INSTITUT SUPERIEUR INFORMATIQUE ISI
المعهد العالي للإعلامية
UNIVERSITÉ TUNIS EL MANAR
INSTITUT SUPÉRIEUR D'INFORMATIQUE

MINES ParisTech
MATHEMATIQUES ET SYSTEMES
CENTRE DE RECHERCHE EN INFOMATIQUE

E/302/CRI
ISI/MGL/01/09

MÉMOIRE DE MASTERE

Présenté en vue de l'obtention du
Diplôme de mastère de recherche Génie Logiciel

Par

Amira MENSI
INGÉNIEUR EN INFORMATIQUE, ISI

Pointeurs et aliasing en C
Une étude bibliographique et expérimentale

Soutenu le 06 Avril 2009, devant le jury d'examen

MM.	**Samir BEN AHMED**	**Président**
	Ezzeddine ZAGROUBA	**Membre**
	François IRIGOIN	**Directeur de mémoire**
	Moncef TEMANI	**Co-directeur du mémoire**

Année universitaire : 2008/2009

Remerciements

Au terme de ce stage, je tiens à adresser mes remerciements les plus sincères au professeur Robert Mahl, Directeur du Centre de Recherche en Informatique de l'école des Mines de Paris, pour m'avoir accueillie au sein de son centre.

Je remercie aussi monsieur François Irigoin Directeur-Adjoint du Centre de Recherche en Informatique de l'école des Mines de Paris, qui m'a accueillie au sein de son équipe de recherche, m'a encadré tout au long de ce stage. Il m'a consacré énormément de son temps et n'a cessé de me conseiller et de me faire profiter de son expérience et de son savoir faire.

Je veux remercier particulièrement Pierre Jouvelot pour ses conseils judicieux, les conversations enrichissantes qu'on a eu. Il m'a conseillé tout au long de ce stage et m'a apporté son aide précieuse pendant la préparation du rapport et les présentations.

Je veux remercier également le professeur Moncef Temani, directeur de l'Institut Supérieur de l'Informatique pour la formation et l' encadrement dont j'ai bénéficiés au sein de son institut. Je le remercie aussi pour m'avoir encadré tout au long de ce stage. Il l'a dirigé, m'a conseillé et m'a apporté son aide pendant la rédaction de ce rapport.

Je souhaite exprimer ma gratitude à madame Corinne Ancourt Enseignante-chercheur au Centre de Recherche en Informatique de l'Ecole des Mines de Paris, pour sa disponibilité.

Je veux remercier toutes les personnes du CRI, qui ont contribué de près ou de loin au bon déroulement de ce travail par leurs encouragements, en particulier Jacqueline Altimira pour son aide amicale.

Je remercie également les membres du jury : monsieur Samir Ben Ahmed directeur de l'Institut National des Sciences Appliquées et de Technologie, ainsi que monsieur Ezzedine Zabrouba professeur et directeur des études à l'Institut Supérieur d'Informatique.

Dédicaces

Je dédie ce mémoire à ceux que j'ai de plus cher, qui ont partagé mes joies et effacé mes peines, à mes très chers parents.

Je dédie ce mémoire à ceux qui ont toujours été à mes côtés ma sœur Emna et mon frére Mehdi.

Je dédie ce mémoire à ceux qui malgré la distance m'ont toujours accordé leur tendresse et soutien ; mes chers amies : Safa Ben Braik, Salima Landoulsi.

Je dédie ce mémoire à mes amis avec qui j'ai partagé des moments des plus agréables : Brice Hoffmann, Gilles Pelfrene, Xavier Courtial et particulièrement Estelle Bonnaud.

Abstract

Static analysis programs strive to extract the information necessary for the understanding and optimization of programs at compile time. The potential values of the variables of type pointer are the most difficult information's to determine. This information is used to determine if two pointers are potential aliases, i.e. if they can point to the same memory area. An analysis of pointers may provide more precision to other analysis such as constant propagation, analysis of dependencies or the analysis of live variables. The analysis of pointers is very important for the exploitation of parallelism in C programs since the most important structure is arrays, which are accessed by pointer. It is necessary to analyse the dependencies between arrays in order to exploit the parallelism between loops. Analysis of pointers can also handle recursive data structure and other structures that are accessed by pointer. This work provides litterature review analysis pointers for the C language. These analysis are grouped into two types :

- Flow-insensitive analysis.
- Flow-sensitive analysis.

For the first type of analysis we introduce Steensgaard's algorithm and Andersen's algorithm, which is incorporated into the CLA architecture (compile-link-analysis) and the gcc compiler. Both algorithms are used to extract information on pointers without taking into account the order of execution of instructions. Although they are effective in terms of execution time, the outcome remains less precise in comparison with other analysis. For the second type of analysis, we introduced Wilson's algorithm which was integrated in the SUIF compiler. This algorithm uses Partial Transfer Functions to describe procedures given conditions in their call. We also studied Emami's analysis which is integrated in the McCAT compiler. This analysis can take into account the context of procedures using a special data structure, which is the graph of invocation. This data structure captures the activation order of the functions' calls of a given program. This literature review also includes an analysis of the needs of signal processing applications to determine the structures of the C language, accessible by pointer, which are the most recurrent. Eventually, this study will guide the choice of the analysis of pointers that we will be implemented at PIPS (parallélisation interprocédurale de programmes scientifiques), a framework which deals with applications of signal processing in order to analyze, to optimize and parallelize them.
KeyWords: static analysis, flow-sensitive analysis, pointers analysis, aliasing.

Résumé

L'analyse statique des programmes permet d'extraire toutes les informations nécessaires à la compréhension ainsi qu'à l'optimisation des programmes au moment de la compilation. Une information des plus difficiles à déterminer concerne les valeurs potentielles des variables de type pointeur. Cette information est utilisée afin de déterminer si deux pointeurs sont potentiellement aliasés; c'est à dire s'ils peuvent pointer vers la même zone mémoire. Cette information est fournie par l'analyse des pointeurs qui apporte beaucoup de précision à d'autres analyses comme la propagation de constante, l'analyse des dépendances ou bien l'analyse des variables vivantes. L'analyse des pointeurs est très importante pour l'exploitation du parallélisme dans les programmes C dont la structure la plus importante est le tableau, parce qu'il est fréquent que des pointeurs soient utilisés pour y accéder. Elle est nécessaire pour l'analyse des dépendances entre tableaux requise pour exploiter le parallélisme entre les boucles. L'analyse des pointeurs permet aussi de traiter les structures de données récursives et les autres structures qui sont accessibles par pointeur. Ce travail consiste en une étude bibliographique des analyses des pointeurs pour le langage C. Ces analyses sont regroupées en deux types :

- les analyses insensibles au contexte et au flot de données,
- les analyses sensibles au contexte et au flot de données.

Pour le premier type d'analyses, on a introduit l'algorithme de Steensgaard ainsi que l'algorithme d'Andersen intégré dans l'architecture CLA (compile-link-analysis) et dans le compilateur gcc. Ces deux algorithmes permettent d'extraire des informations sur les pointeurs sans prendre en compte l'ordre d'exécution des instructions. Bien qu'elles soient efficaces en terme de temps d'exécution, leur résultat reste néanmoins peu précis en comparaison avec les autres analyses. Pour le deuxième type d'analyses, on a introduit l'algorithme de Wilson, qui a été fondé au sein du compilateur SUIF. Cet algorithme utilise les fonctions de transfert partielles afin de décrire le comportement des procédures, sous certaines conditions dans leur contexte d'appel. Dans le cadre des analyses sensibles au contexte et au flot de données, on a aussi étudié l'analyse d'Emami qui est intégrée au sein du compilateur McCAT. Cette analyse permet de prendre en compte le contexte des procédures via une structure de données propre à elle, qui est le graphe d'invocation. Cette structure de données capture l'ordre d'activation des appels des fonctions d'un programme donné. Cette étude bibliographique comporte aussi une analyse des besoins des applications de traitement de signal, afin de déterminer les structures du langage C, accessibles par pointeur, qui sont les plus récurrentes. En effet, cette étude va permettre d'orienter le choix de l'analyse des pointeurs qu'on voudrait implémenter au niveau de PIPS (parallélisation interprocédurale de programmes scientifiques), qui traite des applications de traitement de signal, dans le but de les analyser, de les optimiser et de les paralléliser.
Mots-clés :analyse statique, analyses sensible aux flots de données, analyse de pointeurs, alias.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Contexte

L'analyse de flots de données est un ensemble de techniques permettant d'extraire des informations sur l'évolution des données à travers plusieurs chemins d'exécution d'un programme. Par exemple, si le résultat d'une affectation n'est utilisé sur aucun des chemins d'exécution, alors l'affectation peut être considérée comme du code mort et peut être éliminée. Quand on veut analyser le comportement d'un programme, on doit considérer tous ses chemins d'exécution à travers le graphe de flot de données et extraire par la suite les données adéquates pour le problème d'analyse qu'on veut résoudre. Supposons qu'on s'intéresse à la propagation de constantes. Si l'utilisation d'une variable x est atteinte par une seule définition et que cette définition affecte une constante à x, on pourrait remplacer simplement x par cette constante. Mais si plusieurs définitions de x peuvent atteindre son utilisation, on ne pourrait pas effectuer une propagation de constante. Ainsi, pour la propagation de constante, on voudrait déterminer ces définitions qui sont l'unique définition de leur variable et qui atteignent un point programme donné, indépendamment du chemin d'exécution choisi. Une tâche rendue encore plus difficile par la présence de pointeurs qui nécessite une analyse de pointeurs précise. Le but de l'analyse des pointeurs est d'analyser statiquement un programme en entrée, et de fournir en résultat une approximation des valeurs de ses variables pointeurs, plus précisément les alias entre eux. Cette analyse est considérée comme l'une des analyse de programmes les plus délicates et l'information qu'elle apporte est très précieuse pour un grand nombre d'autres analyses. En effet, elle augmente l'efficacité des optimisations au niveau des compilateurs, comme la propagation de constante et conditionne la parallélisation automatique des programmes. Plusieurs travaux ont été élaborés dans ce domaine qui ne cesse de progresser, allant des analyses rapides mais insensibles au flot de données comme celles d'Andersen, de Steensgaard, de Burke aux analyses plus précises comme celles introduite par Emami ou Wilson bien implémentée au niveau du compilateur SUIF. On s'est intéressé particulièrement aux analyses sensibles au flot de données parce que les analyses de PIPS sont interprocédurales.

1.2 Motivations

Le langage C utilise les pointeurs, ce qui rend les tâches d'analyse difficiles en absence de toute information sur les valeurs de ces pointeurs. Sans cette information, on est toujours obligé d'émettre l'hypothèse du plus mauvais cas et de générer en conséquence une approximation conservatrice. Mais en profitant d'une information plus précise, les compilateurs C pourraient exploiter davantage du parallélisme dans les programmes, et même d'autres outils comme les debuggeurs pour détecter les erreurs, par exemple une mauvaise allocation mémoire ou l'utilisation de données non initialisées. Les outils de mesure de performance pourraient aussi tirer avantage de cette information afin de fournir de meilleurs résultats, ainsi que les outils de «slicing»[5].

L'information sur les pointeurs est très importante lors de l'exploitation du parallélisme dans les programmes C dont la structure la plus importante est les tableaux, surtout que, dans les programmes C, on utilise les pointeurs pour accéder aux tableaux. L'analyse des pointeurs est nécessaire pour l'analyse des dépendances de tableaux où l'on peut exploiter le parallélisme entre les boucles. Outre les tableaux, l'analyse des pointeurs permet aussi de traiter les structures de données récursives et les autres structures qui sont accessibles par pointeur.

On sait que la présence de pointeurs rend l'analyse de flot de données plus compliquée. Et que sans une information précise, on est dans l'obligation d'émettre l'hypothèse du pire des cas concernant les pointeurs, qui est de supposer que la variable pointeur peut pointer vers n'importe quelle autre variable ou zone mémoire dans le programme. Plusieurs analyses peuvent profiter de la présence d'une information précise sur les pointeurs, parmi elles l'analyse des variables vivantes, l'analyse des dépendances de données, la vérification des bornes des tableaux, la propagation de constante, l'élimination du code mort et bien d'autres.

Une des analyses qu'on peut améliorer avec l'analyse des pointeurs est la propagation de constantes qui permet de remplacer les expressions qui ont été évaluées à une constante chaque fois qu'elles sont rencontrées par la valeur de cette constante. Le problème se présente sous la forme d'une analyse de flot de données en avant. Prenons comme exemple le programme suivant et l'information sur les constantes dans le cas où l'information sur les alias (deux pointeurs pointant vers une même zone mémoire) est disponible et dans le cas où elle est inexistante.

```
int *a, b, c, d ;
c = 3;          /* S1 */
a = &c;         /* S2 */
read(d);        /* S3 */
b= *a;          /* S4 */
```

	Constantes sans information sur alias	Constantes avec information sur alias
S1 :	{ c=3 }	{ c=3 }
S2 :	{ c=3 }	{ c=3 }
S3 :	{ c=3, d = ? }	{ c=3, d = ? }
S4 :	{ c=3, d = ?, b = ? }	{ c=3, d = ?, b = 3}

Considérons l'instruction au niveau de S1; la variable c reçoit la valeur de 3. Au niveau de S2, les deux variables *a et c forment une paire d'alias. Puis, avec l'instruction au niveau de S3, la

variable d reçoit une valeur qui lui ait affectée par la fonction read(). C'est l'instruction au niveau de S4 qui démontre l'importance d'avoir une information sur les alias dans un programme : sans cette information, on considère le pire des cas et on émet l'hypothèse que la variable a peut pointer vers n'importe quelle des variables b, c ou d. Mais dans le cas où on dispose de cette information, on peut utiliser le fait que *a est identique à c et en déduire que b vaut 3. Cette information aide à optimiser le code et permet de remplacer la variable *a par 3.

1.3 Problématique

Le problème d'aliasing apparaît lorsqu'on a deux variables, comme deux expressions calculant une adresse, qui pointent vers le même espace mémoire; les deux variables sont considérées comme une paire d'alias. Ce problème peut inhiber les optimisations qu'on veut effectuer sur le code comme l'élimination du code mort ou la parallélisation des boucles contenant des alias. Ou supposons qu'on est en présence d'un programme contenant des millions de lignes de code et qu'on voudrait changer le type d'un ensemble de variables de type1 à type2; pour assurer la consistance du type, on est dans l'obligation de déterminer tous les objets dont le type devrait être changé. Plus concrètement supposons que la taille des valeurs stockées dans une variable doit être étendue à une taille plus grande, par exemple de short vers int; pour ne pas perdre les données on doit déterminer toutes les variables qui récupèrent leurs contenus de la variable dont le type a été changé.

En effet, quand on a deux pointeurs p et q pointant vers le même espace mémoire comme le montre la figure 1.1 , si on veut déterminer vers quels objets p et q pointent, il faut mettre en oeuvre une analyse de pointeurs «points-to analysis», ce qui se fait dans le cadre de l'analyse statique des programmes.

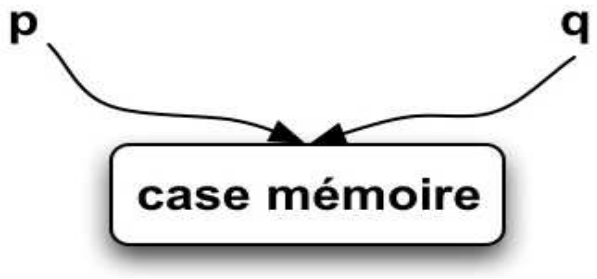


Figure 1.1: Deux pointeurs vers la même zone mémoire

Cette analyse permet l'optimisation des programmes et sert comme entrée pour d'autres analyses comme l'analyse des objets modifiés/référencés dans chaque noeud du graphe de flot de contrôle, l'analyse des variables vivantes, l'analyse des «reaching-definition» et la propagation interprocédurale de constante. Plusieurs analyses ont été mises en oeuvre pour traiter ce problème; elles différent dans les facteurs qu'elles prennent en compte. Tandis que certaines de ces analyses ne prennent pas en considération le flot de données ni le contexte des procédures, privilégiant ainsi la rapidité et l'efficacité des résultats, d'autres analyses prennent en compte ces facteurs en offrant ainsi des résultats plus précis. Dans la suite nous allons introduire ces deux types d'analyses en

mettant d'abord l'accent sur l'algorithme d'Andersen qui a été implémenté et amélioré au sein de l'architecture CLA (compile-link-analysis) et qui fait partie des analyses insensibles au flot de données et au contexte. Par la suite nous introduirons l'analyse d'Emami qui prend en compte le contexte des procédures ainsi que le flot des données.

1.4 But du stage

On s'est fixé deux objectifs pour le stage :

- Effectuer en premier lieu une étude bibliographique ciblée des analyses de pointeurs pour le langage C. En fait, les analyses étudiées vont inspirer le traitement des pointeurs au niveau du logiciel PIPS (Parallélisation interprocédurale des programmes scientifiques)[14] dans le cadre du projet FREIA. PIPS a pour but de paralléliser des programmes en détectant les dépendances entre les données au niveau des boucles imbriquées, plus précisément les dépendances entre les éléments des tableaux. Une tâche rendue plus difficile par les programmes écrits en C, car on peut avoir accès aux éléments des tableaux par pointeur. Ainsi on expose les analyses qui pourraient traiter au mieux les programmes analysés par PIPS. Essentiellement des programmes issus d'applications de traitement de signal où domine le calcul matriciel et les structures de données (le type struct du langage C);

- Effectuer une analyse des besoins des applications de traitement de signal. Cette partie s'est faite sur deux étapes. la première consiste à l'exploration du code source des applications de traitement de signal, dans le but d'en extraire les constructions syntaxiques les plus récurrentes (les pointeurs vers les tableaux, les tableaux de structures ...). La deuxième étape expérimentale, consiste à mettre en oeuvre des programmes C afin de tester le traitement actuel des pointeurs au niveau de PIPS. Cette étape facilite le choix de l'analyse qu'on voudrait implémenter par la suite au niveau du logiciel.

1.5 Organisation du rapport

Le rapport est organisé en 3 parties :

Une première partie est consacrée aux analyses de pointeurs insensibles au flot de données et au contexte, elle inclut un chapitre sur l'algorithme d'Andersen et son intégration au sein de l'architecture CLA (compile-link-analysis), qui est utilisé par le compilateur gcc. Cette partie introduit au préalable une présentation de l'algorithme de Steensgaard, qui est moins précis que l'algorithme d'Andersen mais plus rapide;

Dans la deuxième partie, on s'intéresse aux analyses de pointeurs sensibles au flot de données ainsi qu'au contexte. Un chapitre est consacré à l'analyse introduite par Wilson et intégrée au compilateur SUIF. Cette analyse utilise les fonctions de transfert pour modéliser les relations *points-to* et sert comme information en entrée pour l'analyse des dépendances.

Toujours dans le cadre des analyses sensibles au flot de données et au contexte on consacre un chapitre à la présentation de l'analyse d'Emami intégrée au compilateur McCAT. Cette analyse abstrait l'espace pile afin de modéliser les relations points-to entre les variables de type pointeur.

Cette analyse permet de traiter les tableaux, en traitant l'indice zéro différemment des autres valeurs, qui sont traités de la même manière. Elle permet aussi de traiter aussi les structures agrégées, mais reste imprécise au niveau du traitement des données allouées dans le tas.

Dans la troisième partie, on s'intéresse aux besoins en analyse de pointeurs au sein des applications, en particulier de traitement de signal. Nous étudions ces applications afin de déterminer les usages les plus fréquents des pointeurs, la fréquence des appels à malloc et la fréquence des structures de données récursives.

Chapter 2

L'algorithme d'Andersen dans L'architecture Compile-Link-Analysis

Dans la littérature, il y a deux algorithmes qui dominent les analyses insensibles aux flots de données et au contexte des fonctions : l'algorithme d'Andersen implémenté au niveau de gcc et de celui de Steensgaard. Ces deux algorithmes effectuent des analyses intraprocédurales sur des programmes de millions de lignes de code C et ont prouvé leur valeur en terme d'efficacité et de temps d'exécution. Dans le cadre des analyses insensibles au flot de données et au contexte on introduit en premier lieu l'algorithme Steensgaard, qui est basé sur des équations; ainsi pour deux pointeurs p et q, si on a p = q, l'analyse donne points_to(p) = points_to(q). En deuxième lieu on introduira l'algorithme d'Andersen, basé sur des sous-ensembles tels que si,pour deux variables de type pointeur p et q, on a p = q, l'analyse donne points_to(p) $\subseteq$ points_to(q).

