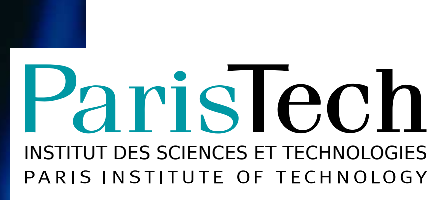


École doctorale n°432 : Sciences des Métiers de l'Ingénieur

Doctorat ParisTech

THÈSE

pour obtenir le grade de docteur délivré par

l'École nationale supérieure des mines de Paris

Spécialité « Informatique temps-réel, robotique et automatique »

présentée et soutenue publiquement par

Amira MENSI

24 juin 2013

Analyse des pointeurs pour le langage C

Directeur de thèse : **François IRIGOIN**
Co-encadrement de la thèse : **Fabien COELHO**
Corinne ANCOURT

Jury

M. Denis BARTHOU, Professeur des universités, ENSEIRB	Président
M. François IRIGOIN, Professeur , CRI MINES ParisTech	Directeur
M. Paul FEAUTRIER, Professeur, École normale supérieure de Lyon	Rapporteur
M. Lionel LACASSAGNE, Maître de conférence, Université Paris Sud	Rapporteur
Mlle. Elisabeth BRUNET, Maître de conférence, Télécom SudParis	Examinateur
Mme. Corinne ANCOURT, Maître de conférence, CRI MINES ParisTech	Invité
M. Fabien COELHO, Maître de conférence, CRI MINES ParisTech	Invité

THÈSE

MINES ParisTech
Centre de Recherche en Informatique
35 rue Saint-Honoré, 77305 Fontainebleau, France

Résumé

Les analyses statiques ont pour but de déterminer les propriétés des programmes au moment de la compilation. Contrairement aux analyses dynamiques, le comportement exact du programme ne peut être connu. Par conséquent, on a recours à des approximations pour remédier à ce manque d'information. Malgré ces approximations, les analyses statiques permettent des optimisations et des transformations efficaces pour améliorer les performances des programmes. Parmi les premières analyses du processus d'optimisation figure l'analyse des pointeurs. Son but est d'analyser statiquement un programme en entrée et de fournir en résultat une approximation des emplacements mémoire vers lesquels pointent ses variables pointeurs. Cette analyse est considérée comme l'une des analyses de programmes les plus délicates et l'information qu'elle apporte est très précieuse pour un grand nombre d'autres analyses clientes. En effet, son résultat est nécessaire à d'autres optimisations, comme la propagation de constante, l'élimination du code inutile, le renommage des scalaires ainsi que la parallélisation automatique des programmes.

Le langage C présente beaucoup de difficultés lors de son analyse par la liberté qu'il offre aux utilisateurs pour gérer et manipuler la mémoire par le biais des pointeurs. Ces difficultés apparaissent par exemple lors de l'accès aux tableaux par pointeurs, l'allocation dynamique «malloc» ainsi que les structures de données récursives. L'un des objectifs principaux de cette thèse est de déterminer les emplacements mémoire vers lesquels les pointeurs pointent. Ceci se fait en assurant plusieurs dimensions comme :

- la sensibilité au flot de contrôle, c'est-à-dire la mise à jour des informations d'un point programme à un autre ;
- la non-sensibilité au contexte, c'est-à-dire l'utilisation de résumés au lieu de l'analyse du corps de la fonction à chaque appel ;
- la modélisation des champs pointeurs des structures de données agrégées, dans laquelle chaque champ représente un emplacement mémoire distinct.

D'autres aspects sont pris en compte lors de l'analyse des programmes écrits en C comme la précision des emplacements mémoire alloués au niveau du tas, l'arithmétique sur pointeurs ou encore les pointeurs vers tableaux. Notre travail, implémenté dans le compilateur paralléliseur PIPS (Parallélisation interprocédurale de programmes scientifiques), permet d'analyser les applications scientifiques de traitement du signal tout en assurant une analyse intraprocédurale précise et une analyse interprocédurale efficace via les résumés.

Mots clés : analyse statique de programmes, analyse de pointeurs, sensibilité au flot de contrôle, sensibilité au contexte.

Abstract

Static analysis algorithms 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 to determine. This information is often used to assess if two pointers are potential aliases, i.e. if they can point to the same memory area. An analysis of pointers, also called points-to analysis, may provide more precision to other analyses such as constant propagation, analysis of dependencies or analysis of live variables. The analysis of pointers is very important for the exploitation of parallelism in scientific C programs since the most important structures they manipulate are arrays, which are typically accessed by pointers. It is necessary to analyse the dependencies between arrays in order to exploit the parallelism between loops. Points-to analysis may also attempt to handle recursive data structures and other structures that are accessed by pointers. This work provides a points-to analysis which is :

- flow-sensitive, by taking into account the order of execution of instructions ;
- field-sensitive, since structure fields are treated as individual locations ;
- context-insensitive, because functions summaries are computed to avoid re-analyzing functions bodies.

Other issues such as heap modeling, pointer arithmetics and pointers to arrays are also taken into account while analyzing C programs. Our intraprocedural analysis provides precise results to client analyses, while our interprocedural one allows to propagate them efficiently. Our work is implemented within the PIPS (Parallélisation interprocédurale de programmes scientifiques) parallelizer, a framework designed to analyze, optimize and parallelize scientific and signal processing applications.

Keywords : static analysis, points-to analysis, flow-sensitive, context-insensitive, field-sensitive.

Remerciements

Cette thèse a été préparée au Centre de recherche en Informatique (CRI) de l'École des mines de Paris (MINES ParisTech), sous la direction de François Irigoin.

Je voudrais tout d'abord exprimer ma reconnaissance à François Irigoin, pour m'avoir accueilli au CRI, de m'avoir proposé un sujet de thèse très riche et très motivant. Durant toute la thèse il a été disponible, patient et les nombreux échanges d'idées et de conseils que nous avons eu ont grandement contribué à l'avancement de mes travaux. Son aide et ses conseils ont été inestimables.

Je remercie mes encadrants de thèse Corinne Ancourt et Fabien Coelho pour m'avoir suivi tout au long de cette thèse et pour leurs conseils éclairés et leur encouragements. Je suis très reconnaissante envers Pierre Jouvelot pour toute l'aide apportée au cours de cette thèse depuis l'étude bibliographique jusqu'à la formalisation et la relecture de la thèse ainsi que Béatrice Creusillet pour l'attention qu'elle a portée à mes travaux, pour sa disponibilité et ses contributions qui ont permis d'intégrer et d'exploiter mon travail dans le compilateur PIPS.

Je suis très reconnaissante envers les membres du CRI, je me rappellerai toujours avec un grand plaisir les bons moments passés avec Robert Mahl, Claire Médrala, Dounia Khaldi, Mehdi Amini, Vivien maisonneuve, Olivier Hermont, Claude Tadonki, Samuel Benveniste, Antoniu Pop, Benoît Pin et en particulier Jacqueline Altimira dont le support moral et la bonne humeur ne m'ont jamais fait défaut.

Merci aussi aux membres du département informatique de Telecom SudParis qui m'ont soutenu durant les derniers mois de préparation de soutenance, je pense en particulier à Elisabeth Brunet, Rachid Habel et Chantal Taconet.

Un grand merci à ma famille, et aux amis Elise, Safa, Nidhal, Xavier, Manon, Sana et Brunozzo pour leur soutien inconditionnel et leurs encouragements. Enfin un merci particulier à celui qui m'a soutenu et motivé durant les moments de doute. Et qui a été présent dès le premier jour jusqu'à celui de la soutenance, mon amour, Gilles.

Table des matières

Table des matières ix

Table des figures xv

Liste des tableaux xvii

1 Introduction 1

1.1 Contexte . . . 1

1.1.1 L'importance du langage C . . . 1

1.1.2 Les pointeurs en C . . . 2

1.2 Motivations . . . 2

1.2.1 Importance de l'optimisation des programmes . . . 2

1.2.2 La mémoire d'un programme . . . 3

1.2.3 Optimisation des programmes en présence de pointeurs . . . 4

1.3 Problématiques . . . 5

1.3.1 Détermination des cibles de pointeurs . . . 5

1.3.2 Analyse de pointeurs ou d'alias ? . . . 5

1.3.3 Analyser les structures de données récursives . . . 5

1.3.4 Abstraction de la mémoire . . . 6

1.4 Contributions . . . 6

1.4.1 Traiter toutes les instructions du langage C ? . . . 6

1.4.2 Concevoir une analyse intraprocédurale précise . . . 7

1.4.3 Concevoir une analyse interprocédurale modulaire . . . 7

1.4.4 Répondre aux besoins des applications scientifiques . . . 8

1.4.5 Modéliser l'allocation dynamique au niveau du tas . . . 8

1.5 Structure de la thèse . . . 8

2 Les analyses clientes et les objectifs pour le compilateur 11

2.1 La représentation des arcs *points-to* . . . 11

2.1.1 Structure de données . . . 12

2.1.2 Affichage de la relation *points-to* . . . 12

2.1.3 Conclusion . . . 13

2.2 Les effets mémoire et l'analyse des pointeurs . . . 13

2.2.1 Définitions . . . 13

2.2.2 Les effets en présence de pointeurs . . . 13

2.3 Les chaînes « use-def » et la suppression du code inutile . . . 15

2.3.1 Les chaînes « use-def » . . . 15

2.3.2 Suppression de code inutile . . . 15

2.3.3 Les chaînes *use-def* et la suppression de code inutile sans information *points-to* . . . 16

2.3.4 Chaînes *use-def* et suppression de code inutile avec l'information *points-to* 18

2.3.5 Conclusion . . . 19

2.4 L'analyse des dépendances et la parallélisation . . . 19

2.4.1 L'analyse des dépendances et la parallélisation sans information *points-to* . 21

2.4.2 L'analyse des dépendances et la parallélisation avec l'information *points-to* 22

2.4.3 Conclusion . . . 24

2.5 La suppression de sous-expressions communes . . . 24

2.5.1 La suppression de sous-expressions communes sans information sur les pointeurs . . . 24
2.5.2 La suppression de sous-expressions communes avec l'information sur les pointeurs . . . 24
2.5.3 Conclusion . . . 26
2.6 Le renommage des scalaires . . . 26
2.6.1 Le renommage des scalaires sans information *points-to* . . . 26
2.6.2 Le renommage des scalaires avec l'information *points-to* . . . 29
2.7 Conclusion . . . 33

3 État de l'art : Les analyses de pointeurs 35
3.1 Les analyses insensibles au flot et au contexte . . . 35
3.1.1 L'analyse d'Andersen . . . 36
3.1.2 L'analyse de Steensgaard . . . 38
3.1.3 Conclusion . . . 40
3.2 Les analyses sensibles au contexte et au flot de contrôle . . . 40
3.2.1 L'analyse de Wilson . . . 40
3.2.2 L'analyse d'Emami . . . 46
3.2.3 Comparaison des analyses insensibles au flot et au contexte . . . 52
3.2.4 Comparaison des analyses sensibles au flot et au contexte . . . 53
3.3 Autres travaux . . . 54
3.3.1 L'analyse « pointer values » . . . 54
3.3.2 L'allocation dynamique . . . 54
3.3.3 L'union, l'arithmétique sur pointeurs et le cast . . . 55
3.4 Conclusion . . . 56

4 L'analyse intraprocédurale simplifiée 57
4.1 Définition de la syntaxe abstraite utilisée pour `C` . . . 57
4.2 Définition du langage $\mathcal{L}_0$ . . . 59
4.2.1 Définition des domaines . . . 59
4.2.2 Syntaxe du langage $\boldsymbol{\mathcal{L}}_0$ . . . 61
4.2.3 Typage des expressions . . . 62
4.3 Le schéma général de l'analyse *points-to* . . . 63
4.3.1 Graphe des appels de l'analyseur et résultat de l'analyse . . . 63
4.3.2 Étapes de l'analyse *points-to* . . . 66
4.3.3 Traduction des expressions d'adresses en chemins d'accès mémoire constants (CP) . . . 67
4.4 Définition de l'analyse *points-to* pour le langage $\mathcal{L}_0$ . . . 72
4.4.1 L'opérateur d'assignation d'un opérateur . . . 72
4.4.2 Exemples . . . 76
4.5 Définition du langage $\mathcal{L}_1$ . . . 79
4.5.1 Langage $\mathcal{L}_1$ et notations . . . 79
4.5.2 Les points de séquence . . . 81
4.5.3 Rupture de contrôle : `exit()` . . . 82
4.5.4 Rupture de contrôle : `return` . . . 82
4.6 Les expressions en langage $\mathcal{L}_1$ . . . 82
4.6.1 Définition de l'analyse *points-to* pour le langage $\mathcal{L}_1$ . . . 83
4.6.2 Cas des branchements conditionnels «if then...else » . . . 87
4.7 Conclusion . . . 89

5 L'analyse intraprocédurale étendue 91
5.1 Cas particuliers d'emplacements mémoire abstraits . . . 91
5.1.1 Modélisation de NULL . . . 91
5.1.2 Modélisation des pointeurs non initialisés . . . 92
5.1.3 Modélisation des tableaux de pointeurs . . . 92
5.1.4 Modélisation des constantes entières . . . 92
5.1.5 Conclusion . . . 93
5.2 Modélisation de l'allocation dynamique . . . 93
5.2.1 Modélisation du tas . . . 93
5.2.2 Modélisation d'un appel à « malloc » . . . 94
5.2.3 Modélisation d'un appel à free . . . 94
5.2.4 Détection des erreurs liées à l'allocation mémoire . . . 97
5.2.5 Conclusion . . . 99
5.3 Abstraction de l'ensemble des adresses sous forme de treillis . . . 99
5.3.1 Opérateurs et structure du treillis des chemins constants CP . . . 101
5.3.2 Le sous-treillis $Module$ . . . 102
5.3.3 Le sous-treillis des noms de variables $Name$ . . . 103
5.3.4 Le treillis $Type$ . . . 107
5.3.5 Le treillis des indices $Vref$ . . . 108
5.3.6 Le prédicat d'atomicité sur le treillis CP . . . 110
5.3.7 Conclusion . . . 110
5.4 Les choix de conception . . . 110
5.4.1 L'initialisation des paramètres formels à NULL . . . 111
5.4.2 Impact de l'initialisation à NULL sur l'interprétation des conditions . . . 111
5.4.3 L'utilisation des alias . . . 115
5.4.4 Conclusion . . . 115
5.5 Treillis des relations *points-to* $\mathcal{PT}$ . . . 115
5.5.1 Abstraction de la valeur d'un pointeur avec une relation *points-to* . . . 115
5.5.2 Domaine des relations *points-to* . . . 116
5.5.3 Typage d'un arc *points-to* . . . 116
5.5.4 Ordre sur les relations *points-to* . . . 116
5.5.5 Finitude des relations . . . 116
5.5.6 Typage des arcs *points-to* . . . 117
5.6 Utilisation des treillis CP et $\mathcal{PT}$ . . . 117
5.6.1 Calcul des ensembles Kill et Gen : affectation d'un pointeur . . . 118
5.6.2 Cas des boucles . . . 119
5.6.3 Cas de la boucle « while » . . . 120
5.6.4 Cas de la boucle *for* . . . 125
5.6.5 La boucle *do...while* . . . 127
5.6.6 L'algorithme général any_loop_to_points_to() . . . 127
5.6.7 Conclusion sur les boucles . . . 131
5.6.8 Cas des graphes de flot de contrôle, « $unstructured$ » . . . 131
5.7 Finitude de la représentation/abstraction de la relation *points-to* . . . 133
5.7.1 Nombre de déréférencements . . . 135
5.7.2 Longueur des chemins dans le graphe . . . 135
5.7.3 Nombre de nœuds du graphe *points-to* . . . 136
5.7.4 Nombre d'arcs *points-to* sortants d'un nœud . . . 136
5.7.5 Conclusion . . . 137
5.8 Conclusion . . . 137

6 L'analyse interprocédurale : traitement des sites d'appels 139
6.1 Les différentes approches pour le traitement des sites d'appels . . . 140
6.1.1 L'approche insensible au contexte . . . 141
6.1.2 Limitation de l'ensemble Gen aux paramètres et variables globales . . . 141
6.1.3 L'approche utilisant l'ensemble des adresses écrites par le site d'appel . . . 142
6.1.4 L'approche par résumés . . . 144
6.1.5 Ordonnancement des phases pour l'analyse *points-to* . . . 145
6.2 Vers un transformeur de *points-to* : les difficultés . . . 148
6.2.1 Affectation à un pointeur de la valeur retournée par une fonction . . . 148
6.2.2 Affectation indirecte d'un pointeur dans une procédure . . . 149
6.2.3 Interruption de l'exécution . . . 151
6.2.4 Impact d'un test . . . 154
6.2.5 Libération d'une zone mémoire . . . 155
6.2.6 Conclusion sur les difficultés de l'analyse *points-to* interprocédurale . . . 163
6.3 Le schéma interprocédural ascendant général . . . 163
6.3.1 Notations . . . 166
6.3.2 Algorithme de traitement d'un site d'appel . . . 168
6.3.3 Fonction de transfert *points-to* associée à une fonction `C` . . . 169
6.3.4 Construction de pt_{bound} et de binding . . . 169
6.4 Compatibilité entre binding et In : correction de la fonction de traduction . . . 173
6.4.1 Exemple pour le test de compatibilité de binding . . . 173
6.4.2 Deuxième exemple . . . 173
6.5 Calcul de Kill, Gen et pt_{end} . . . 174
6.5.1 Calcul de Kill . . . 174
6.5.2 Calcul de Gen . . . 176
6.5.3 Calcul de pt_{end} . . . 177
6.6 Preuve de correction . . . 178
6.7 Conclusion . . . 178

7 Implémentation et expériences 179
7.1 Implémentation de l'analyse *points-to* . . . 179
7.1.1 Le compilateur PIPS . . . 179
7.2 Chaînage d'une structure de données . . . 182
7.2.1 Analyse intraprocédurale de la fonction `chain` . . . 183
7.2.2 Analyse intraprocédurale de la fonction `main` . . . 183
7.2.3 Analyse interprocédurale . . . 183
7.3 Échange indirect . . . 185
7.3.1 Analyse intraprocédurale de la fonction `swap` . . . 185
7.3.2 Analyse intraprocédurale de la fonction `main` . . . 186
7.3.3 Analyse interprocédurale . . . 187
7.4 Tableau dynamique . . . 188
7.4.1 Analyse intraprocédurale de la fonction `allocate_array` . . . 189
7.4.2 Analyse interprocédurale . . . 189
7.5 Affectation de structures . . . 190
7.5.1 Analyse intraprocédurale de la fonction `main` . . . 190
7.5.2 Analyse intraprocédurale de la fonction `assignment` . . . 191
7.5.3 Analyse interprocédurale . . . 191
7.6 Exemple de Wilson . . . 193
7.6.1 Analyse intraprocédurale de la fonction `f` . . . 194
7.6.2 Analyse intraprocédurale de la fonction `main` . . . 195
7.6.3 Analyse interprocédurale pour le premier site d'appel . . . 196

7.6.4 Analyse interprocédurale pour le deuxième site d'appel 198
7.6.5 Analyse interprocédurale pour le troisième site d'appel 200
7.6.6 Conclusion sur l'exemple de Wilson 201
7.7 Conclusion 202

8 Contributions et perspectives 203
8.1 Contexte de la thèse 203
8.1.1 Profil des applications scientifiques 203
8.1.2 Besoins des analyses clientes 204
8.1.3 Intégration des résultats dans le compilateur PIPS 204
8.2 Contributions 204
8.2.1 Étude des besoins 204
8.2.2 État de l'art 204
8.2.3 Conception de l'analyse intraprocédurale 205
8.2.4 Conception de l'analyse interprocédurale 205
8.3 Nouveaux concepts 205
8.3.1 Formulation de l'analyse de pointeurs 206
8.3.2 Le treillis des emplacements mémoire 206
8.3.3 Implémentation dans un compilateur source-à-source 206
8.3.4 Utilisation des chemins d'accès constants 207
8.3.5 Analyse modulaire 207
8.3.6 Analyse paramétrable 208
8.3.7 Détection des erreurs 208
8.4 Perspectives 209
8.4.1 Modélisation du tas 209
8.4.2 Les instructions du langage C 209
8.4.3 Conception d'une analyse des structures de données récursives 209

Bibliographie 211

Table des figures

1.1 Deux pointeurs pointant vers le même emplacement mémoire 4

2.1 Le graphe des « use-def » **sans** l'information *points-to* pour le programme 3.6 . . 17
2.2 Le graphe de dépendances **avec** l'information *points-to* pour le programme 3.6 . . 20
2.3 Le graphe de dépendances **sans** l'information *points-to* 22
2.4 Le graphe de dépendances **avec** l'information *points-to* 23
2.5 Le graphe de dépendances pour le programme `scalar_renaming03` sans l'information *points-to* 28
2.6 Le graphe de dépendances avant le renommage des scalaires 31
2.7 Le graphe de dépendances après le renommage des scalaires 34

3.1 Les contraintes de l'analyse d'Andersen 36
3.2 Contraintes pour le programme 3.2 37
3.3 Graphe résultat de l'analyse de Andersen sur notre exemple 37
3.4 Les règles de production de Steensgaard 39
3.5 La règle de typage pour l'affectation $x = y$ 39
3.6 La règle d'inférence pour l'affectation $x = y$ 39
3.7 Graphe résultat de l'analyse de Steensgaard 40
3.8 Analyse interprocédurale de Wilson 43
3.9 Graphe résultat de l'analyse de Wilson 46
3.10 Représentation SIMPLE 47
3.11 L'abstraction au niveau de la pile 47
3.12 Graphe résultat de l'analyse d'Emami 48
3.13 Le graphe d'invocation 48
3.14 Les séquences d'appels dans un graphe d'invocation 49
3.15 Le processus d'association 50
3.16 État de la pile 50
3.17 Relations *points-to* au niveau de la pile 51
3.18 Les relations au niveau du tas 52

4.1 Syntaxe abstraite de l'ensemble du langage `C` 58
4.2 Syntaxe abstraite de l'ensemble du langage $\mathcal{L}_0$ 62
4.3 le graphe des appels 65
4.4 Traduction de `***p` 68
4.5 Syntaxe du sous-langage $\boldsymbol{\mathcal{L}_1}$ 80
4.6 Les expressions en $\mathcal{L}_1$ 83

