mathcal{R}_r}[\![\texttt{write}\ exp]\!] & = & \underline{\mathcal{R}_r}[\![exp]\!] \\
\underline{\mathcal{R}_r}[\![S_1;S_2]\!] & = & \underline{\mathcal{R}_r}[\![S_1]\!] \,\underline{\cup}\, \underline{\mathcal{R}_r}[\![S_2]\!] \,\underline{\bullet}\, \overline{\mathcal{T}}[\![S_1]\!] \,\underline{\circ}\, \underline{\mathcal{C}}[\![S_1]\!] \\
\underline{\mathcal{R}_r}[\![\texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] & = & \\
\multicolumn{3}{l}{\quad \underline{\mathcal{R}_r}[\![C]\!] \,\underline{\cup}\, (\underline{\mathcal{R}_r}[\![S_1]\!] \,\underline{\circ}\, \underline{\mathcal{E}_c}[\![C]\!]) \,\underline{\cup}\, (\underline{\mathcal{R}_r}[\![S_2]\!] \,\underline{\circ}\, \underline{\mathcal{E}_c}[\![\texttt{.not.}C]\!]) \,\underline{\cup}\, (\underline{\mathcal{R}_r}[\![S_1]\!] \,\underline{\cap}\, \underline{\mathcal{R}_r}[\![S_2]\!])} \\
\underline{\mathcal{R}_r}[\![\texttt{do while}(C)\ S]\!] & = & \\
\multicolumn{3}{r}{\underline{\mathrm{lfp}}\big(\lambda f.\underline{\mathcal{R}_r}[\![C]\!] \,\underline{\cup}\, (\underline{\mathcal{R}_r}[\![S]\!] \,\underline{\cup}\, f \,\underline{\bullet}\, \overline{\mathcal{T}}[\![S]\!] \,\underline{\circ}\, \underline{\mathcal{C}}[\![S]\!]) \,\underline{\circ}\, \underline{\mathcal{E}_c}[\![C]\!]\big)}
\end{array}$$

Table E.3: Under-approximate READ regions

$$\mathcal{R}_w : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \wp(\mathbb{Z}^{d_i}))$$

$$
\begin{array}{lcl}
\mathcal{R}_w[\![\texttt{continue}]\!] & = & \lambda\sigma.\varnothing \\
\mathcal{R}_w[\![\texttt{stop}]\!] & = & \lambda\sigma.\varnothing \\
\mathcal{R}_w[\![var(exp_1,\ldots,exp_k) = exp]\!] & = & \lambda\sigma.\{var(\mathcal{E}[\![exp_1,\ldots,exp_k]\!]\sigma)\} \\
\mathcal{R}_w[\![var = exp]\!] & = & \lambda\sigma.\{var\} \\
\mathcal{R}_w[\![\texttt{read}\ var]\!] & = & \lambda\sigma.\{var\} \\
\mathcal{R}_w[\![\texttt{read}\ var(exp_1,\ldots,exp_k)]\!] & = & \lambda\sigma.\{var(\mathcal{E}[\![exp_1,\ldots,exp_k]\!]\sigma)\} \\
\mathcal{R}_w[\![\texttt{write}\ exp]\!] & = & \lambda\sigma.\varnothing \\
\mathcal{R}_w[\![S_1;S_2]\!] & = & \mathcal{R}_w[\![S_1]\!] \cup \mathcal{R}_w[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!] \\
\mathcal{R}_w[\![\texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] & = & \\
\multicolumn{3}{r}{(\mathcal{R}_w[\![S_1]\!] \circ \mathcal{E}_c[\![C]\!]) \cup (\mathcal{R}_w[\![S_2]\!] \circ \mathcal{E}_c[\![\texttt{.not.}C]\!])} \\
\mathcal{R}_w[\![\texttt{do while(}C\texttt{)}\ S]\!] & = & \mathrm{lfp}\big(\lambda f.(\mathcal{R}_w[\![S]\!] \cup f \circ \mathcal{T}[\![S]\!]) \circ \mathcal{E}_c[\![C]\!]\big)
\end{array}
$$

Table E.4: Exact WRITE regions

$$\overline{\mathcal{R}}_w : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \tilde{\wp}(\mathbb{Z}^{d_i}))$$

$$\begin{array}{lcl}
\overline{\mathcal{R}}_w[\![\texttt{continue}]\!] & = & \lambda\sigma.\varnothing \\
\overline{\mathcal{R}}_w[\![\texttt{stop}]\!] & = & \lambda\sigma.\varnothing \\
\overline{\mathcal{R}}_w[\![var(exp_1,\ldots,exp_k) = exp]\!] & = & \lambda\sigma.\overline{\{var(\mathcal{E}[\![exp_1,\ldots,exp_k]\!]\sigma)\}} \\
\overline{\mathcal{R}}_w[\![var = exp]\!] & = & \lambda\sigma.\overline{\{var\}} \\
\overline{\mathcal{R}}_w[\![\texttt{read}\ var]\!] & = & \lambda\sigma.\overline{\{var\}} \\
\overline{\mathcal{R}}_w[\![\texttt{read}\ var(exp_1,\ldots,exp_k)]\!] & = & \lambda\sigma.\overline{\{var(\mathcal{E}[\![exp_1,\ldots,exp_k]\!]\sigma)\}} \\
\overline{\mathcal{R}}_w[\![\texttt{write}\ exp]\!] & = & \lambda\sigma.\varnothing \\
\overline{\mathcal{R}}_w[\![S_1;S_2]\!] & = & \overline{\mathcal{R}}_w[\![S_1]\!]\,\overline{\cup}\,\overline{\mathcal{R}}_w[\![S_2]\!]\,\overline{\bullet}\,\overline{\mathcal{T}}[\![S_1]\!]\,\overline{\circ}\,\overline{\mathcal{C}}[\![S_1]\!] \\
\overline{\mathcal{R}}_w[\![\texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] & = & \\
\multicolumn{3}{r}{(\overline{\mathcal{R}}_w[\![S_1]\!]\,\overline{\circ}\,\overline{\mathcal{E}}_c[\![C]\!])\,\overline{\cup}\,(\overline{\mathcal{R}}_w[\![S_2]\!]\,\overline{\circ}\,\overline{\mathcal{E}}_c[\![\texttt{.not.}C]\!])} \\
\overline{\mathcal{R}}_w[\![\texttt{do while}(C)\ S]\!] & = & \\
\multicolumn{3}{r}{\overline{\mathrm{lfp}}\big(\lambda f.(\overline{\mathcal{R}}_w[\![S]\!]\,\overline{\cup}\,f\,\overline{\bullet}\,\overline{\mathcal{T}}[\![S]\!]\,\overline{\circ}\,\overline{\mathcal{C}}[\![S]\!])\,\overline{\circ}\,\overline{\mathcal{E}}_c[\![C]\!]\big)}
\end{array}$$