2.1 L'algorithme de Steensgaard

L'algorithme de Steensgaard[20] a une complexité linéaire au prix d'une précision inférieure par rapport à l'algorithme d'Andersen[4] qui est de complexité cubique.

Les deux algorithmes génèrent des graphes pour modéliser les relations «points-to» entre les variables. Un noeud est supposé représenter une variable et l'arc la relation «points-to» entre deux variables. Une différence entre les deux algorithmes existe au niveau du nombre d'arcs sortant d'un noeud. En effet au niveau de l'algorithme d'Andersen nous avons un nombre arbitraire d'arcs sortants, alors que Steensgaard permet seulement un seul arc; donc un noeud représente plus d'une variable (les variables pointant vers les mêmes cibles sont fusionnées). Considérons le programme C suivant:

```
int *a, *b, *d, c, e;
a = &b; /* S1 */
b = &c; /* S2 */
d = &e; /* S3 */
a = &d; /* S4 */
```

L'algorithme de Steensgaard génère le graphe de la figure 2.1 pour S1 et S2 :

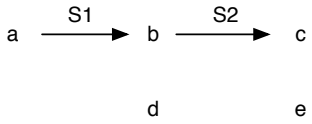


Figure 2.1: Les relations points-to pour S1 et S2

Après analyse des affectations S2 et S3 on obtient le graphe de la figure 2.2 :

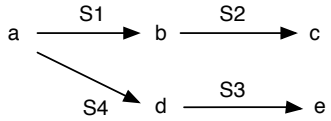


Figure 2.2: Les relations points-to pour S3 et S4

L'algorithme de Steensgaard fusionne par la suite les deux noeuds b et d en un seul, de même pour les noeuds c et e, pour aboutir à la fin de l'analyse au graphe de la figure 2.3 :

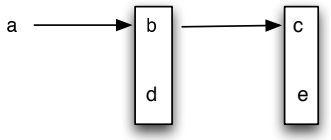


Figure 2.3: Le graphe final

On voit sur cet exemple le prix qu'il faut payer en précision puisque b pointe probablement vers e, ce qui ne correspond pas à la réalité.

2.2 Intégration de l'algorithme d'Andersen

2.2.1 L'algorithme d'Andersen : une analyse basée sur les contraintes

Introduite par Andersen [4], cette approche permet l'analyse des programmes de millions de lignes de code. On peut considérer que, pour un programme donné, contenant des pointeurs, les relations entre eux peuvent être formulées sous la forme de combinaisons des contraintes suivantes :

$p \supseteq q \mid p \supseteq \{q\} \mid p \supseteq *q \mid *p \supseteq q \mid *p \supseteq \{q\}$

où q et p sont les variables contraintes et * l'opérateur de déréférencement; on peut considérer que chaque variable contient l'ensemble vers lequel elle pointe; donc $p \supseteq \{x\}$ stipule que p pointe vers x. En effet, on commence par traduire le programme source dans ce langage en liant chaque variable à une unique variable contrainte et en convertissant les affectations en des contraintes.

Pour la représentation sous la forme de graphe, nous avons un nombre arbitraire d'arcs où chaque noeud représente une seule variable, ce qui accroît la précision des résultats.

L'analyse d'Andersen se fait vers l'avant; elle n'est ni sensible au contexte ni sensible au flot de données. L'analyse ne représente pas les alias sous forme de paire comme la plus part des analyses d'aliasing mais comme un lien entre la variable pointeur et l'ensemble de variables vers lesquelles elle pointe. Par exemple quand on a p = &x et p = &y, l'analyse génère le lien [p → { x,y }], qui est une représentation plus compacte, plus économique en terme d'espace de stockage et plus adéquate pour les autres analyses. Au sein de cette analyse les affectations du programme en cours seront vues comme des contraintes et classifiées en trois types : de base, simple et complexe suivant la topologie suivante [20]:

- des contraintes de base a := &b, quand on a l'opérateur & au niveau de la partie droite de l'affectation ;
- des contraintes simples c := a, quand on a une affectation simple sans variable de type pointeur;
- des contraintes complexes *a := z, quand on a l'opérateur * dans la partie gauche de l'affectation.

Les affectations de type *a = *b peuvent être décomposées en contraintes complexes en introduisant des variables temporaires; on aura *a := z et z := *b .

Prenons comme exemple le programme suivant :

```
a = &b;
c = a;
a = &d;
e = a;
```

Au niveau de ce programme on n'a que des affectations de base ou de type simple; pour analyser ce programme on associe une contrainte à chaque affectation pour aboutir aux contraintes suivantes :

- $a \supseteq \{b, d\}$: la variable a pointe vers b et d;
- $c \supseteq \{a\}$: la variable c pointe vers la variable a, et donc à son tour la variable c pointe vers b et d;
- $e \supseteq \{a\}$: la variable e pointe vers la variable a,et donc à son tour la variable c pointe vers b et d;

Et par la suite par itération sur l'ensemble «points-to» :

- $a \rightarrow \{b, d\}$
- $c \rightarrow \{a\} \rightarrow \{b, d\}$
- $e \rightarrow \{a\} \rightarrow \{b, d\}$

On ne génère plus de nouvelles relations «points-to» et on atteint un point fixe, ce sont les relations «points-to» finales:

- $a \rightarrow \{b, d\}$;
- $c \rightarrow \{b, d\}$;
- $e \rightarrow \{b, d\}$;

Algorithme d'Andersen : prise en compte des fonctions

L'analyse d'Andersen permet aussi le traitement des fonctions, ce qui se fait par génération de contraintes pour les paramètres formels et effectifs. Considérons la fonction foo suivante :

```
foo(int* x)

{
  ...
  return x;

}
a = foo(&b);
```

L'analyse fait pointer les paramètres formels vers les paramètres effectifs, pour aboutir aux contraintes suivantes:

- $x \supseteq \{b\}$
- $a \supseteq \{x\}$

Algorithme d'Andersen : prise en compte des structures de données

Dans [7], à part le traitement des pointeurs, on détaille encore le traitement des variables de type struct. Le traitement adopte une nouvelle approche qui est simple et efficace à la fois et étend le langage d'ensemble de contraintes pour aboutir à une analyse «field-sensitive» pour le langage C. Par ce mécanisme les pointeurs sur les fonctions sont aussi supportés sans sur-coût. En effet pour traiter les variables de type *struct*, on peut appliquer soit l'approche «field-insensitive», où le traitement des champs des structures est agréée à la structure en entier, soit l'approche «field-based» où toutes les instances d'un même champ sont modélisées comme une seule variable. Selon [7] on adopte l'approche «field-sensitive», où chaque instance d'un champ est modélisée avec une variable séparée.

2.3 L'architecture CLA

Dans cette section nous allons introduire l'architecture CLA (compilation-link-analyse) [18], introduite par Tardieu et Heintz, qui s'inspire du processus de compilation et d'exécution et améliore l'algorithme d'Andersen présenté précédemment en terme de temps d'exécution. Le processus de compilation et d'exécution classique fait intervenir trois phases : chaque fichier source est compilé en un fichier objet contenant les informations de liens ainsi que le code machine; puis l'ensemble des fichiers objets est lié pour former un exécutable (CLA); finalement l'exécutable est chargé et exécuté. Par analogie, une analyse associe à chaque fichier source un fichier objet lors de la phase de compilation où on sauvegarde les informations nécessaires à l'édition des liens ainsi que le graphe d'appels et les arcs représentant les relations «points-to» à la place du code machine. Lors de la phase d'édition de liens, on fusionne les fichiers dans une seule base de données afin de fusionner les symboles globaux qui se trouvent dans différents fichiers objets et finalement la phase d'analyse est effectuée.

En résumé la première phase transforme les fichiers source et en extrait les affectations et la phase de lien relie ensemble ces affectations sans effectuer d'analyse, ce qui conduit à l'un des

avantages de cette architecture, qui est que les phases de compilation et d'édition de liens restent inchangées pour différentes types d'analyse de pointeurs et même différents types d'analyse. Du fait, l'architecture CLA a été l'infrastructure de différentes variantes d'analyses de pointeurs comme l'analyse basée sur le graphe, l'analyse «field-independent» ou encore l'analyse «field-based»[18].

2.3.1 Les règles d'inférence

Pour mettre en oeuvre l'analyse des pointeurs, nous avons considéré que toute expression du programme peut être sous l'une des formes suivantes :

$e ::= x | *x | \&x$

Et nous avons exprimé des règles d'inférence STAR1, STAR2, ASSIGN, TRANS pour chaque affectation qui apparaît dans un programme P comme suit :

$$\frac{x \rightarrow \&y}{y \rightarrow e} \quad (if * x := e \ dans \ P)(STAR1)$$

$$\frac{x \rightarrow \&y}{e \rightarrow y} \quad (ife := *x \ dans \ P)(STAR2)$$

$$\frac{}{e1 \rightarrow e2} \quad (ife1 := e2 \ dans \ P)(ASSIGN)$$

$$\frac{e1 \rightarrow e2 \quad e2 \rightarrow e3}{e1 \rightarrow e3} \quad (TRANS)$$

L'exemple suivant montre comment en appliquant les règles d'inférence on arrive à déduire que $y \rightarrow \&x$:

```
int x, *y;
int **z;        /*   z->&    (ASSIGN) */
z = &y;         /* *z ->&x &(ASSIGN) */
*z = &x ;       /* &y ->&x & (STAR 1) */
```

2.3.2 Le format du fichier objet

Le format du fichier objet est illustré par la figure suivante:

header section: taille et décalage des segments
global section: information pour l'édition de liens
static section: adresse des opérations ; toujours chargée pour l'analyse des pointeurs
string section: chaînes de caractères communes
target section: table de hachage pour retrouver les cibles
dynamic section:les éléments sur chargés selon la demande, organisés par objet

Au niveau du fichier objet, on retrouve les sections suivantes :

- la section «header» qui contient le numéro de la version et une liste de l'offset et de la taille des autres sections du fichier.

- la section «global» qui contient la liste des variables globales et leurs offset dans le fichier.
- la section «static» qui contient les dépendances statiques.
- la section «string» qui contient une liste de chaînes qui pourrait être utilisée par d'autres sections du fichier.
- la section «target» qui contient une table de hachage qui fournit un mapping rapide entre le nom d'une cible et les variables de même nom.
- la section «dynamic» qui contient les relations de dépendances élémentaires du programme.

Les fichiers objets sont indépendants de l'implémentation et par conséquent les détails de l'implémentation peuvent être changés sans changer le format du fichier objet.

2.3.3 L'algorithme d'Andersen au niveau de la CLA

Au niveau de l'architecture CLA, des optimisations de l'algorithme d'Andersen, dont le coeur est une structure en graphe, ont été effectuées. La principale est que l'on ne calcule pas tout le graphe, seulement les arcs des variables demandées via l'architecture «demand-driven»[10].

L'analyse «points-to» dans l'architecture CLA commence à partir de la section statique du fichier objet qui contient les affectations de la forme q :=&y. En effet, à partir de cette affectation, on obtient la contrainte $q \supseteq y$ et par conséquent $q \rightarrow y$; donc on doit rechercher et charger toutes les affectations où q est la source et générer les contraintes associées.

L'algorithme d'Andersen au niveau de la CLA permet de générer un graphe réduit car initialement il ne contient que les affectations simples et de base. Pour construire le graphe, on aura un arc de x vers y pour chaque affectation de type $x = y$. Pour chaque noeud x du programme P, on associe l'ensemble $baseElement(n_x) = \{y | x := \&y$apparaît dans P $\}$. Quant aux affectations complexes elles sont regroupées dans un ensemble C. L'algorithme est illustré par la figure 2.4.

```
do{
  Nochange = true;
  for each complex assignment *x = y in C
             for each &z in getLvals(nx) //&...affecté à x
                         add an edge nz -> ny to G;
  for each complex assignment x = *y in C
             add an edge nx -> n*y;
  for each &z in getLvals(ny)
             add an edge n*y -> nz;
 }until no change
```

Figure 2.4: L'algorithme d'Andersen

L'algorithme procède par itération sur l'ensemble des affectations complexes et ajoute les arcs en se basant sur les informations contenues dans le graphe G; le graphe est maintenu en forme pré-transitive; la fermeture transitive n'est pas effectuée. Au niveau de la fonction getLvals, on optimise le graphe par élimination des cycles.

2.4 Critique de l'analyse

Cette étude bibliographique a pour but de faire l'état de l'art des analyses des pointeurs, qui répondent le mieux aux besoins des applications de traitements d'images, traitées par PIPS (parallélisation interprocédurale des programmes scientifiques) dans le cadre du projet FREIA. Ces besoins, ainsi que les structures les plus récurrentes, qui sont référencées par des pointeurs, seront détaillés dans le chapitre 5. Concernant l'analyse présentée dans ce chapitre, elle a l'avantage d'être efficace en terme de temps d'exécution, ainsi qu'en terme de support de réels programmes C de grande tailles. Mais les analyses effectuées au niveau du projet PIPS, et qui devront utiliser l'information fournie par l'analyse des pointeurs sont sensibles au flot de données.

L'ordre d'exécution des instructions, ainsi que le contexte des appels des fonctions sont des informations primordiales pour la parallélisation des programmes. Or l'analyse d'Andersen n'est ni sensible au flot de données ni au contexte des appels. Ceci a pour résultats des alias superflus fournis en résultats. En effet, l'analyse ne peut distinguer entre les différents contextes d'une fonction et permet la propagation d'information imprécise d'un site d'appel à un autre.

Pour conclure, l'analyse d'Andersen, même si elle permet d'avoir des résultats rapides, n'est pas la plus adéquate pour être intégrée à l'outil PIPS compte tenu de son aspect «flow-insensitive» «context-insensitive».

Chapter 3

La fonction partielle de transfert dans le compilateur SUIF

3.1 Introduction

Dans les Chapitres 3 et 4 on introduit deux analyses statiques, c'est à dire qui analysent le programme au moment de la compilation (par opposition aux analyses runtime, au moment de l'exécution). L'analyse se fait par inspection du code source dans le but de vérifier certaines propriétés comme les invariants de boucles ou les debordements de tableaux. Les deux analyses : celle de Wilson et d'Emami s'inspirent de l'analyse statique par interpretation abstraite. L'interpretation abstraite consiste à construire une sur-approximation de la sémantique du programme que l'on désire analyser, de telle sorte que, vis à vis de cette approximation, les propriétés que l'on souhaite déterminer soient décidables[24]. Elle peut être définie comme une exécution partielle d'un programme pour obtenir des informations sur sa sémantique (par exemple, sa structure de contrôle, son flot de données) sans avoir à en faire le traitement complet[1].

L'algorithme de Wilson[22] a été developpé pour le compilateur SUIF en 1995, c'est une méthode efficace, en terme de précision, pour analyser les pointeurs des programmes C. La précision des résultats est assurée par l'aspect interprocédurale et la sensibilité au contexte de l'analyse. En effet, on résume les effets des procédures en utilisant les fonctions partielles de transfert, qui décrivent le comportement d'une procédure en y émettant l'hypothèse qu'il y a des relations d'aliasing entre ses paramètres qui se mettent en place lors de l'appel à cette dernière.

Le but de cette analyse implémentée au sein du compilateur SUIF est d'identifier les valeurs potentielles des pointeurs au niveau de chaque bloc d'instructions du programme. Étant donné qu'il n'est pas nécessaire de résumer toute la procédure pour tous les alias potentiels mais seulement pour ceux qui se produisent dans le programme, l'idée est de générer une fonction de transfert partielle et incomplète qui couvre seulement les conditions d'entrée qui existent dans le programme.

L'analyse des pointeurs doit satisfaire certaines conditions, parmi elles le fait que l'analyse doit être efficace en terme de temps d'exécution sans pour autant sacrifier la précision des résultats et être suffisamment robuste pour supporter des programmes C et des millions de lignes de code. La technique proposée ici est complètement «context-sensitive» sans pour autant sacrifier l'efficacité; elle englobe toutes les fonctionnalités du langage C qui sont fréquemment utilisées et difficiles à

analyser comme le *cast*des types, les unions et l'arithmétique des pointeurs.

3.2 Le compilateur SUIF

La conception SUIF avait pour but de développer un système extensible supportant un large nombre de sujets de recherche qui sont d'actualité, incluant le domaine de la parallélisation, les langages de programmation orienté-objet, l'optimisation scalaire et machine cible. Le compilateur SUIF tend à développer une architecture modulaire, facile à étendre, à maintenir et à réutiliser. SUIF effectue des analyses et des optimisations indépendantes des machines qui permettent :

- l'élimination du code mort;
- l'élimination partielle des redondances;
- la propagation conditionnelle des constantes;

et des optimisations dépendantes des machines comme :

- l'ordonnancement des instructions;
- l'allocation des registres.

L'analyse des pointeurs introduite dans ce chapitre a été implémentée au niveau du compilateur SUIF, qui a comme but principal de paralléliser automatiquement des programmes Fortran. Par la suite il a été étendu afin de supporter la parallélisation automatiques des programmes écrits en C. La parallélisation des programmes au niveau de SUIF se fait par étude des dépendances entre les nids de boucles où on accède à des éléments de tableaux. Pour les programmes écrits en C on accède aux tableaux par pointeurs; c'est à ce niveau que l'analyse des pointeurs intervient. L'analyse des pointeurs introduite par Wilson[23] va servir d'information d'entrée pour l'analyse des dépendances. Pour deux références sur tableau, on va essayer de déterminer si ces deux pointeurs font référence au même tableau. Si c'est le cas on vérifie si les deux pointent vers le début du tableau. Étant donné cette information, l'analyse des dépendances appliquée aux programmes Fortran devient applicable aux programmes écrits en C. Cependant l'utilisation de l'information fournie par l'analyse des pointeurs reste limitée par le compilateur SUIF parce qu'elle n'agit que sur la parallélisation intraprocédurale des boucles. Néanmoins elle reste éventuellement utilisable par d'autres analyses, qui sont implémentées au niveau du compilateur, comme la *transformation* des boucles.

3.3 Les concepts de base

3.3.1 Introduction

Afin d'étudier les relations d'alias entre les paramètres des fonctions, prenons comme exemple le programme C de la figure 3.1 :

```
int f (int **p, int** q, int ** r) {
  *p = *q;
  *q = *r;
}

int  x, y, z;
int *x0, *y0, *z0;

main () {
  x0 = &x;
  y0 = &y;
  z0 = &z;
  if(test1)
    f(&x0, &y0, &z0); // S1
  else if (test2)
    f(&z0, &x0, &y0); // S2
  else
    f(&x0, &y0, &x0); // S3
}
```

Figure 3.1: Le programme C

Voici les résumés informels correspondants aux trois sites d'appels S1, S2, S3 avec comme hypothèse : la cible de p pointe vers ce que initialement la cible de q pointait.

- Cas 1 : si r et p ne pointent vers aucun même espace, la cible de q pointe vers ce que initialement la cible de r pointait.
- Cas 2 : si r et p pointent exactement vers le même espace, alors la cible de q pointe vers sa valeur originale.
- Cas 3 : si r et p pointent vers le même espace mais leur cible n'est pas forcément la même, alors la cible de q pourrait retenir sa valeur originale ou bien pointer vers quoi la cible de r pointait initialement.

L'exemple met en évidence deux points importants :

- l'aliasing en entrée d'une procédure détermine son comportement.
- le résumé complet reste complexe même pour une simple procédure.

L'idée est donc de résumer seulement les aliasings qui se produisent dans le programme.