5.1 Graphe des arcs *points-to* avant l'appel à `free` 94
5.2 Graphe des arcs *points-to* après l'appel à `free` 96
5.3 État du tas avant l'appel à `free` 98
5.4 État du tas après l'appel à `free` 98
5.5 État du tas pour le programme 5.4 99
5.6 État du tas pour le programme 5.4 après l'appel à `free` 99
5.7 Treillis des emplacements mémoire 100
5.8 treillis Module 102
5.9 Le treillis Name 104
5.10 Le treillis type 107

5.11 Un autre choix pour le treillis des emplacements mémoire 114
5.12 Exemple de graphe non structuré sans cycle 132
5.13 Exemple de graphe non structuré complet pour la fonction 22 132

6.1 Graphe des arcs *points-to* avant l'appel à `swap01` 145
6.2 Graphe des arcs *points* après l'appel à `swap01` 145
6.3 Les dépendances entre phases d'analyse 147
6.4 Le schéma interprocédural 165
6.5 L'ensemble In et pt_{bound} pour la variable `pi` 172
6.6 L'ensemble In et pt_{bound} pour la variable `pj` 172

7.1 Architecture logicielle de PIPS 181
7.2 Graphe *points-to* avant l'appel à `chain` 184
7.3 Graphe *points-to* après l'appel à `chain` 185
7.4 Etat de la mémoire après l'analyse intraprocédurale de `swap` 186
7.5 Etat de la mémorie après l'appel à `swap` 188
7.6 L'ensemble pt_{caller} avant l'appel à `assignment` et sans les champs des structures 191
7.7 L'ensemble Out à la sortie de la fonction à `assignment` et sans les champs des structures 191
7.8 Construction de l'ensemble pt_{bound} à l'entrée de la fonction `assignment` 192
7.9 Graphe des arcs *points-to* après le traitement de l'appelée 193
7.10 Graphe des arcs *points* avant le site d'appel 196
7.11 Graphe des arcs *points* pour la construction de binding pour le premier site 197
7.12 Graphe des arcs *points-to* pour le premier site d'appel 198
7.13 Graphe des arcs *points* pour la construction de binding pour le deuxième site . . 199
7.14 Graphe des arcs *points-to* pour le deuxième site d'appel 200
7.15 Graphe des arcs *points* pour la construction de binding pour le troisième site . . . 201
7.16 Graphe des arcs *points* pour le troisième site d'appel 202

Liste des tableaux

3.1 Les règles *points-to* de Wilson 41
3.2 Exemples d'ensembles d'emplacements 42
3.3 Tableau comparatif des analyses de pointeurs 56

4.1 Domaines simples 59
4.2 Exemples de calcul d'adresses abstraites correspondant à des expressions 71
4.3 Exemples de calcul des valeurs correspondantes à des expressions 74
4.4 Les notations du chapitre intraprocédural 80

5.1 Opérateur $inter_M$ 103
5.2 L'opérateur max_N 105
5.3 Opérateur $inter_N$ 105
5.4 L'opérateur $killable_{MAYN}$ 106
5.5 L'opérateur $killable_{EXACTN}$ 106
5.6 Opérateur max_T 107
5.7 Opérateur $inter_T$ 107
5.8 Opérateur $killable_{MAYT}$ 108
5.9 Opérateur $killable_{EXACTT}$ 108

6.1 Les ensembles du chapitre interprocédural 167
6.2 Les notations du chapitre interprocédural 167

Liste des Algorithmes

1 Fonction γ . . . 63
2 Fonction type_of . . . 64
3 Fonction eval_local . . . 67
4 Fonction eval . . . 69
5 Fonction stub_cp . . . 70
6 Fonction eta . . . 71
7 Fonction etv . . . 73
8 Fonction $T_{\mathcal{PT}}$ . . . 76
9 Fonction $T_{\mathcal{PT}ass}$ . . . 76
10 Fonction $T_{\mathcal{PT}test}$ . . . 76
11 Fonction $T_{\mathcal{PT}}$ . . . 81
12 Fonction $T_{\mathcal{PT}}$ d'une expression . . . 84
13 Fonction transformeur_pts_to_call . . . 85
14 Fonction $T_{\mathcal{PT}}$ . . . 87

15 Fonction $points_to_free$ . . . 97
16 Fonction $inter_{Vref}$ . . . 109
17 Fonction $killable_{killMAYV}$ . . . 109
18 Fonction $killable_{killEXACTV}$ . . . 109
19 Fonction atomic_location_p( ) . . . 110
20 Fonction merge . . . 125
21 Fonction any_loop_to_points_to($\mathcal{S} : b, \mathcal{E} : c, \mathcal{E} : init, \mathcal{E} : inc, \mathcal{PT}$: In) . . . 130
22 Fonction points_to_cyclic_graph(graph G(V,E), $\mathcal{PT}$ In) . . . 133
23 Fonction points_to_graph(graph G(V,E), $\mathcal{PT}$ In_u) . . . 134
24 Fonction u_ntp($s : \mathcal{S}, b : \mathcal{B}, P : \mathcal{PT}$) . . . 134

25 Fonction points_to_function_projection($\mathcal{PT} : P$) . . . 168
26 Fonction points_to_call_site . . . 169
27 Fonction compute_pt_bound . . . 170

Exemples et programmes

1.1 Accès à un pointeur et à un élément de tableau avec la même notation 2
1.2 `p` et `q` pointent vers la même zone mémoire après `S2` 4
2.1 Structure de données *points-to* 12
2.2 Affichage des arcs *points-to* 12
2.3 Exemple 1 avec les effets propres **sans** l'information *points-to* 14
2.4 Exemple 1 avec les résultats *points-to* 14
2.5 Exemple 1 avec les effets propres calculés **avec** l'information *points-to* 15
2.6 Exemple C pour la suppression de code inutile 16
2.7 Les effets propres **sans** information *points-to* 16
2.8 Les arcs *points-to* pour le programme 3.6 18
2.9 Les effets propres obtenus **avec** l'information *points-to* 18
2.10 Résultat de la suppression de code inutile **avec** l'information *points-to* 19
2.11 Programme C à paralléliser 21
2.12 Effets propres **sans** l'information *points-to* pour le programme 2.11 21
2.13 Les arcs *points-to* pour le programme 2.11 22
2.14 Les effets propres **avec** les arcs *points-to* pour le programme 2.11 23
2.15 Programme parallélisé 23
2.16 Programme avec expression commune j+2 24
2.17 Les effets propres pour le programme 2.16 25
2.18 Les arcs *points-to* pour le programme 2.16 25
2.19 Les effects propres avec l'information *points-to* pour le programme 2.16 25
2.20 Renommage des variables scalaires 26
2.21 Les effets propres **sans** l'information *points-to* pour le programme 2.20 27
2.22 Les arcs *points-to* pour le programme 2.20 29
2.23 Les effets propres **avec** l'information *points-to* pour le programme 2.20 30
2.24 Le programme après le renommage de scalaires 32
3.1 Exemple C pour l'analyse intraprocédurale 36
3.2 Instructions `C` génératrices de contraintes Andersen 36
3.3 Fonction foo 38
3.4 Modélisation des champs pointeurs 38
3.5 Algorithme de Wilson [Wil97] 45
3.6 Exemple `C` pour l'aspect interprocédural de l'analyse d'Emami 48
3.7 L'algorithme interprocédural d'Emami 49
3.8 Exemple pour le processus d'association 50
3.9 Exemple `C` pour l'association inverse 51
3.10 Exemple C pour le traitement du tas 52
3.11 Exemple `java` de l'allocation dynamique au niveau des boucles 55
4.1 Traduction des accès mémoire en chemins constants 60
4.2 Structure de données récursive 67
4.3 Traduction des chemins constants 68
4.4 Exemple d'expression complexe 68
4.5 Exemple 1 77
4.6 Exemple 2 78
4.7 L'opérateur « virgule » 82
4.8 Structure de données d'une expression 83
4.9 arcs *points-to* pour l'arithmétique des pointeurs 86
4.10 Exemple C avec une instruction `if then...else` 88

4.11 Les arcs *points-to* pour une instruction `if then...else` . . . 88
5.1 Exemple de libération de mémoire . . . 94
5.2 Le résultat de l'analyse après l'appel à `free` . . . 96
5.3 Programme `C` contenant une fuite mémoire . . . 98
5.4 Les arcs *points-to* pour le programme contenant la fuite mémoire . . . 98
5.5 Modélisation des paramètres formels . . . 111
5.6 Analyse sans une initialisation des paramètres formels à `NULL` . . . 112
5.7 Analyse avec une initialisation des paramètres formels à `NULL` . . . 113
5.8 Exemple de *cast* . . . 117
5.9 Exemple de tableau de pointeurs . . . 119
5.10 Exemple d'éléments d'un tableau de pointeurs . . . 120
5.11 Définition de la structure de données . . . 121
5.12 Exemple de liste chaînée : compteur d'entiers . . . 121
5.13 Les arcs *points-to* pour le programme 5.12 . . . 124
5.14 Définition de la structure de données . . . 124
5.15 Exemple de liste chaînée : champ valeur de type pointeur . . . 124
5.16 Les arcs *points-to* pour le programme 5.15 . . . 126
5.17 Exemple de liste chaînée : allocation dynamique . . . 127
5.18 Les arcs *points-to* pour le programme 5.17 . . . 128
5.19 Exemple boucle *for* . . . 128
5.20 Les arcs *points-to* pour le programme 5.19 . . . 129
5.21 Structure de données pour les graphes de contrôle . . . 131
5.22 Initialisation des éléments d'un tableau de pointeurs . . . 136
5.23 Libération d'une liste chaînée . . . 136
5.24 Les arcs *points-to* pour le programme 5.23 . . . 136
6.1 L'exemple interprocédural à analyser . . . 141
6.2 Les arcs *points-to* pour l'exemple interprocédural . . . 142
6.3 Les effets résumés pour le programme 6.1 . . . 143
6.4 Les arcs *points-to* pour l'approche utilisant seulement l'ensemble Kill . . . 143
6.5 Les arcs *points-to* pour le programme 6.1 . . . 144
6.6 Exemple d'allocation dynamique en interprocédural . . . 149
6.7 La version « inlinée » du programme 6.6 . . . 149
6.8 Les arcs *points-to* pour le programme 6.6 . . . 149
6.9 La version « inlinée » du programme 6.6 avec les arcs *points-to* . . . 150
6.10 Affectation d'un pointeur dans une procédure . . . 151
6.11 La version « inlinée » du programme 6.10 . . . 151
6.12 Les arcs *points-to* pour le programme 6.10 . . . 152
6.13 Les arcs *points-to* OUT pour le programme 6.10 . . . 152
6.15 Exemple d'interruption d'exécution en interprocédural . . . 152
6.16 La version « inlinée » du programme 6.15 . . . 152
6.14 La version « inline » du programme 6.10 avec les arcs *points-to* . . . 153
6.17 Les arcs *points-to* pour le programme 6.15 . . . 154
6.18 La version « inlinée » du programme 6.15 avec les arcs *points-to* . . . 154
6.19 Exemple de test en interprocédural . . . 155
6.20 La version « inlinée » du programme 6.19 . . . 155
6.21 Les arcs *points-to* pour le programme 6.19 . . . 156
6.22 La version « inline » du programme 6.19 avec les arcs *points-to* . . . 156
6.23 Libération de zone mémoire en interprocédural 1 . . . 157
6.24 La version « inlinée » du programme 6.23 . . . 157
6.27 Libération de zone mémoire en interprocédural . . . 157
6.28 La version « inlinée » du programme 6.27 . . . 157

6.25 Les arcs *points-to* pour le programme 6.23 158
6.26 Les arcs *points-to* pour la version « inlinée » du programme 6.23 158
6.29 Les arcs *points-to* pour le programme 6.27 159
6.30 La version « inlinée » du programme 6.27 159
6.31 Allocation mémoire après un appel à `free` 160
6.32 Libération mémoire après une allocation 160
6.33 Version expansée du programme 6.31 160
6.34 La version expansée du programme 6.32 160
6.35 Les arcs *points-to* pour la version expansée du programme 6.31 161
6.36 Les arcs *points-to* pour la version expansée du programme 6.32 161
6.37 Les arcs *points-to* pour la programme 6.31 162
6.38 Les arcs *points-to* pour le programme 6.32 162
6.39 Appel conditionnel à `free` 162
6.40 Exemple pour le calcul de pt_{bound} 171
6.41 Les arcs *points-to* pour la fonction `foo` 171
6.42 Une séquence `C` 173
6.43 Exemple de traduction imprécise 176
6.44 Exemple d'analyse imprécise 176
6.45 Les arcs *points-to* pour le programme 6.43 176
6.46 Les arcs *points-to* pour le programme 6.44 176
7.1 Chaînage de deux éléments 182
7.2 In pour l'exemple 7.1 183
7.3 Out pour l'exemple 7.1 183
7.4 pt_{caller} pour l'exemple 7.1 183
7.5 Effets résumés pour l'exemple 7.1 184
7.6 Les arcs *points-to* pour le programme 7.1 185
7.7 Échange indirect 186
7.8 In pour l'exemple d'échange indirect 186
7.9 Out pour l'exemple d'échange indirect 186
7.10 pt_{caller} pour l'exemple d'échange indirect 187
7.11 Les effets résumés pour l'exemple d'échange indirect 187
7.12 Tableau dynamique 188
7.13 In pour l'exemple de tableau dynamique 189
7.14 Out pour l'exemple de tableau dynamique 189
7.15 les effets résumés pour la fonction `allocate_array` 189
7.16 Affectation de structures 190
7.17 pt_{caller} pour l'exemple d'affectation de structures 191
7.18 In pour l'exemple d'affectation de structures 191
7.19 Out pour l'exemple d'affectation de structures 191
7.20 Exemple de Wilson 193
7.21 Premier site d'appel de l'exemple Wilson 194
7.22 Deuxième site d'appel de l'exemple Wilson 194
7.23 Troisième site d'appel de l'exemple Wilson 195
7.24 In pour l'exemple de Wilson 195
7.25 Out pour l'exemple de Wilson 195
7.26 pt_{caller} pour l'exemple de Wilson 196
7.27 Les opérations pour le premier site d'appel 196
7.28 Les effets résumés pour la fonction `f` 197
7.29 les arcs *points-to* pour le premier site d'appel 198
7.30 Les opérations pour le deuxième site d'appel 198
7.31 Les effets résumés pour la fonction `f` 199

7.32 Les arcs *points-to* pour le deuxième site d'appel 200
7.33 Les opérations pour le troisième site d'appel 200
7.34 Les arcs *points-to to* pour le troisième site d'appel 201
8.1 Double déréférencement 207
8.2 Introduction d'une variable temporaire 207
8.3 Modélisation des champs pointeurs 207
8.4 Appel à `malloc` dans une boucle 209

CHAPITRE 1

Introduction

1.1 Contexte

De très nombreux travaux de recherche ont déjà été consacrés à l'analyse statique des pointeurs utilisés dans le langage C, et ce depuis plus d'une vingtaine d'années. Ce sujet est réputé difficile à cause de sa complexité en temps et en espace et à cause de la sémantique du langage C. De nombreuses analyses publiées ont de surcroît la réputation de contenir des erreurs, et seules les analyses les plus simples sont actuellement intégrées dans des compilateurs de production.
Nos motivations pour aborder à nouveau le sujet sont essentiellement constituées de besoins industriels actuels dans le domaine du traitement du signal. Ce dernier comporte de nombreuses applications écrites en langage C où des descripteurs de tableaux sont codés sous la forme d'un champ de structure de données contenant des pointeurs rendant les tâches d'optimisation de code et de parallélisation impossibles aux compilateurs. Les tableaux de structures de données sont aussi fréquemment utilisés dans les codes scientifiques, entre autres pour représenter les nombres complexes. La parallélisation de tels codes nécessite donc que l'analyse prenne en compte les champs de structures. Par ailleurs, certains codes scientifiques sont écrits de manière à pouvoir se passer d'un optimiseur dans le compilateur. Ils s'appuient uniquement sur des pointeurs pour représenter les tableaux et sur l'arithmétique des pointeurs pour les parcourir. L'analyse des opérations sur pointeurs est donc nécessaire pour leur parallélisation ou optimisation. Malheureusement, les travaux de recherche déjà effectués manquent parfois d'explications et de résultats concrets, ce qui ne permet pas de les reprendre, de les étendre et de les intégrer aux compilateurs et outils actuels. Enfin, une difficulté supplémentaire est que nous souhaitons effectuer l'analyse au niveau du code source tel qu'il a été écrit par le programmeur plutôt que sur une représentation trois adresses afin de permettre un retour simple vers le programmeur et une intégration dans un outil source-à-source.

Nous présentons maintenant le contexte scientifique actuel dans lequel s'inscrit cette thèse, qui est motivée par les applications embarquées nécessitant des puissances de calcul importantes.

1.1.1 L'importance du langage C

Le langage C a l'avantage d'être un langage de bas niveau, dont la syntaxe contient de nombreuses constructions avancées, et qui bénéficie aussi des avantages d'un langage de bas niveau, proche du matériel. De plus, le langage C est très portable : presque toutes les plateformes ont un compilateur C. Il permet la manipulation de la mémoire, de périphériques comme les disques durs ainsi que la communication avec les contrôleurs ou les processeurs. Durant ces dernières années il s'est imposé comme le langage de référence pour l'écriture d'applications scientifiques et il a été classé premier au classement 2012 des langages de programmation [1],dans des domaines comme la mécanique de fluides, l'algorithmique géométrique, la biologie algorithmique, le traitement d'images, les mathématiques appliquées ou encore le traitement du signal [gnu91] [gnu96] [mat88] [mat70].

1. http ://www.tiobe.com/index.php/content/paperinfo/tpci/

1.1.2 Les pointeurs en C

Une des particularités du langage C est la présence des pointeurs, qui sont des valeurs contenant l'adresse ou l'emplacement d'un objet ou d'une fonction dans la mémoire. C'est en « déréférençant » ces pointeurs que nous pouvons écrire ou lire les variables dites pointées. L'utilisation des pointeurs est fréquente en C notamment pour manipuler des chaînes de caractères, des tableaux dynamiques, des structures de données récursives dont les graphes et les arbres pour lesquels les nœuds sont créés dynamiquement, mais aussi pour parcourir efficacement les tableaux statiques. Le langage offre la possibilité de manipuler la mémoire en allouant et libérant des zones mémoire via des routines comme `malloc` ou `free`, mais cette liberté n'est pas sans danger. En effet, la manipulation des pointeurs est une opération délicate qui requiert expertise et attention de la part des développeurs. Beaucoup d'erreurs résultant d'une mauvaise gestion des pointeurs ne sont pas détectables à la compilation et aboutissent à l'interruption de l'exécution. Parmi ces erreurs figurent les pointeurs non initialisés (appelés « wild pointer »), les pointeurs libérés puis réutilisés (appelés « dangling pointer »), les zones mémoire dont les références sont perdues avant qu'elles ne soient libérées (appelées fuites mémoire), ainsi que les pointeurs vers des variables de la pile disparues après une sortie de bloc ou de fonction. Un autre avantage et en même temps une difficulté de l'utilisation des pointeurs est l'usage de la notation tableau pour accéder au contenu d'un pointeur et vice-versa (voir par exemple le programme 1.1).

```
int main(){
int i = 1, *X = &i, Y[10] ;

X[i] = 2;
Y[i] = 1;

return 0;
}
```

Prog 1.1 – Accès à un pointeur et à un élément de tableau avec la même notation

Malgré cette équivalence au niveau de la notation, il y a trois cas à distinguer :

1. un tableau dynamique, créé au niveau du tas et dont les éléments sont accessibles via un pointeur. Par exemple un tableau dynamique de 30 entiers est alloué par appel à `malloc(30*sizeof(int))` ;
2. un tableau local, de taille fixe, ses éléments sont contigus en mémoire et son nom est un pointeur constant vers son premier élément. Comme il est constant il ne peut être ni changé ni étendu ; il est déclaré par `int a[30]` ;
3. un tableau alloué par le mot-clé `static int a[30]`, qui a les mêmes propriétés que le tableau local mais avec une longueur de vie qui dépasse l'exécution de la fonction où il est déclaré.

1.2 Motivations

Nous présentons dans cette section les motivations de notre travail, qui sont essentiellement liées aux applications scientifiques et de traitement de signal écrites en langage C.

1.2.1 Importance de l'optimisation des programmes

Malgré l'évolution des architectures parallèles, illustrées par le circuit CELL [PBD], BlueGene/L [TDD+02] ou CRAY [JVIW05]), et des langages parallèles, comme par exemple OpenMP [DM98],

MPI [SOHL+98] ou CUDA [NBGS08], de nombreux programmeurs continuent à réfléchir et concevoir leurs applications de manière séquentielle. En effet la parallélisation et l'optimisation requièrent des connaissances sur les dépendances entre données et instructions, la synchronisation ainsi que la gestion des communications inutiles en général pour le développeur et créent des problèmes supplémentaires de mise au point. À cela s'ajoute aussi le nombre important d'applications scientifiques existantes et qui sont candidates aux optimisations. Pour ces raisons, les développeurs préfèrent laisser le soin de la parallélisation et de l'optimisation aux compilateurs actuels qui sont maintenant munis d'options permettant d'effectuer ces tâches comme par exemple pour le compilateur gcc. L'utilisation de l'option « gcc icc xlc + OpenMP ou MPI » permet de gérer dans le code les paradigmes de parallélisation OpenMP ou MPI.

D'autres outils de recherche permettant la parallélisation et l'optimisation automatiques existent. PSarmi eux nous citons PIPS [AAC+] [ACSGK11], PLUTO [BHR08], Polaris [BEF+94], Omni OpenMP [KSS00] et SUIF [AALwT93]. Ces outils agissent soit au moment de la compilation pour réaliser des analyses statiques, soit au moment de l'exécution pour réaliser des analyses dynamiques. Elles détectent les portions de code qui peuvent être optimisées ou parallélisées comme par exemples les nids de boucles.

Les outils d'optimisation sont utilisés essentiellement pour améliorer les performances des application 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. D'autres applications sont des candidates aux optimisations comme par exemple 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 convolutions appliquées à l'image représentée sous la forme d'un tableau multidimensionnel. Donc ces applications contiennent beaucoup de calcul sur des éléments de tableaux, ce qui fait d'elles les candidates idéales pour la parallélisation des nids de boucles. La phase de parallélisation nécessite de vérifier que les instructions ou occurrences d'instructions sont indépendantes les unes par rapport aux autres. Ceci concerne en particulier les itérations de boucles. C'est ce qu'on appelle l'analyse de dépendances. D'autres optimisations ont besoin d'une information précise sur les pointeurs ; parmi elles nous citons la suppression du code inutile, la propagation de constantes ou encore la suppression de sous-expressions communes (voir le chapitre 2).