$$\underline{\mathcal{R}}_w : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \tilde{\wp}(\mathbb{Z}^{d_i}))$$

$$\begin{array}{lcl}
\underline{\mathcal{R}}_w[\![\texttt{continue}]\!] & = & \lambda\sigma.\varnothing \\
\underline{\mathcal{R}}_w[\![\texttt{stop}]\!] & = & \lambda\sigma.\varnothing \\
\underline{\mathcal{R}}_w[\![var(exp_1,\ldots,exp_k) = exp]\!] & = & \lambda\sigma.\underline{\{var(\mathcal{E}[\![exp_1,\ldots,exp_k]\!]\sigma)\}} \\
\underline{\mathcal{R}}_w[\![var = exp]\!] & = & \lambda\sigma.\underline{\{var\}} \\
\underline{\mathcal{R}}_w[\![\texttt{read}\ var]\!] & = & \lambda\sigma.\underline{\{var\}} \\
\underline{\mathcal{R}}_w[\![\texttt{read}\ var(exp_1,\ldots,exp_k)]\!] & = & \lambda\sigma.\underline{\{var(\mathcal{E}[\![exp_1,\ldots,exp_k]\!]\sigma)\}} \\
\underline{\mathcal{R}}_w[\![\texttt{write}\ exp]\!] & = & \lambda\sigma.\varnothing \\
\underline{\mathcal{R}}_w[\![S_1;S_2]\!] & = & \underline{\mathcal{R}}_w[\![S_1]\!]\,\underline{\cup}\,\underline{\mathcal{R}}_w[\![S_2]\!]\,\underline{\bullet}\,\overline{\mathcal{T}}[\![S_1]\!]\,\underline{\circ}\,\underline{\mathcal{C}}[\![S_1]\!] \\
\underline{\mathcal{R}}_w[\![\texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] & = & \\
\multicolumn{3}{r}{(\underline{\mathcal{R}}_w[\![S_1]\!]\,\underline{\circ}\,\underline{\mathcal{E}}_c[\![C]\!])\,\underline{\cup}\,(\underline{\mathcal{R}}_w[\![S_2]\!]\,\underline{\circ}\,\underline{\mathcal{E}}_c[\![\texttt{.not.}C]\!])\,\underline{\cup}\,(\underline{\mathcal{R}}_w[\![S_1]\!]\,\underline{\cap}\,\underline{\mathcal{R}}_w[\![S_2]\!])} \\
\underline{\mathcal{R}}_w[\![\texttt{do while}(C)\ S]\!] & = & \\
\multicolumn{3}{r}{\underline{\mathrm{lfp}}\big(\lambda f.(\underline{\mathcal{R}}_w[\![S]\!]\,\underline{\cup}\,f\,\underline{\bullet}\,\overline{\mathcal{T}}[\![S]\!]\,\underline{\circ}\,\underline{\mathcal{C}}[\![S]\!])\,\underline{\circ}\,\underline{\mathcal{E}}_c[\![C]\!]\big)}
\end{array}$$

Table E.5: Approximate WRITE regions

$$\mathcal{R}_i : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \wp(\mathbb{Z}^{d_i}))$$

$$
\begin{array}{lcl}
\mathcal{R}_i[\![\texttt{continue}]\!] & = & \lambda\sigma.\varnothing \\
\mathcal{R}_i[\![\texttt{stop}]\!] & = & \lambda\sigma.\varnothing \\
\mathcal{R}_i[\![\mathit{ref} = \mathit{exp}]\!] & = & \mathcal{R}_r[\![\mathit{ref} = \mathit{exp}]\!] \\
\mathcal{R}_i[\![\texttt{read}\ \mathit{ref}]\!] & = & \mathcal{R}_r[\![\texttt{read}\ \mathit{ref}]\!] \\
\mathcal{R}_i[\![\texttt{write}\ \mathit{exp}]\!] & = & \mathcal{R}_r[\![\texttt{write}\ \mathit{exp}]\!] \\
\mathcal{R}_i[\![S_1; S_2]\!] & = & \mathcal{R}_i[\![S_1]\!] \cup \big((\mathcal{R}_i[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]) \boxminus \mathcal{R}_w[\![S_1]\!]\big) \\
\mathcal{R}_i[\![\texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] & = & \\
\multicolumn{3}{r}{\mathcal{R}_r[\![C]\!] \cup (\mathcal{R}_i[\![S_1]\!] \circ \mathcal{E}_c[\![C]\!]) \cup (\mathcal{R}_i[\![S_2]\!] \circ \mathcal{E}_c[\![\texttt{.not.}C]\!])} \\
\mathcal{R}_i[\![\texttt{do while(}C\texttt{)}\ S]\!] & = & \\
\multicolumn{3}{r}{\mathrm{lfp}\big(\lambda f.\mathcal{R}_i[\![C]\!] \cup \big\{\mathcal{R}_i[\![S]\!] \cup \big((f \circ \mathcal{T}[\![S]\!]) \boxminus \mathcal{R}_w[\![S]\!]\big)\big\} \circ \mathcal{E}_c[\![C]\!]\big)}
\end{array}
$$