Une analyse interprocédurale Pour aboutir à des résultats précis, le programme en entier y compris les bibliothèques incluses doivent être fournis à l'analyse des pointeurs. Sinon on sera dans l'obligation d'émettre des hypothèses sur les effets de bord des procédures. En effet, on doit tenir compte des deux contraintes suivantes:

- une procédure peut modifier les valeurs des pointeurs qui sont visibles au niveau de son contexte d'appel. Par conséquent, les instructions qui viennent après son appel ne peuvent être correctement analysées tant que les effets de l'appel ne sont pas connus.

- Le comportement d'une procédure dépend aussi de son contexte d'appel; l'information sur les alias à partir du contexte d'appel doit être disponible.

Ces contraintes ne permettent ni une approche «top-down» ni une approche «bottom-up», mais plutôt une approche itérative qui combine l'analyse intraprocédurale et interprocédurale.

Une analyse sensible au contexte Au cours d'une analyse statique, des informations peuvent être perdues au cours des différents chemins permettant d'atteindre un point particulier du programme. Et parce que un programme a généralement plusieurs chemins d'exécution, l'analyse doit pouvoir combiner différentes informations provenantes de ces chemins. Par conséquent, l'analyse interprocédurale doit aussi supporter le traitement de différents sites d'appels. Néanmoins reste le problème de l'identification du contexte d'appel, qui est une tâche facile au moment de l'exécution,parce qu'elle se fait par consultation de l'état de la pile. Cependant, pour une analyse statique, il existe deux méthodes pour distinguer entre les contextes :

- l'approximation des chemins d'appels, dont la première raison est la récursivité; en absence de cette dernière l'analyse pourrait facilement énumérer tous les chemins menant à chaque procédure. Cette approche a été adoptée par l'analyse d'Emami, qui sera introduite dans le chapitre 4, via une structure de données particulière qui est le graphe d'invocation.
- l'utilisation de valeurs calculées par l'analyse : on pourrait identifier les contextes d'appels par les alias calculés au niveau du site d'appel. Et s'il y a un nombre fini de valeurs, ceci pourrait être plus efficace que l'approximation des chemins.

3.3.2 Fonction de transfert

Cette solution utilise les fonctions «points-to» pour déterminer les alias intraprocéduralement, en appliquant les règles suivantes[21] :

- $x = \&y : pts\_to(x) = \{y\}$
- $x = y : pts\_to(x) = pts\_to\{y\}$
- $x = *y : pts\_to(x) = \cup\{z\}, pour\ tout\ z \in pts\_to\{y\}$
- $*x = y : pts\_to(z) = \cup pts\_to(y), pour\ tout\ z \in pts\_to(x)$

Pour l'analyse interprocédurale, on construit le graphe d'appel pour identifier les relations d'appel entre les méthodes. Et pour prendre en compte le contexte des fonctions, la solution proposée par Wilson[23] se base sur le calcul de fonctions de transfert. Une fonction de transfert permet de représenter la relation entre l'entrée et la sortie d'un système. Cette analyse s'inspire du calcul des effets mémoires qui se fait au niveau du compilateur PIPS (parallélisation interprocédurale de programmes scientifiques) [16]. Un effet mémoire est l'action d'écriture ou de lecture sur une variable scalaire. Au niveau de PIPS, les instructions sont étiquetées par des prédicats qui expriment des contraintes sur les variables scalaires qui sont utilisées comme références (indices de tableaux) ou comme bornes de boucles. Les prédicats permettent d'améliorer la précision de l'information sur les effets mémoires, de fournir de meilleurs informations sur les tests de dépendances et de

choisir les meilleures transformations des programmes. En fait, les prédicats au niveau de PIPS sont une abstraction[13][14] des commandes permettant le mappage d'un emplacement à un ensemble d'emplacements. Ils permettent d'établir une relation entre les valeurs des variables scalaires entières dans un emplacement initial et un emplacement final.

Dans le cas de l'analyse de Wilson, les fonctions de transfert résument les effets des procédures sous certaines conditions dans le contexte d'appel. Les fonctions de transfert établissent une relation entre les fonctions «points-to», récupérées via l'analyse intraprocédurale, et les fonctions «points-to» finales après l'analyse du corps de la procédure. Ainsi une fonction de transfert a pour rôle de résumer une procédure et de spécifier les conditions du contexte d'appel qui définissent le domaine d'application de la fonction de transfert. La fonction de transfert doit décrire le comportement de la procédure pour toutes les entrées possibles. Pour les analyses interprocédurales, la fonction de transfert aboutit à de bonnes performances en absence de récursivité. L'analyse ne requiert que deux passes sur le programme: une «bottom-up» pour calculer les fonctions de transfert et une «top-down» pour les appliquer. Ainsi au niveau de chaque site d'appel, les effets de chaque appel peuvent être déterminés à partir de la fonction de transfert de l'appelée. Et puisque les fonctions de transfert sont paramétrées, les résultats sont entièrement sensibles au contexte. Par la suite, le coût de production d'une fonction de transfert est amorti par tous ses appels; reste seulement le coût de l'application de la fonction de transfert multiplié par le nombre de contextes. Pour l'analyse de pointeurs, énumérer tous les alias possibles dans un programme n'est pas pratique, surtout quand la majorité de ces alias ne se produisent pas vraiment. L'analyse a seulement besoin de calculer les effets des procédures pour les alias qui se produisent vraiment, dits alias pertinents.

3.3.3 L'espace de stockage

Afin de pouvoir supporter les fonctions du langage C, on abstrait l'espace de stockage en utilisant une représentation de bas niveau de la mémoire. Cette représentation fournit des résultats proches de ceux de la représentation de haut niveau par type, cependant elle peut aboutir à des résultats de précision moindre. On définit un espace de nommage pour l'espace de stockage de chaque PTF (partial transfert function). En effet, au niveau de chaque appel de procédure, l'analyse doit traduire de l'espace de nommage de l'appelant vers celui de la PTF utilisé pour décrire l'appelée.

L'abstraction de l'espace de stockage a recours à une représentation qui modélise la mémoire sous forme de blocs où on accède à un élément via sa position dans le bloc et non pas via le nom du champ. Ainsi pour représenter les blocs, on utilise un triplet (b, f, s) où b est le nom du bloc mémoire, f est l'offset de l'élément au sein du bloc et s la taille de l'élément. L'utilisation de l'espace de nommage est une caractéristique de l'analyse. Alors que l'espace mémoire d'un programme est composé de blocs contigus (les variables locales, les variables globales et les espaces alloués dynamiquement), l'espace de nommage d'une PTF, lui, se compose de blocs abstraits : chaque bloc représente un ou plusieurs blocs réels.

Au niveau de l'espace de stockage de la PTF on considère deux types de blocs abstraits : local ou non local. Les blocs non locaux représentent l'espace qui existait dans le contexte d'appel, ils sont référencés via des pointeurs pour être accessible au niveau de la procédure. Puisque différents contextes d'appel peuvent avoir différentes valeurs de pointeur, les blocs non locaux sont considérés comme des paramètres de la PTF. Et concernant les variables globales, on les traite comme des

blocs non locaux afin d'augmenter la réutilisation des PTF pour différents contextes d'appels.

Quant aux blocs locaux ils représentent l'espace alloué au sein de la procédure, qui est indépendant du contexte d'appel. Il en existe deux types :

- les blocs des variables locales, pour représenter les paramètres formels et les variables locales de la procédure;
- les blocs pour le tas, pour représenter les portées du tas alloué au sein d'une procédure ou au sein des procédures appelées à son niveau.

On définit aussi les paramètres étendus qui sont liés à l'espace de nommage. Un paramètre étendu représente l'ensemble des valeurs potentielles d'un pointeur contenu dans un emplacement abstrait, à l'entrée d'une procédure. Cette notion permet de différencier entre les valeurs et les emplacements. Ainsi on utilise les paramètres étendus pour nommer les valeurs à la portée de la fonction «points-to» et les blocs non locaux pour nommer les emplacements de l'espace de stockage abstrait de la PTF. En effet, quand la valeur d'un pointeur, représentée par un paramètre étendu, est référencée l'analyse doit traduire ce paramètre au niveau du bloc non local correspondant dans l'espace de stockage abstrait de la PTF. La section 3.3.3 détaillera la notion de paramètres étendus.

3.3.4 Modélisation de la mémoire

L'analyse des pointeurs calcule un ensemble d'emplacements vers lesquels un pointeur pourrait pointer, ce qui pourrait introduire des ambiguïtés surtout qu'on s'intéresse aux emplacements alloués dynamiquement ou contenant des structures agrégées ou des tableaux. L'idée est d'abstraire la mémoire est de lier des emplacements concrets à un ou plusieurs emplacements abstraits.

Le but de cette analyse est de pouvoir distinguer entre différents champs d'une structure mais pas entre les différents éléments d'un tableau; en introduisant une nouvelle abstraction qui est un ensemble d'emplacements pour décrire en même temps un bloc de mémoire ainsi que les emplacements au sein de ce bloc. En effet, on divise la mémoire en blocs, chaque bloc représente un ensemble d'adresses contigues dont la position de l'un par rapport à l'autre est indéfinie.

Un bloc mémoire pourrait être soit une variable locale, soit un bloc tas alloué localement soit un paramètre étendu. Les blocs de tas sont seulement ceux alloués dans une procédure ou bien à travers ses appelants. Ceux qui sont transmis à partir d'un contexte d'appel sont considérés comme des paramètres étendus. Ils sont distingués par leur contexte d'allocation, en groupant ensemble tous les blocs alloués dans chaque contexte, l'information minimale sur le contexte est l'instruction qui a créé le bloc.

On représente les positions au sein d'un bloc par des ensembles d'emplacements où un ensemble d'emplacement est un triplet (b, f, s). Dans le tableau suivant, on expose les triplets associés à différentes expressions :

Expression	Ensemble d'emplacement
scalaire	(scalaire, 0, 0)
struct.F	(struct, f, 0)
tableau	(tableau, 0, 0)
tableau[i]	(tableau, 0, s)
tableau[i].F	(tableau, f, s)
struct.F[i]	(struct, f%s, s)

Table 3.1: Exemples d'ensembles d'emplacements

Où f est l'offset du champ F et s est la taille de l'élément du tableau.
Ainsi on remarque qu'un champ dans une structure est identifié par son offset à partir du début de cette dernière. La référence à un tableau par contre est mise à zéro. Et quand la position d'un emplacement au sein d'un bloc est inconnue, on met le pas s à 1. Cela veut dire que l'ensemble des emplacements inclut tous les emplacements du bloc. Ceci permet d'approximer de manière conservatrice les résultats de n'importe quelle expression arithmétique; en gardant la même base b mais avec un pas s mis à un. Et parce que la position d'un bloc par rapport à un autre est indéfinie, on n'a pas à se soucier de l'arithmétique de pointeurs qui déplace un pointeur vers un différent bloc.

Cette représentation de la mémoire a pour buts :

- distinguer entre les différents champs d'une structure, toujours dans le but de fournir des résultats plus précis.
- représenter les tableaux comme un seul élément.
- supporter l'arithmétique pointeur en offrant la possibilité de représenter des emplacements inconnus dans un bloc.

Étant donné que les valeurs des ensembles d'emplacement sont dérivées de la manière avec laquelle on y accède. Cette représentation sous forme d'ensemble d'emplacement omet des informations liées aux types. Ceci peut, en comparaison à d'autres analyses, induire des résultats moins précis, mais permet par contre une analyse sure des programmes où il y a violation des types déclarés.

3.3.5 Le traitement des données allouées dans le tas

L'analyse représente les emplacements alloués dynamiquement (le tas/heap) par des blocs tas ou bien par des blocs non locaux, selon l'endroit où l'emplacement a été référencé.
Quand l'emplacement est référencé au sein d'une procédure, il est considéré comme un bloc tas. Il n'est visible des autres procédures que s'il leur est passé par référence. Les autres procédures traitent le bloc comme un bloc non local, comme tous les blocs passés par référence. Ainsi le bloc tas pour une procédure représente seulement l'espace tas alloué au niveau de ses descendants du graphe d'appel. Mais même avec cette simplification, on peut se retrouver avec un grand nombre d'emplacements représentés par un bloc tas. C'est pour cette raison que l'analyse partitionne l'espace en un nombre fini de classes qui seront décrits par la suite.

Le but de l'analyse est de distinguer entre les emplacements mémoire à un niveau de granularité d'une structure de données complète. Pour le traitement du tas, le but est d'avoir un seul bloc

pour chaque structure de données.

Pour identifier les structures de données allouées au niveau du tas, sans avoir recourir à l'information sur les types, on utilise deux heuristiques pour partitionner le tas :

1. Si l'espace est alloué dans le même contexte d'appel relatif au contexte courant, il est supposé faire partie de la même structure de donné et est placé en conséquence dans la même classe. En effet, l'idée est de grouper l'espace tas par contexte d'allocation., où le contexte d'allocation est identifié par l'instruction qui fait appel à la routine d'allocation. Avec une information supplémentaire, le chemin à travers le multigraphe d'appel.
   Cette heuristique découle du fait que l'espace alloué dans le même contexte appartient presque toujours à la même structure de données.

2. Combiner tout l'espace tas qui est potentiellement référencé par un pointeur dans un contexte donné, en émettant l'hypothèse que ce pointeur contient des références à une seule structure de données. En effet, nommer l'espace tas par le contexte d'allocation n'est pas suffisant pour aboutir à un seul bloc par structure de données. Il permet d'avoir des blocs de tas pour différents structures de données mais n'empêche pas d'avoir plusieurs blocs tas pour la même structure de données. Ce cas peut se produire quand les éléments d'une structure de données sont alloués dans différents contextes.

   L'heuristique définit chaque bloc tas qui représentera l'espace à partir de plusieurs contextes d'allocation de sorte que les blocs tas utilisés dans une PTF ne sont jamais aliasés. Par conséquent l'espace d'un contexte d'allocation particulier n'est jamais représenté par plus d'un bloc tas dans la même PTF.

   Chaque fois qu'on a un pointeur qui référence plusieurs blocs tas, on combine l'ensemble des contextes d'allocation en un nouveau bloc tas, comme dans le cas du programme suivant :

```
void *alloc(){
  void *p;
  if(test())
    p=malloc(1); /* contexte x */
  else
    p=malloc(1); /* contexte y */
  return p;
}
```

   Un bloc h0 va être créé au niveau du contexte x, lors du deuxième appel à malloc, au niveau du contexte y, un deuxième bloc h1 va être crée. En appliquant l'heuristique, les deux blocs vont être combinés et vont être représenté par un nouveau bloc tas unique.

Ces heuristiques ont montré leur efficacité, surtout en présence d'appel aux fonctions d'allocation de tas standards du langage C, exemple la fonction *malloc*.

3.3.6 Les paramètres étendus

Un paramètre étendu représente l'ensemble des valeurs potentielles d'un pointeur contenu dans un emplacement abstrait à l'entrée d'une procédure. Les paramètres étendus sont une généralisation des paramètres formels. Ils représentent des noms symboliques pour toutes les valeurs qui passées directement à une procédure ou bien les valeurs passées indirectement via les pointeurs.
Chaque emplacement de l'espace de stockage d'une PTF (partial transfert function) a son propre paramètre étendu afin de représenter ses valeurs initiales.

Les valeurs représentées par les paramètres étendus ne changent pas suite aux affectations dans la procédure. Elles reflètent l'état des pointeurs dans le contexte d'appel et par conséquent restent constantes tout au long de la procédure. Pour simplifier l'analyse on fait pointer chaque nouvelle entrée vers un unique paramètre étendu. Dans le cas où la valeur initiale est aliasée avec plus d'une valeur, on crée un nouveau paramètre étendu qui contient tous les paramètres aliasés. Dans le cas où les valeurs initiales sont aliasées avec un paramètre existant et incluent aussi de nouvelles valeurs, on introduit un nouveau paramètre qui contiendra les anciens alias.

Les paramètres étendus permettent de représenter les pointeurs passés à une procédure, ne se référant pas à un bloc non local. Ceci est important pour l'analyse qui ne veut considérer que les alias pertinemment; où le pointeur est déférencé. En effet, l'analyse identifie les alias pertinents en cherchant les valeurs «points-to» initiales qui font référence au même bloc non local. On ne veut sauvegarder que les valeurs «points-to» initiales qui font référence à un bloc non local à moins que les pointeurs soient déréférencés. Les paramètres étendus ont trois rôles :

1. Ils font partie de l'espace de nommage d'une procédure. En effet chaque procédure a son propre espace de nommage constitué des paramètres étendus, des variables locales, de l'espace de stockage alloué par la procédure et ses descendants. On dérive un mappage sur chaque site d'appel pour lier l'espace de nommage de l'appelant et celui de l'appelé.

2. Les «points-to» initiaux précisent le domaine d'entrée. En effet, les alias à l'entrée de la PTF forment la majeure partie de son domaine de spécification. Les fonctions «points-to» initiaux capturent précisément cette information. Ceci est illustré par les figures 3.1 et 3.2 où les fonctions «points-to» relatives aux appels S1 et S2 du programme précédent montrent que les paramètres p, r et sont disjoints et pointent respectivement vers les paramètres étendus 1_p, 1_q et 1_r. Tandis que pour l'appel S3, étant donné que p et r sont aliasés et pointent en conséquence vers le même paramètre étendu 1_p.

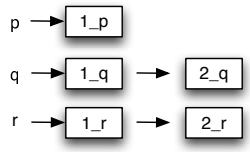


Figure 3.2: Les valeurs initiales au niveau de S1 et S2

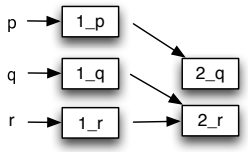


Figure 3.3: Les valeurs initiales au niveau de S3

3. Les «points-to» finaux au niveau de la sortie de la procédure résument les affectations aux pointeurs, qui sont facilement déduites à partir des «points-to» initiaux. les figures 3.3 et 3.4

illustrent les fonctions «points-to» finaux pour les appels S1, S2 et S3.

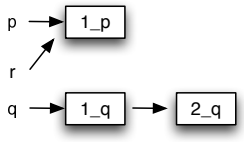


Figure 3.4: Les valeurs finales au niveau de S1 et S2

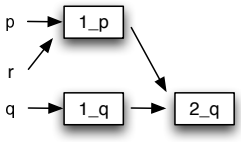


Figure 3.5: Les valeurs finales au niveau de S3

Les paramètres étendus interviennent aussi dans ce qu'on appelle les mises à jour «fortes». En effet, quand l'analyse peut déterminer qu'une affectation aura définitivement lieu, qu'elle va affecter une seul emplacement, l'analyse peut effectuer une mise à jour «forte» où les affectations écrasent l'ancienne valeur de leur destination au niveau de l'espace abstrait de stockage. En cas d'incertitude, on doit supposer de manière conservative que la destination conserve son ancienne valeur, cumulées avec les nouvelles valeurs potentielles de l'affectation.

Ces mises à jour affectent la terminaison de l'algorithme, ce qui nous obligent à introduire des contraintes pour s'assurer de la terminaison de l'algorithme. Ces contraintes introduisent un ordre sur l'évaluation des noeuds du graphe de flot; on s'assure qu'aucun noeud n'est évalué à moins que l'un de ses prédécesseurs soit évalué et on n'évalue aucune affectation à moins de connaître sa destination.

Les grandes lignes de l'algorithme

L'algorithme de la figure 3.5 est une version simplifié de la fonction EvalProc() qui permet l'analyse d'une fonction en faisantappel à la fonction adéquate pour chaque type d'instruction. En effet , pour une affection simple à un scalaire on fait appel à EvalScalarAssign(), pour le traitement des structures agrégées on fait appel à EvalAggregateAssign(), pour le traitement des noeuds qui se trouvent à une jointure de deux chemins du graphe de flot de données on fait appel à EvalJoin() et finalement en présence d'un appel à une procédure on fait appel à EvalCall().

3.4 L'algorithme de Wilson : un exemple

Dans [23] on introduit l'algorithme de l'analyse qui procède par descente récursive sur le graphe d'appel. Au sein de chaque procédure, on utilise une analyse de flot de données itérative pour calculer les fonctions «points-to». A la création d'une nouvelle PTF, les fonctions «points-to» et la table des valeurs des pointeurs sont vides.