1.2.2 La mémoire d'un programme

Avant de préciser les optimisations faites sur les codes, précisons que, lors de l'exécution d'un programme, une plage mémoire lui est allouée. Elle est généralement divisée en cinq segments, chacun ayant une fonctionnalité bien précise :

1. « text » : appelé aussi segment code. Il sert à stocker les instructions machine du programme ;
2. « data » : il contient les variables globales initialisées au lancement du programme, les chaînes de caractères et les constantes globales ;
3. « bss » : contient le reste des variables globales et statiques (non-initialisées) ;
4. « heap » : appelé aussi tas, c'est une zone mémoire utilisée pour l'allocation dynamique. Cette zone permet au programmeur, à des moments arbitraires de l'exécution, de demander au système l'allocation de nouvelles zones de mémoire, et de restituer au système ces zones (libérer la mémoire) ; la manipulation de la mémoire est possible via les routines de la bibliothèque standard de C.
5. « stack » : appelé pile, il a pour objectif le stockage des variables locales des fonctions ainsi que le contexte de ces dernières. Chaque fois qu'une fonction est activée, une zone

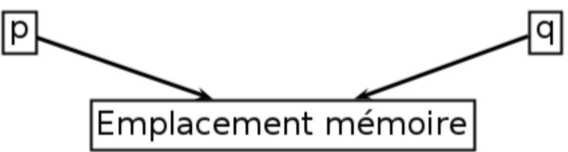


FIGURE 1.1 – Deux pointeurs pointant vers le même emplacement mémoire

mémoire appelée *frame* lui est allouée dans la pile, ce qui permet par exemple d'avoir des contextes complètement différents pour la même fonction et donc des comportements différents. Quand la fonction se termine, son contexte d'exécution, *frame*, dans la pile est détruit.

Pour un programme écrit en langage C, la manipulation de la mémoire se fait souvent via des pointeurs et l'allocation dynamique. Analyser l'état de la mémoire d'un programme est une tâche difficile à réaliser mais permet d'optimiser le coût des accès mémoire qui se font par le déréférencement de pointeurs, d'où la nécessité d'avoir une analyse de pointeurs ou d'alias[2] afin de de déterminer les cibles des pointeurs au niveau de la pile ou encore les cibles des pointeurs sur fonctions.

1.2.3 Optimisation des programmes en présence de pointeurs

La présence des pointeurs dans un programme créé une réelle incertitude sur les données accédées par les instructions. Les variables sont modifiées via le déréférencement de pointeurs. En l'absence d'information sur l'emplacement mémoire vers lequel le pointeur pointe, il est nécessaire d'analyser le programme sous l'hypothèse que toutes les variables du programme peuvent être lues ou écrites par chaque déréférencement. Deux pointeurs peuvent pointer vers le même emplacement ; par exemple pour le programme 1.2, chaque déréférencement du pointeur `p` ou `q` pourrait être considéré comme une lecture (ou écriture) des variables `i` ou `j`.

```
int main()
{
int i = 1, j=0, *p, *q;
 p = &i;  // S1
 q = p;   // S2
*q = 2;   // S3
 j = *p;  // S4

return 0;
}
```

Prog 1.2 – `p` et `q` pointent vers la même zone mémoire après `S2`

De tels pointeurs sont dits en « alias », cet état est illustré par la figure 1.1.
Le pointeur `p` peut lire la variable dont la valeur vient d'être changée par `q`. En effet, `p` et `q` pointent vers la variable `i`. Le pointeur `q` change la valeur de cette dernière au niveau de l'instruction `S3`. Quand le pointeur `p` est déréférencé et son contenu est lu ce n'est plus la valeur 1 qui est obtenue mais 2. En absence d'information sur les relations entre `p` et `q` et leur cibles respectives le compilateur ne peut analyser correctement le programme. Il faut lui fournir l'information `p->i` et `q->i`, où `->`[3] désigne la relation pointe vers. La relation « `p` pointe vers un emplacement mémoire » est appelée dans la littérature *points-to*.

Cette incertitude sur les cibles des pointeurs inhibe plusieurs optimisations de code comme :

2. Deux pointeurs sont dits en « alias » ou synonymes s'ils contiennent l'adresse de la même zone mémoire.
3. La notation`->` n'a pas le même sens que celui de l'accès à un champ de structure en C.

- la propagation de constantes ;
- l'élimination des expressions communes ;
- la détection de code inutile ;
- ou le calcul des chaînes *use-def*.

Toutes ces analyses et bien d'autres seraient possibles en présence d'une analyse de pointeurs qui fournirait une information suffisamment précise sur les emplacements mémoire pointés par les pointeurs. L'impact de l'analyse de pointeurs sur ces optimisations est étudié dans le chapitre 2. Bien-sûr, le compromis entre une analyse précise ou efficace devrait être fait et sera discuté dans ce manuscrit.

1.3 Problématiques

La difficulté majeure du langage `C` vient de la liberté qu'il offre aux développeurs pour manipuler la mémoire. Cette manipulation de la mémoire via les pointeurs doit être analysée pour pouvoir déterminer les accès en lecture et écriture.

1.3.1 Détermination des cibles de pointeurs

Déterminer pour chaque pointeur, à la compilation, la liste des emplacements vers lesquels il pointera au moment de l'exécution est un problème indécidable. La difficulté provient des différents chemins d'exécution qui ne peuvent être prédits ainsi que l'allocation dynamique dont la réussite ne peut être vérifiée qu'au moment de l'exécution. Pour pouvoir répondre à cette problématique il faut avoir recours à l'interprétation abstraite [Cou00], qui est une théorie d'approximation de la sémantique de programmes. L'interprétation abstraite s'inscrit dans le cadre de l'analyse statique dont le but est l'extraction automatique d'informations sur les exécutions possibles d'un programme. L'interprétation abstraite va permettre de simuler le comportement du programme et par conséquent l'état de la mémoire allouée au programme, que ce soit au niveau de la pile ou au niveau du tas (voir sous-section 1.2.1).

1.3.2 Analyse de pointeurs ou d'alias ?

L'information `p->i` et `q->i` obtenue sur le programme 1.2 peut être aussi écrite sous la forme `<p,q>` qui signifie que les deux pointeurs sont en alias. Dans la littérature les deux analyses existent et peuvent être combinées. Il faut savoir qu'il est plus facile de passer d'une analyse *points-to*, via l'application de certaines règles de traduction, à une analyse d'alias que l'inverse [Ema93]. Le choix de la représentation de l'analyse des pointeurs peut affecter les performances de cette dernière. En général les analyses choisissent les représentations les plus compactes pour ne pas consommer un grand espace mémoire et une représentation en table de hachage pour accélérer les accès. Pour notre analyse, nous avons choisi l'analyse *points-to* avec la représentation `p->i` qui est plus lisible pour l'utilisateur et qui en appliquant des règles simples de traduction peut se transformer en représentation sous forme d'alias.

1.3.3 Analyser les structures de données récursives

L'analyse des structures de données récursives, appelée classiquement *shape analysis* [SA], est une analyse statique dont le but est de vérifier certaines propriétés des structures récursives allouées dynamiquement et de permettre la parallélisation sur liste ou encore la conversion de listes en tableaux. Cette analyse intervient au moment de la compilation pour détecter les erreurs suivantes, dont une partie qui peut être détectée par l'analyse des pointeurs :

- les fuites mémoires ;

- les cases mémoire libérées plus d'une fois ;
- les pointeurs pendants (*dangling pointers*) ;
- les accès tableaux hors limites ;
- la cohérence des listes chaînées.

L'analyse des structures de données récursives est aussi une analyse de pointeurs mais restreinte au sous-ensemble des pointeurs alloués dynamiquement. Si l'analyse de pointeurs restreint le domaine de nommage aux variables du programme, l'analyse des données récursives utilise quant à elle un système de nommage plus flexible pour pouvoir nommer tous les objets alloués. Par exemple, chacun des éléments d'un tableau dynamique initialisé au niveau d'une boucle aura un nom différent avec la possibilité de fusionner, si nécessaire, en un seul objet un ensemble d'éléments. Notre analyse ne comporte pas de « shape analysis » ; néanmoins elle abstrait les objets alloués en les nommant avec le même nom s'ils ont été créés au niveau de la même ligne de code. Pour le moment, cette solution répond aux besoins des applications que nous traitons. Une analyse des données récursives est une analyse à part qui a fait l'objet de plusieurs articles et thèses [SJR10] [SRW02] [HHN92]. Elle pourrait faire partie des suites de notre travail, mais même sans une « shape analysis » notre analyse permet de détecter un nombre important des erreurs énumérées ci-dessus.

1.3.4 Abstraction de la mémoire

Comme il n'est pas possible de représenter tous les emplacements mémoire d'un programme, une solution est de les regrouper selon la zone dans laquelle ils ont été alloués. Chaque zone représenterait une classe de variables ; cette solution est donc une abstraction des zones mémoire. Elle est nécessaire pour l'analyse de pointeurs et fera l'objet du chapitre 5 où sont détaillées les zones mémoire ainsi que la relation d'ordre établie entre elles, sous la forme d'un treillis.

L'analyse de pointeurs doit aussi satisfaire plusieurs critères comme la précision des résultats, le temps d'exécution, l'espace mémoire utilisé ainsi que la représentation des résultats. La plupart des analyses ont échoué à cause du compromis précision/efficacité [Hin01]. Nous avons systématiquement favorisé la précision pour que l'analyse intraprocédurale permette de détecter le parallélisme des fonctions feuilles de l'arbre des appels, fonctions qui contiennent le gros des calculs en traitement de signal. Nous nous sommes aussi intéressés au traitement des appels de fonctions, mais, faute de temps, nous ne sommes préoccupés ni du temps ni de l'espace mémoire nécessaire à notre analyse.

1.4 Contributions

Nous présentons dans cette section les principales contributions de la thèse qui sont 1) la prise en compte de toutes les instructions du langage `C`, 2) la conception d'une analyse intraprocédurale précise et 3) un analyse interprocédurale ascendante locale.

1.4.1 Traiter toutes les instructions du langage `C` ?

Notre analyse de pointeurs permet de traiter toutes les instructions du langage `C`. Cependant, toutes ces instructions n'ont pas le même effet sur l'analyse. L'instruction principale génératrice d'arcs *points-to* est l'assignation avec en partie gauche un pointeur. Mais les autres instructions ont quand même des effets de bord sur les arcs *points-to*, comme par exemple l'arithmétique de pointeurs, les appels de fonctions, ou encore les structures de contrôle. Comme toutes les instructions en `C` sont aussi des expressions et qu'il est possible d'en combiner plusieurs, par exemple `++a[i-j]`, il a fallu respecter la priorité des opérateurs ainsi que les points de sé-

quence [4]. Grâce à une représentation interne proche du langage, notre analyse effectue le calcul des arcs *points-to* en une seule passe, sans décomposer les expressions. Elle prend en compte les tableaux et les structures de données de type `struct`. Cependant, les `unions` et les `enum` ne sont pas encore pris en compte dans la version actuelle. Quant aux « cast » ils ne sont pris en compte que partiellement.

1.4.2 Concevoir une analyse intraprocédurale précise

La contribution la plus importante de notre travail est notre analyse intraprocédurale. C'est une analyse sensible au flot de contrôle ; c'est-à-dire que les arcs *points-to* sont mis à jour d'un point programme à un autre, ce qui garantit un résultat précis permettant la prise de décision des autres analyses. L'analyse prend aussi en compte les structures de contrôle et introduit une approximation sur les relations pour exprimer l'incertitude due aux conditions. Notre analyse prend en entrée un programme traduit dans la représentation interne du compilateur PIPS [JT89] et se déroule en une seule passe sans décomposer les instructions complexes, contrairement à l'analyse d'Emami [Ema93].

Garantir la précision des résultats intraprocéduraux a ainsi permis de répondre aux besoins des analyses clientes comme le montre le chapitre 2.

1.4.3 Concevoir une analyse interprocédurale modulaire

Adopter la même approche pour l'analyse interprocédurale, qui consiste à propager en avant l'information *points-to* pour tout le programme, aurait provoqué la dégradation des résultats et des performances (temps d'exécution). Si une analyse sensible au contexte avait été adoptée, les résultats obtenus auraient été très précis mais au prix d'un temps d'exécution considérable et d'une grande complexité. En effet, réanalyser une fonction à chacun de ses appels est une mauvaise solution, surtout si la précision accrue de cette approche n'améliore pas significativement les résultats des analyses clientes.

Pour minimiser les réanalyses, Wilson [WHI95] a proposé d'accumuler pour chaque fonction des *résumés* pouvant être réutilisés lors des sites d'appels ultérieurs dans la mesure où l'aliasing de deux sites d'appel est identique. Malheureusement, Wilson n'a donné d'information précise ni sur ses résumés ni sur les algorithmes nécessaires à leur utilisation.

Notre analyse est fondamentalement différente parce qu'elle est ascendante afin de limiter le volume des structures de données à manipuler. Elle nécessite d'une part une hypothèse forte de non aliasing [5] des paramètres formels et des variables globales, et d'autre part un mécanisme de *transformeur* de *points-to*, remplissant la même fonction que les résumés de Wilson.

Pour chaque fonction et après une analyse de son corps, nous créons un *transformeur* des arcs *points-to* qui peut être transmis au niveau de l'appelant.

Ce transformeur contient simultanément des information sur les relations *points-to* en entrée et sortie de procédure, sur les pointeurs modifiés et sur les cellules mémoire libérées alors que le résumé de Wilson ne comporte que l'effet de la procédure sur les arguments et leur cibles, qui peuvent être de type pointeur aussi. Seules les variables visibles au niveau du site d'appel sont conservées ; ces variables incluent les valeurs de retours, les paramètres formels pouvant contenir des pointeurs (pointeur, structure, tableau de pointeurs), les variables globales et les variables allouées au niveau du tas. Ce mécanisme permet de mettre à jour le site d'appel, de garder une liste d'arcs *points-to* pertinents et surtout d'offrir une analyse de pointeurs performante.

4. Voir la section 5.1.2.3 de la norme C
5. aliasing : quand deux pointeurs pointent vers la même case mémoire.

1.4.4 Répondre aux besoins des applications scientifiques

La meilleure analyse de pointeurs n'est pas celle qui fournit les résultats les plus précis, ni celle qui s'exécute rapidement mais plutôt celle qui répond à des besoins bien spécifiques. En effet, concevoir une analyse précise alors que les applications qu'elles ciblent ne nécessitent que des approximations sur les pointeurs est une erreur de conception.

Notre analyse répond aux besoins des applications scientifiques, décrites précédemment au niveau de la sous-section 1.2.1 ainsi que par les travaux suivants [Rak01], [GH98], [HHN92], [ADLL05]. Le but est de cibler les analyses clientes comme l'analyse de variables vivantes, la détection de code mort ou inutile ou encore l'analyse de dépendances. Une première étude a été faite sur les besoins des applications [Ami08] afin de repérer les structures les plus récurrentes et d'orienter le choix de l'analyse *points-to* qui répond le mieux aux profils de ces applications. Les structures les plus fréquentes sont les pointeurs sur tableaux, les tableaux dynamiques créés au niveau du tas ainsi que les boucles imbriquées.

1.4.5 Modéliser l'allocation dynamique au niveau du tas

Malgré l'absence de « shape analysis » dans notre analyse, celle-ci apporte une précision sur les sites d'allocation dynamique et la modélisation du tas [Ghi96]. Cette précision est paramétrable ; le tas peut être considéré comme un objet unique et donc tous les pointeurs alloués via la routine `malloc` pointent vers la même destination `*heap*` ou bien les pointeurs ciblent non seulement le tas mais aussi la ligne à laquelle `malloc` a été appelée ainsi que le nom du module en cours. La modélisation du tas est détaillée dans le chapitre 5.

1.5 Structure de la thèse

Dans le premier chapitre, consacré aux analyses clientes, nous montrons l'intérêt de concevoir une analyse de pointeurs via des exemples d'analyses et d'optimisations. Les résultats de ces dernières sont significativement améliorés, quand l'information *points-to* permet de préciser les zones mémoire lues et écrites par les références.

Dans le deuxième chapitre, nous reprenons les principales analyses de pointeurs. Les analyses se divisent en deux catégories selon leur sensibilité au flot de contrôle. Les analyses insensibles sont connues et utilisées grâce à leurs efficacité et complexité polynomiale, contrairement aux analyses sensibles au flot, connues pour la précision de leur résultat et leur complexité exponentielle. Cette étude nous a permis de comparer les différents algorithmes et de nous en inspirer pour dépasser leurs limitations.

L'analyse *points-to* intraprocédurale du langage `C` est extrêmement complexe. Nous avons décomposé la présentation en utilisant successivement plusieurs sous-ensembles du langage, appelés $\mathcal{L}_0$, $\mathcal{L}_1$ et $\mathcal{L}_2$ de complexité croissante.

Ainsi le troisième chapitre décrit l'analyse intraprocédurale [6] que nous avons conçue et qui reprend le principe de la sensibilité au flot de contrôle, comme l'analyse d'Emami [EGH94], tout en proposant un algorithme plus précis et général que cette dernière. Notre analyse est une analyse descendante (appelée aussi « top-down »). L'information *points-to* de l'instruction en cours est calculée en fonction de l'instruction précédente. Dans ce chapitre, nous présentons les différentes équations utiles au calcul des arcs *points-to* et qui permettent de couvrir les instructions du langage $\mathcal{L}_0$ et $\mathcal{L}_1$.

Dans le quatrième chapitre, nous décrivons l'analyse intraprocédurale étendue pour le langage $\mathcal{L}_2$, c'est-à-dire qui prend en compte l'abstraction de la mémoire ainsi que les opérateurs de comparaison. Ce chapitre comporte aussi les algorithmes de point fixe qui calculent les arcs

6. Sans prendre en compte les appels de fonction

points-to au niveau des boucles. La modélisation du tas ainsi que le traitement des structures de données récursives sont aussi abordés.

Le dernier chapitre décrit l'analyse interprocédurale qui prend en compte finement les appels de fonction. Notre analyse s'inspire du travail de Wilson [Wil97] qui propose une analyse insensible au contexte et qui se fonde sur les résumés calculés précédemment pour chaque fonction. L'analyse met à jour les sites d'appel sans recalculer les arcs *points-to* des appelés, quand c'est possible. En effet, contrairement à Wilson [WSR00], nous avons essayé de mettre clairement en évidence les difficultés de la traduction des sites d'appels et de décrire en détail les solutions retenues.

Notre analyse offre enfin l'avantage d'être développée dans PIPS qui est un compilateur source-à-source. Une fois notre analyse terminée, le programme peut être soumis directement à une autre analyse ou optimisation utilisant ses résultats. Notre analyse permet de fournir des résultats intraprocéduraux précis et une propagation interprocédurale locale efficace.

CHAPITRE 2

Les analyses clientes et les objectifs pour le compilateur

Pour évaluer une analyse de pointeurs, la métrique la plus utilisée est la taille de l'ensemble des arcs *points-to* [HP00] [Rak01]. Plus cet ensemble est petit, plus l'analyse était jugée précise.

Cependant, cet ensemble peut être petit parce qu'il fait pointer tous les pointeurs vers la même zone mémoire, comme par exemple dans l'analyse d'Emami [Ema93], où tous les pointeurs alloués dynamiquement via la fonction `malloc` pointent vers le même emplacement abstrait `HEAP`. Par conséquent cette métrique n'est pas suffisante pour juger de l'efficacité et surtout de la précision d'une analyse de pointeurs. C'est pour cette raison que les chercheurs [HP00] se sont tournés vers d'autres métriques plus significatives. Dans ce chapitre nous allons évaluer la précision de notre analyse, non pas avec des métriques de graphe, mais plutôt en se fondant sur des exemples qui illustrent l'impact des informations relatives aux pointeurs sur d'autres analyses, des optimisations et des transformations de code. Ces optimisations de code sont indépendantes de l'architecture et font donc partie de la partie intermédiaire du compilateur.

L'analyse *points-to* intervient au début de cette phase et ses résultats sont fournis comme entrée pour ces optimisations. L'impact de cette information nous fournit des informations sur la précision et surtout sur l'utilité de notre analyse *points-to*. Dans ce chapitre nous étudions l'impact de l'information *points-to* sur les analyses et optimisations suivantes :

1. le calcul des effets de lecture/écriture sur la mémoire ;
2. le calcul des *use-def chains* ;
3. la suppression du code inutile ;
4. le calcul des dépendances ;
5. la parallélisation ;
6. la suppression des expressions communes ;
7. et le renommage des scalaires.

2.1 La représentation des arcs *points-to*

Dans la littérature plusieurs représentations des arcs *points-to* ont été utilisées. Que ce soit sous la forme de couples d'alias [1] `<p,q>` ou sous la forme de relation *points-to* `<p->i>`. Nous avons choisi d'utiliser la notation *points-to* avec une information en plus sur l'exactitude de l'arc. Les détails de la structure de données et de son affichage sont fournis dans les sections 2.1.1 et 2.1.2.

1. synonymes

2.1.1 Structure de données

Un arc *points-to* relie une source, appelée aussi origine, une référence constante dont le type final est un pointeur, à un `sink`, qui représente un emplacement mémoire vers lequel pointe la source. L'emplacement mémoire est une référence constante correspondant à une adresse mémoire, si on la connaît. Sinon c'est un emplacement abstrait de la mémoire représentant un ensemble plus ou moins grand d'adresses. L'abstraction de la mémoire est détaillée à la section 5.3.1.

L'implémentation des structures de données est fondée sur l'outil `NewGen` [IJT91] qui permet la modélisation et la génération automatique de fonctions de manipulation (voir sous 7.1.1.3).

```
points_to = source:cell x sink:cell x approximation x descriptor;
```

Prog 2.1 – Structure de données *points-to*

Pour l'analyse de pointeurs, le domaine *cell* qui représente une cellule mémoire, va se restreindre à une référence. Le domaine *reference* dans PIPS [pip] est utilisé pour représenter une référence à un élément de tableau par exemple `a[5]`. C'est une variable avec un nom et une liste d'indices.