Table E.6: Exact IN regions

$$\overline{\mathcal{R}}_i : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \tilde{\wp}(\mathbb{Z}^{d_i}))$$

$$\begin{array}{lcl}
\overline{\mathcal{R}}_i[\![\texttt{continue}]\!] & = & \lambda\sigma.\varnothing \\
\overline{\mathcal{R}}_i[\![\texttt{stop}]\!] & = & \lambda\sigma.\varnothing \\
\overline{\mathcal{R}}_i[\![\mathit{ref} = \mathit{exp}]\!] & = & \overline{\mathcal{R}}_r[\![\mathit{ref} = \mathit{exp}]\!] \\
\overline{\mathcal{R}}_i[\![\texttt{read } \mathit{ref}]\!] & = & \overline{\mathcal{R}}_r[\![\texttt{read } \mathit{ref}]\!] \\
\overline{\mathcal{R}}_i[\![\texttt{write } \mathit{exp}]\!] & = & \overline{\mathcal{R}}_r[\![\texttt{write } \mathit{exp}]\!] \\
\overline{\mathcal{R}}_i[\![S_1; S_2]\!] & = & \\
\multicolumn{3}{r}{\overline{\mathcal{R}}_i[\![S_1]\!] \mathbin{\overline{\cup}} \big((\overline{\mathcal{R}}_i[\![S_2]\!] \mathbin{\overline{\bullet}} \overline{\mathcal{T}}[\![S_1]\!] \mathbin{\overline{\circ}} \overline{\mathcal{C}}[\![S_1]\!]) \mathbin{\overline{\boxminus}} \underline{\mathcal{R}}_w[\![S_1]\!]\big)} \\
\overline{\mathcal{R}}_i[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] & = & \\
\multicolumn{3}{r}{\overline{\mathcal{R}}_r[\![C]\!] \mathbin{\overline{\cup}} (\overline{\mathcal{R}}_i[\![S_1]\!] \mathbin{\overline{\circ}} \overline{\mathcal{E}}_c[\![C]\!]) \mathbin{\overline{\cup}} (\overline{\mathcal{R}}_i[\![S_2]\!] \mathbin{\overline{\circ}} \overline{\mathcal{E}}_c[\![\texttt{.not.}C]\!])} \\
\overline{\mathcal{R}}_i[\![\texttt{do while}(C)\ S]\!] & = & \\
\multicolumn{3}{r}{\overline{\mathrm{lfp}}\big(\lambda f.\overline{\mathcal{R}}_i[\![C]\!] \mathbin{\overline{\cup}} \big\{\overline{\mathcal{R}}_i[\![S]\!] \mathbin{\overline{\cup}} \big((f \mathbin{\overline{\bullet}} \overline{\mathcal{T}}[\![S]\!] \mathbin{\overline{\circ}} \overline{\mathcal{C}}[\![S]\!]) \mathbin{\overline{\boxminus}} \underline{\mathcal{R}}_w[\![S]\!]\big)\big\} \mathbin{\overline{\circ}} \overline{\mathcal{E}}_c[\![C]\!]\big)}
\end{array}$$

$$\underline{\mathcal{R}}_i : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \tilde{\wp}(\mathbb{Z}^{d_i}))$$

$$\begin{array}{lcl}
\underline{\mathcal{R}}_i[\![\texttt{continue}]\!] & = & \lambda\sigma.\varnothing \\
\underline{\mathcal{R}}_i[\![\texttt{stop}]\!] & = & \lambda\sigma.\varnothing \\
\underline{\mathcal{R}}_i[\![\mathit{ref} = \mathit{exp}]\!] & = & \underline{\mathcal{R}}_r[\![\mathit{ref} = \mathit{exp}]\!] \\
\underline{\mathcal{R}}_i[\![\texttt{read } \mathit{ref}]\!] & = & \underline{\mathcal{R}}_r[\![\texttt{read } \mathit{ref}]\!] \\
\underline{\mathcal{R}}_i[\![\texttt{write } \mathit{exp}]\!] & = & \underline{\mathcal{R}}_r[\![\texttt{write } \mathit{exp}]\!] \\
\underline{\mathcal{R}}_i[\![S_1; S_2]\!] & = & \\
\multicolumn{3}{r}{\underline{\mathcal{R}}_i[\![S_1]\!] \mathbin{\underline{\cup}} \big((\underline{\mathcal{R}}_i[\![S_2]\!] \mathbin{\underline{\bullet}} \overline{\mathcal{T}}[\![S_1]\!] \mathbin{\underline{\circ}} \underline{\mathcal{C}}[\![S_1]\!]) \mathbin{\underline{\boxminus}} \overline{\mathcal{R}}_w S_1\big)} \\
\underline{\mathcal{R}}_i[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] & = & \\
\multicolumn{3}{r}{\underline{\mathcal{R}}_r[\![C]\!] \mathbin{\underline{\cup}} (\underline{\mathcal{R}}_i[\![S_1]\!] \mathbin{\underline{\circ}} \underline{\mathcal{E}}_c[\![C]\!]) \mathbin{\underline{\cup}} (\underline{\mathcal{R}}_i[\![S_2]\!] \mathbin{\underline{\circ}} \underline{\mathcal{E}}_c[\![\texttt{.not.}C]\!]) \mathbin{\underline{\cup}} (\underline{\mathcal{R}}_i[\![S_1]\!] \mathbin{\underline{\cap}} \underline{\mathcal{R}}_i[\![S_2]\!])} \\
\underline{\mathcal{R}}_i[\![\texttt{do while}(C)\ S]\!] & = & \\
\multicolumn{3}{r}{\underline{\mathrm{lfp}}\big(\lambda f.\underline{\mathcal{R}}_i[\![C]\!] \mathbin{\underline{\cup}} \big\{\underline{\mathcal{R}}_i[\![S]\!] \mathbin{\underline{\cup}} \big((f \mathbin{\underline{\bullet}} \overline{\mathcal{T}}[\![S]\!] \mathbin{\underline{\circ}} \underline{\mathcal{C}}[\![S]\!]) \mathbin{\underline{\boxminus}} \overline{\mathcal{R}}_w[\![S]\!]\big) \mathbin{\underline{\circ}} \underline{\mathcal{E}}_c[\![C]\!]\big\}\big)}
\end{array}$$

Table E.7: Approximate IN regions

$$\mathcal{R}_o : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \wp(\mathbb{Z}^{d_i}))$$