L'analyse itère par la suite sur les instructions de la procédure; en les évaluant et en mettant à jour les fonctions «points-to» associées. Au niveau des appels, l'analyse retrouve les appelés potentiels, puis évalue l'effet de l'appel. Pour chaque appelé potentiel, l'analyse essaye d'appliquer une PTF existante. Si elle ne trouve pas une PTF applicable, une nouvelle PTF est créée en analysant l'appelé. Une fois qu'on a appliqué une PTF, l'effet de la procédure, comme décrit à la sortie de la PTF, est traduit vers le contexte d'appel et utilisé pour mettre à jour la fonction «points-to» au niveau de ce point.

```
/* la fonction reçoit comme paramètre une PTF, qui pour un premier
   appel peut être initialement vide */
void EvalProc (PTF ptf) {
  do {
    changed = FALSE;
    /* parcours itérat if du graphe de flot de données */
    foreach Node n in ptf.proc.flowGraph
      if (no predecessors of n evaluated)
        continue;
    if (n is scalar assignment)
      /* EvalScalarAssign évalue les expressions, établit le mappage
         entre les valeurs et les emplacements abstraits et mets à jour les
         fonctions points-to */
      EvalScalarAssign(n, ptf);
    else if (n is aggregate assignment)
      /* EvalAggregateAssign traite les affectations aux structures
         agrégées */
      EvalAggregateAssign(n, ptf);
    else if (n is  join)
      /* EvalJoin traite les noeuds qui sont à  la jointure de deux chemins
         du graphe de flot de données */
      EvalJoin(n, ptf);
    else if (n is call)
      /* evalCall traite les appels aux fonctions en déterminant les
         effets de cet appel sur les fonctions points-to, ceci se fait
         par détermination de quelle procédure va être appelée,
         détermination de quelle PTF va être appliquée aux appelées
         potentielles et finalement par l'ajout de l'information
         points-to au niveau du site d'appel */
      EvalCall(n, ptf);
}while{!changed }
}
```

Figure 3.6: L'algorithme simplifié de Wilson

exemple On reprend l'algorithme de la section précédente : l'analyse commence par itérer sur la fonction main, jusqu'à atteindre l'appel à la fonction f au niveau de l'instruction S1. Puisque c'est le premier appel à la fonction et qu'elle n'a pas été analysée précédemment; une nouvelle PTF est créée avec comme entrée :

- $p \rightarrow x0$
- $q \rightarrow y0 \rightarrow y$
- $r \rightarrow z0 \rightarrow z$

Les figures 3.7 et 3.8 illustrent les fonctions «points-to» initiaux au niveau de l'instruction S1, S2 et S3. Les variables globales, sont mises entre parenthèses et représentées par des blocs non locaux. et comme sortie :

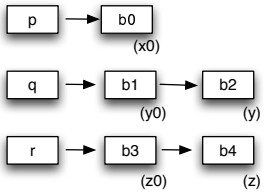


Figure 3.7: Les valeurs initiales au niveau de S1 et S2

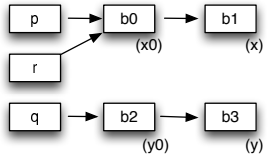


Figure 3.8: Les valeurs initiales au niveau de S3

- $p \rightarrow x0 \rightarrow y$
- $q \rightarrow y0 \rightarrow z$
- $r \rightarrow z0 \rightarrow z$

Les figures suivantes illustrent les fonctions «points-to» finales au niveau de l'instruction S1, S2 et S3.

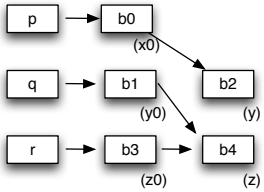


Figure 3.9: Les valeurs finales au niveau de S1 et S2

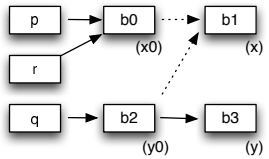


Figure 3.10: Les valeurs finales au niveau de S3

Après calcul de la PTF, sa sortie est traduite au niveau de l'appel. Comme résultat, la fonction «points-to» suivante va inclure les relations suivantes :

- $x0 \rightarrow \{y\}$,
- $y0 \rightarrow \{z\}$,

- $z0 \rightarrow \{z\}$.

Pour l'appel au niveau de l'instruction S2, la même PTF est appliquée; pour aboutir mettre à jour les relations et aboutir aux résultats suivantes :

- $x0 \rightarrow \{y\}$,
- $y0 \rightarrow \{y\}$,
- $z0 \rightarrow \{x\}$.

Pour l'appel au niveau de l'instruction S3; une nouvelle PTF est requise pour traiter les alias entre les paramètres formels; pour aboutir aux relations suivantes :

- $x0 \rightarrow \{x, y\}$,
- $y0 \rightarrow \{x, y\}$.

Après l'évaluation de tous les appels, l'analyse termine par fusionner les relations résultats de chaque appel, on aura ainsi comme résultat final, les relations «points-to» suivantes :

- $x0 \rightarrow \{x, y\}$,
- $y0 \rightarrow \{x, y, z\}$,
- $z0 \rightarrow \{x, z\}$.

Les itérations suivantes sur la fonction main n'aboutissent pas à de nouvelles relations et l'analyse se termine.

3.5 Critique de l'analyse

La majorité des analyses sensibles au contexte ont comme défaut qu'au pire des cas ils peuvent avoir une complexité exponentielle.
Les raisons de cette complexité sont que :

- les algorithmes ne peuvent obtenir la fonction de transfert complète d'une procédure, il est difficile de résumer les effets d'une procédure à un état donné du programme, tout en respectant un état initial arbitraire;
- chaque procédure doit être analysée à nouveau pour chaque contexte d'appel.

L'analyse de Wilson permet de remédier à ce dernier défaut en calculant la fonction partielle de transfert. Mais le coût de propagation des effets de l'appelée vers l'appelant reste élevé, puisque les relations «points-to» de l'appelant doivent être appliquées au niveau de chaque site d'appel.

Cependant l'analyse de Wilson fournit des résultats précis, compte tenu de sa sensibilité au flot de données et au contexte d'appel. Ses résultats peuvent éventuellement être exploités par l'analyse de dépendances de PIPS. Surtout s'ils peuvent être obtenus en temps d'exécution raisonnable.

Chapter 4

L'analyse d' Emami au niveau du compilateur McCAT

4.1 Introduction

L'analyse de Emami[8] de 1995 introduit une nouvelle approche pour étudier les relations «points-to». Cette approche se fonde sur une abstraction de l'espace pile et sur les relations entre les élements de la pile. L'analyse permet de modéliser ces relations sous la forme d'un triplet (x, y, rel) pour exprimer une relation «points-to» qui stipule que x pointe vers l'emplacement de y dans la pile avec une approximation rel. L'approximation rel permet de préciser la certitude de la relation : «definitely» quand on est sûr de la relation ou bien «possibly» dans le cas contraire.

Cette analyse est sensible au flot de données et au contexte. En effet, afin de prendre en compte les contextes des fonctions, l'analyse introduit une structure de données propre à elle, qui est le graphe d'invocation. Le graphe d'invocation modélise les séquences d'appels et de retour des fonctions.

Mise à part l'analyse interprocédurale, l'algorithme permet aussi le support des structures de donnés du langage C, les données allouées au niveau du tas sont représentées par un seul emplacement et les tableaux et les structures de données agrégées sont traités afin d'exploiter le parallélisme du code.Dans la suite, on explique en détails l'analyse de Emami.

4.2 Le compilateur McCAT

Le compilateur McCAT [11] a été conçu dans le but de pouvoir supporter des analyses de haut niveau et de bas niveau à la fois. Le compilateur a été mis en place avec comme principaux buts de conception :

- une famille de représentations intermédiaires : ces représentations s'étendent des représentations haut niveau qui capturent précisément le programme aux présentations bas niveau qui permettent l'allocation de registres ainsi que la planification des registres et la génération de code;
- une analyse de pointeurs précise : une représentation intermédiaire a été spécialement conçue

afin de permettre une analyse de pointeurs précise; l'analyse devait être précise afin de tirer profit du parallélisme du code.

- un développement d'outils de compilation : la conception du compilateur visait à utiliser une représentation intermédiaire structurée afin de développer des outils de compilation avancés comme un générateur de code et d'analyses.

Il est important de préciser que le compilateur ne traite que du code structuré ou qu'on peut structurer. C'est dans ce but qu'un ensemble raisonnable d'instructions de contrôle a été sélectionné, et chaque programme comportant des sauts doit d'abord être convertit en un programme équivalent structuré.

Une vue générale du compilateur est illustrée par la figure 4.1.

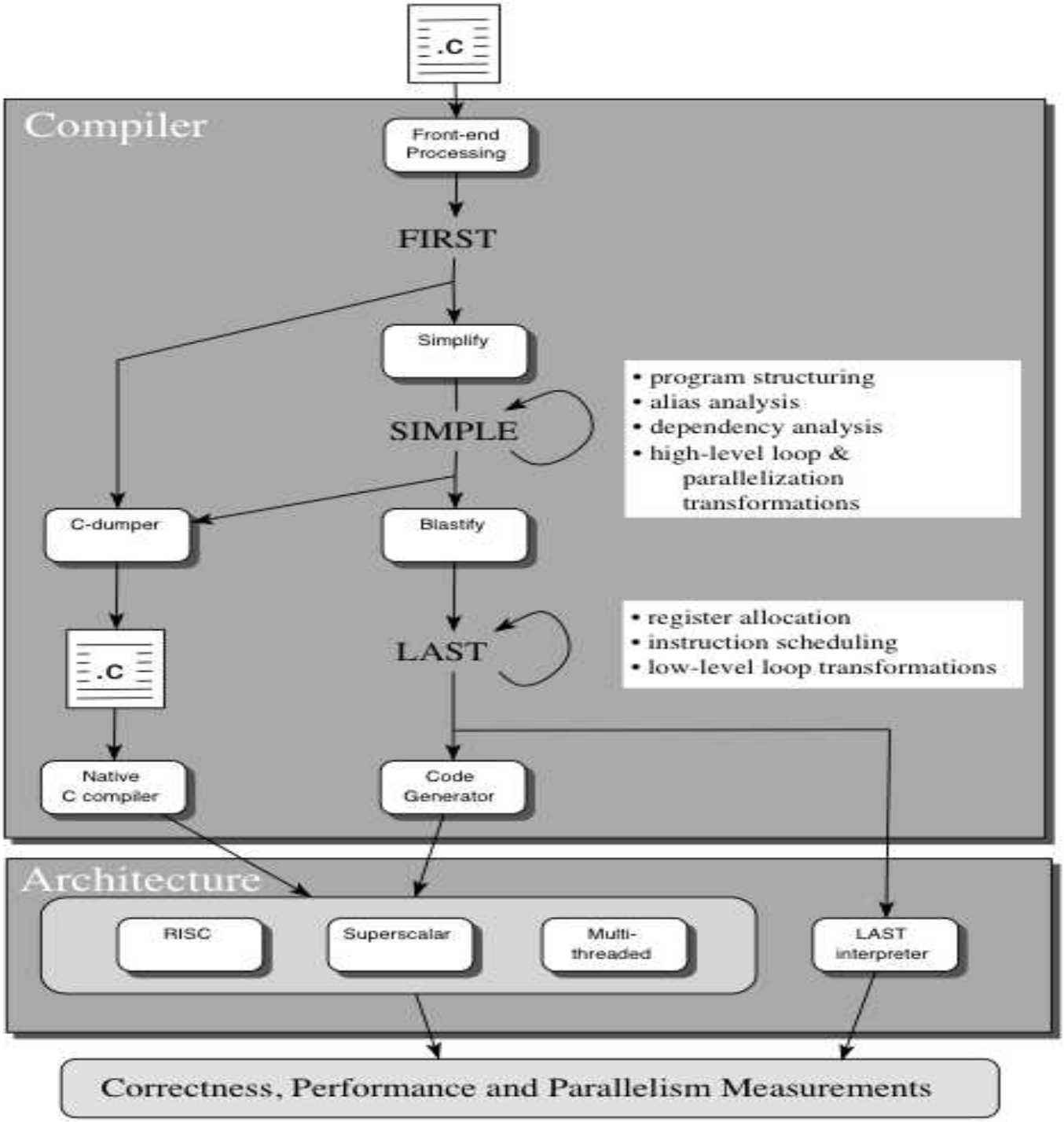


Figure 4.1: Vue générale du compilateur McCAT

4.3 Problématique

On pourrait analyser un programme en vue de déterminer les paires d'aliasing sans prendre en considération le flot de données ni le contexte des procédures, ce qu'on appelle l'analyse «context-insensitive flow-insensitive», comme c'est le cas des analyses de Steensgaard et Andersen, qui ont l'avantage d'être rapides et simples. Mais, en privilégiant une analyse efficace, on perd au niveau

de la précision; le compromis entre l'efficacité et la précision dépend des besoins de l'application.

Dans ce qui suit nous allons nous intéresser à une analyse plus précise qui prend en considération le flot des données et le contexte des procédures.

En effet le résultat de chaque procédure diffère selon le contexte dans lequel elle est invoquée. L'exemple suivant où on fait appel 3 fois à la même fonction f(), qui incrémente simplement son argument de un, démontre qu'on doit différencier entre chaque appel, même si la valeur de retour est identique.

```
for (i = 0; i < n; i++) {
 C1 :        t1 = f(0);    //f retourne 1 au niveau du site d'appel c1
 C2 :        t2 = f(243);  //f retourne 244 au niveau du site d'appel c2
 C3 :        t3 = f(243);  //f retourne 244 au niveau du site d'appel c3
 }
```

Une analyse imprécise aurait conclu que f retourne soit 1 ou 244 de n'importe quel site d'appel. D'autre part le langage C pose des difficultés comme :

- l'opérateur &;
- plusieurs niveaux de référencement **a;
- les pointeurs sur les fonctions int(*p)(int).

Afin de pouvoir différencier entre les sites d'appels des fonctions et traiter les difficultés que pose le langage C, on doit mettre en oeuvre des analyses précises, interprocédurales, et sensibles au contexte.

4.4 La représentation intermédiaire SIMPLE

L'analyse des pointeurs d'Emami a été implémentée dans compilateur McCAT qui traduit les programmes C dans son propre langage intermédiaire SIMPLE. La représentation SIMPLE introduit des variables temporaires pour traiter les tableaux, les doubles pointeurs ainsi que les arguments des fonctions qui sont de type pointeurs.

Le but de cette représentation est de transformer les expressions complexes en expressions basiques pour faciliter leur analyse; dans la suite on introduit des exemples illustrant comment au niveau du SIMPLE on représente les tableaux, les appels aux fonctions et les doubles pointeurs.

Les tableaux	x = a[5].b.c	temp0 = &a[5] a = (*temp0).b.c
Les doubles pointeurs	a = **b	temp0 = *b a =*temp0
Les fonctions avec un pointeur en argument	f(3, &a, *b)	temp0 = &a temp1 = *b f(3, temp0, temp1)
Les fonctions avec un tableau en argument	f(a.b, c[7], «abc»)	temp0 = a.b temp1 = c[7] temp2 = «abc» f(temp0, temp1, temp2)

La représentation SIMPLE permet une représentation de la composition du programme où le flot de contrôle est explicite, ce qui permet par exemple d'analyser une boucle en analysant simplement ses composants : la condition et le corps de la boucle. De cette représentation découlent trois principaux avantages : le flot de contrôle est structuré et explicite; les outils d'analyse structurée peuvent être utilisés pour analyser le flot de contrôle structuré; et enfin il est facile de retrouver et transformer les boucles imbriquées.

4.5 Abstraction de l'espace pile et les règles basiques de l'analyse

Traditionnelement les analyses d'aliasing représentent les alias sous forme d'ensembles de paires d'alias. Ici on présente une méthode pour sur estimer les relations entre variables pointeurs du langage C. Elle consiste à abstraire les relations entre les espaces mémoires dans la pile. On effectue une analyse structurée pour approximer les relations entre les espaces abstraits de la pile.

L'abstraction de la pile se fait par:

- association de noms symboliques aux variables de type pointeur;
- introduction d'une propriété de précision $rel \in \{D, P\}$;
- relation points-to PI = {(x,y,rel)}.

On dit que x pointe vers y certainement(D) ou possiblement(P); ceci se traduit au niveau de la pile par un lien de x vers y comme le montre la figure 4.2:

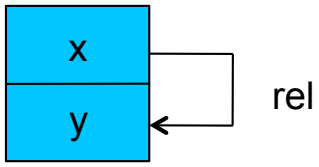


Figure 4.2: L'abstraction au niveau de la pile

L'analyse possède les deux propriétés suivantes :

Propriété 1 : Chaque emplacement réel dans la pile, qu'il soit la source ou la cible d'une référence pointeur, à un point p du programme, est représenté par exactement un seul emplacement abstrait nommé de la pile.

Propriété 2 : Chaque emplacement abstrait nommé de la pile, à un point p du programme, représente un ou plusieurs emplacements réels de la pile.

Chaque emplacement abstrait correspond au nom d'une variable locale, globale, paramètre ou un nom symbolique, ce qui permet de garantir qu'on fournit toutes les relations «points-to» en utilisant les emplacements abstraits nommés de la pile.

L'analyse s'effectue au niveau de chaque point du programme où on collecte l'information «points-to» qui abstrait la relation entre chaque paire d'emplacements abstraits de la pile. Des propriétés précédentes découlent les définitions suivantes :

Définition 1 : L'emplacement abstrait x de la pile pointe certainement vers l'emplacement abstrait de la pile y, en fonction d'un contexte d'invocation particulier, si x et y représentent exactement un seul emplacement réel de la pile dans ce contexte et l'emplacement réel qui correspond à x contient l'adresse de l'emplacement réel correspondant à y; la relation est représentée par (x,y,D).

Définition 2 : L'emplacement abstrait x de la pile pointe peut être vers l'emplacement abstrait de la pile y, en fonction d'un contexte d'invocation particulier, s'il est possible que l'un des emplacements réels de la pile correspondants à x contienne l'adresse d'un des emplacements réels correspondants à y dans ce contexte; la relation est représentée par (x,y,P). Pour mieux comprendre ces définitions, considérons l'exemple de la figure 4.3.

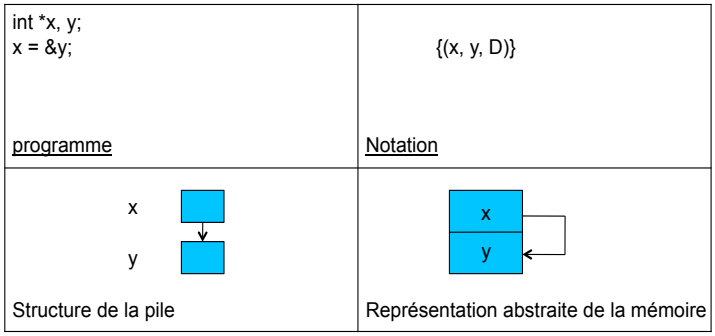


Figure 4.3: Illustration de la relation(x,y,D)

L'exemple illustre la relation D= «definetely» où on est sûr que la variable x pointe vers la variable y. Au niveau du deuxième exemple de la figure 4.4 on analyse un programme avec des branchements conditionnels qui aboutissent à des relations de type (x,y,P) avec P=«possibly».

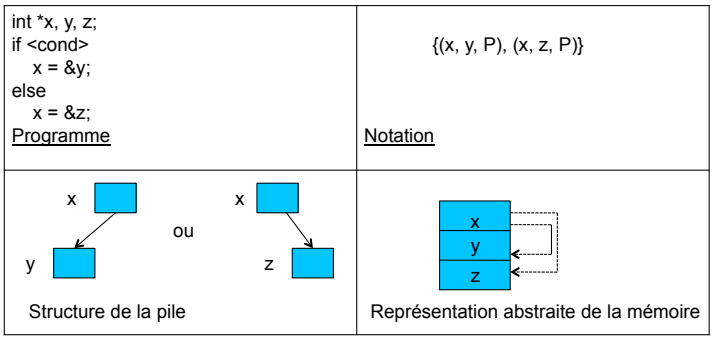


Figure 4.4: Illustration de la relation(x,y,P)

Dans cette section nous avons présenté brièvement l'analyse intraprocédurale des pointeurs. On s'est intéressé aux blocs de base qui contiennent des instructions impliquant des variables scalaires, sans s'intéresser aux tableaux, ni aux structures agrégées ni aux appels de fonctions.