Les champs de structures, c'est-à-dire le type `struct` du langage `C`, sont représentés comme une concaténation du nom de la structure avec le caractère « `.` » et le nom du champ, par exemple un champ `age` d'une structure `UnePersonne` est noté `UnePersonne.age`. Le domaine *cell* va être aussi utilisé pour représenter des emplacements mémoire abstraits lorsque la case mémoire vers laquelle pointe le pointeur n'est pas une variable du programme (comme la valeur indéfinie ou le tas). Pour apporter plus de précision à l'arc *points-to* une approximation est attachée à chaque arc. Elle apporte une information sur le degré de certitude et sur la manière dont la source pointe vers la destination (« sink »), avec une approximation « `MAY` » ou « `EXACT` ». Les relations « `MAY` » sont générées entre autres par les structures de contrôle (union des *points-to* des branches « true » et « false »). Mais des incertitudes sur l'origine ou la destination d'un arc conduisent aussi à une approximation « `MAY` ». Elles sont le résultat de la traduction des déréférencements de pointeurs ou d'évaluations d'indices en chemins d'accès mémoire constants. En effet, la traduction peut produire un ensemble de chemins dont la cardinalité est supérieure à un. Dans ce cas les arcs portent l'approximation « `MAY` ». Un autre cas générateur d'arcs « `MAY` » se présente quand la destination est un emplacement abstrait représentant plusieurs emplacements réels comme les cellules allouées au niveau du tas. Le champ *descriptor* permet de définir des contraintes qui pourront éventuellement être utilisées par la suite pour traiter les arcs *points-to* impliquant des tableaux. Ce champ est ignoré par nos algorithmes et notre implantation. L'ensemble des emplacements mémoire, abstraits ou concrets, est modélisé sous la forme d'un treillis présenté dans le chapitre 5.

2.1.2 Affichage de la relation *points-to*

L'affichage des arcs *points-to*, au niveau du compilateur PIPS, est effectué sous forme de post-conditions données en commentaires au niveau du code source. Le code originel est décoré par les arcs *points-to*. Par exemple pour une instruction `p = &i`, qui a pour effet de faire pointer le pointeur `p` vers la case mémoire contenant la valeur de la variable `i`, le résultat est le suivant :

```
p = &i;
// Points To:
// p -> i , EXACT
```

Prog 2.2 – Affichage des arcs *points-to*

La flèche `->` désigne la relation « pointe vers » et `EXACT` l'approximation de la relation ; dans ce cas on est sûr que `p` pointe vers `i` après l'exécution de l'affectation.

2.1.3 Conclusion

Une fois la structure de données des arcs *points-to* ainsi que le format des résultats définis, nous pouvons présenter les grandes lignes de notre analyse de pointeurs. Pour cela nous définissons un sous-ensemble réduit du langage `C` comportant seulement trois instructions, le langage $\mathcal{L}_0$ (voir la section 4.2).

2.2 Les effets mémoire et l'analyse des pointeurs

La première analyse dont l'amélioration des résultats dépend de la précision d'une analyse de pointeurs est l'analyse des *effets* [IJT91] [Cre96]. Dans la littérature cette analyse correspond à la phase de calcul des ensembles *Kill* et *Gen*, appelé aussi calcul des « use-def chains », pour les blocs de base [2] du programme.

2.2.1 Définitions

Pour effectuer la parallélisation automatique PIPS [IJT91] [pip] [AAC$^+$] [ACSGK11] calcule les effets des instructions sur la mémoire. On distingue les effets propres, cumulés et résumés, tout en faisant la distinction entre les effets en lecture (`READ`), les effets en écriture (`WRITE`), les effets qui ont toujours lieu (`EXACT`) et les effets qui peuvent avoir lieu (`MAY`). Remarquons que les effets propres mémoire correspondent aux ensembles Gen et Kill utilisés lors de l'analyse de flot de données [ASU86].

Effets propres Les effets propres d'une instruction sont la liste des variables lues ou écrites directement par cette dernière. Ils sont utilisés pour construire les chaînes *use-def* et le graphe de dépendance par la suite. Nous désignons par effets propres les effets d'un bloc composé, sans prendre en compte les effets des blocs inférieurs. Par exemple le corps d'une boucle, les branchements vrais et faux d'un bloc de test et les nœuds de contrôle dans les blocs non structurés sont ignorés lors du calcul des effets propres.

Effets cumulés Les effets cumulés sont aussi des listes de variables lues et écrites. Il s'agit d'effets cumulés lorsque les effets d'un bloc composé, d'une boucle, d'un test ou d'un bloc non structuré, incluent les effets des blocs inférieurs comme le corps d'une boucle ou une branche d'un test.

Effets résumés Les effets résumés d'un module sont les effets cumulés de son bloc supérieur en ignorant les effets sur les variables dynamiques locales qui ne sont pas visibles aux appelants.

2.2.2 Les effets en présence de pointeurs

Un pointeur contient l'adresse d'une variable du programme ou d'une zone mémoire allouée dynamiquement. En le déférençant, c'est la variable vers laquelle il pointe qui est lue ou écrite. Lors de l'analyse des effets propres, il devient impossible de déterminer la variable qui est lue

2. Un bloc de base est une séquence maximale d'instructions où le flot de contrôle ne peut entrer que par la première instruction et ne peut pas quitter le bloc par un branchement sauf si c'est la dernière instruction de ce dernier [ASU86].

ou écrite lors d'un déréférencement de pointeur, sans une information au préalable sur ce dernier. L'analyse adopte dans ce cas une approche conservatrice en désignant toute la mémoire, par l'emplacement abstrait `*ANY_MODULE*:*ANYWHERE*`, comme lue ou écrite. Prenons comme exemple le programme 2.3. Les effets propres y apparaissent sous la forme de prédicats en commentaires `C`, avec, pour chaque variable, l'action de l'instruction (WRITE ou READ) : en déréférençant `b` on ne sait pas quelle case mémoire est écrite.

```
int exemple01() {
int a, *b;
//             <    is written>: b
   b = &a;
//             <may be written>: *ANY_MODULE*:*ANYWHERE*
//             <    is read   >: b
   *b = 2;
//             <    is read   >: a
    return a;
}
```

Prog 2.3 – Exemple 1 avec les effets propres **sans** l'information *points-to*

Dans le reste de ce chapitre nous présentons seulement les résultats de l'analyse *points-to* du compilateur PIPS [pip] et leur impact sur les phases qui en dépendent. L'analyse est détaillée dans les chapitres 4, 5 et 6. Il faut retenir pour l'instant que l'analyse fournit en résultat un programme décoré d'arcs *points-to*, sous la forme :

source -> destination, approximation

avec source et destination des emplacements mémoire du programme et approximation qui peut être `EXACT` ou `MAY` . Maintenant notre analyse de pointeurs est exécutée pour obtenir le programme 2.4 décoré avec des arcs *points-to*, qui apparaissent en précondition[3] pour chaque instruction.

```
int exemple01() {
int a, *b;

   b = &a;
// Points To:
// b -> a , EXACT
   *b = 2;
// Points To:
// b -> a , EXACT
   return a;
}
```

Prog 2.4 – Exemple 1 avec les résultats *points-to*

L'arc `b -> a , EXACT` se lit : le pointeur `b` pointe exactement vers la variable `a`. Si les résultats de l'analyse de pointeurs sont fournis au calcul des effets, le programme 2.5 est obtenu.

En effet grâce à l'information *points-to*, nous savons que `b` pointe vers `a` ; donc en déréférençant `b` la variable `a` est écrite. D'où la disparition du `*ANY_MODULE*:*ANYWHERE*` qui induit trop d'imprécision. L'amélioration du calcul des effets propres, lorsque l'information *points-to* est fournie, améliore aussi le calcul des effets cumulés et résumés qui en sont dérivés.

3. Pour améliorer la lisibilité des résultats nous avons supprimé les arcs *points-to* associés aux déclarations

```
int exemple01() {
int a, *b;
//          <   is written>: b
   b = &a;
//          <   is read   >: b
//          <   is written>: a
  *b = 2;
//          <   is read   >: a
   return a;
}
```

Prog 2.5 – Exemple 1 avec les effets propres calculés **avec** l'information *points-to*

2.3 Les chaînes « use-def » et la suppression du code inutile

Pour certaines optimisations et en particulier pour la suppression du code inutile, le compilateur doit connaître les relations existantes entre définitions et utilisations des variables. Dans cette section nous allons présenter le calcul des chaînes « use-def » et la suppression du code inutile ainsi que l'amélioration de leurs résultats en présence de l'analyse de pointeurs.

2.3.1 Les chaînes « use-def »

Les chaînes « use-def », ou plus précisément le graphe de « use-def » [ASU86], est un graphe où les arcs relient chaque définition dans le programme à chaque potentielle utilisation. En d'autres termes l'analyse « use-def » fournit aux transformations une liste d'utilisations pour chaque définition du programme. Pour cela il faut calculer pour chaque bloc du programme :

- les utilisations des variables, pour chaque instruction qui lit une variable ;
- l'ensemble des variables tuées, pour chaque variable qui a été réécrite ;
- les « reaching definitions ».

Les « reaching definitions » [ASU86] sont calculées en considérant qu'une définition d'une variable `x` est une instruction qui affecte ou peut affecter une valeur à `x`. Les définitions les plus communes sont celles qui affectent directement une valeur à `x` ; elles sont considérées comme des définitions non ambiguës. Par opposition aux définitions non ambiguës, il existe les définitions ambiguës qui peuvent affecter une valeur à `x`, comme les appels à des procédures avec `x` comme paramètre (autre que le passage par valeur) ou bien une procédure qui peut avoir accès à la valeur de `x` parce que `x` est dans la portée de cette dernière. Comme autre définition ambiguë, nous pouvons citer les pointeurs qui font référence à `x`. Par exemple l'affectation `*q=y` est une définition de `x` si `q` pointe sur `x`. Cette ambiguïté peut être levée en utilisant l'information *points-to* comme nous le verrons avec l'exemple 3.6. Une définition `d` atteint (reach) un point `p` s'il existe un chemin à partir du point qui suit immédiatement `d` jusqu'à `p`, durant lequel `d` n'est pas « tuée ». Une définition d'une variable `a` est tuée si entre deux points il y a eu une lecture ou une affectation de la variable `a`. Par la suite, il a été jugé plus judicieux de conserver les informations apportées par les « reaching definitions » comme des chaînes « use-def » qui sont des listes pour chaque utilisation d'une variable ou bien pour toutes les définitions qui atteignent cette utilisation. Dans le compilateur PIPS, les chaînes « use-def » sont utilisées comme une première approximation du graphe de dépendance de données.

2.3.2 Suppression de code inutile

Si le résultat d'une instruction n'est jamais utilisé, cette instruction peut être considérée comme « morte » et supprimée [ASU86], sous l'hypothèse que cette instruction ne contienne pas de bugs provoquant ainsi une erreur d'exécution et provoquant l'arrêt de l'exécution. L'optimisation

qui permet la suppression de ces instructions peut être appelée suppression de code inutile ou suppression de code mort. Plus généralement cette optimisation concerne les instructions jamais atteintes, comme par exemple une instruction dans un test qui est toujours évalué à faux ou encore l'élimination des « use-def ».

2.3.3 Les chaînes *use-def* et la suppression de code inutile sans information *points-to*

Prenons comme exemple le programme 3.6, où la variable `i` est écrite une première fois avec l'instruction `i = 2` puis une deuxième fois avec l'instruction `*y = 1`. En effet, comme `x` pointe sur `i` et `x` et `y` sont des alias[4] l'instruction `*y=1` écrit dans la variable `i`.

```
int use_def_elim01() {
  int i, *x, *y;
  i = 2;
  x = &i;
  y = x;
  *y = 1;
  return *y;
}
```

Prog 2.6 – Exemple C pour la suppression de code inutile

Donc l'instruction `i = 2` est inutile parce cette valeur sera écrasée plus tard et n'est pas utilisée entre temps. La variable `i` sera mise à jour par le déréférencement du pointeur `y`. La phase de suppression de code inutile requiert le calcul des effets propres. Le résultat des effets propres sans l'information *points-to* est illustré par le programme 2.7.

```
int use_def_elim01() {
   int i, *x, *y;
//             <    is written>: i
   i = 2;
//             <    is written>: x
   x = &i;
//             <    is read   >: x
//             <    is written>: y
   y = x;
//             <may be written>: *ANY_MODULE*:*ANYWHERE*
//             <    is read   >: y
   *y = 1;
//             <may be read   >: *ANY_MODULE*:*ANYWHERE*
   return *y;
}
```

Prog 2.7 – Les effets propres **sans** information *points-to*

Sans information *points-to*, la case mémoire écrite par l'instruction `*y = 1` ne peut être connue. Par conséquent le calcul des chaînes *use-def* conclut à une dépendance entre la première et la deuxième affectation de `i` ; la suppression de l'affectation inutile ne peut être effectuée. Le graphe des chaînes *use-def*, ou plus précisément le graphe des dépendances, est illustré par la figure 2.1.

4. Deux pointeurs sont dits des alias si ils pointent vers le même emplacement mémoire

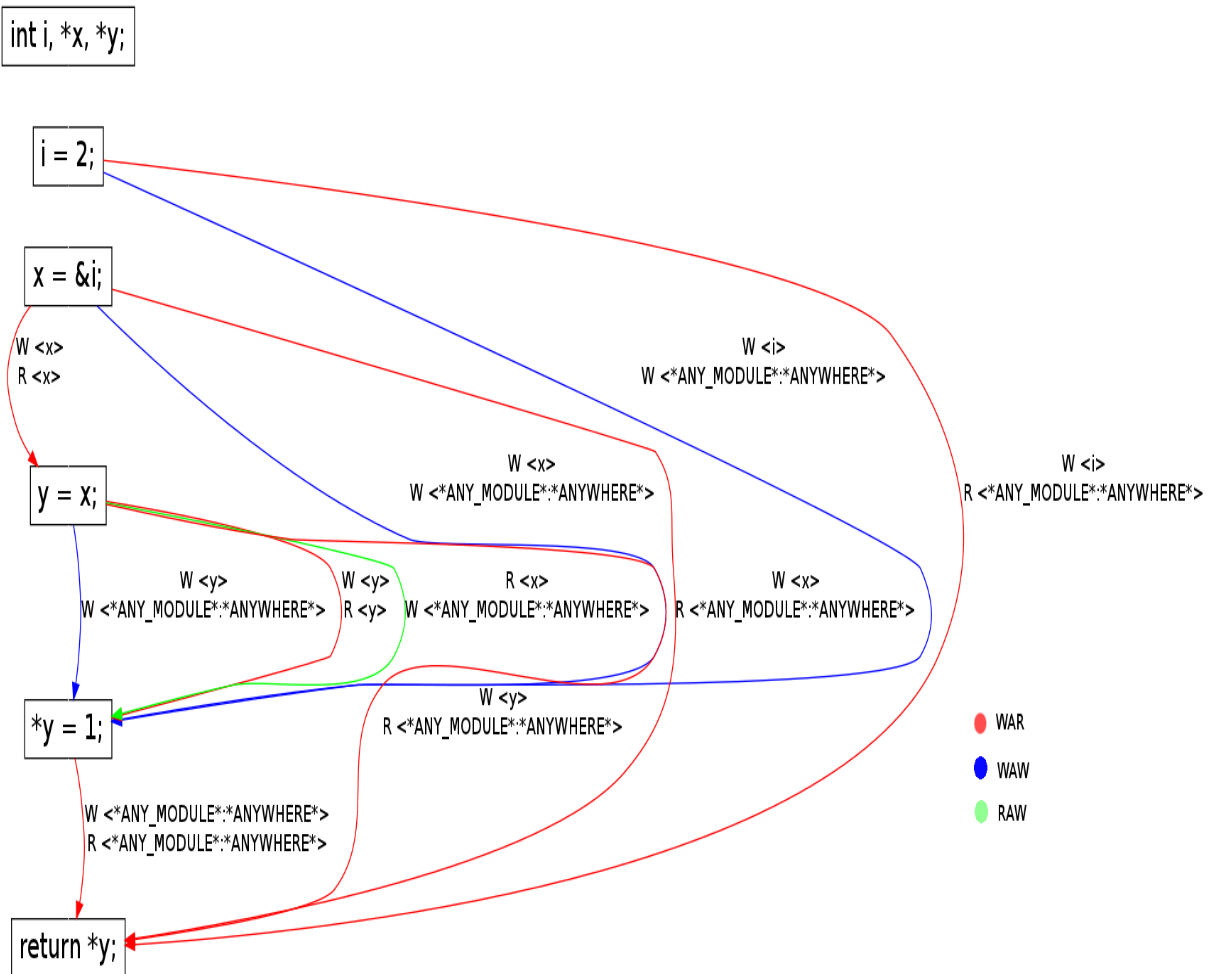


FIGURE 2.1 – Le graphe des « use-def » **sans** l'information *points-to* pour le programme 3.6

2.3.4 Chaînes *use-def* et suppression de code inutile avec l'information *points-to*

Les arcs *points-to* sont calculés au niveau du programme 2.8 pour pouvoir fournir à l'analyse des effets propres les cases mémoires qui sont lues ou écrites par les instructions de déréfrencement de pointeurs. Ici l'instruction `*y = 1` écrit la variable `i`.
Les effets propres calculés en utilisant les arcs *points-to* sont illustrés par le programme 2.9.

```
int use_def_elim01() {
 int i, *x, *y;

 i = 2;
 x = &i;
 y = x;

// Points To:
// x -> i , EXACT
// y -> i , EXACT
 *y = 1;

// Points To:
// x -> i , EXACT
// y -> i , EXACT
  return *y;
}
```

Prog 2.8 – Les arcs *points-to* pour le programme 3.6

Cette fois-ci, l'information que l'expression `*y` pointe la variable `i` est obtenue, ainsi que le fait qu'il y a une action d'écriture sur cette variable.

Le graphe de chaînes « use-def » est calculé une nouvelle fois, figure 2.2, où les emplacements abstraits `*ANY_MODULE*:*ANYWHERE*` ont disparu et un nouvel arc écriture après écriture de la variable `i` apparaît entre les instructions `i = 2` et `*y = 1`.

```
int use_def_elim01() {
   int i, *x, *y;
//            <    is written>: i
   i = 2;
//            <    is written>: x
   x = &i;
//            <    is read   >: x
//            <    is written>: y
   y = x;
//            <    is read   >: y
//            <    is written>: i
   *y = 1;
//            <    is read   >: i y
   return *y;
}
```

Prog 2.9 – Les effets propres obtenus **avec** l'information *points-to*

Nous observons maintenant que la première affectation à `i` n'est pas utilisée avant la deuxième affectation. Par conséquent la phase de suppression de code inutile avec l'information *points-to* permet la suppression de l'instruction `i = 2`.

```
int use_def_elim01(){
//           <    is declared>: i x y
   int i, *x, *y;

//           <    is written>: x
//           <    is referenced>: i x
   x = &i;
//           <    is read   >: x
//           <    is written>: y
//           <    is referenced>: x y
   y = x;
//           <    is read   >: y
//           <    is written>: i
//           <    is referenced>: y
   *y = 1;
//           <    is read   >: i y
//           <    is referenced>: y
   return *y;
```

Prog 2.10 – Résultat de la suppression de code inutile **avec** l'information *points-to*

2.3.5 Conclusion

Le résultat obtenu illustré par le programme 2.10 montre que nous pouvons utiliser les améliorations apportées à l'optimisation de suppression de code inutile comme métrique pour évaluer une analyse de pointeurs. Par exemple, le nombre d'instructions supprimées et le nouveau temps d'exécution après l'application de l'optimisation permettent de comparer deux analyses de pointeurs.

2.4 L'analyse des dépendances et la parallélisation

L'analyse de dépendances renseigne le compilateur sur l'ordre des exécutions des instructions. Ils existent deux types de dépendances :

1. les dépendances de contrôle : une instruction `i2` est dite dépendante de l'instruction `i1` si l'exécution ou non de la première est conditionnée par le résultat de la seconde ; cette dernière doit alors impérativement être exécutée en premier ;
2. les dépendances de données : deux instructions sont dépendantes si elles accèdent en lecture ou en écriture à la même donnée (sauf si ce sont deux lectures) ; leur ordre d'exécution doit être préservé.

L'analyse des dépendances s'étend à l'analyse des boucles et permet de déterminer les relations de dépendances entre les différentes occurrences des instructions du corps de la boucle. Elle permet de décider si une boucle peut être parallélisée ou non. Mais quand le programme contient des pointeurs, l'analyse a besoin d'une analyse de pointeurs effectuée au préalable. En effet, la parallélisation et l'optimisation se trouvent pénalisées par la présence de pointeurs , plus précisément, au niveau des boucles simples ou imbriquées, qui sont très présentes dans les applications de calcul scientifique ou du traitement de signal via des descripteurs de tableaux qui sont de type pointeur. Comme le langage `C` permet que l'accès à un élément de tableau et le déréférencement d'un pointeur aient la même notation, la parallélisation des boucles se trouve en

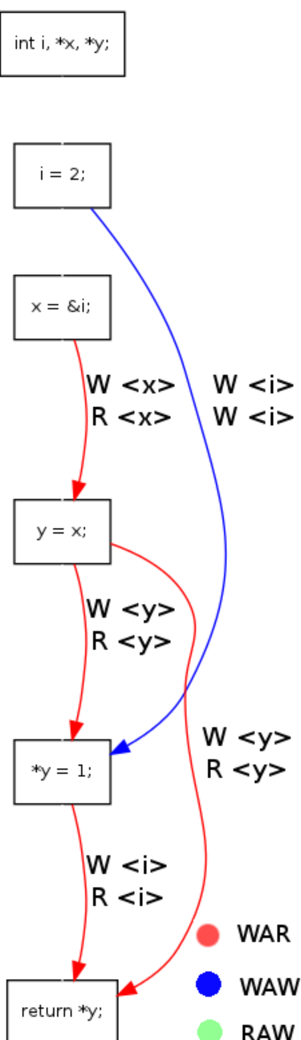


FIGURE 2.2 – Le graphe de dépendances **avec** l'information *points-to* pour le programme 3.6

conséquence inhibée. En effet, tant nous ne savons pas plus précisément quelle case mémoire est écrite ou lue, les algorithmes de parallélisation ne peuvent être exécutés.

2.4.1 L'analyse des dépendances et la parallélisation sans information *points-to*

Pour illustrer le problème de la parallélisation en absence d'information sur les pointeurs, prenons comme exemple le programme 2.11.

```
void pointer01(int n, float *p, float *q)
{
  int i;

  for(i=0; i<n; i++)
    p[i] = q[i];
}
```

Prog 2.11 – Programme C à paralléliser

Les accès `p[i]` et `q[i]` au niveau de la boucle sont des accès aux cases mémoires pointées par les pointeurs `p` et `q`. En absence d'information *points-to*, on se retrouve avec les effets propres du programme 2.12.

```
void pointer01(int n, float *p, float *q){
//             <    is declared>: i
   int i;
//             <    is read   >: n
//             <    is written>: i
//             <    is referenced>: i n

   for(i = 0; i <= n-1; i += 1)
//             <may be read  >: *ANY_MODULE*:*ANYWHERE*
//             <may be written>: *ANY_MODULE*:*ANYWHERE*
//             <    is referenced>: i n p q
      p[i] = q[i];
}
```

Prog 2.12 – Effets propres **sans** l'information *points-to* pour le programme 2.11

Les effets propres ainsi calculés ne peuvent donner d'information précise sur les variables écrites ou lues. Le compilateur conclut que toute la mémoire peut être lue et écrite. Le calcul des effets propres et celui du graphe de dépendances sont des phases requises pour pouvoir effectuer la parallélisation. Ils démontrent entre autres la nécessité de connaître les cibles des pointeurs. La figure 2.3 illustre le graphe de dépendances calculé sans l'information *points-to*.