$S_1; S_2$

$$\begin{array}{lcl}
\mathcal{R}_o[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!] & = & \mathcal{R}_o[\![S_1;S_2]\!] \cap \mathcal{R}_w[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!] \\
\mathcal{R}_o[\![S_1]\!] & = & \mathcal{R}_w[\![S_1]\!] \cap \big((\mathcal{R}_o[\![S_1;S_2]\!] \boxminus \mathcal{R}_o[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]) \cup \mathcal{R}_i[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]\big)
\end{array}$$

`if` C `then` S_1 `else` S_2

$$\begin{array}{lcl}
\mathcal{R}_o[\![S_1]\!] & = & (\mathcal{R}_o[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] \cap \mathcal{R}_w[\![S_1]\!]) \circ \mathcal{E}_c[\![C]\!] \\
\mathcal{R}_o[\![S_2]\!] & = & (\mathcal{R}_o[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] \cap \mathcal{R}_w[\![S_2]\!]) \circ \mathcal{E}_c[\![\texttt{.not.}C]\!]
\end{array}$$

`do while(`C`)` S

$$\begin{array}{r}
\mathcal{R}_o[\![S]\!]_k \circ \mathcal{E}_c[\![C]\!] \circ \big(\underbrace{(\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!]) \circ \ldots \circ (\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!])}_{k-1}\big) = \\
\mathcal{R}_w[\![S]\!] \circ \mathcal{E}_c[\![C]\!] \circ \big(\underbrace{(\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!]) \circ \ldots \circ (\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!])}_{k-1}\big) \cap \\
\Bigg((\mathcal{R}_o[\![\texttt{do while(}C\texttt{) } S]\!] \boxminus \mathcal{R}_w[\![\texttt{do while(}C\texttt{) } S]\!] \circ \\
\big(\underbrace{(\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!]) \circ \ldots \circ (\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!])}_{k}\big)) \\
\cup \mathcal{R}_i[\![\texttt{do while(}C\texttt{) } S]\!] \circ \big(\underbrace{(\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!]) \circ \ldots \circ (\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!])}_{k}\big)\Bigg)
\end{array}$$

Table E.8: Exact OUT regions

$$\overline{\mathcal{R}}_o : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \tilde{\wp}(\mathbb{Z}^{d_i}))$$

$S_1; S_2$

$$\begin{array}{lcl}
\overline{\mathcal{R}}_o[\![S_2]\!] & = & \overline{\mathcal{R}}_o[\![S_1;S_2]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}^{-1}[\![S_1]\!] \,\overline{\cap}\, \overline{\mathcal{R}}_w[\![S_2]\!] \\
\overline{\mathcal{R}}_o[\![S_1]\!] & = & \overline{\mathcal{R}}_w[\![S_1]\!] \,\overline{\cap}\, \big((\overline{\mathcal{R}}_o[\![S_1;S_2]\!] \,\overline{\boxminus}\, \underline{\mathcal{R}}_o[\![S_2]\!] \,\underline{\bullet}\, \overline{\mathcal{T}}[\![S_1]\!]) \,\overline{\cup}\, \overline{\mathcal{R}}_i[\![S_2]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}[\![S_1]\!] \big)
\end{array}$$

`if` C `then` S_1 `else` S_2

$$\begin{array}{lcl}
\overline{\mathcal{R}}_o[\![S_1]\!] & = & (\overline{\mathcal{R}}_o[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] \,\overline{\cap}\, \overline{\mathcal{R}}_w[\![S_1]\!]) \,\overline{\circ}\, \overline{\mathcal{E}}_c[\![C]\!] \\
\overline{\mathcal{R}}_o[\![S_2]\!] & = & (\overline{\mathcal{R}}_o[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] \,\overline{\cap}\, \overline{\mathcal{R}}_w[\![S_2]\!]) \,\overline{\circ}\, \overline{\mathcal{E}}_c[\![\texttt{.not.}C]\!]
\end{array}$$

`do while(`C`)` S

$$\begin{array}{lcl}
\overline{\mathcal{R}}_o[\![S]\!]_{\text{inv}} & = & \overline{\mathcal{R}}_w[\![S]\!] \,\overline{\cap}\, \Big((\overline{\mathcal{R}}_o[\![\texttt{do while(}C\texttt{) } S]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}^{-1}_{\text{inv}}[\![\texttt{do while(}C\texttt{) } S]\!] \,\overline{\circ}\, \overline{\mathcal{E}}_c[\![C]\!] \\
 & & \overline{\boxminus}\, \underline{\mathcal{R}}_w[\![\texttt{do while(}C\texttt{) } S]\!] \,\underline{\bullet}\, \overline{\mathcal{T}}[\![S]\!]) \,\overline{\cup}\, \overline{\mathcal{R}}_i[\![\texttt{do while(}C\texttt{) } S]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}[\![S]\!] \Big)
\end{array}$$

$$\underline{\mathcal{R}}_o : \boldsymbol{S} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \prod_{i=1,n} \tilde{\wp}(\mathbb{Z}^{d_i}))$$

$S_1; S_2$

$$\begin{array}{lcl}
\underline{\mathcal{R}}_o[\![S_2]\!] & = & \underline{\mathcal{R}}_o[\![S_1;S_2]\!] \,\underline{\bullet}\, \overline{\mathcal{T}}^{-1}[\![S_1]\!] \,\underline{\cap}\, \underline{\mathcal{R}}_w[\![S_2]\!] \\
\underline{\mathcal{R}}_o[\![S_1]\!] & = & \underline{\mathcal{R}}_w[\![S_1]\!] \,\underline{\cap}\, \big((\underline{\mathcal{R}}_o[\![S_1;S_2]\!] \,\underline{\boxminus}\, \overline{\mathcal{R}}_o[\![S_2]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}[\![S_1]\!]) \,\underline{\cup}\, \underline{\mathcal{R}}_i[\![S_2]\!] \,\underline{\bullet}\, \overline{\mathcal{T}}[\![S_1]\!] \big)
\end{array}$$

`if` C `then` S_1 `else` S_2

$$\begin{array}{lcl}
\underline{\mathcal{R}}_o[\![S_1]\!] & = & (\underline{\mathcal{R}}_o[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] \,\underline{\cap}\, \underline{\mathcal{R}}_w[\![S_1]\!]) \,\underline{\circ}\, \underline{\mathcal{E}}_c[\![C]\!] \\
\underline{\mathcal{R}}_o[\![S_2]\!] & = & (\underline{\mathcal{R}}_o[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] \,\underline{\cap}\, \underline{\mathcal{R}}_w[\![S_2]\!]) \,\underline{\circ}\, \underline{\mathcal{E}}_c[\![\texttt{.not.}C]\!]
\end{array}$$