Dans le tableau de la figure 4.5, on définit les règles à appliquer pour chaque bloc de base, les règles sont détaillées, plus tard, au niveau de l'annexe.

lhs \ rhs	&y	y	*y
x	Règle 1	2	3
*x	4	5	6

Figure 4.5: Les règles des blocs de base

L'exemple de la figure 4.6 illustre l'application de la règle 1 qui est la suivante:

Règle 1 : < x = &y >

$kill = \{(x, x1, rel)|(x, x1, rel) \in input\}$;

$gen = \{(x, y, D)\}$;

$PI = (gen \bigcup (input - kill))$

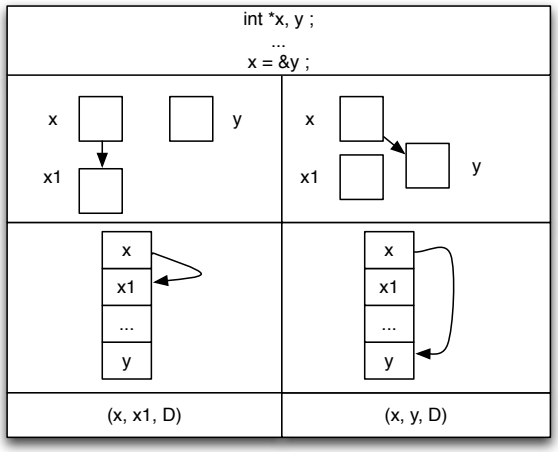


Figure 4.6: Application de la règle 1

En appliquant la règle 1 avec comme :

- ensemble *input* la relation (x,x1,D);
- ensemble *kill* la relation (x,x1,D);
- ensemble *gen* la relation (x,y,D).

On obtient en retour la relation (x,y,D). Donc pour effectuer l'analyse intraprocédurale il suffit d'appliquer la règle adéquate pour chaque affectation.

Ceci concernait les blocs de base où il n'y a pas de structures de contrôle ou de conditions comme «if» ou la boucle «while». Mais quand on doit traiter de telles instructions la recherche d'un point fixe s'impose. Ceci est illustré par le graphe de la figure 4.7 qui met en évidence le déroulement de l'algorithme qui analyse le corps de la boucle «while».

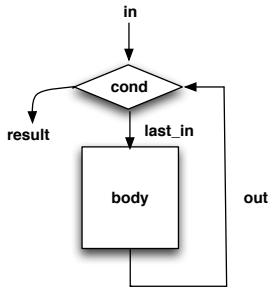


Figure 4.7: Déroulement de l'algorithme sur la boucle while

L'algorithme est le suivant :

```
process_while(cond, body, in) {
  {    last_in = in ;
    out = points_to(body, in) ;
    in = merge_info(in, out) ;
  } while (last_in != in) ;
  result = in ;
  return(result) ;
}
```

L'exemple du programme suivant montre la recherche du point fixe pour une boucle «while» comportant une instruction de type x = &y où il faut appliquer la règle 1.

```
int  *b, c, d;
b = &c ;      /* S1 */
while (cond){  /* S2 */
  b = &d ;    /* S3 */
 }            /* S4 */
```

Première approximation	Deuxième approximation
S1 : { (b, c, D) }	S1 : { (b, c, D) }
S2 : { (b, c, D) }	S2 : { (b, c, P), (b, d, P) }
S3 : { (b, d, D) }	S3 : { (b, d, P), (b, c, P), (b, d, P)}
S4 : { (b, c, P), (b, d, P) }	S4 : { (b, c, P), (b, d, P) }
	→ Point fixe atteint

4.6 L'analyse interprocédurale

Dans ce qui suit, nous allons nous intéresser à une analyse interprocédurale de la pile abstraite, qui prend en considération les appels aux fonctions[9] et par conséquent présente plus de difficultés comme le support du mappage entre les paramètres formels et effectifs ou bien l'effet des procédures sur les variables globales.

4.6.1 Le graphe d'invocation

L'analyse utilise une structure de données bien spécifique qui capture l'ordre d'activation et la structure des appels des fonctions d'un programme donné.Cette structure est le graphe d'invocation dont la stratégie adoptée est de représenter explicitement tous les chemins d'invocation des fonctions, comme pour les fonctions f et g du programme suivant. En utilisant le graphe d'invocation on peut différencier les appels d'une même procédure (appels à la fonction g()) et distinguer entre deux invocations d'une même fonction d'un même site d'appel qui sont atteignables selon des chaînes d'invocation différentes (appel à la fonction f()). On peut utiliser le graphe d'invocation pour

```
main()
{
  g();
  g();
}
g()
{
  f();

}
```

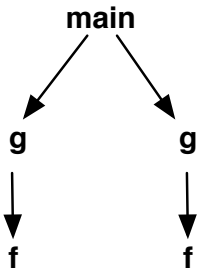


Figure 4.8: Le graphe d'invocation

représenter les fonctions récursives aussi. Dés qu'on rencontre un deuxième appel de la même fonction sur le même chemin d'invocation on le renomme en noeud approximatif puis on renomme le noeud ancêtre en noeud récursif. Ainsi pour le programme suivant :

```
main() {
  f() ;
}

f(){
  if(cond)
    f() ;
}
```

On obtient un graphe d'invocation de la figure 4.9 :

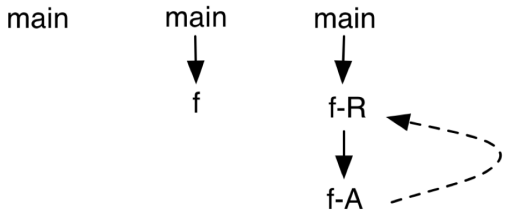


Figure 4.9: Le graphe d'invocation pour une fonction récursive

4.6.2 L'analyse interprocédurale et le graphe d'invocation

L'analyse utilise le graphe d'invocation pour suivre la séquence d'appels des fonctions comme le montre la figure 4.10. En fait l'analyse commence par la fonction main. Chaque instruction est

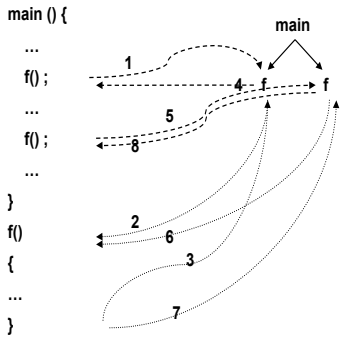


Figure 4.10: Les séquences d'appels dans un graphe d'invocation

analysée intraprocéduralement jusqu'à atteindre un appel à une fonction. C'est là que l'analyse fait appel au graphe d'invocation. Si c'est un noeud ordinaire, on analyse le corps de la fonction et le résultat est retourné via le graphe vers le site d'appel. Après, l'analyse se poursuit dans le corps du main jusqu'à atteindre un autre appel et ainsi de suite jusqu'à terminer tout le programme.Contrairement à l'analyse de SUIF, on n'utilise pas de fonction de transfert mais une réanalyse de l'appelée.

Quant aux fonctions récursives, leur traitement est similaire à celui de la boucle «while» où la recherche d'un point fixe s'impose. Cependant reste à noter qu'il n'y a pas d'analyse du corps de la fonction au niveau du noeud approximatif et que le processus de mappage reste le même. Cette démarche garantit que l'information «points-to» provenant de différents sites d'appels ne va pas être utilisée en même temps pour collecter de nouvelles information «points-to» et que cette dernière retourne toujours au site d'appel approprié.

Les grandes lignes de l'algorithme

L'algorithme de la figure 4.11 est une version simplifié de la fonction func_pts_to(), qui permet le calcul interprocédural des informations «points-to» en absence de récursivité.

Quant au processus de mappage entre les variables formels et effectifs, il est illustré par la figure 4.12.

Le processus de mappage compare les paramètres actuels et formels et établit le lien entre eux en appliquant la même règle qu'au niveau de l'analyse intraprocédurale qui stipule que le paramètre doit pointer vers le même emplacement que l'argument correspondant. Le processus de la fonction analyse intraprocéduralement le corps de la fonction et le processus de mappage inverse prend le résultat de la fonction traitée et change les noms des variables de l'appelée vers l'appelant.

Afin de suivre le processus de mappage, prenons l'exemple de la figure 4.13 :

Au niveau de la pile on aura les relations suivantes :

Au niveau de l'instruction S1, avant d'analyser la fonction f(), on a les relations (a,y,D) et (x,y,D). On effectue par la suite le processus de mappage sur les paramètres: pour cela on ajoute

```
/* la fonction reçoit comme paramètres le noeud d'appel au niveau de
   l'appelant et l'information «points-to» en entrée */
func_pts_to(call_expr_node, in_data)
{
  /* on récupère le noeud du graphe d'invocation associé à l'appelé à
     partir de l'appelant*/
  ig_node = get_related_ig_node(in_data_cur_ig_node,call_expr_num);
  /* récuperer la liste des arguments de la fonction*/
  arg_lst = get_arg_lst(call_expr_node);
  /* établir un mappage entre les informations «points-to» en entrée
     (in_data) et retourner l'information «points-to» mappée*/
  [map_info, func_in_data.in] = map_process(func_node, arg_lst, in_data.in);
  /*récuperer la liste des arguments de la fonction*/
  arg_lst = get_arg_lst(call_expr_node);
  /* récuperer le corps de la procédure*/
  func_body = get_func_body(func_node);
  /* analyser intraprocéduralement le corps de la fonction et
     retourner l'information  «points-to» */
  func_out_data = points-to(func_body, func_in_data);
  /* sauvegarder l'information  «points-to»  au niveau du noeud du
     graphe d'invocation*/
  save_out_info(ig_node, func_out_data.in);
  /* lancer le processus de unmappage qui à partir des informations
     «points-to» de l'appelé et de l'appelant va créer le nouvel
     ensemble  «points-to» */
  out_data.in = unmap_process(in_data, func_out_data.in, map_info);
  retrun(out_data);
}
```

Figure 4.11: L'algorithme interprocédural d'Emami

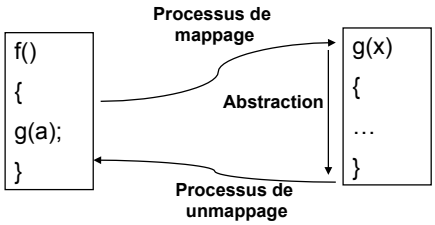


Figure 4.12: Le processus de mappage

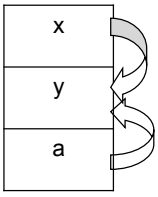
les relations «points-to» qui résultent de l'affectation m=a; en d'autres termes m doit pointer vers les mêmes emplacements que a. On obtient comme résultat la relation (m,y,D).

Maintenant pour illustrer le processus de «mappage inverse» prenons l'exemple suivant où il n'y a pas de relations «points-to» avant l'appel à la fonction f() au niveau de l'instruction S1; c'est pour cette raison que le processus de mappage n'ajoute aucune information à la fonction f(). Le programme est celui de la figure 4.14 :

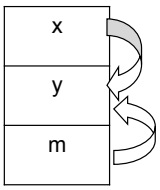
Au niveau de la pile, on aura les relations suivantes :

```
int *x,y ;
main()
{
  int *a ;
  a = &y ;
  x = &y ;
  f(a) ;  /* S1 */
}
f(m)
int *m;  /* S2 */
{
  ...
    }
```

Figure 4.13: Le programme à abstraire



La pile abstraite de main au niveau de S1

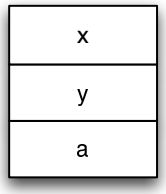


La pile abstraite de f() au niveau de S2

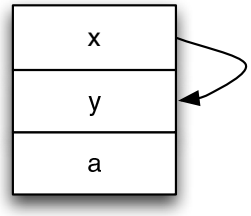
```
int *x,y ;
main()
{
  int *a ;
  f() ;        /* S1 */
  a = x ;      /* S3 */
               /* S4 */
}

f()
{
  x = &y ;
  /* S2 */
}
```

Figure 4.14: Le programme à analyser

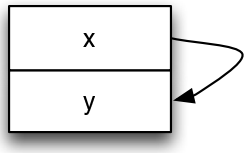
Au niveau de S2 on obtient la relation (x,y,D). Puis au niveau de S3 on applique le processus de «mappage inverse» pour obtenir les relations au niveau de la pile. Après S3 on obtient: (a,y,D) puisque x pointe vers y, la variable a va aussi pointer vers y.



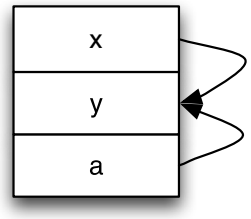
Au niveau de S1 avant l'appel à f



Au niveau de S3 après l'appel à f



Au niveau de S2



Au niveau de S4

4.6.3 Support des pointeurs sur les fonctions

Une autre difficulté qu'on peut rencontrer est constitué par les pointeurs sur fonctions qui, contrairement aux appels des fonctions standards, ne peuvent en général être associés à une seule fonction au moment de la compilation. En présence de pointeurs sur fonctions le graphe d'invocation ne peut être construit par une simple passe sur le programme. Afin de pouvoir supporter ces fonctions, on suit les étapes suivantes :

- construction d'un graphe incomplet au niveau des appels à des pointeurs sur fonction;
- Analyse des «points-to» en utilisant ce graphe incomplet;
- détermination des fonctions vers lesquelles pointe un pointeur sur fonctions en se basant sur les «points-to» déjà calculés;
- mise à jour du graphe d'invocation pour indiquer que l'appel peut invoquer une de ces fonctions; simultanément chaque fonction est analysée dans le contexte de l'appel;
- fusion de tous les «points-to» en sortie obtenus par analyse des fonctions invoquées.

Pour le programme de la figure 4.15 on peut effectuer une analyse conservatrice qui stipule que fp() peut être n'importe quelle fonction du programme: dans ce cas là g() ou bien h(). Quand on a plusieurs choix, il faut fusionner les résultats des différents appels, ce qui peut aboutir à des résultats moins précis.

En adoptant cette approche conservatrice, on aboutit aux résultats de la figure 4.16

```
int *x,
int *x, y ;
int *a, b ;
main() {
  void (*fp) ();
  fp = g ; /* S1 */
  fp() ;   /* S2 */
}
g()
{
  x = &y ;
}
h()
{
  a = &b ;
}
```

Figure 4.15: Programme avec pointeur sur les fonctions

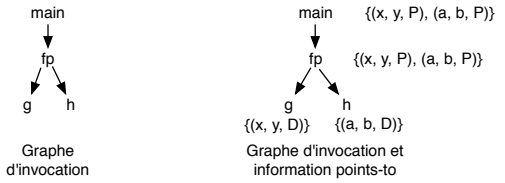


Figure 4.16: Approche conservatrice

La solution adoptée par l'analyse est de mettre à jour le graphe d'invocation pendant que l'analyse des pointeurs est encore en cours. D'abord le graphe est construit avec un parcours en profondeur, puis on le laisse incomplet aux niveaux des points où on rencontre un appel à un pointeur sur une fonction. Après, l'analyse des pointeurs est effectuée en utilisant ce graphe incomplet; une fois que l'appel au pointeur sur la fonction est rencontré, toutes les fonctions vers lesquelles il peut pointer selon l'analyse des pointeurs sont déterminées. Le graphe d'invocation est mis à jour pour indiquer que toutes ces fonctions peuvent être invoquées à partir de ce site d'appel. L'analyse des pointeurs considère que toutes ces fonctions peuvent être invoquées et en conséquence fusionne leur information en sortie afin d'obtenir l'information «points-to» du point qui vient juste après l'appel.

Pour l'exemple, on construit d'abord un graphe incomplet. Ensuite l'analyse des pointeurs est effectuée. Quand on rencontre l'appel au pointeur sur la fonction fp() au niveau de S2, on utilise l'information (fp,g,D) pour mettre à jour le graphe en ajoutant g comme fils du noeud fp. On continue après l'analyse des pointeurs pour obtenir des résultats plus précis que l'approche conservatrice.

4.7 Le traitement des tableaux

Au niveau de l'analyse, les tableaux sont traités comme des variables scalaires en rajoutant des règles spécifiques au traitement des blocs basiques introduit précédemment. En effet on traite a[i]

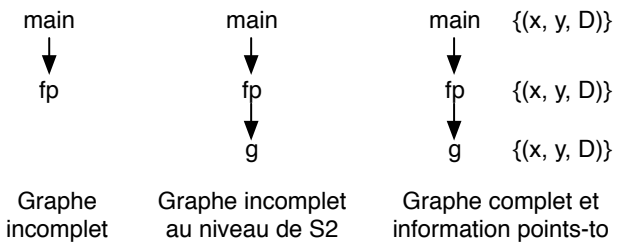


Figure 4.17: Approche de l'analyse

comme une variable scalaire pour i = 0; sinon la relation a comme précision la valeur «possibly» et par conséquent plus de parallélisation possible. Prenons comme exemple le programme de la figure 4.18 :

```
main() {
  struct {
   int *f1 ;
  } x, *z ;
 int y[70], w ;
  z =&x ;            /* S1 */
  (*z).f1 = &w ;      /* S2 */
  x.f1 = &y[0] ;     /* S3 */
}
```

Figure 4.18: Traitement des tableaux

L'analyse définit la règle suivante à appliquer pour le cas de l'instruction S3, où on est en présence d'une structure de données x avec un champ nommé a :
Règle 1.8 appliquée au niveau de S3
$kill = \{(x.a, x1, rel)|(x.a, x1, rel)\}$
$gen = \{(x.a, y, first\_elem([j]))\}$
$PI = gen \bigcup (input - kill)$
En appliquant la règle 1.8 on obtient les ensembles suivants :
$input = \{(z, x, D), (x.f1, w, D)\}$
$kill = \{(x.f1, w, D)\}$
$gen = \{(x.f1, y, D)\}$
$output = \{(z, x, D), (x.f1, y, D)\}$

4.8 Le traitement des structures

Les structures agrégées sont composées de différents emplacements dans la pile; le but de l'analyse est d'avoir un emplacement nommé pour chaque champ. Dans le but d'avoir un emplacement nommé pour tous les champs possibles de la structure, on énumère tous ses champs, comme par exemple pour la structure root de la figure 4.19 dont la pile est illustrée par la figure 4.20:

```
struct {
  int int *field1 ;
  float field2 ;
} root ;
```

Figure 4.19: Structure de données

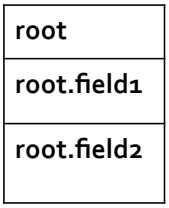


Figure 4.20: La pile abstraite pour une structure agrégée

Mais certaines structures sont définies récursivement, par exemple les listes chaînées comme la liste foo suivante :

```
struct foo{
  int data ;
 struct  foo *next ;
} cell ;
```

Pour ces structures on énumère comme décrit précédemment tous les champs de la structure et on introduit une variable invisible cell.1_next qui va pointer vers cell comme le montre la figure 4.21 :

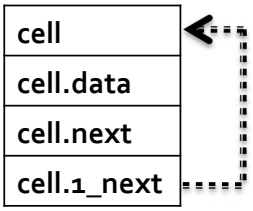


Figure 4.21: La pile abstraite pour une structure agrégée récursive

4.9 Gestion des données stockées dans le tas

Concernant le traitement des données qui pointent vers le tas (heap), l'analyse adopte une approche conservatrice en introduisant un emplacement unique, non typé, nommé «heap» dans la pile. Au niveau de l'emplacement «heap» on ne peut avoir que des relations «possibly».Ainsi pour le programme de la figure 4.22 :

On obtient la pile suivante :

4.10 Étude de cas

Dans cette section on exécute à la main l'analyse de Emami appliquée à un programme C nécessitant une analyse interprocédurale, un traitement des structures agrégées récursives ainsi qu'un

```
typedef struct foo {
  int a ;
  int *b ;
} FOO;
void main()
{
  FOO *x ;
  FOO *y ;
  x = (FOO*) malloc (sizeof(FOO)) ; /* S1 */
  y =& x->a ;                       /* S2 */
}
```