Comme le graphe de dépendance est calculé à partir des effets propres, les dépendances qu'il affiche résultent de l'effet « `*ANY_MODULE*:*ANYWHERE*` ». On se retrouve avec des dépendances entre les itérations ainsi qu'entre l'indice de boucle, `p[i]` et `q[i]`. La parallélisation de la boucle ne peut pas être appliquée.

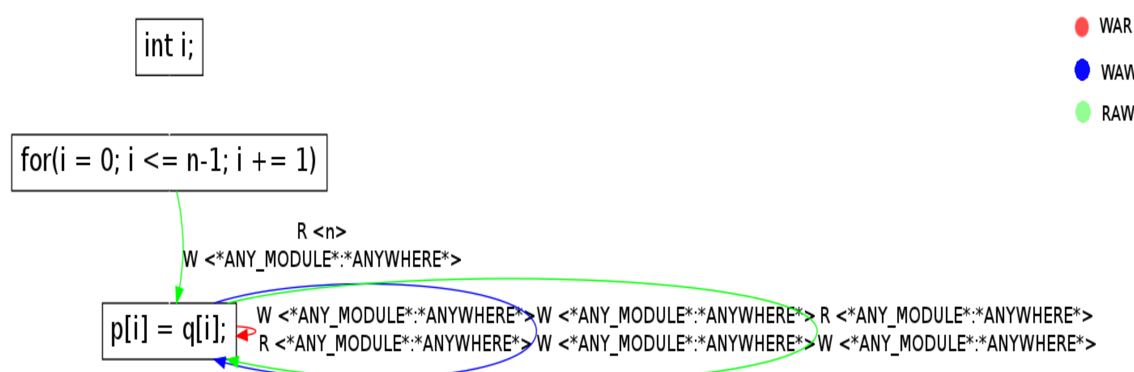


FIGURE 2.3 – Le graphe de dépendances **sans** l'information *points-to*

2.4.2 L'analyse des dépendances et la parallélisation avec l'information *points-to*

Nous calculons maintenant les arcs *points-to* (voir programme 2.13). Comme nous sommes dans le cadre de l'analyse intraprocédurale, nous n'avons pas le contexte d'appel de la fonction. C'est pour cette raison que nous créons des emplacements abstraits qui représentent les cibles des paramètres effectifs ; leurs noms sont dérivés à partir de ces derniers. La construction du contexte d'appel est détaillé dans le chapitre 4. L'analyse nous permet de déterminer quelle case mémoire est écrite par le déréférencement du pointeur `p` et quelle case mémoire est lue par le déréférencement du pointeur `q`.

```
void pointer01(int n, float *p, float *q){
   int i;
// Points To:
// p -> _p_2 , EXACT
// q -> _q_3 , EXACT

   for(i = 0; i <= n-1; i += 1)
// Points To:
// p -> _p_2 , EXACT
// q -> _q_3 , EXACT
      p[i] = q[i];
}
```

Prog 2.13 – Les arcs *points-to* pour le programme 2.11

On fournit maintenant cette information à l'analyse des effets, programme 2.14 ; on a maintenant l'information que `p[i]` écrit `_p_2[i]` et que `q[i]` lit `_q_3[i]`.

Le calcul de dépendances peut maintenant nous fournir les dépendances réelles et significatives entre les variables et les itérations de boucle. Comme on n'écrit pas la même case en déréférençant les pointeurs `p` et `q` [5], le graphe de dépendance, figure 2.4, ne montre aucune dépendance au niveau de la boucle. La parallélisation du code est maintenant possible ; la boucle peut être parallélisée grâce à l'application de l'algorithme de Allen & Kennedy ou celui de la parallélisation gros grain, implémenté par des phases de PIPS. Le résultat est un code décoré avec des pragmas OpenMP [ope] au niveau des boucles qui peuvent être exécutées en parallèle (voir le programme 2.15).

5. Nous travaillons sous l'hypothèse qu'il n'existe pas d'aliasing entre les paramètres

```
void pointer01(int n, float *p, float *q){
//             <    is declared>: i
   int i;
//             <    is read   >: n
//             <    is written>: i
//             <    is referenced>: i n

   for(i = 0; i <= n-1; i += 1)
//             <    is read   >: _q_3[i] i n p q
//             <    is written>: _p_2[i]
//             <    is referenced>: i n p q
      p[i] = q[i];
}
```

Prog 2.14 – Les effets propres **avec** les arcs *points-to* pour le programme 2.11

FIGURE 2.4 – Le graphe de dépendances **avec** l'information *points-to*

```
void pointer01(int n, float *p, float *q)
{
   int i;
#pragma omp parallel for
   for(i = 0; i <= n-1; i += 1)
      p[i] = q[i];
}
```

Prog 2.15 – Programme parallélisé

2.4.3 Conclusion

Avec une information *points-to* précise sur les accès aux tableaux via des pointeurs, la parallélisation des boucles devient possible. Par conséquent le nombre de boucles parallèles dans un programme peut être utilisé comme une métrique pour comparer deux analyses de pointeurs. Bien sûr cette métrique doit être suivie d'une comparaison du temps d'exécution pour décider de l'efficacité de la parallélisation.

2.5 La suppression de sous-expressions communes

Les expressions ou les sous-expressions communes sont des expressions préalablement calculées et dont les variables n'ont pas changé depuis le dernier calcul. On désigne ici par sous-expressions locales les expressions calculées à l'intérieur d'un bloc unique, et par sous-expressions globales les expressions préalablement calculées pour plusieurs blocs consécutifs.

2.5.1 La suppression de sous-expressions communes sans information sur les pointeurs

En présence de pointeurs, la tâche d'évaluation des expressions devient plus délicate. En l'absence d'information sur les pointeurs, le résultat est sur-approximé et ne permet aucune optimisation, comme on le voit au niveau du programme 2.16.

```
int cse_wpt01()
{
   int i = 0;
   int j = 1;
   int k;
   int *p = &j;
   int *q = &k;
   i = 2*(j+2);
   *q = *p;
   k = 3*(j+2);
}
```

Prog 2.16 – Programme avec expression commune j+2

Une première lecture du programme conclut à la présence d'une expression commune pour les instructions `i = 2*(j+2)` et `k = 3*(j+2)`, en considérant qu'entre les deux instructions il n y a pas eu d'écriture de la variable `j`. Mais si les effets propres sont calculés pour déterminer la liste des variables lues et écrites par les instructions, nous obtenons les effets du programme 2.17.

Sans l'information *points-to*, les déréférencements des deux pointeurs `p` et `q` sont considérés comme des lectures et des écritures sur toute la mémoire modélisée par `*ANYWHERE*` et donc sur la variable `j`.

2.5.2 La suppression de sous-expressions communes avec l'information sur les pointeurs

Les arcs *points-to* sont calculés au niveau du programme 2.18. L'analyse *points-to* permet de s'apercevoir que par l'instruction `*q = *p` c'est la variable `k` qui est écrite et donc l'expression `j+2` est commune. Sans cette information la phase de calcul des effets n'autorise pas la suppression des expressions communes par sûreté pour ne pas produire de code potentiellement

```
int cse_wpt01()
{
//             <may be read  >: *ANY_MODULE*:*ANYWHERE*
//             <may be written>: *ANY_MODULE*:*ANYWHERE*
//             <    is written>: i j
   int i = 0, j = 1, k;
//             <    is written>: p
   int *p = &j;
//             <    is written>: q
   int *q = &k;
//             <    is read   >: j
//             <    is written>: i
   i = 2*(j+2);
//             <may be read  >: *ANY_MODULE*:*ANYWHERE*
//             <may be written>: *ANY_MODULE*:*ANYWHERE*
   *q = *p;
//             <    is read   >: j
//             <    is written>: k
   k = 3*(j+2);
}
```

Prog 2.17 – Les effets propres pour le programme 2.16

```
int cse_wpt01()
{
   int i = 0, j = 1, k;
   int *p = &j;

// Points To:
// p -> j , EXACT
   int *q = &k;

// Points To:
// p -> j , EXACT
// q -> k , EXACT
   i = 2*(j+2);

// Points To:
// p -> j , EXACT
// q -> k , EXACT
   *q = *p;

// Points To:
// p -> j , EXACT
// q -> k , EXACT
   k = 3*(j+2);
}
```

Prog 2.18 – Les arcs *points-to* pour le programme 2.16

```
int cse_wpt01()
{
//             <    is written>: i j
   int i = 0, j = 1, k;
//             <    is written>: p
   int *p = &j;
//             <    is written>: q
   int *q = &k;
//             <    is read   >: j
//             <    is written>: i
   i = 2*(j+2);
//             <    is read   >: j p q
//             <    is written>: k
   *q = *p;
//             <    is read   >: j
//             <    is written>: k
   k = 3*(j+2);
}
```

Prog 2.19 – Les effects propres avec l'information *points-to* pour le programme 2.16

faux. Avec l'information *points-to*, nous connaissons exactement la variable qui a été écrite et le programmeur ou le compilateur peut réorganiser le code, quand c'est possible, pour permettre l'optimisation.

2.5.3 Conclusion

L'impact de l'analyse de pointeurs sur la suppression des sous-expressions communes peut être utilisé comme métrique pour l'évaluer. Plus le nombre de sous-expressions communes découvertes est important, plus l'analyse est potentiellement utile.

2.6 Le renommage des scalaires

Cette optimisation consiste à introduire des variables scalaires temporaires dans le programme pour stocker des résultats temporaires. Le but est de donner un nom unique à chaque valeur d'un scalaire. Le renommage des scalaires permet de minimiser les dépendances entre instructions.

2.6.1 Le renommage des scalaires sans information *points-to*

Le renommage produit un nouveau code à partir de l'ancien en se fondant sur les effets propres ainsi que sur le graphe de dépendances. Comme il a déjà été montré dans les sections 2.2 et 2.4, l'analyse des pointeurs améliore notablement le calcul des effets propres et le graphe de dépendances et, par conséquent, le renommage des scalaires. Le programme 2.20 [Ran01] présente des dépendances qui peuvent être éliminées. Calculons les effets propres du pro-

```
int scalar_renaming01()
{
  int *p, i, T, A[100], B[100], C[100], D[100];
  p = &T;
  for(i = 0; i<100; i++) {
    T = A[i] + B[i];             /* S1 */
    C[i] = T + T;             /* S2 */
    T = D[i] - B[i];           /* S3 */
    *p = 1;                 /* S4 */
    A[i] = T*T;               /* S5 */
  }

  return T;
}
```

Prog 2.20 – Renommage des variables scalaires

gramme sans calculer au préalable les arcs *points-to*. Il y a un effet `*ANYWHERE*` qui apparaît au niveau de l'instruction `*p = 1`. Le déréférencement du pointeur `p` écrit la variable `T`. Comme l'emplacement `*ANYWHERE*` désigne n'importe quel emplacement de la mémoire, le graphe de dépendances montre les dépendances entre l'instruction de déréférencement et toutes les autres instructions.

```
int scalar_renaming01()
{
   int *p, i, T, A[100], B[100], C[100], D[100];
//             <    is written>: p
   p = &T;
//             <    is written>: i
   for(i = 0; i <= 99; i += 1) {
//             <    is read   >: A[i] B[i] i
//             <    is written>: T
      T = A[i]+B[i];
//             <    is read   >: T i
//             <    is written>: C[i]
      C[i] = T+T;
//             <    is read   >: B[i] D[i] i
//             <    is written>: T
      T = D[i]-B[i];
//             <may be written>: *ANY_MODULE*:*ANYWHERE*
//             <    is read   >: p
      *p = 1;
//             <    is read   >: T i
//             <    is written>: A[i]
      A[i] = T*T;
   }
//             <    is read   >: T

   return T;
}
```

Prog 2.21 – Les effets propres **sans** l'information *points-to* pour le programme 2.20

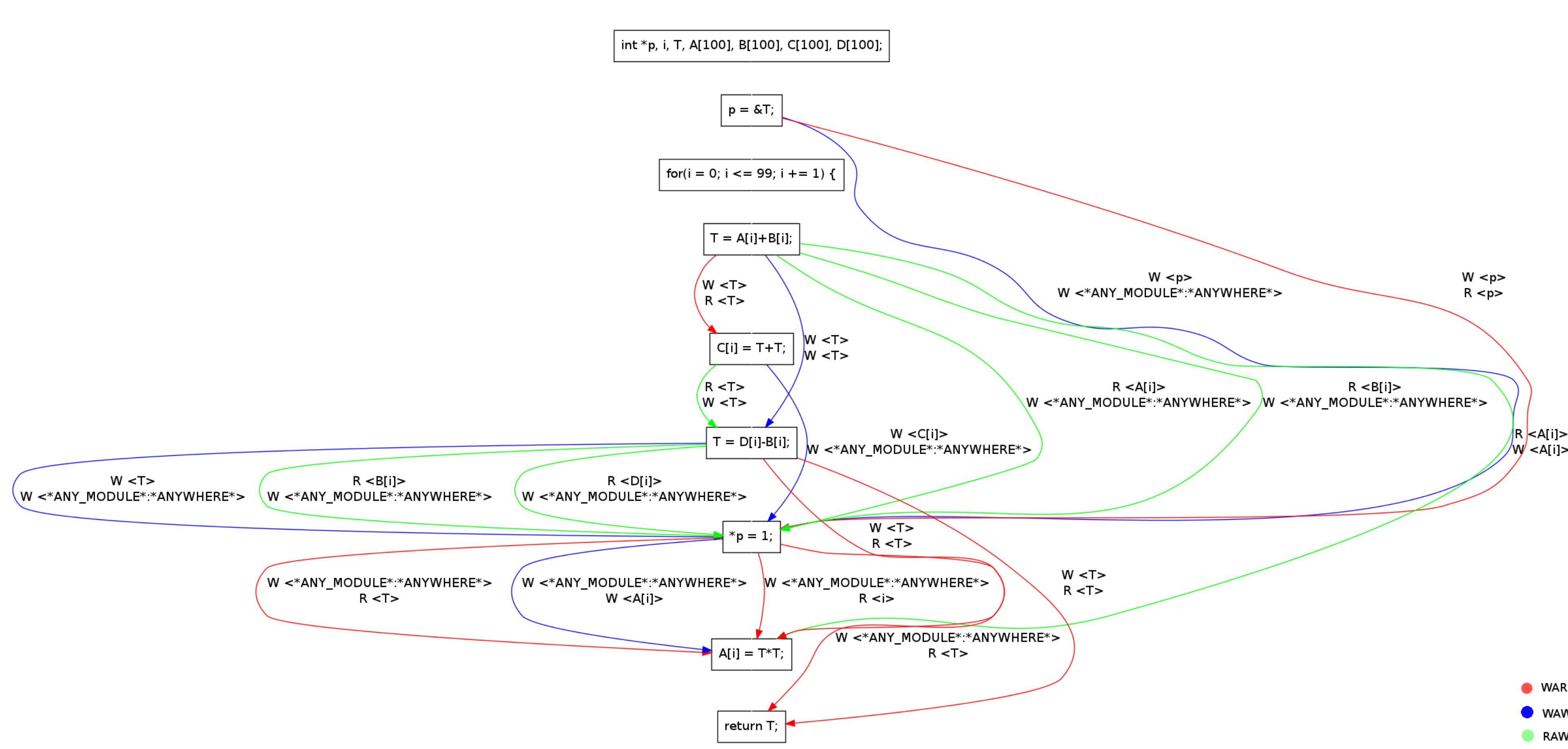


FIGURE 2.5 – Le graphe de dépendances pour le programme `scalar_renaming03` sans l'information *points-to*

Des fausses dépendances, lecture après écriture, entre S1 et S5 peuvent apparaître alors que T a été mis à jour au niveau de S3 puis au niveau de S4 lors du déréférencement du pointeur p. Mais sans l'information *points-to*, dans le programme 2.21, le calcul des effets propres nécessaires à la construction du graphe de dépendance est imprécis.

2.6.2 Le renommage des scalaires avec l'information *points-to*

Pour supprimer les dépendances qui sont le résultat de l'effet d'écriture sur *ANYWHERE*, nous calculons en premier lieu les arcs *points-to*.

```
int scalar_renaming01()
{
   int *p, i, T, A[100], B[100], C[100], D[100];
   p = &T;

// Points To:
// p -> T , EXACT
   for(i = 0; i <= 99; i += 1) {
// Points To:
// p -> T , EXACT
      T = A[i]+B[i];
// Points To:
// p -> T , EXACT
      C[i] = T+T;
// Points To:
// p -> T , EXACT
      T = D[i]-B[i];
// Points To:
// p -> T , EXACT
      *p = 1;
// Points To:
// p -> T , EXACT
      A[i] = T*T;
   }
// Points To:
// p -> T , EXACT

   return T;
}
```

Prog 2.22 – Les arcs *points-to* pour le programme 2.20

Le programme génère un seul arc *points-to* entre le pointeur p et la variable T. Cela implique que tout déréférencement du pointeur est équivalent à une écriture ou lecture de T.

Les effets propres sont calculés après la génération des arcs *points-to* au niveau du programme 2.23. On retrouve les dépendances précédemment décrites, en particulier les dépendances écriture après écriture entre les instructions :

- `T = A[i]+B[i];`
- `T = D[i]-B[i];`
- `*p = 1;`

Ces dépendances sont calculées sous forme de graphe de dépendance, voir figure 2.5.

Le renommage de scalaire peut être maintenant appliqué, voir programme 2.24. Ceci va permettre de supprimer les fausses dépendances au niveau de la boucle et aussi d'appliquer, par la suite, une autre optimisation : la vectorisation.

```
int scalar_renaming01()
{
   int *p, i, T, A[100], B[100], C[100], D[100];
//             <    is written>: p
   p = &T;
//             <    is written>: i
   for(i = 0; i <= 99; i += 1) {
//             <    is read   >: A[i] B[i] i
//             <    is written>: T
      T = A[i]+B[i];
//             <    is read   >: T i
//             <    is written>: C[i]
      C[i] = T+T;
//             <    is read   >: B[i] D[i] i
//             <    is written>: T
      T = D[i]-B[i];
//             <    is read   >: p
//             <    is written>: T
      *p = 1;
//             <    is read   >: T i
//             <    is written>: A[i]
      A[i] = T*T;
   }
//             <    is read   >: T

   return T;
}
```

Prog 2.23 – Les effets propres **avec** l'information *points-to* pour le programme 2.20

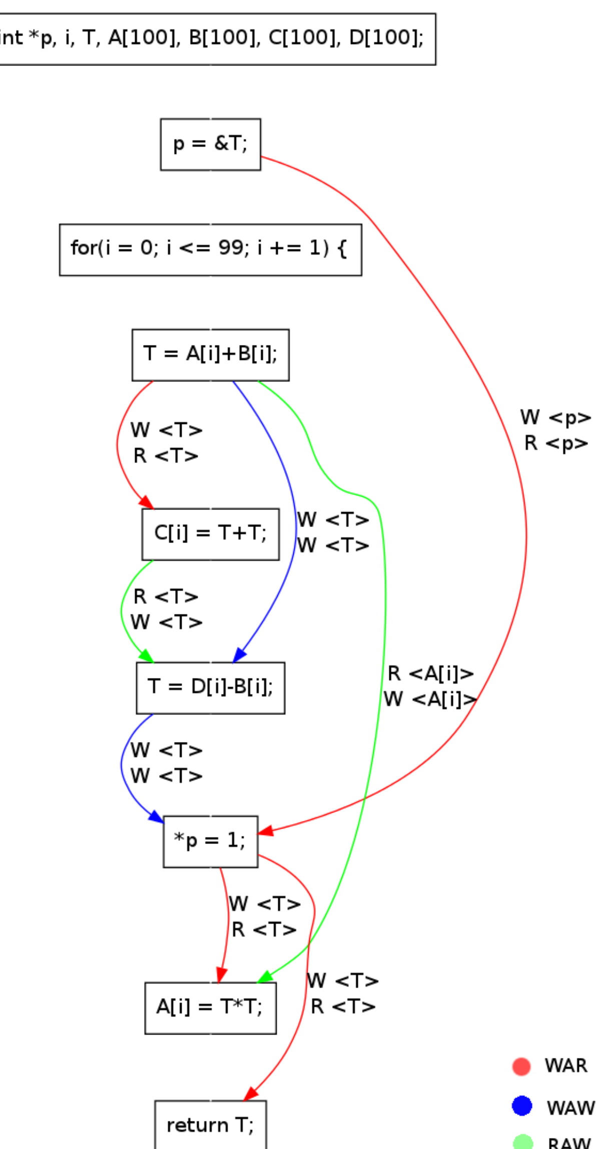


FIGURE 2.6 – Le graphe de dépendances avant le renommage des scalaires

```
int scalar_renaming01()
{
//             <    is declared>: A B C D T p
   int *p, T, A[100], B[100], C[100], D[100];
   //PIPS generated variable
//             <    is declared>: T0 T1 T2 i0
   int i0, T0, T1, T2;
//             <    is written>: p
//             <    is referenced>: T p
   p = &T;
//             <    is written>: i0
//             <    is referenced>: i0
   for(i0 = 0; i0 <= 99; i0 += 1) {
//             <    is read   >: A[i0] B[i0] i0
//             <    is written>: T0
//             <    is referenced>: A B T0 i0
      T0 = A[i0]+B[i0];
//             <    is read   >: T0 i0
//             <    is written>: C[i0]
//             <    is referenced>: C T0 i0
      C[i0] = T0+T0;
//             <    is read   >: B[i0] D[i0] i0
//             <    is written>: T1
//             <    is referenced>: B D T1 i0
      T1 = D[i0]-B[i0];
//             <    is read   >: p
//             <    is written>: T
//             <    is referenced>: p
      *p = 1;
//             <    is read   >: T2 i0
//             <    is written>: A[i0]
//             <    is referenced>: A T2 i0
      A[i0] = T2*T2;
   }
//             <    is read   >: T2
//             <    is referenced>: T2

   return T2;
}
```

Prog 2.24 – Le programme après le renommage de scalaires

Le graphe de dépendance est généré une nouvelle fois, figure 2.7, et les dépendances entre les instructions écrivant `T` y ont disparu. Des optimisations sur le calcul de données au niveau des instructions du corps de la boucle peuvent être maintenant appliquées.