`do while(`C`)` S

$$\begin{array}{lcl}
\underline{\mathcal{R}}_o[\![S]\!]_{\text{inv}} & = & \underline{\mathcal{R}}_w[\![S]\!] \,\underline{\cap}\, \Big((\underline{\mathcal{R}}_o[\![\texttt{do while(}C\texttt{) } S]\!] \,\underline{\bullet}\, \overline{\mathcal{T}}^{-1}_{\text{inv}}[\![\texttt{do while(}C\texttt{) } S]\!] \,\underline{\circ}\, \underline{\mathcal{E}}_c[\![C]\!] \\
 & & \underline{\boxminus}\, \overline{\mathcal{R}}_w[\![\texttt{do while(}C\texttt{) } S]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}[\![S]\!]) \,\underline{\cup}\, \underline{\mathcal{R}}_i[\![\texttt{do while(}C\texttt{) } S]\!] \,\underline{\bullet}\, \overline{\mathcal{T}}[\![S]\!] \Big)
\end{array}$$

Table E.9: Approximate OUT regions

Bibliographie
Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

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

[3] J. Allen and S. Johnson. Compiling C for vectorization, parallelization and inline expansion. In *International Conference on Programming Language Design and Implementation*, pages 241–249, June 1988.

[4] J. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4), October 1987.

[5] S. Amarasinghe, J. Anderson, C. Wilson, S.-W. Liao, R. French, and M. Lam. Multiprocessors from a software perspective. *IEEE Micro*, 16(3):52–61, June 1996.

[6] Saman Amarasinghe and Monica Lam. Communication optimization and code generation for distributed memory machines. In *International Conference on Programming Language Design and Implementation*, pages 126–138, June 1993. In ACM SIGPLAN Notices.

[7] Zahira Amarguellat. A control-flow normalization algorithm and its complexity. *IEEE Transactions on Software Engineering*, 18(3):237–251, March 1992.

[8] American National Standard Institute. *Programming Language FORTRAN, ANSI X3.9-1978, ISO 1539-1980*, 1983.

[9] Corinne Ancourt. *Génération de code pour multiprocesseurs à mémoires locales.* PhD thesis, Université Paris VI, mars 1991.

[10] Corinne Ancourt, Fabien Coelho, François Irigoin, and Ronan Keryell. A Linear Algebra Framework for Static HPF Code Distribution. In *Fourth International Workshop on Compilers for Parallel Computers*, December 1993. To appear in Scientific Programming. Available on `http://www.cri.ensmp.fr`.

[11] Corinne Ancourt, Fabien Coelho, François Irigoin, and Ronan Keryell. A Linear Algebra Framework for Static HPF Code Distribution. *Scientific Programming*, 1997. To appear.

[12] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *Symposium on Principles and Practice of Parallel Programming*, pages 39–50, April 1991.

[13] Corinne Ancourt and François Irigoin. Automatic code distribution. In *Third Workshop on Compilers for Parallel Computers*, July 1992.

[14] Jennifer Anderson, Saman Amarasinghe, and Monica Lam. Data and computation transformations for multiprocessors. In *Symposium on Principles and Practice of Parallel Programming*, July 1995.

[15] Applied-Parallel Research. *The Forge User's Guide, Version* 8.0, 1992.

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

[17] Béatrice Apvrille-Creusillet. Régions exactes et privatisation de tableaux (Exact array region analyses and array privatization). Master's thesis, Université Paris VI, France, September 1994. Available via `http://www.cri.ensmp.fr/~creusil`.

[18] Béatrice Apvrille-Creusillet. Calcul de régions de tableaux exactes. *TSI, Numéro spécial RenPar'6*, 14(5):585–600, mai 1995.

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

[20] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston / Dordrecht / London, 1988.

[21] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David A. Padua. Automatic program parallelisation. *Proceedings of the IEEE*, 81(2), February 1993.

[22] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 1981.

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

[24] Denis Barthou, Jean-François Collard, and Paul Feautrier. Applications of fuzzy array data-flow analysis. In *Europar'96*, August 1996.

[25] M. Berry, D. Chen, P. Koss, D. Kuck, V. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orzag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin. The PERFECT Club benchmarks : Effective performance evaluation of supercomputers. Technical Report CSRD-827, CSRD, University of Illinois, May 1989.

[26] David Binkley. Interprocedural constant propagation using dependence graphs and a data-flow model. In *International Conference on Compiler Construction*, April 1994.

[27] Garrett Birkhoff. *Lattice Theory*, volume XXV of *AMS Colloqium Publications*. American Mathematical Society, Providence, Rhode Island, third edition, 1967.

[28] Bill Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris : The next generation in parallelizing compilers. CSRD report 1375, CSRD, University of Illinois, 1994.

[29] W. Blume and R. Eigenmann. Performance analysis of parallelizing compilers on the Perfect Benchmarks programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, November 1992.

[30] William Blume and Rudolf Eigenmann. The range test: A dependence test for symbolic, non-linear expressions. In *International Conference on Supercomputing*, pages 528–537, November 1994.

[31] F. Bodin, C. Eisenbeis, W. Jalby, T. Montaut, P. Rabain, and D. Windheiser. Algorithms for data locality optimization. Deliverable CoD3, APPARC, 1994.

[32] François Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 2(4):407–435, 1992.

[33] François Bourdoncle. *Sémantique des Langages Impératifs d'Ordre Supérieur et Interprétation Abstraite*. PhD thesis, École Polytechnique, November 1992.

[34] Thomas Brandes. The importance of direct dependences for automatic parallelization. In *International Conference on Supercomputing*, pages 407–417, July 1988.

[35] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. *ACM SIGPLAN Notices*, 21(7):162–175, July 1986.

[36] D. Callahan. The program summary graph and flow-sensitive interprocedural data-flow analysis. *ACM SIGPLAN Notices*, 23(7):47–56, July 1988. Also published in the proceedings of the International Conference on Programming Language Design and Implementation.