Figure 4.22: Données allouées dans le tas

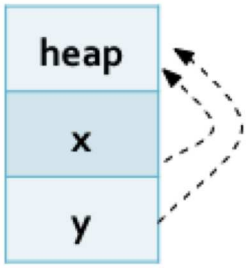


Figure 4.23: Les relations au niveau du tas

traitement des données au niveau du tas. Le programme qu'on veut analyser est celui de la figure 4.24.

```
#include <stdlib.h>

struct cons_t {
  int ** ppval;
  struct cons_t * next;
};

static int a = 1, c = 3;

struct cons_t * cons(int ** ppval, struct cons_t * next)   /* S13 */
{
  struct cons_t * p = (struct cons_t *) malloc(sizeof(struct cons_t)); /* S14 */
  p->ppval = ppval;                                                   /* S15 */
  p->next = next;                                                     /* S16 */
  return p;                                                           /* S17 */
}

int main(void)
{
  int b = 2, d = 4;

  /* pointeurs... */
  int * pb = &b;    /* S1 */
  int ** ppb = &pb; /* S2 */
  int * pa = &a;    /* S3 */
  int ** ppa = &pa; /* S4 */
  int * pc = &c;    /* S5 */
  int ** ppc = &pc; /* S6 */
  int * pd = &d;    /* S7 */
  int ** ppd = &pd; /* S8 */


  struct cons_t * l = cons(ppb, NULL);  /* S9 */
  l = cons(ppc, l);                     /* S10 */

  l->ppval = ppb;   /* S11 */

  *(l->ppval) = pa; /* S12 */

  return  0;
}
```

Figure 4.24: Programme à analyser

Avant d'entamer l'analyse du programme il est important de rappeler que la représentation intermédiaire SIMPLE simplifie les instructions complexes : ainsi les instructions de type l → ppval = ppb seront remplacées par *(l).ppval = ppb. Un autre détail important est celui des règles à appliquer; au niveau de la fonction main (les instructions S1 à S8) la règle à appliquer est la règle 1 introduite précédemment. Quant aux instructions impliquant des structures, la règle à appliquer est la suivante :

règle < (*x).a, y >

$kill = \{(x1.a, x2, rel) | (x, x1, D), (x1.a, x2, rel) \in input\}$

$gen = \{(x1.a, y, rel1 \bowtie rel2)\}$

$changed\_input = (input - \{(x1.a, x2, D) | (x, x1, P), (x1.a, x2, D) \in input\})$

$\cup \{(x1.a, x2, P) | (x, x1, P), (x1.a, x2, D) \in input\}$

$points - to \;\; PI = gen \cup (changed\_input - kill)$

Étant donné qu'une analyse interprocédurale est nécessaire, on commence par construire le graphe d'invocation illustré par la figure 4.25 :

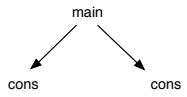


Figure 4.25: Le graphe d'invocation

On commence l'analyse par la fonction main de l'instruction S1 à S8 (avant l'appel à la fonction cons) qui sont des instructions basiques où on applique la règle 1 pour aboutir à l'ensemble des «points-to» suivant :

{ (pb, b, D), (ppb, pb, D), (pa, a, D), (pc, c, D), (ppc, pc, D), (pd, d, D), (ppd, pd, D) }.

Quand l'appel à la fonction cons() est rencontré au niveau de l'instruction S9, on fait appel à la fonction func_points_to() avec en paramètres la fonction cons et l'ensemble «points-to» calculé précédemment. La fonction récupère le noeud représentant la fonction *cons* ainsi que ses arguments afin d'effectuer un mappage entre les arguments formels et effectifs via la fonction map_process() qui fait appel à son tour à la fonction map_function. Le retour de cette fonction est un ensemble de relations entre les variables invisibles de l'appelée et les variables locales de l'appelant; c'est à ce niveau qu'on traite les variables statiques, en introduisant des variables invisibles les représentant au niveau de la fonction. Les variables invisibles sont nommés en associant un chiffre au nom du parmétre formel. On aura ainsi comme information de mappage : 1_ppval → pb et 2_ppval → b. On vérifie aussi à ce niveau s'il y a des variables globales à traiter, qui nécessitent à leur tour l'introduction de variables globales. Dans notre cas il n'y a pas de variables globales à traiter.

L'analyse de la fonction se poursuit. Etant donné que ce n'est pas une fonction récursive, on traite le corps de la fonction comme un ensemble de blocs basiques. L'instruction S14 aboutit aux relations «points-to» suivantes : (p, heap, P). Les instructions S15 et S16 ne génèrent pas de nouvelles relations «points-to». Avant de revenir vers le site de l'appelant (la fonction main dans ce cas là), on appelle la fonction unmap_process en lui passant comme paramètres l'information de mappage déjà calculée ainsi que les relations «points-to» générées au niveau de la fonction appelée pour établir cette fois ci un lien entre les paramètres effectifs et formels (la fonction inverse de la

fonction de mappage).

Au niveau du site d'appel, l'instruction S9, on obtient la relation «points-to» suivante (l, heap, p). Au niveau de l'instruction S10, on rencontre un nouveau appel à la fonction cons, on relance alors l'analyse de la fonction. Pour analyser la fonction, les mêmes étapes que la première fois sont effectuées : on commence par un appel à la fonction func_points_to qui récupère sous forme d'une liste les arguments ppc et l, ainsi que les paramètres ppval et next. Ensuite, la fonction func_points_to fait appel au processus de mappage, qui va créer deux variables invisibles pour pc et c. En conséquence l'information de mappage sera : 1_ppval → pc et 2_ppval → et l'analyse du corps de la fonction se poursuit avec comme relation «points-to» : (ppaval, 1_ppval, D), (1_ppval, 2_ppval, D) et (next, heap, P). Le deuxième appel à la fonction cons ne va pas générer de nouvelles relations «points-to»; on aura la relation (p, heap, P), qui après unmappage donnera comme résultat la relation (l, heap, P).

4.11 Critique de l'analyse

L'analyse d'Emami a l'avantage d'être sensible au contexte grâce à sa structure partuculiére de representation des contextes des fonctions qui est le graphe d'invocation. Mais cet avantage peut se voir transformer en inconvenient, car en presence d'un grand nombre de fonctions, l'algorithme peut devenir de complexité exponentielle. Par conséquent l'algorithme ne peut traiter des programmes de taille trés importantes, de millions de lignes de code.
Outre la dégradation de ses perfermances en presence de programmes de taille importante, l'analyse traite les données allouées au niveau du tas d'une maniére peu précise. En effet, le tas est considéré comme un seul emplacement, qui peut pointer vers des emplacements de la pile abstraite et peut être pointés par plusieurs d'entre eux. Ceci peut introduire des imprécisions sur l'information «points-to».

Chapter 5

Analyse des besoins spécifiques au traitement de signal

5.1 Introduction

Dans ce chapitre, nous allons nous intéresser au projet PIPS afin de déterminer ses besoins en terme d'analyse de pointeurs. On s'intéresse plus particulièrement aux applications qu'on voudrait analyser, optimiser et paralléliser en utilisant le logiciel PIPS. Ces application sont essentiellement des applications de traitement de signal. Le traitement du signal est la discipline qui développe et étudie les techniques de traitement (filtrage, détection...), d'analyse et d'interprétation des signaux. Une de ses applications est le traitement d'images, qui désigne une discipline des mathématiques appliquées qui étudie les images numériques et leurs transformations, dans le but d'améliorer leur qualité ou d'en extraire de l'information. L'amélioration ou l'extraction de l'information à partir des images se fait par l'application de filtres, qui sont des matrices appliquées à l'image représentée elle aussi sous la forme d'une matrice. Donc ces applications contiennent beaucoup de calcul matriciel, ce qui fait d'elles les candidates idéales pour la parallélisation des données au niveau des boucles. Étant donné que les tableaux en langage C peuvent être aussi accèdes par des pointeurs, l'étude des pointeurs peut améliorer la phase de parallélisation et apporter des améliorations considérables. En effet, la parallélisation des données au niveau de boucles, nécessite une analyse des dépendances. L'analyse de dépendances peut être considérées comme une analyse cliente de l'analyse de pointeurs. Elle utilise cette information comme entrée lui permettant de déterminer si deux éléments du tableaux pointent vers la même zone mémoire. Outre que les pointeurs sur les tableaux, on recherche les autres structures accessibles par pointeurs, comme les structures agrégées ou bien les données allouées au niveau du tas par la fonction malloc par exemple, on éspere ne pas trouver cette structure dans les applications de traitement de signal. Cette recherche va se faire sur des applications de traitement de signal fournies au projet PIPS, dans le but d'être analysées et parallélisées. Ce chapitre introduit aussi d'autres analyses clientes qui pourraient exploiter les résultats des analyses de pointeurs, ainsi qu'un exemple montrant comment une analyse de dépendances pourrait utiliser l'analyse de pointeurs d'Emami, introduite au niveau du chapitre 4.

5.2 Impact sur les analyses clientes

Dans [17] Hind et Pioli effectuent une comparaison empirique de 5 analyses de pointeurs des programmes C. Les algorithmes varient dans leur utilisation de flots de données, des structures de données et dans leur complexité, allant du linéaire au polynomial. Outre la précision et l'efficacité, la comparaison prend en compte comment les 5 analyses affectent les clients typiques des analyses de pointeurs : l'analyse Mod/Réf, l'analyse des variables vivantes, l'identification des affectations mortes, la propagation des constantes conditionnelle et l'identification du code inaccessible. Pour pouvoir différencier les différentes analyses de pointeur, les auteurs [17] ont pris les facteurs suivants en compte:

- L'analyse interprocédurale des flots de données qui peut être soit «flow-sensitive» soit «flot-insensitive» selon que l'information de flot de contrôle d'une procédure est utilisée lors de l'analyse. L'ignorer et calculer un résumé conservatif permet d'avoir une analyse plus efficace mais moins précise.
- «Context-sensitivity»: le contexte des appels est-il pris en compte lors de l'analyse d'une fonction?
- «Heap-modeling» : les objets sont-ils nommés par site ou bien une analyse plus sophistiquée est elle effectuée?
- Modélisation du struct : les champs sont-ils distingués ou bien encapsulés et vus comme un seul objet?
- La représentation des alias : une représentation explicite ou bien compacte?

On maintient ces facteurs constants pour toutes les analyses : des analyses «flow-insensitive», les objets nommés par leur site, les champs encapsulés et une représentation compacte des alias.

Des exemples d'analyses clientes

Plusieurs analyses utilisent l'analyse «points-to» comme l'analyse Mod/Réf, l'analyse des variables vivantes,l'analyse «reaching-definition», la propagation de constante.

L'analyse Mod/Réf détermine quels objets peuvent être modifiés ou référencés au niveau de chaque noeud du graphe du flot de contrôle. Cette information est aussi utilisée par d'autres analyses comme l'analyse des «reaching-definition» et l'analyse des variables vivantes, toutes nécessaires dans le cadre de la parallélisation des programmes. Cette information est obtenue en visitant chaque noeud du graphe du flot de contrôle et en déterminant quels objets ont été modifiés ou référencés par le noeud.

L'analyse des variables vivantes détermine quels objets peuvent être référencés après un point du programme sans qu'il ait eu une redéfinition des objets. Cette information est utile pour l'allocation des registres et la détection des affectations mortes. L'analyse utilise l'information Mod/Réf : elle associe deux ensembles de variables vivantes avec chaque noeud du graphe de flot de contrôle pour représenter ce qui est vivant avant et après l'exécution du noeud. Tous les objets

de l'ensemble Réf d'un noeud sont vivants avant ce noeud. Un objet est tué s'il est modifié au niveau du noeud.

L'analyse «reaching-definition» détermine quelle définition des objets peut atteindre un point du programme. L'analyse utilise l'information Mod/Réf et associe deux ensembles des définitions accessibles pour chaque noeud du graphe de flot de contrôle. Les paramètres des procédures, les accès aux tableaux ainsi que les références indirectes peuvent introduire des alias et il devient difficile d'assurer qu'une affectation concerne une variable en particulier.

La propagation interprocédurale de constantes traque les valeurs des variables interprocéduralement à travers le programme et utilise cette information pour évaluer les branchements conditionnels. Cette analyse a été mise en oeuvre pour être combinée avec l'analyse des pointeurs de Choi et al's; elle utilise donc directement le résultat de l'analyse des pointeurs, contrairement à l'analyse «reaching-defintion» ou bien l'analyse des variables vivantes.

5.2.1 Les algorithmes

Voici les algorithmes triés par ordre croissant de précision :

- Adress-taken : un algorithme «flow-insensitive» qui enregistre toutes les variables dont les adresses ont été affectées à d'autres variables. Cette analyse est efficace parce qu'elle est linéaire en fonction de la taille du programme mais reste très imprécise.
- Steensgaard : calcule un seul ensemble de solutions pour le programme en entier. Le résultat est obtenu dans un temps presque linéaire.
- Andersen[4] : une implémentation itérative,«flow-insensitive», «context-insensitive» plus performante que Steensgaard, car elle ne fusionne pas les structures de données en un seul objet.
- Burke et al[12] : diffère de l'algorithme d'Andersen par le calcul des alias pour chaque procédure en plus du calcul des alias pour tout le programme.
- Choi et al's[6] : un algorithme «flow-sensitive» qui calcule un ensemble de solutions pour chaque point du programme en associant un ensemble d'alias avec chaque noeud du graphe du flot de contrôle.

5.2.2 Résultats de la comparaison des algorithmes

Une première mesure de la précision des algorithmes est le nombre d'objets aliasés à une expression de pointeur qui apparaît dans le programme. Cette métrique a montré que les analyse Andersen et Burke et al. donnaient le même niveau de précision.C'est pour cela que pour la suite de la comparaison ces deux analyses sont regroupées. La deuxième métrique permet d'estimer l'effet de la précision d'une analyse de pointeurs sur l'analyse Mod/Réf, elle estime la moyenne de la taille de l'ensemble Mod/Réf pour chaque noeud du graphe de flot de contrôle, on remarque :

- une différence substantielle entre l'analyse Adress- Taken et Steensgaard en faveur de Adress-taken;
- une différence mesurable entre l'analyse de Steensgaard et celle d'Andersen/Burke et al en faveur de Steensgaard;
- une légère différence entre l'analyse d'Andersen/Burke et al. et l'analyse de Choi et al. en faveur d'Andersen/Burke et al..

Une autre métrique qui permet d'estimer l'effet des analyses de pointeurs sur l'analyse des variables vivantes. On calcule la moyenne du nombre des variables vivantes au niveau de chaque noeud du graphe de flot de contrôle ainsi que la moyenne de ces moyennes. L'information sur les variables vivantes est utilisée pour identifier les affectations des variables qui ne sont jamais utilisées, l'expérience montre :

- Une différence substantielle entre l'analyse d'Adress-taken et Steensgaard en faveur de l'analyse Adress-taken pour l'identification des variables vivantes mais pas de différence au niveau de l'identification des affectations mortes.
- Une différence significative entre l'analyse Steensgaard et Andersen/Burke et al. en faveur de la première au niveau de l'identification des variables vivantes mais moins importante au niveau de l'identification des affectations mortes.
- Une légère différence entre Choi et al. et Andersen/Burke et al. pour l'identification des variables vivantes mais pas de différence au niveau de l'identification des affectations mortes.

En résumé une analyse de pointeurs précise améliore la précision de l'analyse des variables vivantes sauf pour l'analyse Choi et al. dont l'amélioration n'est pas significative. Par contre l'identification des affectations mortes a été très affecté par l'utilisation de différentes analyses de pointeurs.

La quatrième métrique porte sur l'amélioration de la précision de l'analyse «reaching definition», elle se calcule par la moyenne des définitions qui atteignent un noeud du graphe de flot de contrôle ainsi que la moyenne d'un flot de dépendances entre deux noeuds du graphe de flot de contrôle par fonction. On remarque :

- une différence significative entre l'analyse Adress-taken et Steensgaard en faveur de l'analyse Adress-taken.
- une différence mesurable entre l'analyse d'Andersen/Burke et al et Steensgaard en faveur de l'analyse d'Andersen/Burke et al. pour l'analyse des «reaching definition» ainsi que pour le flot de dépendances.
- Une différence négligeable entre l'analyse d'Andersen/Burke et al et Choi et al mais pas de différence au niveau de flot de dépendances.

En résumé chaque analyse plus précise améliore par rapport à la précédente le calcul des «reaching defintion», mais cette amélioration n'est pas ressentie au niveau du calcul du flot de dépendances. La dernière métrique vise à améliorer l'analyse de propagation de constante et la détection du code non exécuté(code mort), elle se fait en calculant le nombre d'expressions constantes. Au niveau des résultats on obtient :

- une différence significative entre l'analyse de Steensgaard et Adress-taken en faveur de Steensgaard mais pas d'amélioration significative pour la découverte des constantes.
- une différence négligeable entre l'analyse d'Andersen/Burke et al et Steensgaard.
- pas de différence entre l'analyse d'Andersen/Burke et al. et l'analyse de Choi et al..

En résumé à partir de l'analyse de Steensgaard la propagation de constante et la découverte de code mort ne tire plus avantage de l'amélioration de la précision des analyses des pointeurs.

5.3 Exemple d'application cliente : l'analyse des dépendances et l'analyse d'Emami

Une des applications de l'analyse des pointeurs est l'analyse des dépendances, dont la première phase est de générer les ensembles lecture/écriture, qui se fait en utilisant l'information «points-to» telle qu'elle est générée par l'analyse d'Emami au sein du compilateur McCAT. Le tableau suivant résume les ensembles de lecture\écriture pour une instruction s en utilisant l'information «points-to» PI, avec :

- Wp(s,PI)/Wd(s,PI) sont les ensembles écritures «possibles» ou «certaines» au niveau de l'instruction s avec PI comme entrée.
- Pp(x,PI)/Pd(x,PI) est l'ensemble des variables vers lesquelles x pointe possiblement ou certainement.

s	W_p(s,PI)	W_d(s,PI)	R(s,PI)
x = y x = unop y x = y binop z	φ φ φ	{x} {x} {x}	{y} {y} {y,z}
*x = y *x = unop y *x = y binop z	P_p(x,PI) P_p(x,PI) P_p(x,PI)	P_d(x,PI) P_d(x,PI) P_d(x,PI)	{x,y} {x,y} {x,y}
x = &y	φ	{x}	φ
x = *y	φ	{x}	{y} ∪ P(y,PI)

L'exemple de la figure 5.1 montre comment on utilise l'information «points-to» pour analyser les dépendances d'un programme :

Pour traiter le programme la représentation SIMPLE, une représentation intermédiaire introduite par le compilateur, introduit des variables temporaires qu'on a plusieurs niveaux de références. Au niveau de l'instruction S1 la variable temp0 est dans l'ensemble de lecture et toutes les variables vers lesquelles temp0 pointe sont dans l'ensemble d'écriture, dans ce cas là la variable c. On aboutit au programme de la figure 5.2.

Au niveau de la pile on obtient les relations «points-to» de la figure 5.3 :

Quant aux ensembles d'écriture lecture ils sont résumés dans le tableau de la figure 5.4 :

```
main() {
  int **a, *b, c ;
  a = &b ;
  b = &c ;
  **a = 7 ;
}
```

Figure 5.1: Le programme C à analyser

```
main()  {
  int **a, *b, c ;
  int temp0 ;
  a = &b ; /* S1 */
  b = &c ; /* S2 */
  temp0 = *a ;          /* S3 */
  *temp0 = 7 ;          /* S4 */
}
```

Figure 5.2: Programme mis dans la représentation SIMPLE

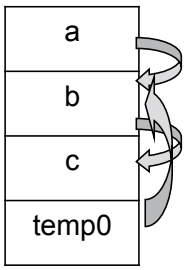


Figure 5.3: Les relations au niveau de la pile

S	Write(S)	Read(S)
a = &b	{a}	φ
b = &c	{b}	φ
temp0 = *a	{temp0}	{a,b}
*temp = 7	{c}	{temp0}

Figure 5.4: Les ensemble d'écriture/lecture

5.4 Les constructions syntaxiques rencontrées

Cette section a pour but d'analyser des applications de traitement de signal et d'en extraire les constructions syntaxiques utilisés des pointeur les plus fréquentes. Avec cette information, et après avoir étudier différentes analyses de pointeurs, on pourra en choisir une ou bien s'inspirer d'elles pour implémenter une analyse de pointeurs au niveau du projet PIPS.