2.7 Conclusion

Pour évaluer une analyse de pointeurs les chercheurs se tournaient jusqu'à vers des métriques comme le temps d'exécution, l'espace mémoire utilisé ou encore la taille de l'ensemble final des arcs *points-to*. Mais cette évaluation est incomplète car une analyse de pointeurs à elle seule ne peut être évaluée utilement. En effet, tant que l'information sur les pointeurs n'a pas été fournie aux autres analyses clientes du compilateur, et tant qu'une comparaison des résultats de ces dernières avec et sans information *points-to* n'a pas été effectuée, nous ne pouvons évaluer la précision et l'intérêt de l'analyse.

Dans ce chapitre nous avons montré l'impact de l'analyse de pointeurs sur les analyses clientes [Rak01] [GH98] [CH00] [DMM98] qui en absence d'information *points-to* produisent des informations imprécises, augmentent considérablement le temps d'exécution, voire ne peuvent pas être appliquées à des programmes contenant des pointeurs. À travers des exemples nous avons vu la précision apportée par l'analyse de pointeurs à :

- la suppression de sous-expressions communes,
- le test de dépendances lors des accès aux tableaux,
- la détection d'invariant de boucle [GH98],
- la facilité de debuggage et la compréhension des programmes,
- la suppression de code inutile.

Ces optimisations peuvent être utilisées comme des métriques pour évaluer et comparer les analyses de pointeurs. En comptant par exemple le nombre d'instructions qui peuvent être supprimées lors de la suppression de code inutile ou le nombre de boucles parallélisables lors de la parallélisation automatique ou encore le nombre des sous-expressions communes découvertes. D'autres études ont aussi été effectuées sur l'amélioration des transformations en présence d'analyse de pointeurs comme la suppression et/ou l'ordonnancement des chargements mémoire, la compilation dynamique et le préchargement des pointeurs vers les structures de données [GH98] et copies mémoire superflues [DMM98] [CH00]. Dans le chapitre 7, on étudiera l'impact sur l'efficacité en terme de temps d'exécution. Le gain en précision pour certaines transformations vaut la peine d'appliquer une analyse très précise et par conséquent coûteuse. Pour d'autres transformations une information imprécise nous permet quand même d'appliquer des optimisations.

Le gain en précision que nous avons observé justifie une analyse *points-to* relativement coûteuse mais dans les limites d'une passe de compilation usuelle. Nous proposons donc d'effectuer des analyses intraprocédurales pour en limiter le coût tout en obtenant une bonne précision, sous une hypothèse de non aliasing des paramètres formels et des variables globales.

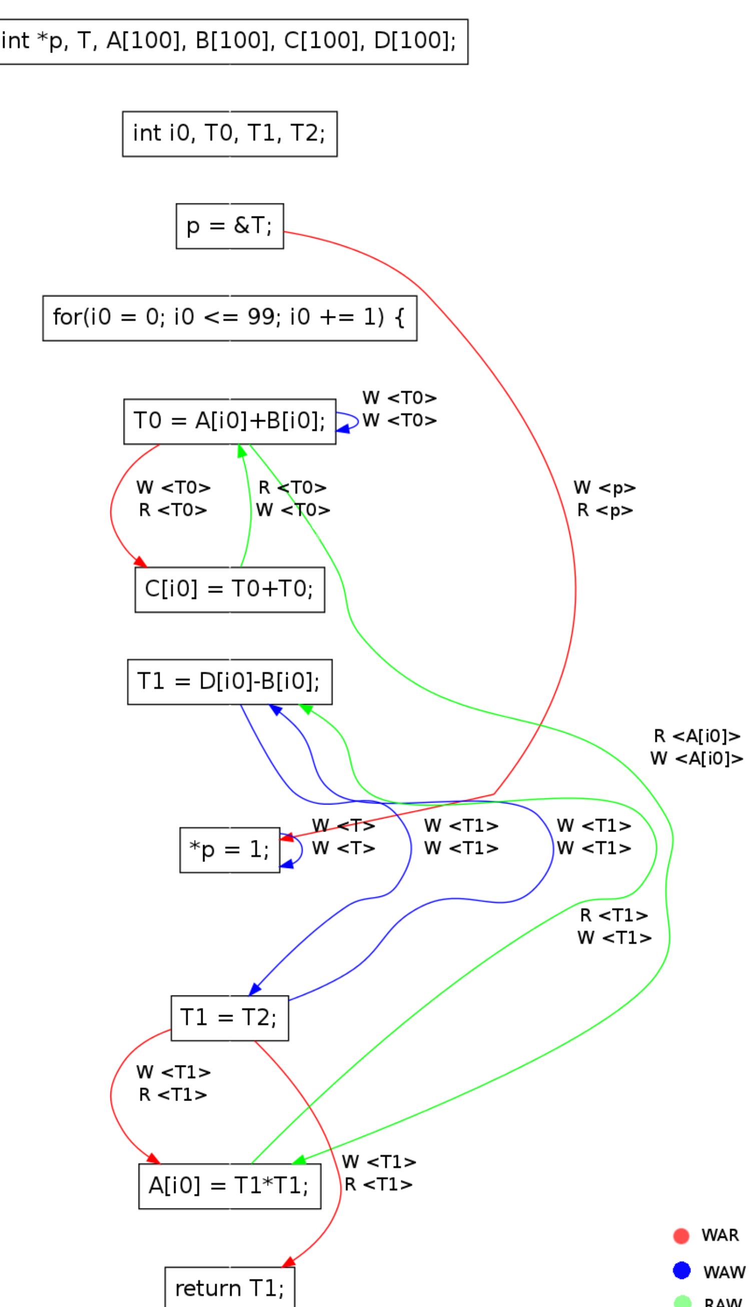


FIGURE 2.7 – Le graphe de dépendances après le renommage des scalaires

CHAPITRE 3

État de l'art : Les analyses de pointeurs

Dans la littérature, plusieurs dimensions permettent de caractériser les analyses de pointeurs. Parmi ces dimensions, citons :

1. le flot de contrôle : s'il est pris en compte, c'est-à-dire que l'ordre des instructions a une importance, et qu'il y a une mise à jour des arcs *points-to* à chaque point de contrôle l'analyse est dite sensible au flot ; dans le cas contraire c'est une analyse insensible au flot ;
2. le contexte d'appel : s'il est pris en compte, c'est-à-dire qu'à chaque appel de fonction les arcs *points-to* sont mis à jour en fonction des *points-to* calculés dans le corps de la fonction, l'analyse est dite sensible au contexte ; sinon elle est dite insensible au contexte ;
3. l'évaluation des conditions dans les branchements : si elle est prise en compte l'analyse est dite sensible aux chemins ;
4. les champs pointeurs dans les structures de données, *field-sensitivity* : ils sont représentés par un seul emplacement ou une modélisation plus sophistiquée ;
5. la mémoire allouée dynamiquement, appelé le tas : il est représenté par un seul emplacement ou une autre modélisation donnant plus d'information ;

Deux analyses insensibles au flot de contrôle dominent : l'analyse d'Andersen [And94] et de Steensagard [Ste96]. La première a été améliorée et intégrée au compilateur gcc [Hei01]. La deuxième est intégrée dans le compilateur LLVM [LLV] avec un meilleur traitement des champs pointeurs de structures. Ces deux algorithmes sont détaillés dans la section 3.1. Quant aux analyses sensibles au flot de contrôle, nous nous sommes intéressés à l'analyse d'Emami [Ema93]. Finalement concernant les analyses sensibles au contexte nous présentons celle de Wilson [WHI95] ainsi que celle d'Emami [Ema93]. Cette dernière prend la plus grande partie de ce chapitre, compte tenu du fait que notre travail s'est initialement inspiré de ce travail avant de l'étendre pour traiter l'ensemble des instructions du langage C ainsi que le type `struct` ou encore les tableaux, sans pour autant passer par une simplification du code mais plutôt en travaillant directement sur le code source.

Dans la suite du manuscrit nous utilisons les différentes analyses sur un exemple écrit en langage `C` qui évoluera au fur et à mesure des analyses. Nous commençons par des affectations de pointeurs simples, puis nous ajoutons des appels à `malloc`, des structures de données récursives pour aboutir à la fin à un exemple interprocédural pour illustrer la propagation des arcs *points-to*. L'exemple initial est donné dans le programme 3.1.

3.1 Les analyses insensibles au flot et au contexte

Une analyse est dite sensible au flot lorsque son calcul suit le flot de contrôle et calcule ce vers quoi pointe chaque variable après chaque instruction. C'est-à-dire que l'ensemble des arcs *points-to* augmente en fonction des arcs générés au niveau de l'instruction en cours mais diminue aussi par l'élimination des arcs supprimés par cette dernière.

```
int main()
{
  int *p, *q, i, j;

  i = 1;
  j = 2;
  p = &i;
  q = &j;
  p = q;

  return 0;
}
```

Prog 3.1 – Exemple C pour l'analyse intraprocédurale

Ceci est appelé mise à jour forte en opposition à la mise à jour faible où l'instruction en cours ne fait qu'ajouter de nouveaux arcs [ASU86]. Ici c'est la mise à jour faible qui est adoptée ; les instructions ne suppriment pas les anciens arcs [ASU86].

Quant à l'analyse interprocédurale une approche simpliste consisterait à traiter les sites d'appel comme de simples instructions, sans chercher à modéliser le comportement de la procédure ni son impact sur l'appelant.

3.1.1 L'analyse d'Andersen

Introduite par Andersen [And94], et reprise par Heintz [Hei99] [Hei01] [HT01], cette approche permet l'analyse de programmes de plusieurs millions de lignes de code. On peut considérer que, pour un programme donné, contenant des pointeurs, les arcs entre eux peuvent être formulés sous la forme de combinaisons des contraintes suivantes :

$$p \supseteq q \;\mid\; p \supseteq \{q\} \;\mid\; p \supseteq *q \;\mid\; *p \supseteq q \;\mid\; *p \supseteq \{q\} \tag{3.1}$$

FIGURE 3.1 – Les contraintes de l'analyse d'Andersen

où q et p sont les variables contraintes, « $*$ » l'opérateur de déréférencement et la contrainte « $p \supseteq \{q\}$ » signifiant que q appartient à l'ensemble des variables pointées par p. Par conséquent $p \supseteq \{x\}$ stipule que p pointe vers x. On traduit alors le programme source dans ce langage en liant chaque variable à une unique variable contrainte et en convertissant les affectations en des contraintes.

L'analyse d'Andersen se fait vers l'avant ; elle n'est sensible ni au contexte ni au flot de contrôle. L'analyse ne représente pas les alias sous forme de paires comme la plupart des analyses d'aliasing [1], mais comme un lien entre la variable pointeur et l'ensemble des variables vers lesquelles elle pointe. Par exemple, pour les instructions du programme 3.2.

```
p = &x;
p = &y;
```

Prog 3.2 – Instructions C génératrices de contraintes Andersen

l'analyse génère les contraintes de la figure 3.2. Ces contraintes peuvent être traduites sous la forme d'une fonction Mem, avec $\{x, y\} \in Mem(p)$, qui est une représentation plus compacte, plus économique en terme d'espace de stockage et plus adéquate pour les autres analyses. Au

1. Appelées analyses de synonymes en français

$$p \supseteq \{x\}$$
$$p \supseteq \{y\}$$

FIGURE 3.2 – Contraintes pour le programme 3.2

sein de cette analyse, les affectations du programme en cours sont vues comme des contraintes et classifiées en trois types : type de base, type simple et type complexe suivant la topologie [Ray05] :

- des contraintes de base `a = &b`, dans le cas où l'opérateur « & » se situe au niveau de la partie droite de l'affectation ;
- des contraintes simples `c = a`, dans le cas d'une affectation simple sans variable de type pointeur ;
- des contraintes complexes `*a = z`, dans le cas où l'opérateur « * » se situe dans la partie gauche de l'affectation.

Les affectations de type `*a = *b` peuvent être décomposées en contraintes complexes en introduisant des variables temporaires, pour obtenir `*a = z` et `z = *b`. Plus généralement, les expressions sont simplifiées par introduction de temporaires. Pour notre exemple `C`, l'analyse d'Andersen génère les contraintes suivantes :

$$\begin{aligned} p &\supseteq \{i\} \\ q &\supseteq \{j\} \\ p &\supseteq q \\ p &\supseteq \{i, j\} \end{aligned}$$

L'analyse itère sur l'ensemble *points-to* jusqu'à atteindre un point fixe.

Les analyses de pointeurs peuvent être aussi représentées sous forme de graphe où les nœuds représentent les cases mémoire et les arcs les relations *points-to*. Cette représentation permet une première évaluation de la précision de l'analyse ; si un nœud représente plus d'un emplacement mémoire, cela veut dire que des emplacements ont été fusionnés et que des relations *points-to* ont été sur-approximées. Dans le cas de l'analyse d'Andersen, nous avons un nombre arbitraire d'arcs et, malgré la multiplication des arcs, chaque nœud représente une seule variable, ce qui accroît la précision des résultats.

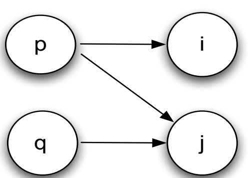


FIGURE 3.3 – Graphe résultat de l'analyse de Andersen sur notre exemple

3.1.1.1 Algorithme d'Andersen : prise en compte des appels de 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` du programme 3.3. L'analyse fait pointer les paramètres formels vers les paramètres effectifs, pour

```
int* foo(int *x)
{
  x = malloc(4*sizeof(int));
  return x;
}
int main()
{
  int *b, *a;
  a = foo(b);
  return 0;
}
```

Prog 3.3 – Fonction foo

aboutir aux contraintes suivantes :

$$x \supseteq b \equiv x = b$$
$$a \supseteq x \equiv a = x$$

L'ensemble des arcs *points-to* est augmenté par ceux générés par les paramètres formels.

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

Dans [PKH04], à part le traitement des pointeurs, l'auteur détaille aussi le traitement des variables de type structure de données, `struct`. Pour comprendre les différentes approches, prenons l'exemple tiré de l'article de Ghiya [Rak01] :

```
int main(){
struct foo {int *p; int *q;} s1, s2;
int x,y;
s1.p = &x;
s2.q = &y;
}
```

Prog 3.4 – Modélisation des champs pointeurs

- la structure `foo` peut être modélisée comme un seul objet représentant la structure ainsi que ses champs : cette approche est appelée *field-insensitive* ;
- toutes les instances d'un même champ sont modélisées comme une seule variable c'est à dire que `s1.p` et `s2.p` sont fusionnés en un seul objet : cette approche est appelée *field-based* ;
- chaque instance d'un champ est modélisée avec une variable séparée : cette approche est appelée *field-sensitive*.

C'est la dernière approche qui a été adoptée par [PKH04]. Cette approche apporte beaucoup plus de précision à l'analyse mais peut devenir vite coûteuse si l'application contient un nombre important de structures de données récursives.

3.1.2 L'analyse de Steensgaard

L'analyse de Steensgaard [Ste96] [Ray05] est la même que celle d'Andersen [And94], mais elle est simplifiée par la fusion de deux emplacements mémoire en une seule classe d'équivalence s'ils sont pointés par le même pointeur (la nature des nœuds est modifiée). Un algorithme *union-find* est appliqué. L'analyse est formalisée sous la forme d'un système de typage, en appliquant des contraintes d'unification à des affectations de pointeurs. Elle est fondée sur trois composantes : un système de typage, des règles de typages et des règles d'inférence.

1. Le système de typage n'est pas standard, et il permet de décrire un *storage shape graph* (SSG), où les nœuds représentent un ou plusieurs emplacements mémoires et les arcs des relations *points-to*. Les types sont affectés aux valeurs, aux emplacements mémoires ainsi qu'aux fonctions. Les types qui sont affectés aux valeurs sont sous la forme de paires :

$$(type\ de\ l'emplacement\ \times\ type\ de\ la\ signature\ de\ fonction)$$

Les types sont définis par des règles de production :

$$\begin{aligned} \alpha &::= \tau \times \lambda \\ \tau &::= \bot | \text{ref}(\alpha) \\ \lambda &::= \bot | \lambda(\alpha_1..\alpha_n)(\alpha_{n+1}..\alpha_{n+m}) \end{aligned} \tag{3.2}$$

FIGURE 3.4 – Les règles de production de Steensgaard

où τ est un pointeur vers une variable, λ est un pointeur vers une fonction et $\bot$ la valeur désignant un type non pointeur.

2. Les règles de typage sont fondées sur le système de typage et permettent d'attester qu'un programme est correctement typé. Un programme est bien typé, dans un environnement A, si toutes les instructions sont correctement typées. Voici par exemple la règle de typage pour l'affectation $x = y$:

$$\frac{\begin{array}{r} A \vdash x : \text{ref}(\alpha_1) \\ A \vdash y : \text{ref}(\alpha_2) \\ \alpha_1 \trianglelefteq \alpha_2 \end{array}}{A \vdash \text{welltyped}(x = y)} \tag{3.3}$$

FIGURE 3.5 – La règle de typage pour l'affectation $x = y$

3. Les règles d'inférence pour résoudre le système de typage permettent de fusionner des types en classe d'équivalence « ecr » (*equivalence class representative*), comme par exemple pour l'affectation $x = y$:

$$\begin{array}{r} \text{let ref}(\tau_1 \times \lambda_1) = \text{type}(\text{ecr}(x)) \\ \text{ref}(\tau_2 \times \lambda_2) = \text{type}(\text{ecr}(y)) \\ \text{if } \tau_1 \neq \tau_2 \text{ then cjoin}(\tau_1, \tau_2) \\ \text{if } \lambda_1 \neq \lambda_2 \text{ then cjoin}(\lambda_1, \lambda_2) \end{array}$$

FIGURE 3.6 – La règle d'inférence pour l'affectation $x = y$

L'algorithme commence par une phase d'initialisation de toutes les variables au type $(\bot \times \bot)$. Ensuite chaque instruction est analysée une seule fois en appliquant les règles de typage, pour affecter à chaque variable le type correspondant. En dernier lieu, les règles d'inférence sont appliquées afin de permettre d'unifier les classes d'équivalence.

Pour le programme 3.1, après application de l'algorithme de Steensagaard [Ste96] nous obtenons :

$$p = q \Rightarrow Points_to(p) = Points_to(q)$$

Ceci permet de construire le graphe de la figure 3.7 où les variables `p` et `q` sont fusionnées en un seul nœud, tout comme les variables `i` et `j`.

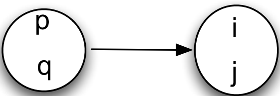


FIGURE 3.7 – Graphe résultat de l'analyse de Steensgaard

On voit sur cet exemple la sur-approximation des arcs *points-to*, due à la fusion des emplacements mémoires, puisque `q` ne pointe pas vers `i`. Cette fusion, par rapport à l'algorithme d'Andersen, permet un gain en termes de temps de calcul d'où son efficacité.

3.1.2.1 Algorithme de Steensgaard : prise en compte des structures de données et des appels de fonction

L'algorithme de Steensgaard [Ste96] ne permet pas une prise en compte précise des structures de données ; les différents champs de structures sont représentés par un seul type, pareillement pour les éléments de tableaux. Les appels de fonction sont traités de la même manière qu'avec l'algorithme d'Andersen [And94], en appliquant les règles d'inférence aux affectations des paramètres effectifs aux paramètres formels.

3.1.3 Conclusion

Les deux analyses Andersen et Steensgaard sont efficaces en termes de temps d'exécution et d'espace mémoire. Bien qu'elles soient insensibles au flot de contrôle, elles ont été souvent reprises dans les compilateurs, mais en augmentant une des autres dimensions des analyses de pointeurs, par exemple la dimension représentation des champs de structures de données récursives ou le calcul interprocédural des arcs *points-to*.

3.2 Les analyses sensibles au contexte et au flot de contrôle

La sensibilité au contexte rend les analyses interprocédurales difficiles et complexes à effectuer, car le comportement de chaque procédure dépend du contexte d'appel. Une des approches précises visant à récupérer les relations générées par le site d'appel est l'analyse sensible au contexte par résumés. Cette dernière consiste à créer une description concise du comportement de la procédure sous la forme d'un résumé qui sera utilisé lors de tous les appels à la procédure sans avoir à réanalyser son corps [ASU86].

Nous présentons deux analyses comme exemples d'analyses sensibles au contexte et/ou au flot de contrôle.

La première analyse, qui est celle de Wilson [WHI95], a été développée dans le contexte de la parallélisation automatique, comme composant du projet SUIF [SUI]. La deuxième analyse est celle d'Emami [Ema93] développée dans le cadre du compilateur McCAt [HDE+92].

3.2.1 L'analyse de Wilson

L'analyse de Wilson [WHI95] est fondée sur l'interprétation abstraite [Cou00], qui 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 à déterminer soient décidables [Zog]. 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 contrôle) sans avoir à en faire le traitement complet.

La précision des résultats de l'analyse est assurée par son aspect interprocédural et la sensibilité au contexte des appels des fonctions. En effet, on résume les effets des procédures en utilisant les fonctions partielles de transfert *partial transfert function* (PTF), qui décrivent le comportement d'une procédure en faisant l'hypothèse que les mêmes relations d'aliasing entre les paramètres sont présentes lors des différents appels à cette dernière.

Le but de cette analyse, implémentée au sein du compilateur SUIF [SUI], 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'aliasing d'entrée qui existent dans le programme.

3.2.1.1 Analyse intraprocédurale de Wilson

L'algorithme de Wilson utilise les fonctions *points-to* pour déterminer les alias intraprocéduralement, en appliquant les règles [Rug05a] illustrées dans le tableau 3.1.

Assignation	Règle *points-to*	Graphe *points-to*
$x = \&y$	$pts_to(x) = \{y\}$	x → y
$x = y$	$pts_to(x) = pts_to(y)$	x → z, y → z
$x = *y$	$pts_to(x) = \cup pts_to(z)$ $pour\ tout\ z \in pts_to(y)$	y → z, z → z1, x → z1
$*x = y$	$pts_to(z) = \cup pts_to(y)$ $pour\ tout\ z \in pts_to(x)$	x → z, y → z1, z → z1

Tableau 3.1 – Les règles *points-to* de Wilson

Modélisation de la mémoire dans l'analyse de Wilson L'analyse des pointeurs fournit un ensemble d'emplacements vers lesquels un pointeur pourrait pointer, ce qui introduit des ambiguïtés surtout quand on s'intéresse aux emplacements alloués dynamiquement ou contenant

Expression	Ensemble d'emplacements
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)

Tableau 3.2 – Exemples d'ensembles d'emplacements

des structures agrégées ou des tableaux. L'idée est alors d'abstraire la mémoire et de regrouper plusieurs emplacements concrets en un ou plusieurs emplacements abstraits.