[37] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5:517–550, 1988.

[38] Pierre-Yves Calland, Alain Darte, Yves Robert, and Frédéric Vivien. On the removal of anti and output dependences. Rapport de Recherche 2800, INRIA, February 1996.

[39] Pierre-Yves Calland, Alain Darte, Yves Robert, and Frédéric Vivien. Plugging anti and output dependence removal techniques into loop parallelization algorithms. Rapport de Recherche 2914, INRIA, June 1996.

[40] Philippe Chassany. Les méthodes de parallélisation interprocédurales. Master's thesis, Université Paris VI, Septembre 1990. DEA Systèmes Informatiques. Rapport interne CRI-ENSMP EMP-CAI-I E/129.

[41] J Choi, R. Cytron, and J Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Symposium on Principles of Programming Languages*, pages 55–66, January 1991.

[42] Michał Cierniak and Wei Li. Recovering logical structures of data. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 362–376. Springer-Verlag, August 1995.

[43] Ph. Clauss and V. Loechner. Parametric analysis of polyhedral iteration spaces. In *International Conference on Application Specific Array Processors*, August 1996.

[44] Philippe Clauss. Couting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyse and transform scientific programs. In *International Conference on Supercomputing*, May 1996.

[45] Fabien Coelho. Experiments with HPF Compilation for a Network of Workstations. In *High Performance Computing and Networking Conference*, pages 423–428, April 1994.

[46] Fabien Coelho. Compilation of I/O communications for HPF. In *Frontiers'95*, pages 102–109, February 1995. Available via `http://www.cri.ensmp.fr/~coelho`.

[47] Fabien Coelho and Corinne Ancourt. Optimal compilation of HPF remappings. Technical Report A-277-CRI, CRI, École des Mines de Paris, October 1995. To appear in JPDC in 1996.

[48] Fabien Coelho and Corinne Ancourt. Optimal Compilation of HPF Remappings. CRI TR A 277, CRI, École des Mines de Paris, October 1995. To appear in JPDC, 1996.

[49] Jean-François Collard. Space-time transformation of while-loops using speculative execution. In *Scalable High Performance Computing Conference*, pages 429–436, May 1994.

[50] Jean-François Collard. Automatic parallelization of while-loops using speculative execution. *International Journal of Parallel Programming*, 23(2):191–219, 1995.

[51] Jean-François Collard, Denis Barthou, and Paul Feautrier. Fuzzy array dataflow analysis. In *Symposium on Principles and Practice of Parallel Programming*, July 1995.

[52] Jean-François Collard and Martin Griebl. Array dataflow analysis for explicitely parallel programs. In *Europar'96*, August 1996.

[53] K. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *IEEE International Conference on Computer Language*, April 1992.

[54] Keith Cooper, Ken Kennedy, and Nathaniel McIntosh. Cross-loop reuse analysis and its application to cache optimizations. In *Languages and Compilers for Parallel Computing*, August 1996.

[55] Patrick Cousot. *Méthodes Itératives de Construction et d'Approximation de Points Fixes d'Opérateurs Monotones sur un Treillis, Analyse Sémantique des Programmes.* PhD thesis, Institut National Polytechnique de Grenoble, March 1978.

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

[57] Patrick Cousot and Radhia Cousot. Abstract interpretation framework. *Journal of Logic and Computation*, 2(5):511–547, August 1992.

[58] Patrick Cousot and Radhia Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages). In *International Conference on Computer Languages, IEEE Computer Socitey Press*, pages 95–112, May 1994.

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

[60] Béatrice Creusillet. Analyse de flot de données : Régions de tableaux IN et OUT. In *Rencontres Francophones du Parallélisme*, mai-juin 1995.

[61] Béatrice Creusillet. Array regions for interprocedural parallelization and array privatization. Report A-279, CRI, École des Mines de Paris, November 1995. Available at `http://www.cri.ensmp.fr/~creusil`.

[62] Béatrice Creusillet. IN and OUT array region analyses. In *Fifth International Workshop on Compilers for Parallel Computers*, pages 233–246, June 1995.

[63] Béatrice Creusillet and François Irigoin. Interprocedural array region analyses. In *Languages and Compilers for Parallel Computing*, number 1033 in Lecture Notes in Computer Science, pages 46–60. Springer-Verlag, August 1995.

[64] Béatrice Creusillet and François Irigoin. Exact vs. approximate array region analyses. In *Languages and Compilers for Parallel Computing*, August 1996.

[65] Béatrice Creusillet and François Irigoin. Interprocedural array region analyses. *International Journal of Parallel Programming (special issue on LCPC)*, 24(6):513–546, 1996. Extended version of [63].

[66] R. Cytron and J. Ferrante. What's in a name? or the value of renaming for parallelism detection and storage allocation. In *International Conference on Parallel Processing*, pages 19–27, August 1987.

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

[68] Alain Dumay. *Traitement des indexations non linéaires en parallélisation automatique : une méthode de linéarisation contextuelle.* PhD thesis, Université Paris VI, December 1992.

[69] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four perfect-benchmark programs. In *Languages and Compilers for Parallel Computing*, number 589 in Lecture Notes in Computer Science, pages 65–83. Springer-Verlag, August 1991.

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

[71] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, pages 243–268, September 1988.

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

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

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

[75] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[76] High Performance Fortran Forum. *High Performance Fortran Language Specification.* Rice University, Houston, Texas, November 1994. *Version 1.1.*

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

[78] Geoffrey C. Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Ulrich Kremer, Chau-Wen Tseng, and M. Wu. The Fortran D Language Specification. TR 90079, CRPC, December 1990.

[79] Kyle Gallivan, William Jalby, and Dennis Gannon. On the problem of optimizing data transfers for complex memory systems. In *International Conference on Supercomputing*, pages 238–253, July 1988.

[80] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations. *Journal of Parallel and Distributed Computing*, 5(5):587–616, October 1988.

[81] R.E. Gomory. Solving linear programming problems in integer. In *Combinatorial Analysis (proceedings of Symposia in Applied Mathematics)*, pages 211–215, 1960.

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

[83] Susan Graham, Steven Lucco, and Oliver Sharp. Orchestrating interactions among parallel computations. In *International Conference on Programming Language Design and Implementation*, pages 100–111, June 1993. ACM SIGPLAN Notices.