5.4.1 Tableaux de structures

Le premier benchmark auquel on s'intéresse est le nouveau standard JPEG2000. C'est le nouveau système de codage d'images utilisant l'état de l'art des techniques de compression et basé sur la transformée en ondelettes. Son architecture devrait être appropriée à un grand nombre d'applications depuis les appareils photos digitaux jusqu'à l'imagerie médicale et d'autre secteurs clé[3]. Le code source de l'application fait partie du domaine publique ce qui nous autorise à l'exploiter (disponible au niveau de l'URL :http://www.ece.uvic.ca/ mdadams/jasper/).

En effet, le code source de l'application, qui une application type de traitement d'images, nous permet de déterminer les constructions syntaxiques, les plus fréquentes. Celles auxquelles on s'intéresse sont essentiellement qui font appel aux structures agrégées, et qui peuvent poser problème lors de l'analyse du programme dans le but de le paralléliser.

La structure récurrente, au niveau du code source du projet JPEG2000, est le tableau de structures. On retrouve la définition de cette structure au niveau du fichier jasper/jas_image.h.

```
main() {
/* Image format information. */

typedef struct {
  int id;
  /* The ID for this format. */
  char *name;
  /* The name by which this format is identified. */
  char *ext;
  /* The file name extension associated with this format. */
  char *desc;
  /* A brief description of the format. */
  jas_image_fmtops_t ops;
  /* The operations for this format. */
} jas_image_fmtinfo_t;
```

Figure 5.5: Définition de la structure jas_image_fmtinfo_t

Puis la déclaration d'un tableau de structure, de type jas_image_fmtpos_t au niveau du fichier jasper/jas_image.c.

```
static jas_image_fmtinfo_t jas_image_fmtinfos[JAS_IMAGE_MAXFMTS];
```

Figure 5.6: Déclaration de la variable statique jas_image_fmtinfos

Puis le tableau de structure est utilisé au niveau de la fonction jas_image_clearfmts().

```
/***** File format operations.*****/

void jas_image_clearfmts()
{
  int i;
  jas_image_fmtinfo_t *fmtinfo;
  for (i = 0; i < jas_image_numfmts; ++i) {
    fmtinfo = &jas_image_fmtinfos[i];
    if (fmtinfo->name) {
      jas_free(fmtinfo->name);
      fmtinfo->name = 0;
    }
    if (fmtinfo->ext) {
      jas_free(fmtinfo->ext);
      fmtinfo->ext=0;
    }
    if(fmtinfo->desc){
      jas_free(fmtinfo->desc);
      fmtinfo->desc = 0;
    }
  }
  jas_image_numfmts = 0;
}
```

Figure 5.7: La fonction jas_image_clearfmts()

De retour au tableau 3.1 de l'analyse de Wilson, on remarque que ce cas de figure a été pris en compte par la représentation des emplacements mémoire. En effet, pour un tableau de structures le champ offset est celui de la structure.

Quant à l'analyse d'Emami, étant donné qu'un tableau est une séquence de variables de même type, elle considère tous les éléments du tableau comme un seul espace au niveau de la pile abstraite. Ainsi l'analyse «points-to» définit les relations entre les tableaux en entier, puis cette information est transmise à l' analyse de dépendances. L'analyse des dépendances devra à son tour affiner ces relations au sein des éléments du tableau. L'analyse des «points-to» peut être considérée comme un prétraitement pour une analyse des dépendances entre les éléments de tableau dans des boucles imbriquées. Et dans le cas de la fonction jas_image_clearfmts() l'analyse d'Emami peut être utilisée pour paralléliser la boucle d'initialisation.

5.4.2 Structure de tableau

Toujours dans le cadre de l'imagerie, on s'intéresse à Fulguro, une bibliothèque de traitement d'images sous des contraintes temps-réel. Cette bibliothèque utilise des optimisations SIMD, quand cela est possible et offre la possibilité d'utiliser des threads, toujours dans le but d'accélérer le calcul[2].

L'analyse du code source de la bibliothèque, qui fait partie du domaine publique, a permis l'extraction de la structure FLGR_Data2D qui a parmi ses champs le champ row de type la structure FLGR_Data1D, toutes deux définies au niveau du fichier flgrCoreData.h.

```
typedef STRUCT {
int dim;
int size_struct;
void* array;
void* array_phantom;
}FLGR_Data1D;
typedef STRUCT {
int dim;
int size_struct;
int link_overlap;
int size_y;
int size_x;
FLGR_Data1D **row;
void** array;
}FLGR_Data2D;
```

Figure 5.8: La déclaration des structures de données

On s'intéresse plus particulièrement au champ row de type FLGR_Data1D, qui est utilisé au niveau du fichier flgrCoreData.c.

```
  FLGR_Data2D *flgr2d_create_pixmap_link(FLGR_Data2D *datain,
  int partsNumber, int partIndex, int overlapSize) {

  FLGR_Data2D *dat;

  int i,k,nbrow,startin;

  dat = (FLGR_Data2D*) flgr_malloc(sizeof(FLGR_Data2D));

  dat->link_overlap  = overlapSize;


  for(i=0 ; i<nbrow-overlapSize ; i++) {

    dat->row[i] = datain->row[i];

    dat->row[i]->ref2d = i;

    dat->array[i] = dat->row[i]->array;

 }
}
```

Figure 5.9: La fonction flgr2d_create_pixmap_link()

Il est essentiel pour l'analyse des dépendances au niveau des nids de boucles, qui permettra la parallélisation par la suite, de pouvoir déterminer si deux références pointent vers le même élément du tableau. Cet exemple présente une autre difficulté; chaque élément du tableau est en fait une référence à une structure de données. Cette structure est prise en compte par l'analyse de Wilson, au niveau du tableau 3.1. Mais cette prise en compte reste approximative;une référence à un tableau dans une structure peut avoir accès à n'importe quel élément de la structure en utilisant des indices dépassant les limites du tableau. Ceci est possible parce que le langage C n'effectue pas de vérification sur les indices de tableau, contrairement au langage java par exemple. L'analyse traite le tableau imbriqué dans une structure comme s'il englobait la structure en entier. Pour cela on met le pas dans la représentation des ensemble à une valeur différente de zéro. Ce traitement peut aboutir à des imprécisions. Au niveau de l'analyse d'Emami, on peut traiter avec plus de précision les éléments des structures mais pas les éléments des tableaux. Ainsi le tableau row ne sera pas traité en fonction de l'indice i. En effet, l'analyse distingue seulement entre l'indice zéro et tous le reste est considéré comme un seul élément. Cet exemple reste donc difficile à traiter et par conséquent à paralléliser.

5.4.3 Descripteur de tableau

Pour le cas du descripteur de tableau, on extrait l'exemple du code source du projet Ter@ps. L'exemple figure 5.10 est une fonction qui crée un tableau via un pointeur puis effectue des calculs sur les éléments du tableau au niveau d'une boucle. Cette construction est très récurrente dans les applications de traitement de signal,et qu'on voudrait traiter avec PIPS.

```
byte** bmatrix(long nrl, long nrh, long ncl, long nch)
/* ------------------------------------------------ */
/*allocate an uchar matrix with subscript range m[nrl..nrh][ncl..nch]*/
{
  long i, nrow=nrh-nrl+1,ncol=nch-ncl+1;
  byte **m;

  /* allocate pointers to rows */
  m=(byte **) malloc((size_t)((nrow+NR_END)*sizeof(byte*)));
  for(i=nrl+1;i<=nrh;i++)
    m[i]=m[i-1]+ncol;
  /* return pointer to array of pointers to rows */
  return m;
}
```

Figure 5.10: Exemple de descripteur de tableau

5.4.4 Pointeurs sur fonctions, structures récursives et allocation dynamique des données

Le calcul matriciel est dominant dans les applications de traitement de signal. Pour orienter notre choix d'analyse de pointeurs on a parcourut le code des applications à la recherche des constructions syntaxiques qui pourraient introduire des imprécisions sur les résultats. la première construction qui peut introduire des imprécisions est l'allocation dynamique des données (le tas ou le heap, alloué par la fonction malloc), qui après exploration des applications s'est révélée être d'une utilisation très peu fréquente. Pour gérer les emplacements au niveau du tas, l'analyse d'Emami choisit d'abstraire l'emplacement comme un seul bloc; contrairement à l'analyse de Wilson qui crée un emplacement pour chaque contexte. L'approche d'Emami, malgré son aspect conservatif est plus adéquate pour les applications de traitement de signal car elle est moins coûteuse.

La deuxième construction, qui elle aussi est peu utilisée dans les applications de traitement de signal est le pointeur sur fonction. Même si cette construction est très bien gérée par l'analyse d'Emami grâce au graphe d'invocation, elle est presque absente des applications de traitement de signal parce contrairement aux autres appels de fonctions, on ne peut pas déterminer au préalable la fonction cible.

En troisième lieu on s'est intéressé aux structures de données récursives. Même si les structures de données (le type *struct* du langage C) sont utilisées largement dans les applications, leur utilisation sous une forme récursive reste néanmoins limitée. L'absence presque totale de *malloc*, de pointeur sur fonction et de structures de données récursives facilite l'analyse de pointeur et par conséquent oriente notre choix sur une analyse qui traite ces constructions de façon conservatrice et traite avec plus de précision les autres constructions syntaxiques récurrentes citées dans la section 5.4.

5.5 Exemples de parallélisation C avec PIPS

Dans cette section nous entamons une autre partie expérimentale du stage. En effet, on met en place des programmes C contenant des constructions syntaxiques qui pourraient introduire des cas d'aliasing. On s'intéresse plus précisément aux effets mémoires calculés par PIPS. Les effets décrivent les opérations mémoires produites par une instruction. Mise à part l'adresse référence à laquelle l'opération est effectuée et sa nature (écriture ou lecture) on peut aussi distinguer entre les effets qui se produisent toujours et ceux qui peuventt se produire[14]. Les effets propres sont les références mémoires locales à un bloc d'instructions, comme l'écriture sur la partie gauche d'une affectation d'un indice de boucle. Les effets cumulés prennent en compte tous les effets de l'instruction y compris ceux des blocs imbriqués au sein d'une instrcution composée comme un test ou une boucle. Les effets propres et cumulés sont calculés par un parcours bottom-up du graphe d'appel.

5.5.1 Les régions convexes de tableaux

Outre les effets, on s'intéresse à l'analyse du flot de données des tableaux. Cette analyse requiert des informations intraprocédurale et interprocédurale. Au niveau de PIPS on calcule des résumés qui représentent exactement les effets des appels de procédures sur des ensembles d'éléments de tableau. Le calcul se fait par introduction de deux types de régions exactes, pour n'importe quel bloc d'instruction ou procédure :

- région IN qui contient les éléments importés du tableau ;
- région OUT qui représente l'ensemble des éléments vivants d'un tableau.

Les régions READ et WRITE

Une région de tableau est définie comme un ensemble de tableau décrit par un polyèdre convexe contenant des égalités et des inégalités qui lient les paramètres des régions (les variables phi) qui représentent les dimensions du tableau aux valeurs des variables scalaires entières du programme. Deux caractéristiques à prendre en compte :

- le type de la région : READ (R) ou WRITE (W) pour représenter les effets des blocs d'instructions et des procédures; IN et OUT pour représenter le flot des éléments du tableau;
- l'approximation de la région : EXACT quand la région représente exactement l'ensemble requis d'éléments du tableau, ou MAY si c'est une sous-approximation.

Prenons comme exemple la région:

<A(phi1,phi2)-W-MUST-phi1==I, phi1==phi2>

où phi1 et phi2 représentent respectivement la première et la deuxième dimensions de A. elle correspond à une affectation de l'élément A(I,I).

Les régions IN et OUT

Les régions READ et WRITE résument les effets exactes des blocs d'instructions et des procédures sur les éléments de tableau. Cependant, elles ne représentent pas le flot des éléments. C'est

dans ce but qu'on introduit les régions IN et OUT qui prennent en compte l'ensemble KILL du tableau. Les régions IN contiennent les éléments du tableau, dont les valeurs sont (EXACT) ou peut être (MAY) importées par le morceau de code actuel. Ce sont les éléments qui sont lus avant d'être potentiellement redéfinis par une autre instruction du même fragment. Les régions OUT correspondent au morceau de code contenant les éléments du tableau qu'il définit, qui sont (EXACT) ou peut être (MAY) utilisés par la suite, dans l'ordre de l'exécution du programme. Ces éléments sont les éléments du tableau dits vivants ou exportés.

5.5.2 Un tableau de structures

On introduit ici un programme C (figure 5.11), contenant un tableau de structures de type complexe. La structure complexe est composé de deux champs : un champ réel et un champ imaginaire.

```
void test(){
  typedef struct {
    int  img;
    int  reel;
  } complexe;
  complexe UnComplexe[100];
  int N=10;
  int i;
  for (i=0; i<N; i++)
    {
      UnComplexe[i].img=0;
      UnComplexe[i].reel=0.;
    }
}
```

Figure 5.11: Exemple de tableau de structures

Voici le fichier tpips correspondant, où on a activé les options de calcul des prédicats (les préconditions), des effets propres et cumulés ainsi que des régions. En dernier lieu on génère du code parallèle en distribuant la boucle.

```
delete test
create test test.c
activate C_PARSER
setproperty PRETTYPRINT_C_CODE TRUE
setproperty PRETTYPRINT_STATEMENT_NUMBER FALSE
setproperty FOR_TO_DO_LOOP_IN_CONTROLIZER   TRUE
echo
echo Controlizer output
echo
display PRINTED_FILE[test]
echo
echo Preconditions
echo
activate PRINT_CODE_PRECONDITIONS
display PRINTED_FILE[test]
echo
echo Effects for test
echo
activate PRINT_CODE_PROPER_EFFECTS
display PRINTED_FILE[test]
echo
echo Cumulated Effects for test
echo
activate PRINT_CODE_CUMULATED_EFFECTS
display PRINTED_FILE[test]
echo
echo PRINTED Regions for test
echo
activate PRINT_CODE_REGIONS
display PRINTED_FILE[test]
echo
echo Parallelization
echo
activate PRINT_PARALLELIZEDOMP_CODE
display PARALLELPRINTED_FILE[test]
close
quit
```

On récupère d'abord les préconditions sur le code, illustré par la figure 5.12, où les préconditions se font sur les bornes de la boucle ainsi que l'indice i.

```
Preconditions


//  P() {}

void test()
{
   typedef struct {int img; int reel;} complexe;
   complexe UnComplexe[100];
   int N = 10;
   int i;

//  P() {N==10}

   for(i = 0; i <= N-1; i += 1) {

//  P(i) {N==10, 0<=i, i<=9}

      UnComplexe[i].img = 0;

//  P(i) {N==10, 0<=i, i<=9}

      UnComplexe[i].reel = 0.;
   }
}
```

Figure 5.12: Calcul des préconditions

On rappelle qu'au niveau de PIPS les instructions sont étiquetées avec des prédicats exprimants des contraintes sur les variables scalaires entières qui sont utilisés au niveau des bornes de boucles. Ceci nous est utile pour les références sur les tableaux, pour définir quelle sous partie d'un tableau est référencée par une opération mémoire et pour raffiner les effets cumulés dans les régions. Deuxième analyse qu'on récupère est les effets propres, illustrés par la figure 5.13.

```
Effects for test

void test()
{
   typedef struct {int img; int reel;} complexe;
   complexe UnComplexe[100];
   int N = 10;
   int i;
//                <must be read   >: N
//                <must be written>: i
   for(i = 0; i <= N-1; i += 1) {
//                <must be read   >: N i
//                <must be written>: UnComplexe[i][1]
      UnComplexe[i].img = 0;
//                <must be read   >: N i
//                <must be written>: UnComplexe[i][2]
      UnComplexe[i].reel = 0.;
   }
}
```

Figure 5.13: Calcul des effets propres

D'après l'analyse on remarque que PIPS distingue entre les éléments du tableau grâce aux champs {reel} et {img} de la structure. Une information qui se confirme avec le calcul des effets cumulés, illustrés par la figure 5.14.

```
Cumulated Effects for test

void test()
{
   typedef struct {int img; int reel;} complexe;
   complexe UnComplexe[100];
   int N = 10;
   int i;
//              <may be read    >: i
//              <may be written >: UnComplexe[*][1] UnComplexe[*][2]
//              <must be read   >: N
//              <must be written>: i
   for(i = 0; i <= N-1; i += 1) {
//              <may be written >: UnComplexe[*][1]
//              <must be read   >: N i
      UnComplexe[i].img = 0;
//              <may be written >: UnComplexe[*][2]
//              <must be read   >: N i
      UnComplexe[i].reel = 0.;
   }
}
```

Figure 5.14: Calcul des effets cumulés

Quant aux régions, on remarque clairement que PHI1, la variable qui représente la première dimension du tableau, a la valeur de i. Alors que PHI2, la deuxième dimension du tableau, a pour valeur 1 dans le cas de l'affectation au champ {reel} et la valeur 2 dans le cas de l'affectation au champ {img} (figure 5.15).

```
PRINTED Regions for test

void test()
{
   typedef struct {int img; int reel;} complexe;
   complexe UnComplexe[100];
   int N = 10;
   int i;

//  <UnComplexe[PHI1][PHI2]-W-MAY-{0<=PHI1, PHI1<=9, 1<=PHI2, PHI2<=2,
//    N==10}>

   for(i = 0; i <= N-1; i += 1) {

//  <UnComplexe[PHI1][PHI2]-W-EXACT-{PHI1==i, PHI2==1, N==10, 0<=i,
//    i<=9}>

      UnComplexe[i].img = 0;

//  <UnComplexe[PHI1][PHI2]-W-EXACT-{PHI1==i, PHI2==2, N==10, 0<=i,
//    i<=9}>

      UnComplexe[i].reel = 0.;
   }
}
```

Figure 5.15: Calcul des régions

Cette distinction entre les éléments du tableau prouve qu'il n y a pas d'aliasing et donc une parallélisation de la boucle est possible. La parallélisation se fait par distribution de la boucle. Étant donné qu'il n y pas de dépendances entre les deux éléments du tableau, on peut calculer les affectations chacune dans une boucle à part (figure 5.16).

```
Parallelization

void test()
{
   typedef struct {int img; int reel;} complexe;
   complexe UnComplexe[100];
   int N = 10;
   int i;
#pragma omp parallel for
   for(i = 0; i <= N-1; i += 1)
      UnComplexe[i].img = 0;
#pragma omp parallel for
   for(i = 0; i <= N-1; i += 1)
      UnComplexe[i].reel = 0.;
}
```

Figure 5.16: Distribution des boucles

5.5.3 Une structure contenant un tableau

Dans ce cas, on rajoute un champ de type tableau à la structure de données. Et on lance le même script tpips, à savoir un script qui calcule les préconditions, les effets propres, les effets cumulés, les régions ainsi qu'une parallélisation de la boucle. La structure est illustrée par la figure 5.17.

```
Controlizer output

int test1()
{
   typedef struct {int Eleves[30]; int niveau;} classe;
   classe UneClasse[100];
   int N = 10;
   int i;
   for(i = 0; i <= N-1; i += 1) {
      (UneClasse[i].Eleves)[i] = 0;
      UneClasse[i].niveau = 1;
   }
}
```

Figure 5.17: Exemple de structure contenant un tableau

Pour les préconditions on obtient les résultats illustrés par la figure 5.18, qui sont les mêmes que pour la structure précédente, c'est à dire des prédicats sur les bornes de la boucle et l'indice.

```
Preconditions


//  P() {}

int test1()
{
   typedef struct {int Eleves[30]; int niveau;} classe;
   classe UneClasse[100];
   int N = 10;
   int i;

//  P() {N==10}

   for(i = 0; i <= N-1; i += 1) {

//  P(i) {N==10, 0<=i, i<=9}

      (UneClasse[i].Eleves)[i] = 0;

//  P(i) {N==10, 0<=i, i<=9}

      UneClasse[i].niveau = 1;
   }
}
```