Le but de la modélisation proposée par Wilson [WHI95] est de distinguer entre différents champs d'une structure mais pas entre les différents éléments d'un tableau. Wilson [WHI95] introduit 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, une fois la mémoire divisée en blocs, chaque bloc représente un ensemble d'adresses contiguës dont les positions relatives sont indéfinies.

Un bloc mémoire peut être soit une variable locale, soit un bloc du tas alloué localement, soit un paramètre étendu [2]. Les blocs de tas sont 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 se distinguent par leur contexte d'allocation, en groupant ensemble tous les blocs alloués dans chaque contexte, l'information minimale sur le contexte étant l'instruction qui a créé le bloc. Les positions sont représentées au sein d'un bloc par des ensembles d'emplacements où un ensemble d'emplacements est un triplet (b, f, s), où b représente la base, f l'offset et s la taille de l'élément du tableau. Dans le tableau 3.2, on trouve les triplets associés à différentes expressions.

Ainsi on remarque qu'un champ dans une structure est identifié par son offset à partir du début de cette dernière. L'offset de la référence à un tableau par contre est mis à zéro. Quand la position d'un emplacement au sein d'un bloc est inconnue, on met le pas s à 1, pour 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 1. Et parce que la position d'un bloc par rapport à un autre est indéfinie, nous n'avons pas à nous soucier de l'arithmétique de pointeurs qui déplacerait un pointeur vers un différent bloc.

Cette représentation de la mémoire a pour buts :

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

Cette représentation sous forme d'ensemble d'emplacements ne tient pas compte des informations liées aux types. Ceci peut, par rapport à d'autres modélisations, induire des résultats moins précis, mais permet par contre une analyse sûre des programmes où il y a des violations des types déclarés *via* des *casts*.

2. Ensemble des valeurs abstraites vers lesquelles un paramètre peut pointer.

3.2.1.2 L'analyse interprocédurale de Wilson

Pour aboutir à des résultats précis, le programme entier, y compris les bibliothèques incluses, doit être fourni 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 permettent une approche «top-down» qui combine l'analyse intraprocédurale et interprocédurale en ne réanalysant une fonction que si l'aliasing entre paramètres et variables globales trouvé au niveau d'un site d'appel n'a jamais été rencontré auparavant.

Pour son analyse interprocédurale, Wilson [WHI95] construit le graphe des appels pour identifier les relations d'appel entre les fonctions. Pour prendre en compte le contexte des fonctions tout en profitant des analyses déjà effectuées, sa solution se fonde sur le calcul de fonctions de transfert (PTF). Une fonction de transfert permet de représenter la relation entre l'entrée et la sortie d'un système. L'analyse de Wilson s'inspire du calcul des effets prédicats entiers et des effets mémoire, tel qu'il est implanté au niveau du compilateur PIPS [Men08], où les transformeurs de prédicats sont eux-mêmes des abstractions calculées [Iri93] [IJT91].

Ils permettent d'établir une relation entre les valeurs des variables scalaires entières à un point de contrôle et leurs valeurs à un autre point de contrôle.

Dans le cas de l'analyse de Wilson, illustrée par la figure 3.8, les fonctions de transfert résument les effets des procédures sur la relation *points-to*, sous certaines conditions d'aliasing dans le contexte d'appel.

```
void appelant()
{
int *e1, *e2, *e3 i = 1, j = 2, n = 10;

S1 : e1 = &i;

S2 : e2 = &j;

S3 : e3 = callee(e1, e2, n);
S4:
}
```

// Calcul du résumé

```
int * appele(int *f1, int *f2, int n)
{
S5 : f1 = malloc(n*sizeof(int));
S6 : ...
S7: return f1;
}
```

// Application du résumé

FIGURE 3.8 – Analyse interprocédurale de Wilson

Les fonctions de transfert (PTF) établissent une relation entre la relation *points-to* calculée *via* l'analyse intraprocédurale, juste avant le site d'appel, et la relation *points-to* finale après appel et retour de la fonction avec 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. Elle 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 l'absence de récursivité. L'analyse ne requiert que deux passes sur le programme : une passe *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 la procédure appelée peuvent être déterminés à partir de sa fonction de transfert. Et puisque les fonctions de transfert sont paramétrées, les résultats sont sensibles au contexte. Par la suite, le

coût de production d'une fonction de transfert est amorti par le nombre de ses appels ; il reste seulement le coût de l'application de la fonction de transfert multiplié par le nombre de sites d'appel. Mais si la fonction de transfert disponible à ce moment-là ne correspond pas à l'aliasing du site d'appel, la fonction appelée doit être réanalysée sous ces nouvelles conditions d'aliasing et la fonction de transfert doit être étendue avec la nouvelle condition d'aliasing et une nouvelle fonction de transfert partiel. 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. L'analyse de Wilson a seulement besoin de calculer les effets des procédures pour les alias qui se produisent effectivement, dits alias pertinents.

3.2.1.3 Modélisation du contexte d'appel : 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 sont représentés par des noms symboliques, pour toutes les valeurs passées à une procédure, soit directement *via* les paramètres formels et les variables globales, soit indirectement via les pointeurs. Chaque entrée d'une PTF *partial transfert function* a son propre contexte formel de paramètres étendus 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 et ne se référant pas à un bloc local. Ceci est important pour l'analyse qui ne veut considérer que les alias pertinents, où le pointeur est déréférencé. 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.

1. Les *points-to* 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 de ses descendants. L'algorithme dérive une association 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* initiales capturent précisément cette information.
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 paramètres étendus interviennent aussi dans ce qu'on appelle les mises à jour «fortes». En effet, quand l'analyse peut déterminer qu'une affectation a sûrement lieu et qu'elle va affecter un seul emplacement, elle 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 conservatrice que la destination conserve son ancienne valeur, cumulée avec les nouvelles valeurs potentielles de l'affectation.

Ces mises à jour affectent la terminaison de l'algorithme, d'où la nécessité d'introduire des contraintes pour s'assurer de la sa terminaison. Ces contraintes introduisent un ordre sur l'éva-

luation des nœuds du graphe de flot de contrôle ; il faut s'assurer alors qu'aucun nœud n'est évalué à moins que l'un de ses prédécesseurs soit évalué et qu'aucune affectation n'est évaluée à moins de connaître sa destination.

3.2.1.4 Algorithme de Wilson

L'algorithme suivant est l'application d'une version simplifiée de la fonction `EvalProc()` de Wilson qui permet l'analyse d'une fonction en faisant appel 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 nœuds 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()`.

```
/* la fonction recoit comme parametre une PTF, qui pour un premier
   appel peut etre initialement vide */
void EvalProc (PTF ptf){
  do {
    changed = FALSE;
    /* parcours iteratif du graphe de flot de donnees */

    foreach Node n in ptf.proc.flowGraph
      if (no predecessors of n evaluated)
        continue;
    if (n is scalar assignment)
      /* EvalScalarAssign evalue les expressions, etablit l'association entre
   les valeurs et les emplacements abstraits
   et met a jour les fonctions points-to */
      EvalScalarAssign(n, ptf);

    else if (n is aggregate assignment)
      /* EvalAggregateAssign traite les affectations aux structures agregees */
      EvalAggregateAssign(n, ptf);

    else if (n is join)
      /* EvalJoin traite les noeuds qui sont a la jointure de deux chemins
         du graphe de flot de donnees */
      EvalJoin(n, ptf);

    else if (n is call)
      /* evalCall traite les appels aux fonctions en determinant les effets
   de cet appel sur les fonctions points-to ; ceci se fait par determination
   de quelle procedure va etre appelee, determination de quelle PTF va etre
   appliquee aux appelees potentielles et finalement par l'ajout de
   l'information points-to au niveau du site d'appel */
      EvalCall(n, ptf);
  }while{!changed }
}
```

Prog 3.5 – Algorithme de Wilson [Wil97]

En appliquant ces règles de calcul des arcs *points-to* à notre exemple, le programme 3.1, les

résultats suivants sont obtenus à la fin du programme :

$$pts_to(p) = \{j\}$$
$$pts_to(q) = \{j\}$$

Les résultats sont traduits sous la forme de graphe figure 3.9.

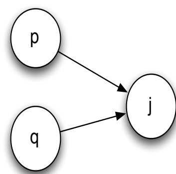


FIGURE 3.9 – Graphe résultat de l'analyse de Wilson

3.2.1.5 Conclusion

L'analyse de Wilson a été testée sur de grands programmes écrits en `C`. L'analyse permet de couvrir toutes les instructions du langage, en particulier l'arithmétique de pointeurs et l'allocation dynamique de mémoire. Cependant les publications de Wilson manquent de description détaillée des fonctions et structures de données utilisées par son algorithme.

3.2.2 L'analyse d'Emami

L'analyse de Emami [Ema93] [HDE+92] est une autre approche pour étudier les relations *points-to*. Cette approche se fonde sur une abstraction de l'espace pile et sur les relations entre les éléments de la pile. L'analyse permet de modéliser ces relations sous la forme d'un triplet `(x, y, rel)` un arc *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 l'arc : «`definitely`» quand nous sommes sûr de l'arc ou bien «`possibly`» dans le cas contraire.

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

A part l'analyse interprocédurale, l'algorithme permet aussi le support des structures de données 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, nous expliquons en détails l'analyse de Emami [Ema93].

3.2.2.1 L'analyse intraprocédurale d'Emami

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. Par exemple les multiples pointeurs sont traduits comme suit :

La représentation SIMPLE est une représentation du programme où le flot de contrôle est explicite, ce qui permet, par exemple, d'analyser une boucle en parcourant simplement ses composants : la condition et le corps de la boucle. De cette représentation découlent trois principaux

a = **b;	temp0 = *b;
	a = *temp0;

FIGURE 3.10 – Représentation SIMPLE

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.

Abstraction de l'espace pile L'analyse d'Emami consiste à abstraire les arcs entre les espaces mémoires dans la pile et à les approximer.

L'abstraction de la pile se fait par :

- l'association de noms symboliques aux variables de type pointeur ;
- l'introduction d'une propriété de précision $rel \in \{D, P\}$;
- des arcs *points-to* valués par la précision `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 3.11 :

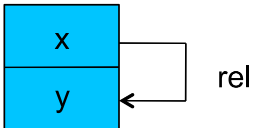


FIGURE 3.11 – L'abstraction au niveau de la pile

L'analyse possède les deux propriétés suivantes :

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

Propriété 2 : Chaque emplacement abstrait nommé de la pile, construit à partir du nom d'une variable du programme, à un point de contrôle 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 tous les arcs *points-to* en utilisant les emplacements abstraits nommés de la pile. L'analyse s'effectue au niveau de chaque point du programme où l'information *points-to* est collectée. En appliquant les règles de calcul de relations *points-to* à notre exemple 3.1 ; les résultats suivants sont obtenus :

```
(p, j, definitely)
(q, j, definitely)
```

Les résultats peuvent être traduits sous la forme de graphe (voir figure 3.12).

3.2.2.2 L'analyse interprocédurale

Le graphe d'invocation Dans ce qui suit, nous nous intéressons à une analyse interprocédurale de la pile abstraite, qui calcule les nouveaux arcs *points-to* résultat des effets des appels aux fonctions [EGH94], comme l'association entre les paramètres formels et effectifs ou bien l'effet des procédures sur les variables globales. L'analyse utilise une structure de données bien spécifique qui capture l'ordre d'activation et la structure des appels d'un programme donné. Cette

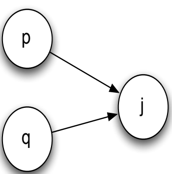


FIGURE 3.12 – Graphe résultat de l'analyse d'Emami

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

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

Prog 3.6 – Exemple C pour l'aspect interprocédural de l'analyse d'Emami

structure est le graphe d'invocation qui consiste à 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 différencie les appels d'une même procédure (appels à la fonction `g()`) et on distingue 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()`), voir figure 3.13.

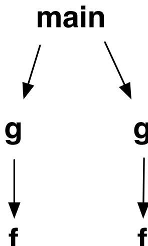


FIGURE 3.13 – Le graphe d'invocation

L'analyse utilise le graphe d'invocation pour suivre la séquence d'appels des fonctions comme le montre la figure 3.14. En commençant par la fonction `main`, chaque instruction est 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 nœud ordinaire [3], le corps de la fonction est analysé 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.

3.2.2.3 Les grandes lignes de l'algorithme

L'algorithme 3.7 est une version simplifiée de la fonction `func_pts_to()`, qui permet le calcul interprocédural des informations *points-to* en absence de récursivité. L'algorithme commence

3. Qui n'est pas un appel récursif.

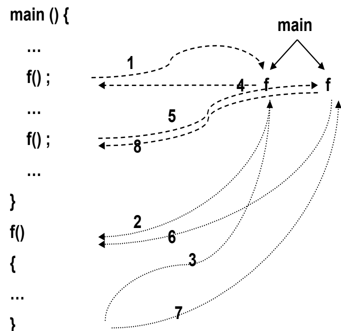


FIGURE 3.14 – Les séquences d'appels dans un graphe d'invocation

par récupérer le nœud du graphe d'invocation associé à l'appelé à partir du site d'appel de l'appelant. Ensuite la liste des arguments de la fonction est obtenue et une association entre eux et les informations *points-to* en entrée (in_data) est établie et c'est l'information *points-to* (mappee) qui est retournée. Puis c'est la liste des paramètres de la fonction ainsi que son corps qui sont récupérés. Le corps de la fonction est analysé et l'information *points-to* est retournée *via* la relation points-to finale. Cette information est sauvegardée au niveau du nœud du graphe d'invocation. Finalement le processus inverse de l'association est déclenché. Il créé, à partir des informations *points-to* de l'appelé et de l'appelant, la nouvelle relation *points-to*.

```
func_pts_to(call_expr_node, in_data)
{
  ig_node = get_related_ig_node(in_data_cur_ig_node,call_expr_num);
  arg_lst = get_arg_lst(call_expr_node);
  [map_info, func_in_data.in] = map_process(func_node, arg_lst, in_data.in);
  arg_lst = get_arg_lst(call_expr_node);
  func_body = get_func_body(func_node);
  func_out_data = points-to(func_body, func_in_data);
  save_out_info(ig_node, func_out_data.in);
  out_data.in = unmap_process(in_data, func_out_data.in, map_info);
  return(out_data);
}
```

Prog 3.7 – L'algorithme interprocédural d'Emami

Quant au processus d'association entre les paramètres formels et effectifs, il est illustré par la figure 3.15. Le processus d'association établit le lien entre les paramètres actuels et formels 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. L'analyse intraprocédurale analyse le corps de la fonction et le processus d'association 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 d'association, prenons comme exemple le programme 3.8. Au niveau de la pile les relations de la figure 3.16 sont obtenues. 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 d'association sur les paramètres.

Pour cela, on ajoute 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 d'«association inverse», prenons l'exemple suivant, le programme 3.9, où il n'y a pas de relations «points-to»

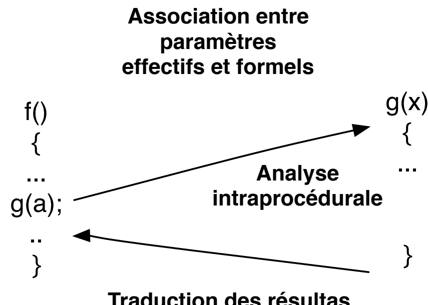
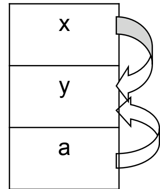


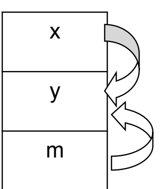
FIGURE 3.15 – Le processus d'association

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

Prog 3.8 – Exemple pour le processus d'association



La pile abstraite de `main` au niveau de S1



La pile abstraite de `f()` au niveau de `S2`

FIGURE 3.16 – État de la pile

avant l'appel à la fonction `f()` au niveau de l'instruction `S1` ; c'est pour cette raison que le processus d'association n'ajoute aucune information à la fonction `f()`.

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

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

Prog 3.9 – Exemple C pour l'association inverse

Au niveau de la pile, on aura les relations de la figure 3.17.

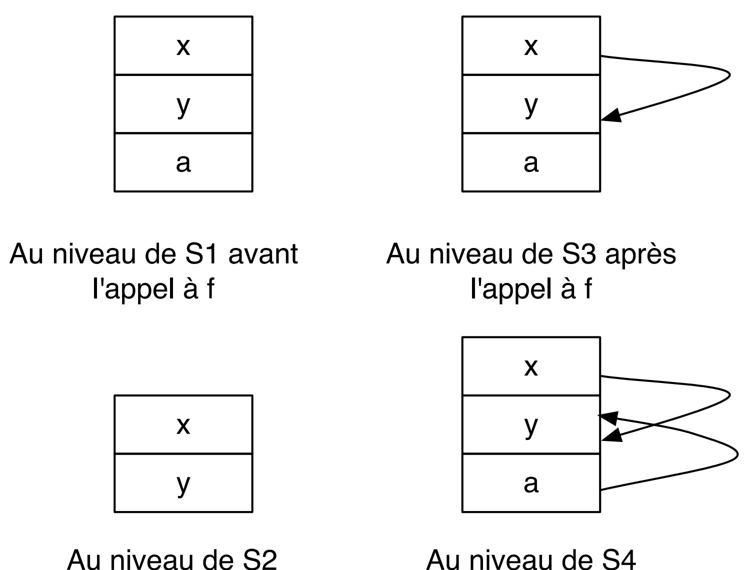


FIGURE 3.17 – Relations *points-to* au niveau de la pile

Au niveau de `S2`, on obtient la relation `(x,y,D)`. Puis au niveau de `S3`, on applique le processus d' association inverse pour obtenir les relations au niveau de la pile. Après `S3` la relation `(a,y,D)` est obtenue puisque `x` pointe vers `y` ; la variable `a` va aussi pointer vers `y`.

3.2.2.4 Gestion des données stockées dans le tas

Concernant le traitement des données qui sont allouées dans 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», il ne peut y avoir que des relations «possibly». Ainsi pour le programme 3.10, la pile de la figure 3.18 est obtenue.

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

Prog 3.10 – Exemple C pour le traitement du tas

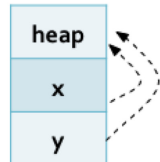


FIGURE 3.18 – Les relations au niveau du tas

Cette partie de l'analyse d'Emami a été reprise par la suite par Ghiya [GH96], qui a apporté une modélisation plus précise des données allouées au niveau du tas en nommant chaque site d'allocation différemment.

3.2.3 Comparaison des analyses insensibles au flot et au contexte

L'analyse de pointeurs dans ce contexte a été le sujet de plusieurs articles [HP00] [Ray05] [Qia] [Rug05b]. Nous avons décidé de reprendre, brièvement, les résultats d'une étude expérimentale effectuée par Hind [HP00] qui s'intéresse à cinq analyses de pointeurs insensibles au flot de contrôle contre une seule sensible au flot. Les algorithmes cités sont les suivants :

1. Andersen [And94] : génération d'un seul ensemble de *points-to* pour le programme, construit en résolvant des contraintes sur les relations entre variables pointeurs ;
2. Burke et al. [HBCC99] : itération sur tout le programme, comme Andersen, mais qui diffère de ce dernier par le calcul des alias pour chaque procédure en plus du calcul des alias pour tout le programme. Cet algorithme est moins efficace que celui d'Andersen à cause de l'itération en plus sur les fonctions ;
3. Steensgaard [Ste96] [HP00] : calcul d'un seul ensemble de solutions pour tout le programme, qui se distinguent d'Andersen par la fusion des structures de données en un seul objet ;
4. «Address-taken » [HP00] : génération d'un seul ensemble regroupant toutes les variables dont l'adresse a été assignée à d'autres variables du programme, incluant les objets alloués dynamiquement ainsi que les paramètres formels. Cette analyse est linéaire en fonction de la taille du programme, d'où son efficacité en terme de temps d'exécution ;
5. Choi et al. [BCCH95] : génération d'un ensemble de solutions pour chaque point du programme en associant un ensemble d'alias à chaque nœud du graphe de flot de contrôle.

Hind [HP00] a comparé les analyses suivant plusieurs critères, parmi lesquels figure le nombre moyen des objets qui sont en alias avec un pointeur. Cependant ce critère peut varier pour les raisons suivantes :

- la modélisation des emplacements mémoire peut augmenter ou diminuer significativement cette moyenne ; une structure de données récursives modélisée comme un seul objet diminue le nombre d'objets aliasés avec le pointeur ;
- la précision de l'analyse devrait se mesurer en fonction de son impact sur les analyses clientes ; si ces dernières ne montrent aucune amélioration, il est difficile de se prononcer sur la précision de l'analyse de pointeurs ;
- les variables globales peuvent induire un grand nombre d'alias, qui n'existent pas forcément au moment de l'exécution.

Pour ces différentes raisons, la comparaison s'est faite en gardant les mêmes caractéristiques pour toutes les analyses :

- l'analyse est insensible au contexte ;
- les objets alloués au niveau du tas sont identifiés par leur site d'allocation ;
- la représentation des structures de données récursives est *field-insensitive* ; la représentation est agrégée à la structure qui apparaît comme un seul objet (voir la section 3.1.1.2) ;
- les arcs *points-to* ont la même représentation compacte.

Les métriques utilisées pour comparer les analyses des pointeurs sont :

1. la précision en termes de nombre d'alias découverts ;
2. la précision en termes d'amélioration de quatre analyses clientes : l'analyse Mod/Ref(détermine pour chaque nœud du graphe de flot de contrôle les objets qui ont été modifiés ou référencés), l'analyse des variables vivantes, l'analyse *reaching definition* et l'analyse interprocédurale de propagation de constantes ;
3. efficacité en terme de temps d'exécution/espace mémoire.

Les résultats obtenus sont mitigés. En terme de précision les analyses insensibles au flot de contrôle se valent presque toutes. L'analyse *Adress-taken* apparaît en dernier lieu, après celle de Steensgaard et celle d'Andersen ; celle Burke et al. est la plus précise.

L'analyse de Choi et al. offre les résultats les plus précis et améliorent nettement les résultats des analyses clientes. Cependant elle utilise six fois plus d'espace mémoire que toutes les autres analyses.