[84] Philippe Granger. Improving the results of static analyses of programs by local decreasing iterations. Research Report LIX/RR/91/08, École Polytechnique, Laboratoire d'Informatique, December 1991.

[85] Elana D. Granston and Alexander V. Veidenbaum. Combining flow and dependence analysis to expose redundant array access. Technical report, Leiden University, October 1992.

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

[87] Jungie Gu, Zhiyuan Li, and Gyungho Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Supercomputing*, December 1995.

[88] C. Gunter and D. Scott. Denotational semantics. In Jan van Leeuwen, editor, *Theoretical Computer Science*, volume B, chapter 12. Elsevier Science Publisher, 1990.

[89] Mohammad R. Haghighat. *Symbolic Analysis for Parallelizing Compilers*. Kluwer Academic Publishers, 1995.

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

[91] Mary Hall. *Managing Interprocedural Optimization*. PhD thesis, Rice University, Houston, Texas, April 1991.

[92] Mary Hall, Saman Amarasinghe, Brian Murphy, Shih-Wei Liao, and Monica Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Supercomputing*, December 1995.

[93] Mary Hall, Brian Murphy, Saman Amarasinghe, Shih-Wei Liao, and Monica Lam. Interprocedural analysis for parallelization. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 61–80. Springer-Verlag, August 1995.

[94] Mary Hall, Brian Murphy, Saman Amarasinghe, Shih-Wei Liao, and Monica Lam. Overview of an interprocedural automatic parallelization system. In *Fifth International Workshop on Compilers for Parallel Computers*, pages 570–579, June 1995.

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

[96] Mary Jean Harrold and Brian Malloy. A unified interprocedural program representation for a maintenance environment. *IEEE Transactions on Software Engineering*, 19(6):584–593, June 1993.

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

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

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

[100] M. Hind, M. Burke, P. Carini, and S. Midkiff. Interprocedural array analysis : How much precision do we need ? In *Third Workshop on Compilers for Parallel Computers*, pages 48–64, July 1992.

[101] Michael Hind, Michael Burke, Paul Carini, and Sam Midkiff. An empirical study of precise interprocedural array analysis. *Scientific Programming*, 3(3):255–271, May 1994.

[102] C. Hoare. An axiomatic basis of computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[103] François Irigoin. Interprocedural analyses for programming environments. In *Workshop on Environments and Tools for Parallel Scientific Computing*, pages 333–350, September 1992.

[104] François Irigoin and Pierre Jouvelot. Projet PIPS, analyseur sémantique, rapport de synthèse. Document EMP–CAI–I E–137, CRI, École des Mines de Paris, décembre 1990. contrat DRET No. 88.017.01.

[105] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *International Conference on Supercomputing*, pages 144–151, June 1991.

[106] François Irigoin, Pierre Jouvelot, Rémi Triolet, Arnauld Leservot, Alexis Platonoff, Ronan Keryell, Corinne Ancourt, Fabien Coelho, and Béatrice Creusillet. *PIPS: Development Environnement.* CRI, École des Mines de Paris, 1996.

[107] Chung-Chi Jim Li, Elliot M. Stewart, and W. Kent Fuchs. Compiler-assisted full checkpointing. *Software : Practice and Experience*, 24(10):871–886, October 1994.

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

[109] Pierre Jouvelot and Rémi Triolet. *NewGen User Manual.* CRI, École des Mines de Paris, December 1990.

[110] A. Kallis and D. Klappholz. Extending conventionnal flow analysis to deal with array references. In *Languages and Compilers for Parallel Computing*, number 589 in Lecture Notes in Computer Science, pages 251–265. Springer-Verlag, August 1991.

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

[112] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The Omega library interface guide. Technical report CS-TR-3445, University of Maryland, College Park, March 1995.

[113] Ken Kennedy. A survey of data flow analysis techniques. In S. Muchnick and Edts. N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 5–54. Prentice-Hall, Inc., 1979.

[114] Ken Kennedy and Kathrin McKinley. Optimizing for parallelism and data locality. In *International Conference on Supercomputing*, pages 323–334, July 1992.

[115] Ronan Keryell. *WPips and EPips User Manual*. CRI, École des Mines de Paris, April 1996. Available from `http://www.cri.ensmp.fr/pips/` and as report A/288.

[116] Ronan Keryell, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, François Irigoin, and Pierre Jouvelot. PIPS: A Framework for Building Interprocedural Compilers, Parallelizers and Optimizers. Technical Report 289, CRI, École des Mines de Paris, April 1996.

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

[118] Kuck & associates, Inc. *KAP Users's Guide*, 1988.

[119] W. Landi and B. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *International Conference on Programming Language Design and Implementation*, 1992.

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

[121] Arnauld Leservot. *Analyses interprocédurales du flot des données*. PhD thesis, Université Paris VI, March 1996.

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

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

[124] C. Livercy. *Théorie des programmes*. Dunod, 1978.

[125] Muniyappa Manjunathaiah and Denis Nicole. Accurately representing the union of convex sections. *Scientific Programming*, ?(?), 1996. Special issue on languages, compilers and run-time systems for scalable computers.

[126] Thomas Marlowe, Barbara Ryder, and Michael Burke. Defining flow sensitivity in data flow problems. Research Technical Report lcsr-tr-249, Rutgers University, Laboratory of Computer Science, July 1995.

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

[128] Vadim Maslov and William Pugh. Simplifying polynomial constraints over integers to make dependence analysis more precise. Technical Report CS-TR-3109.1, University of Maryland, College Park, February 1994.

[129] Dror Maydan, John Hennessy, and Monica Lam. Efficient and exact dependence analysis. In *International Conference on Programming Language Design and Implementation*, June 1991.

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

[131] Brian Mayoh. Attribute grammars and mathematical semantics. *SIAM Journal on Computing*, 10(3):503–518, August 1981.

[132] Peter Mosses. Denotational semantics. In Jan van Leeuwen, editor, *Theoretical Computer Science*, volume B, chapter 11. Elsevier Science Publisher, 1990.

[133] Steven Muchnick and Neil Jones. *Program Flow Analysis (Theory and Applications)*. Prentice-Hall, Inc., 1981.