Figure 5.18: Les préconditions

Pour les effets propres (figure 5.19), on remarque que PIPS rajoute des parenthèses pour traiter (UneClasse[i].Eleves) comme un seul élément.

```
Effects for test1
int test1()
{
   typedef struct {int Eleves[30]; int niveau;} classe;
   classe UneClasse[100];
   int N = 10;
   int i;
//                <must be read   >: N
//                <must be written>: i
   for(i = 0; i <= N-1; i += 1) {
//                <must be read   >: N i
//                <must be written>: UneClasse[i][1][i]
      (UneClasse[i].Eleves)[i] = 0;
//                <must be read   >: N i
//                <must be written>: UneClasse[i][2]
      UneClasse[i].niveau = 1;
   }
}
```

Figure 5.19: Les effets propres

Il va de même pour le calcul des effets cumulés (figure 5.20).

```
Cumulated Effects for test1

int test1()
{
   typedef struct {int Eleves[30]; int niveau;} classe;
   classe UneClasse[100];
   int N = 10;
   int i;
//                 <may be read    >: i
//                 <may be written >: UneClasse[*][1][*] UneClasse[*][2]
//                 <must be read   >: N
//                 <must be written>: i
   for(i = 0; i <= N-1; i += 1) {
//                 <may be written >: UneClasse[*][1][*]
//                 <must be read   >: N i
      (UneClasse[i].Eleves)[i] = 0;
//                 <may be written >: UneClasse[*][2]
//                 <must be read   >: N i
      UneClasse[i].niveau = 1;
   }
}
```

Figure 5.20: Les effets cumulés

Pour le calcul des régions (figure 5.21), PIPS rajoute une troisième variable PHI3 pour prendre en considération le champ de type tableau d'entier de la structure.

```
PRINTED Regions for test1

int test1()
{
   typedef struct {int Eleves[30]; int niveau;} classe;
   classe UneClasse[100];
   int N = 10;
   int i;

//  <UneClasse[PHI1][PHI2][PHI3]-W-MAY-{0<=PHI1, PHI1<=9, 1<=PHI2,
//    PHI2<=2, N==10}>

   for(i = 0; i <= N-1; i += 1) {

//  <UneClasse[PHI1][PHI2][PHI3]-W-EXACT-{PHI1==i, PHI2==1, PHI3==i,
//    N==10, 0<=i, i<=9}>

      (UneClasse[i].Eleves)[i] = 0;

//  <UneClasse[PHI1][PHI2][PHI3]-W-EXACT-{PHI1==i, PHI2==2, N==10,
//    0<=i, i<=9}>

      UneClasse[i].niveau = 1;
   }
}
```

Figure 5.21: Les régions

Même avec cette difficulté rajoutée, on est encore dans la possibilité de distribuer la boucle. A ce niveau PIPS est encore capable de détecter les alias. La distribution de boucles est illustrée par la figure 5.22.

```
Parallelization

int test1()
{
   typedef struct {int Eleves[30]; int niveau;} classe;
   classe UneClasse[100];
   int N = 10;
   int i;
#pragma omp parallel for
   for(i = 0; i <= N-1; i += 1)
      (UneClasse[i].Eleves)[i] = 0;
#pragma omp parallel for
   for(i = 0; i <= N-1; i += 1)
      UneClasse[i].niveau = 1;
}
```

Figure 5.22: La distribution des boucles

5.5.4 Utilisation de tableau via un pointeur

Les pointeurs ne sont pas gérés au niveau de PIPS, on adopte une approche plutôt conservatrice. Afin de montrer l'incapacité de PIPS à traiter des tableaux déclarés par pointeurs, on a mis en place le code de la figure 5.23.

```
void test2(int N){
  int *p, *q;
  int a[N], b[N];
  int i;
  p=a;
  q=b;
  p=q;
  for (i=0; i<= N; i++)
    {
      p[i]=0;
      q[i]=1;
    }

}
```

Figure 5.23: Déclaration de tableau via pointeur

On a déclaré deux pointeurs p et q en leur affectant respectivement l'adresse du premier élément du tableau a et du tableau b. On introduit l'aliasing par l'affectation p=q, en conséquence p et q pointent vers la même zone mémoire. Le calcul des préconditions se fait classiquement; on obtient des prédicats sur la variable N qui est la taille du tableau et sur l'indice de la boucle i (figure 5.24)

Au niveau du calcul des effets propres (figure 5.25) on remarque que PIPS traite p et q comme des tableaux. On obtient les opérations mémoire (lecture/écriture) effectuées sur les variables du programme.

Au niveau des effets cumulés (figure 5.26), on remarque que les opérations sur les pointeurs ont perdu de leur précision. En effet les opérations mémoires sont devenues may.

```
Preconditions


//  P() {}

void test2(int N)
{
   int *p;
   int *q;
   int a[N];
   int b[N];
   int i;

//  P() {}

   p = a;

//  P() {}

   q = b;

//  P() {}

   p = q;

//  P() {}

   for(i = 0; i <= N; i += 1) {

//  P(i) {i<=N, 0<=i}

      p[i] = 0;

//  P(i) {i<=N, 0<=i}

      q[i] = 1;
   }
}
```

Figure 5.24: Le calcul des préconditions

Pour la calcul de régions (figure 5.27), PIPS ne prend en considérations que les tableaux a et b. En conséquence il ne génère de région que pour ces derniers, ce qui peut amener à prédire une erreur de traitement et de parallélisation par la suite.

```
Effects for test2

void test2(int N)
{
   int *p;
   int *q;
   int a[N];
   int b[N];
   int i;
//                <must be read   >: a
//                <must be written>: p
   p = a;
//                <must be read   >: b
//                <must be written>: q
   q = b;
//                <must be read   >: q
//                <must be written>: p
   p = q;
//                <must be read   >: N
//                <must be written>: i
   for(i = 0; i <= N; i += 1) {
//                <must be read   >: N i
//                <must be written>: p[i]
      p[i] = 0;
//                <must be read   >: N i
//                <must be written>: q[i]
      q[i] = 1;
   }
}
```

Figure 5.25: Le calcul des effets propres

En effet, PIPS ne détecte pas l'aliasing entre p et q et en conséquence permet la distribution de la boucle (figure 5.28).

```
Cumulated Effects for test2

//              <may be written >: TOP-LEVEL:*MEMORY*
//              <must be read   >: N
void test2(int N)
{
   int *p;
   int *q;
   int a[N];
   int b[N];
   int i;
//              <must be read   >: a
//              <must be written>: p
   p = a;
//              <must be read   >: b
//              <must be written>: q
   q = b;
//              <must be read   >: q
//              <must be written>: p
   p = q;
//              <may be read    >: i
//              <may be written >: p[*] q[*]
//              <must be read   >: N
//              <must be written>: i
   for(i = 0; i <= N; i += 1) {
//              <may be written >: p[*]
//              <must be read   >: N i
      p[i] = 0;
//              <may be written >: q[*]
//              <must be read   >: N i
      q[i] = 1;
   }
}
```

Figure 5.26: Le calcul des effets cumulés

Les pointeurs p et q sont considérés comme aliasés d'où la nécessité d'une analyse au moins au niveau de propagation de pointeurs constants.

5.6 Choix de l'analyse

Après l'étude des analyses de pointeurs, notre choix s'est tout d'abord orienter vers les analyses sensibles au flot de données, car elles seules pouvaient apporter une meilleure précision à l'analyse de dépendances interprocédurale de PIPS. Entre l'analyse de Wilson et celle d'Emami, le choix s'est fait en fonction des construction syntaxiques les plus récurrentes. En effet, il n'est pas nécessaire d'utiliser une analyse, même si elle est plus précise, si elle ne sera pas exploitée par le compilateur, surtout si elle risque de dégrader les performances de ce dernier. L'analyse de Wilson s'inspire de l'analyse d'Emami, tout en apportant des précisions sur le traitement des données allouées

```
PRINTED Regions for test2

void test2(int N)
{
   int *p;
   int *q;
   int a[N];
   int b[N];
   int i;

//  <a[PHI1]-R-EXACT-{0<=PHI1, PHI1+1<=N}>

   p = a;

//  <b[PHI1]-R-EXACT-{0<=PHI1, PHI1+1<=N}>

   q = b;
   p = q;
   for(i = 0; i <= N; i += 1) {
      p[i] = 0;
      q[i] = 1;
   }
}
```

Figure 5.27: Le calcul des régions

```
Parallelization

void test2(int N)
{
   int *p;
   int *q;
   int a[N];
   int b[N];
   int i;
   p = a;
   q = b;
   p = q;
   for(i = 0; i <= N; i += 1)
      p[i] = 0;
   for(i = 0; i <= N; i += 1)
      q[i] = 1;
}
```

Figure 5.28: La distribution de boucle

dynamiquement au niveau du tas. D'après notre étude expérimentale, plusieurs constructions syntaxiques que l'analyse de Wilson traite ne sont pas présentes dans les applications de traitement de signal. L'analyse d'Emami offre une analyse sensible au flot de données, qui prend en compte

le contexte des fonctions. Par son graphe d'invocation, l'analyse permet de traiter la récursivité dans les programmes, un traitement qu'on voudrait intégrer au niveau de PIPS. Pour conclure, notre choix s'est porté sur l'analyse d'Emami pour être implémentée au niveau de PIPS et dont les résultats seront exploitées par l'analyse de dépendances dans le but de paralléliser les programmes.

Chapter 6

Conclusion et perspectives

Le paralléliser source à source PIPS permet de paralléliser des programmes scientifiques écrits en Fortran et maintenant étendu à paralléliser des programmes écrits en C.

Le langage C comporte beaucoup de fonctionnalités, qui peuvent affecter les performances des analyses de programme. Dans le cadre de ce stage, on s'intéresse particulièrement aux pointeurs dans les programmes C et les effets sur la précision de plusieurs analyses clientes. La particularité du logiciel PIPS est son analyse interprocédurale des programmes qui prend en compte le flot de données. Le but de l'analyse interprocédurale est de pouvoir détecter les dépendances existant entre les variables et plus particulièrement entre les éléments de tableau au niveau des boucles.

Afin d'améliorer la précision de l'analyse interprocédurale de PIPS, il faut étudier le problème d'aliasing introduit par l'utilisation des pointeurs, qui pour le moment ne sont pas traités par PIPS.

Au cours de ce stage, nous avons d'abord effectué une étude bibliographique sur les analyses de pointeurs, dont le but est de déterminer les cibles des pointeurs dans un programme donné. En premier lieu et dans le cadre de l'étude bibliographique, on s'est intéressé à l'analyse d'Andersen intégrée au niveau de l'architecture CLA (compile-link-analysis)[18] et utilisée par le compilateur gcc. Cette analyse n'est pas sensible au flot de données, mais elle a permis de s'initier aux analyses de pointeurs rencontrées par la suite et surtout de voir ce qui était utilisé en pratique par le compilateur gcc.

Dans un deuxième lieu on s'est intéressé aux analyses sensibles au flot de données et au contexte de fonctions parce que les données sont accessibles via les pointeurs et peuvent être transmises à différentes fonctions. Concernant les analyses de pointeurs sensibles au flot de données et au contexte, on a présenté l'analyse de Wilson[22] qui est intégrée dans le compilateur SUIF. Cette analyse permet de traiter avec précision les différents cas d'aliasing en un temps d'exécution raisonnable. Mais on rappelle que le but principal de cette étude bibliographique est de décider quelle analyse pourrait être intégrée au niveau de PIPS et qui répond aux besoins des applications traitées par ce dernier. L'analyse de Wilson apporte des précisions coûteuses en terme d'espace de stockage et de temps d'exécution, alors que ces précisions ne seront pas utilisées par l'analyse de PIPS[14].

Toujours dans le cadre des analyses sensibles au flot de données, on a présenté l'analyse d'Emami[9]qui est intégrée au niveau du compilateur académique McCAT. Cette analyse a l'avantage d'utiliser une structure de données propre à elle, pour modéliser les appels des fonctions dans un programme : le graphe d'invocation. Cette analyse utilise aussi une représentation interne du code source proche de celle de PIPS, qui permet un traitement plus intuitif des alias. Les deux anal-

yses utilisent l'interprétation abstraite qui est une théorie d'approximation de la sémantique de programmes informatiques.

Ensuite dans le cadre d'une étude expérimentale, nous nous sommes intéressés, en premier lieu, aux constructions syntaxiques accessibles par pointeurs qui sont les plus fréquentes. Cette recherche s'est faite sur des applications de traitement de signal, qui sont en cours d'analyse par le projet PIPS. Cette exploration de code source, a été mise en place pour orienter le choix entre l'analyse de Wilson et d'Emami en fonction des structures. Il n'est pas nécessaire d'utiliser une analyse coûteuse en terme d'espace de stockage et de temps d'exécution si le programme ne comporte pas de constructions nécessitants une telle précision. En deuxième lieu, nous avons mis en place des programmes C, pour tester le traitement actuel des pointeurs au niveau de PIPS. On a remarqué que pour les tableaux déclarés via des pointeurs PIPS ne pouvait pas détecter l'aliasing entre les éléments et par conséquent pouvait autoriser une distribution des boucles inadéquate.

L'étude bibliographique et l'étude expérimentale orientent notre choix vers une analyse similaire à celle d'Emami qui est intégrée au niveau du compilateur McCAT, en raison du caractére intuitif de cette dernier ainsi que son traitement de la récursivité et des pointeurs sur fonctions qu'on voudrait aussi pouvoir traiter au niveau de PIPS.

Enfin il y a d'autres travaux auxquels on devrait s'intéresser comme l'analyse utilisant les BDD (Binary Decision Diagram)[19] de Hendren ou bien une analyse utilisant le principe de diviser pour régner introduite par Kahlon[15].

Appendix A

Annexe

A.1 Les règles basiques de l'analyse de Emami

Cas de x[i]=y[j]:

kill = { (x x1 D) (x x1 D) ∈ input ∧ first_elem(x|i|) = D }
gen = { (x y1 rel ⋈ first_elem(|i|) ⋈ first_elem(|j|)) (y y1 rel) ∈ input }
changed_input = (input - { (x x1 D) (x x1 D) ∈ input ∧ first_elem(|i|)= P})
∪ { (x x1 P) (x x1 P) ∈ input ∧ first_elem(|i|)= P}
return(gen ∪ (changed_input - kill))

Figure A.1: Règle du cas x[i]=y[j]

Cas de x=(*y).b:

kill = { (x x1 rel) (x x1 rel) ∈ input }
gen = { (x y2 rel1 ⋈ rel2) (y y1 rel1) (y1 b y2 rel2) ∈ input}
return(gen ∪ (input - kill))

Figure A.2: Règle du casx=(*y).b

Cas de (*x).a=&y :

kill = { (x1 a x2 rel) (x x1 D) (x1 a x2 rel) ∈ input }
gen = { (x1 a y rel) (x x1 rel) ∈ input}
changed_input = (input - { (x1 a x2 D) (x x1 P) (x1 a x2 D) ∈ input})
∪ { (x1 a x2 P) (x x1 P) (x1 a x2 D) ∈ input}
return(gen ∪ (changed_input - kill))

Figure A.3: Règle du cas(*x).a=&y

Cas de (*x).a=y.b :

kill = { (x1 a x2 rel) (x x1 D) (x1 a x2 rel) ∈ input }
gen = { (x1 a y1 rel1 ⋈ rel2) (x x1 rel1) (y b y1 rel2) ∈ input}
changed_input = (input - { (x1 a x2 D) (x x1 P) (x1 a x2 D) ∈ input}
∪ { (x1 a x2 P) (x x1 P) (x1 a x2 D) ∈ input}
return(gen ∪ (changed_input - kill))

Figure A.4: Règle du cas (*x).a=y.

Cas de (*px)[i]=y :

kill = { (x1 x2 D) (px x1 D) (x1 x2 D) ∈ input ∧ first_elem(|i|)=D }
gen = {(x1 y1 rel1 ⋈ rel2 ⋈ first_elem(|i|)) (px x1 rel1) (y y1 rel2) ∈ input }
changed_input = (input - { (x1 x2 D) |(px x1 P) (x1 x2 D) ∈ input|∨
|(px x1 D) (x1 x2 D) ∈ input ∧ first_elem(|i|)=P| })
∪ { (x1 x2 P) |(px x1 P) (x1 x2 D) ∈ input|∨
|(px x1 D) (x1 x2 D) ∈ input ∧ first_elem(|i|)=P| }
return(gen ∪ (changed_input - kill))

Figure A.5: Règle du cas(*px)[i]=y

Cas de (*x)=(*y).b :

kill = { (x1.x2.rel) (x.x1.D).(x1.x2.rel) ∈ input }
gen = { (x1.y2.rel1 ⋈ rel2 ⋈ rel3) (x.x1.rel1).(y.y1.rel2).(y1.b.y2.rel3)
∈ input }
changed_input = (input - { (x1.x2.D) (x.x1.P).(x1.x2.D) ∈ input})
∪ { (x1.x2.P) (x.x1.P).(x1.x2.D) ∈ input}
return(gen ∪ (changed_input - kill))

Figure A.6: Règle du cas(*x)=(*y).b

Cas de *x=&(*y).b :

kill = { (x1 x2 rel) (x x1 D) (x1 x2 rel) $\in$ input }
gen = { (x1 y1 b rel1 $\bowtie$ rel2) (x x1 rel1) (y y1 rel2) $\in$ input}
changed_input = (input - { (x1 x2 D) (x x1 P) (x1 x2 D) $\in$ input})
$\cup$ { (x1 x2 P) (x x1 P) (x1 x2 D) $\in$ input}
return(gen $\cup$ (changed_input - kill))

Figure A.7: Règle du cas(*px)[i]=y

Bibliography

[1] Définition de l'interprétation abstraite. http://fr.wikipedia.org/wiki/Interpretation_abstraite.

[2] Site officiel de fulguro. http://fulguro.sourceforge.net.

[3] Site officiel de jpeg2000. http://www.jpeg.org/jpeg2000.

[4] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[5] Darren C. Atkinson and William G. Griswold. Effective whole-program analysis in the presence of pointers. ACM, 1998.

[6] Michael G. Burke, Paul R. Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. Springer-Verlag, 1995.

[7] D.Pearce, P.Kelly, and C.Hankin. Efficient field-sensitive pointer analysis for c. 2004.

[8] Maryam Emami. A pratical interprocedural alias analysis for an optimizing/parallelizing c compiler. Master's thesis, School of Computer Science, McGill University, Montreal, 1993.

[9] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. 1994.

[10] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, 2001.

[11] Laurie J. Hendren, Guang R. Gao, and Bhama Sridharan. Designing the mccat compiler based on a family of structured intermediate representations. Springer-Verlag, 1992.

[12] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. 1999.

[13] François. Irigoin. Interprocedural analyses for programming environments. Elsevier Science Publisher, 1992.

[14] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: an overview of the pips project. ACM, 1991.

[15] Vineet Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. ACM, 2008.

[16] Amira. Mensi. Analyse des flots de données. Technical report, Ecole Nationale Supérieure des Mines de Paris,Centre de Recherche en Informatique (CRI), 2008.

[17] M.Hind and A.Pioli. Which pointer analysis should i use? 2000.

[18] N.Heintze and O.Tardieu. Ultra-fast aliasing analysis using cla: a million lines of c code in a second. 2001.

[19] Lhoták Ondřej and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. ACM, 2008.

[20] Derek Rayside. Points-to analysis. Technical report, MIT CSAIL, 2005.

[21] Radu. Rugina. Pointer analysis overview and flow-sensitive analysis. Technical report, Cornell University, 2005.

[22] Robert P. Wilson and Monica S.Lam. Efficient context-sensitive pointer analysis for c programs. 1995.

[23] Robert Paul Wilson. *EFFICIENT, CONTEXT-SENSITIVE POINTER ANALYSIS FOR C PROGRAMS*. PhD thesis, Standford University, 1997.

[24] Sam Zoghaib. Analyseurs statiques et parallélisation. Technical report, l'École Polytechnique.