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

[135] Trung Nguyen, Jungie Gu, and Zhiyuan Li. An interprocedural parallelizing compiler and its support for memory hierarchy research. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 96–110. Springer-Verlag, August 1995.

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

[137] Michael Paleczny, Ken Kennedy, and Charles Koelbel. Compiler support for out-of-core arrays on parallel machines. In *Frontiers'95*, pages 110–118, February 1995.

[138] Karen L. Pieper. *Parallelizing Compilers: Implementation and Effectiveness*. PhD thesis, Stanford University, Computer Systems Laboratory, June 1993.

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

[140] Alexis Platonoff. Automatic data distribution for massively parallel computers. In *Fifth International Workshop on Compilers for Parallel Computers*, pages 555–570, June 1995.

[141] Alexis Platonoff. *Contribution à la Distribution Automatique de Données pour Machines Massivement Parallèles.* PhD thesis, Université Paris VI, March 1995.

[142] C. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten. Paraphrase-2: An environment for parallelizing, partitionning, synchronizing and scheduling programs on multiprocessors. In *International Conference on Parallel Processing*, August 1989.

[143] William Pugh. Uniform techniques for loop optimizations. In *International Conference on Supercomputing*, January 1991.

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

[145] William Pugh. Counting solutions to Presburger formulas: How and why. In *International Conference on Programming Language Design and Implementation*, pages 121–134, June 1994. In ACM SIGPLAN Notices.

[146] William Pugh and David Wonnacott. Eliminating false data dependences using the Omega test. In *International Conference on Programming Language Design and Implementation*, pages 140–151, June 1992.

[147] William Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences. In *Languages and Compilers for Parallel Computing*, August 1993.

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

[149] Lawrence Rauchwerger and David Padua. The LRPD test : Speculative run-time parallelization of loops with privatization and reduction parallelization. In *International Conference on Programming Language Design and Implementation*, June 1995.

[150] Xavier Redon. *Détection et exploitation des récurrences dans les programes scientifiques en vue de leur parallélisation.* PhD thesis, Université Paris VI, January 1995.

[151] Carl Rosene. *Incremental Dependence analysis.* PhD thesis, Rice University, Houston, Texas, March 1990.

[152] B. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3):216–225, May 1979.

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

[154] Vivek Sarkar. PTRAN — the IBM parallel translation system. Research report RC 16194, IBM, Research Division, June 1990.

[155] Vivek Sarkar and Guang Gao. Optimization of array accesses by collective loop transformations. In *International Conference on Supercomputing*, pages 194–205, June 1991.

[156] Micha Sharir and Amir Pnuelli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis*, pages 189–233. Prentice-Hall, Inc., 1981.

[157] David Smith. *Denotational Semantics.* Allyn and Bacon, Inc., 1953.

[158] Eric Stoltz, Michael Gelerk, and Michael Wolfe. Extended SSA with factored use-def chains to support optimization and parallelism. In *International Conference on System Sciences*, pages 43–52, 1994.

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

[160] Thinking Machine Corporation, Cambridge, Massachussetts. *CM Fortran Programming Guide*, January 1991.

[161] Rémi Triolet. *Contribution à la parallélisation automatique de programmes Fortran comportant des appels de procédures.* PhD thesis, Paris VI University, 1984.

[162] Rémi Triolet. Interprocedural analysis for program restructuring with Parafrase. Technical report 538, CSRD, University of Illinois, December 1985.

[163] Rémi Triolet, Paul Feautrier, and François Irigoin. Direct parallelization of call statements. In *ACM SIGPLAN Symposium on Compiler Construction*, pages 176–185, 1986.

[164] Peng Tu. *Automatic Array Privatization and Demand-Driven Symbolic Analysis.* PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[165] Peng Tu and David Padua. Array privatization for shared and distributed memory machines (extended abstract). In *Workshop on Languages and Compilers for Distributed Memory Machines*, pages 64–67, 1992.

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

[167] Y. Tuchman, N. Sack, and Z. Barkat. Mass loss from dynamically unstable stellar envelopes. *Astrophysical Journal*, 219:183–194, 1978.

[168] Tia M. Watts, Mary Lou Soffa, and Rajiv Gupta. Techniques for integrating parallelizing transformations and compiler based scheduling methods. In *International Conference on Supercomputing*, pages 830–839, 1992.

[169] Doran K. Wilde. A library for doing polyhedral operations. Publication Interne 785, IRISA, December 1993.

[170] Pugh William and David Wonnacot. Nonlinear array dependence analysis. Technical Report CS-TR-3372, University of Maryland, College Park, November 1994.

[171] Pugh William and David Wonnacot. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Transactions on Programming Languages and Systems*, 16(4):1248–1278, July 1994.

[172] Michael E. Wolf and Monica Lam. A data locality optimizing algorithm. In *International Conference on Programming Language Design and Implementation*, pages 30–44, June 1991. In ACM SIGPLAN Notices.

[173] Michael Wolfe. Scalar vs. parallel optimizations. Technical report, Oregon Graduate Institute of Science and Technology, 1990.

[174] Michael Wolfe. Beyond induction variables. In *International Conference on Programming Language Design and Implementation*, pages 162–174, June 1992.

[175] Michael Wolfe and Chau-Wen Tseng. The power test for data dependence. In *IEEE Transactions on Parallel and Distributed Systems*, September 1992.

[176] David Wonnacott. *Constraint-Based Array Dependence Analysis.* PhD thesis, University of Maryland, College Park, August 1995.

[177] Yi-Qing Yang. *Tests des Dépendances et Transformations de Programme.* PhD thesis, Université Paris VI, Novembre 1993.

[178] Yi-Qing Yang, Corinne Ancourt, and François Irigoin. Minimal data dependence abstractions for loop transformations. In *Languages and Compilers for Parallel Computing*, July 1994.

[179] Lei Zhou. *Statical and Dynamical Analysis of Program Complexity.* PhD thesis, Université Paris VI, September 1994.

[180] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1991.

[181] Hans Zima, Peter Brezany, Barbara Chapman, Piyush Mehrotra, and Andreas Schwald. Vienna Fortran - A Language Specification. ftp `cs.rice.edu`, 1992. Version 1.1.