En terme d'efficacité et dans un ordre croissant, on trouve :

1. l'analyse de Choi et al. [BCCH95], la seule à être sensible au flot de contrôle et qui produit deux ensembles par nœud du graphe de flot de contrôle ;
2. l'analyse Burke et al. qui produit un ensemble par fonction ;
3. l'analyse d'Andersen qui produit un seul ensemble par programme ;
4. l'analyse de Steensgaard, presque linéaire et qui produit un seul ensemble par programme ;
5. l'analyse «Adress-taken» qui est linéaire et qui produit un seul ensemble par programme.

3.2.4 Comparaison des analyses sensibles au flot et au contexte

Dans cette sous-section nous nous intéressons aux analyses sensibles au flot et/ou au contexte. Nous faisons une critique de l'analyse de Wilson et celle d'Emami en présentant les points forts et faibles de chacune d'entre elles.

3.2.4.1 Critique de l'analyse de Wilson

La plus part des analyses sensibles au contexte ont comme défaut 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 ayant un état initial arbitraire ;

- chaque procédure doit être analysée à nouveau pour chaque contexte d'appel (sauf si un mécanisme de résultats résumés est mis en place [IJT91]).

L'analyse de Wilson [Wil97] permet de remédier à ce dernier défaut en calculant la fonction partielle de transfert, même si le coût de propagation des informations à travers le graphe de flot de contrôle reste élevé. En résumé l'analyse de Wilson [Wil97] a l'avantage de prendre en compte le contexte d'appels des fonctions, une modélisation précise du tas et une certaine efficacité en termes de temps d'exécution compte tenu de l'aspect insensible au flot de l'analyse. Ce gain au niveau de l'efficacité vient au prix de la précision des résultats. En effet, les relations *points-to* ne sont pas mises à jour au fur et à mesure du parcours du graphe de flot de contrôle.

3.2.4.2 Critique de l'analyse d'Emami

L'analyse d'Emami [EGH94] est précise par ce qu'elle prend en compte l'ordre d'exécution des instructions, sensibilité au flot de contrôle, et par son aspect contextuel, prise en compte de tous les contextes d'appels. En effet l'analyse propage les relations d'une procédure à une autre par l'introduction du concept de variables invisibles (le même concept que les paramètres étendus de Wilson) qui sont des variables temporaires introduites pour représenter les paramètres formels et les variables globales de type pointeur, et par le graphe d'invocation. Mais cette précision est très coûteuse en terme de mémoire et de temps d'exécution du fait de la construction du graphe d'invocation et du parcours du graphe de flot de contrôle. Par contre l'analyse reste restreinte en ce qui concerne le traitement des structures des données, ainsi que le traitement des tableaux, alors que ce sont deux éléments clés du langage `C`, surtout quand on vise la parallélisation automatique des programmes.

3.3 Autres travaux

Nous présentons dans cette section d'autres travaux auxquels nous nous sommes intéressés et qui traitent essentiellement des relations d'alias, de l'allocation dynamique ou encore de l'arithmétique sur pointeurs.

3.3.1 L'analyse « pointer values »

Cette analyse a été développée par Béatrice Creusillet [Cre96] dans le cadre du compilateur PIPS [AAC+]. Son but principal est de permettre la parallélisation à gros grain en effectuant une étape en moins que l'analyse de pointeurs. En effet, l'analyse de pointeurs remplace les déréférencements de pointeurs par leur cible et, lors des affectations de pointeurs de type `q = p[i]`, où `p` et `q` sont des doubles pointeurs, l'arc *points-to* produit est `q -> cible_p`. L'analyse « pointer values » elle fournira plutôt le résultat `q == p[i]` pour garder trace de la relation d'alias entre les pointeurs du programme. Cette analyse est encore en phase de développement et tests.

3.3.2 L'allocation dynamique

Une autre dimension très importante de l'analyse de pointeurs, et qui apparaît fréquemment dans les applications scientifiques, est l'allocation dynamique au niveau des boucles. Dans la littérature, un travail en particulier [WFPS02] se distingue des autres analyses qui modélisent selon le site d'appel. Pour l'exemple `java` suivant tiré de [WFPS02] :

La plupart des analyses vont conclure que `p` est synonyme[4] d'un seul objet `q`, alors qu'il est synonyme d'un nouvel objet à chaque itération de la boucle. Comme le travail de [WFPS02] traite

4. en alias

```
for(i = 0; i < m; i++) {
 p = q;
 p.x = ...;
 q = new Object();
}
```

Prog 3.11 – Exemple `java` de l'allocation dynamique au niveau des boucles

en particulier le langage `java` et propose aussi une solution pour prendre en compte les tableaux multidimensionnels qui sont codés sous la forme de tableaux de pointeurs, il devient difficile de déterminer les relations de synonymes créées lors de l'allocation des éléments de tableau au niveau d'une boucle. La solution présentée par [WFPS02] est fondée sur l'abstraction `ewpt` (*element-wise points-to*) qui associe à chaque pointeur les objets du tas vers lesquels il pointe et cela pour chaque instance du pointeur dans la boucle et pour chaque élément de tableau référencé par le pointeur.

Fournir ces relations entre les références est nécessaire pour l'analyse des dépendances des boucles qui a besoin de retenir l'information sur les pointeurs des différents points programme ainsi que les pointeurs des différentes itérations de boucles. L'algorithme fournit ces informations sous la forme de résumés précis pour toutes les instances d'un pointeur dans une boucle. Dans ces résumés les objets alloués sont représentés par des régions de tableaux sous une forme telle que $< N_t[x], x \leqslant i >$ où $N_t[x]$ représente l'objet alloué par le « statement » t au niveau de l'itération x et i représente l'index de la boucle. L'algorithme itère sur le programme en considérant chaque assignation de référence comme une fonction de transfert des régions de tableaux qui représentent les relations entre pointeurs alloués dynamiquement. Les résultats de ce travail pourront être intégrés ultérieurement à notre travail et font partie de nos perspectives.

3.3.3 L'union, l'arithmétique sur pointeurs et le cast

Un des axes de recherches liés aux analyses de pointeurs est le traitement des unions, des « casts » ainsi que l'arithmétique sur pointeurs. Comme le langage `C` est faiblement typé, permettant le chevauchement des variables en mémoire ainsi que le changement de type en cours de programme via le cast, Miné [Min07] propose de ne pas se contenter de l'information sur le type obtenu lors de la déclaration. Il définit une analyse de valeur [5] qui comporte trois étapes :

1. prendre en entrée un programme écrit dans le langage `C` (sans récursion ni allocation dynamique) autorisant l'aliasing ;
2. traduire ce programme en instructions sans aliasing où les valeurs de variables ont été transformées en ensembles de cellules mémoire définies par (nom, offset) ;
3. effectuer une interprétation abstraite où ces instructions sont évaluées dans un domaine numérique abstrait quelconque pour aboutir à une description de bas niveau de la mémoire en un point du programme.

Cette abstraction permet de traiter les alias à bas niveau, ainsi que les chevauchements des variables rendus possibles par le langage `C`. Ce travail a été développé dans le cadre de l'analyseur ASTRÉE ; et grâce à cette représentation bas niveau de la mémoire où la valeur de chaque variable est vue comme une séquence d'octets contigus, le traitement des programmes écrits en langage `C` contenant des structures de données récursives `struct`, des unions, l'arithmétique sur pointeurs ainsi que des casts devient possible.

5. Dite « value analysis »

3.4 Conclusion

La littérature classifie les analyses de pointeurs essentiellement selon deux critères :

1. La sensibilité au flot de contrôle ;
2. La sensibilité au contexte d'appel.

Le premier critère indique la mise à jour des arcs *points-to* au fur et à mesure du parcours du graphe de flot de contrôle, une précision coûteuse en terme de temps d'exécution. L'analyse qui répond le mieux à ce critère est l'analyse d'Emami intégrée dans le compilateur McCat [HDE+92]. Le deuxième critère impose d'identifier chaque contexte d'appel et de modéliser plus précisément le tas parce qu'on a plus d'informations sur les routines d'allocation dynamique (exemple `malloc`). L'analyse de Wilson, intégrée dans le compilateur SUIF [SUI], est sensible au contexte tout en étant rapide et non coûteuse. D'autres critères de classification comme le traitement du tas, des structures de données ou la précision apportée aux analyses clientes nous ont permis de classifier des analyses insensibles au flot de contrôle et au contexte, mais qui restent essentielles à notre étude bibliographique.

Parmi ces analyses nous citons Andersen, Steensgaard et Choi et al. L'étude de ces analyses nous a permis de concevoir une analyse qui répond au mieux à nos besoins, notre but final étant de permettre la parallélisation. Ainsi nous avons pu développer une analyse plus précise qu'Emami avec une meilleure modélisation du tas et des structures de données agrégées. Notre analyse interprocédurale reprend l'idée des résumés de Wilson mais permet une vérification plus rigoureuse sur la condition d'aliasing entre les paramètres . Nous ne pouvons pas garantir la même efficacité que celle des analyses insensibles mais nous pouvons garantir des résultats plus précis qui permettent l'amélioration de plusieurs analyses clients. Pour résumer voici un tableau récapitulatif des principales analyses en fonction des dimensions cités ci-dessus :

Analyse/Dimension	sensible au flot de contrôle	sensible au contexte	field-sensitive	modélisation du tas
Andersen	-	-	+	-
Steensgaard	-	-	-	-
Wilson	-	-	+	+
Emami	+	+	+	+

Tableau 3.3 – Tableau comparatif des analyses de pointeurs

CHAPITRE 4

L'analyse intraprocédurale simplifiée

Dans ce chapitre nous introduisons les grandes lignes de notre analyse de pointeurs, implémentée dans PIPS [pip]. L'analyse intraprocédurale que nous y présentons est une analyse descendante qui parcourt l'arbre syntaxique en avant, en utilisant un contexte formel implicite, construit sur l'hypothèse qu'il n'existe pas d'aliasing entre les paramètres formels.

Cette analyse, dont le résultat est une fonction des *statements* vers des ensembles d'arcs *points to* qui sont en préconditions de ce *statement*[1], est sensible au flot de contrôle, c'est-à-dire que les arcs sont mis à jour d'un point de contrôle du programme à un autre. Dans le cadre d'un outil d'optimisation, d'analyse et de parallélisation automatique dans lequel se situe la thèse, une approche insensible au contexte n'aurait pas su répondre aux besoins des analyses clientes présentées dans le chapitre 2.

Dans cette thèse, nous présentons l'algorithme général, les équations de flot de données ainsi que les équations sémantiques permettant d'analyser les programmes écrits en langage C. Vu la complexité de C et de sa syntaxe abstraite (section 4.1), nous commençons par définir, dans la première partie de ce chapitre (section 4.2), un premier sous-ensemble de C, $\mathcal{L}_0$, qui ne comporte que trois types d'instructions :

1. la séquence ;
2. le test ;
3. l'affectation.

ainsi qu'un sous-ensemble des expressions typées du langage C comme l'accès à un élément de tableau ou à un champ de structure.

La section 4.3 décrit le schéma général de l'analyse de pointeurs avec les différentes étapes requises. Et la section 4.4 détaille la fonction transformeur, qui convertit la précondition *points-to* en postcondition pour les instructions du langage $\mathcal{L}_0$.

Dans la deuxième partie du chapitre (section 4.5), nous enrichissons le premier langage analysé, $\mathcal{L}_0$, pour arriver à un second langage, $\mathcal{L}_1$, qui nous rapproche du langage C. Nous y insistons sur la notion d'expressions et sur leur impact dans l'analyse des pointeurs, et nous présentons les équations qui permettent de les traiter (section 4.6).

Le chapitre 5 traite plus en détail les abstractions des emplacements mémoire nécessaires aux traitements des instructions plus complexes comme les boucles et les graphes de flot contrôle, et le chapitre 6 décrit les algorithmes proposés pour traiter les appels de fonctions définies par le programmeur, ce qui conclut notre analyse du langage C.

4.1 Définition de la syntaxe abstraite utilisée pour C

La syntaxe abstraite utilisée dans PIPS est donnée figure 4.1. Le nombre de constructions montre qu'il n'est pas possible de les traiter toutes en un seul chapitre. C'est pourquoi nous avons décidé de les traiter en quatre étapes décrites ci-dessus.

1. Les informations *points-to* sont imprimées dans les listings avant les *statements*.

< *statement* > $\mathcal{S}$ ::= < *expression* > = < *expression* >
| `if` < *expression* > < $s1$ > `else` < $s2$ >
| `do while` (< *expression* >) < s >
| `while` (< *expression* >) < s >
| `for` (< *s1,expression,s2* >) < s >
| `call` < *name* > (< *expression* >*)
| `return` (< *expression* >)
| `exit`
| `sequence` < s >*

< *expression* > $\mathcal{E}$ ::= < *constant* >
| < *reference* >
| (< *expression* >)
| < u_op > < *expression* >
| < *expression* > < b_op > < *expression* >
| < *sizeofexpression* >
| < *cast* >
| < *call* >
| < *application* >
| < *va_args* >

< *unary_op* > u_op ::= `-` | `!` | `&` | `*`

< *binary_op* > b_op ::= `=` | `+` | `-` | `*` | `/` | `<` | `>` | `<=` | `>=` | `==` | `!=`

< *reference* > $\mathcal{R}$ ::= < *name* > | < *name* >[< *expression* >*]

< *sizeofexpression* > Soe ::= < *type* >
| < *expression* >

< *cast* > Ct ::= < *expression* > < *type* >

< *call* > C ::= < *entity* > (< *expression* >*)

< *application* > App ::= < *expression* > (< *expression* >*)

< *v_arg* > $Varg$ ::= (< *sizeofexpression* >*)

< *type* > $\mathcal{T}$ ::= `int` | `float` | `void` | `pointer`(t) | `struct` | `functional`

FIGURE 4.1 – Syntaxe abstraite de l'ensemble du langage C

Tous les sous-langages, $\mathcal{L}_0$, $\mathcal{L}_1$, $\mathcal{L}_2$, sont définis par des sous-ensembles de ces règles. Ce n'est pas qu'au chapitre 6 que le langage C est analysé dans sa totalité.

4.2 Définition du langage $\mathcal{L}_0$

Compte tenu de la richesse du langage C et de la complexité de l'analyse des pointeurs, nous avons choisi de commencer par définir l'analyse pour un sous-ensemble du langage C, appelé $\mathcal{L}_0$. Le sous-langage contient les principaux opérateurs impliquant des pointeurs comme la prise d'adresse « & » et le déréférencement « * ». L'opérateur `p->x` peut être réécrit sous la forme `(*p).x`.

4.2.1 Définition des domaines

Nous commençons par définir les différents domaines des éléments qui interviennent dans notre sous-ensemble du langage $\mathcal{L}_l0$. Les domaines dont nous avons besoin sont essentiellement le domaine des expressions $\mathcal{E}$, le domaine des chemins d'accès mémoire constants $\mathcal{A}$ ainsi que le domaine des *statements* $\mathcal{S}$. Nous définissons tout d'abord à la table 4.1 les domaines simples nécessaires à la construction des domaines plus complexes. La constante « $*$ », à ne pas confondre avec le déréferencement de pointeur, désigne n'importe quelle valeur entière. Quand elle est utilisée comme indice, l'ensemble des valeurs correspondantes est restreint à l'intervalle $[0, N[$ où N est défini par la déclaration du tableau utilisé.

Domaine	**Définition**
Booléen	$\mathcal{B} = \{\texttt{true}, \texttt{false}\}$
Constante	$\mathcal{C} = \mathbb{N} \cup \{\texttt{NULL}, \text{undefined}, *\}$
Identificateur	$\mathcal{I}$

Tableau 4.1 – Domaines simples

4.2.1.1 Définition du domaine $\mathcal{T}$ des types

Les variables et les expressions des programmes que nous traitons sont typées. Comme nous traitons un sous-ensemble du langage C nous avons inclus un sous-ensemble des types en ne gardant parmi les types basiques que le type `int`. Quant aux types agrégés, les constructeurs de types tableaux et structures ont été inclus. Bien sûr le domaine des types comporte aussi le type pointeur.

$$t \in \mathcal{T} ::= \texttt{int}|\text{struct}|\text{pointer}(t')|\text{array}(t', \text{cte})|overloaded \tag{4.1}$$

avec cte $\in \mathcal{C}$, $t' \in \mathcal{T}$ et struct une fonction des identificateurs (un nom de champ) vers les types (le type du champ) struct : $\mathcal{I} \rightarrow \mathcal{T}$. Le type $overloaded$ représente le sommet du treillis type introduit à la sous-section 5.3.4.

4.2.1.2 Définition des éléments de l'ensemble $\mathcal{E}$ des chemins d'accès non constants

Nous commençons par définir le domaine $\mathcal{E}$ des chemins d'accès non constants. Un chemin d'accès est construit à partir d'une expression qui est l'élément le plus fondamental du langage C. Il est dit non constant dès qu'il implique un déréférencement mémoire, même si la case mémoire utilisée pour le déréférencement est constante sur la portée considérée. C'est une définition syntaxique. Le langage comporte les accès aux champs des structures, aux éléments des tableaux

ou encore l'accès à la case mémoire pointée par un pointeur. Mathématiquement, il n'est pas possible de distinguer entre un élément de $\mathcal{E}$ et un élément de $\mathcal{A}$ (langage défini dans la section suivante 4.2.1.3). C'est l'information de type associée implicitement à chaque chemin ou sous-chemin qui permet de distinguer entre déréférencements d'une part, et indexations ou accès à un champ d'autre part.

Au niveau de la section 4.6, nous verrons qu'un *lhs*[2] peut être une expression complexe avec de possibles effets de bords. Les expressions complexes et développées sont définies au niveau de la sous-section 4.6. Dans cette section, nous nous contentons d'expressions simples, sans effets de bords.

$$e \in \mathcal{E} \quad ::= \quad \mathrm{Id}_t | e_1.f_t | *e_t | \&e_1 | e_1[e_2]_t | \mathsf{cte}_t \tag{4.2}$$

avec $t \in \mathcal{T}$, cte $\in \mathcal{C}$, $\mathrm{Id} \in \mathcal{I}$, $f \in \mathcal{I}$, $e \in \mathcal{E}$.

4.2.1.3 Définition des éléments de l'ensemble des chemins constants $\mathcal{A}$

Nous commençons par définir l'ensemble des chemins constants $\mathcal{A}$ qui va permettre de traduire les déréférencements de pointeurs sans passer par un code à 3-adresses. En effet, les expressions gauches, les *lhs*, sont traduites en chemins d'accès dans $\mathcal{E}$ en généralisant la notion d'indexation aux champs de structures et aux déréférencement (voir le programme 4.1).

```
s1          -> s1
s1.val      -> s1[val]
a[1]        -> a[1]
*p          -> p[0]
(*q).next   -> q[0][next]
```

Prog 4.1 – Traduction des accès mémoire en chemins constants

C'est la base du treillis, aussi appelé domaine abstrait, que nous allons utiliser pour abstraire les relations *points-to*. L'ensemble $\mathcal{A}$ est construit à partir de l'ensemble $\mathcal{I}$ des variables d'une fonction et de l'opérateur d'indexation qui sert à représenter les accès aux champs des structures tout comme les accès aux éléments de tableaux. Par exemple, si `a` est déclarée `int a[10]`, l'expression `a[2]` est un élément de $\mathcal{A}$. Dans $\mathcal{A}$, les arguments de l'opérateur d'indexation sont toujours des constantes. Une adresse au sens de $\mathcal{A}$ est donc un *lhs* constant, ou, dans la terminologie interne, un chemin d'accès constant. Le chemin constant correspond à une adresse mémoire unique dans le cas d'un scalaire, ou à un ensemble d'adresses mémoire contigües dans le cas d'une structure ou d'un tableau. Le chemin est valable dans toute la portée (*scope*) du programme où sa variable initiale est définie.

Par définition, les chemins constants sont inclus dans les expressions, $\mathcal{A} \subset \mathcal{E}$. Ils représentent des expressions qui ont été évaluées et où chaque déréférencement de pointeur a été remplacé par l'emplacement mémoire pointé. Avec l'introduction de l'information sur les types, les éléments de $\mathcal{A}$ correspondent à des chemins constants et typés. Le domaine $\mathcal{A}$ est défini récursivement comme suit :

$$a \in \mathcal{A} \quad ::= \quad \mathrm{Id}_t | \mathsf{cte}_t | (a_t.c)_{t'} | a_t[\mathsf{cte}_i] \tag{4.3}$$

avec $t \in \mathcal{T}$, $c \in \mathcal{I}$, cte $\in \mathcal{C}$. Donc, un chemin constant est soit un identificateur, soit une constante qui peut être le pointeur nul, `NULL`, un indice quelconque, $*$, ou encore un pointeur indéfini, undefined, soit encore un chemin postfixé par un champ ou par un indice constant de type entier.

2. *lhs* = *left hand side*, partie gauche d'une affectation.

4.2.1.4 Définition du domaine $\mathcal{P}$ des programmes

Le domaine des programmes correspond à la notion de fonction en C, avec des paramètres formels en entrée ainsi qu'une liste de variables locales ; la valeur de retour est ignoré pour le moment. En plus des déclarations, la fonction comporte un ensemble d'instructions regroupées en un *statement*[3]. Nous commençons par définir les sous-domaines qui constituent le domaine $\mathcal{P}$.

Définition du domaine $\mathcal{D}$ des déclarations Le domaine des déclarations est défini comme une fonction des identificateurs (les noms des variables) vers les types.

$$\mathcal{D} = \mathcal{I} \rightarrow \mathcal{T} \tag{4.4}$$

Les déclarations, notées par la suite δ, concernent les paramètres d'une fonction ou les variables locales. La fonction de déclaration doit être définie pour toutes les variables d'un programme.

Définition du domaine $\mathcal{S}$ des *statements* Le domaine des « statements » est défini comme suit :

$$s \in \mathcal{S} ::= \text{Sequence}(s_1, s_2, ...)|\text{Test}(e_{\mathcal{B}}, s_1, s_2)|\text{Assign}(e_1, e_2) \tag{4.5}$$

avec $e \in \mathcal{E}$. Un *statement* peut être une séquence de *statements* , ou bien une structure de contrôle de type test où une condition est évaluée. Si la condition $e_{\mathcal{B}}$ de type $\mathcal{B}$ est évaluée à vrai, c'est le *statement* s_1 qui est exécuté, sinon c'est le *statement* s_2. Comme c'est l'analyse des pointeurs que nous ciblons, le domaine $\mathcal{S}$ inclut aussi le *statement* d'affectation qui prend comme arguments deux expressions de types compatibles[4]. Enfin, comme expliqué dans l'introduction de ce chapitre, il n'y a pas de boucles dans le langage $\mathcal{L}_0$. Les calculs de points fixes sont traités dans le chapitre 5.

Définition du domaine $\mathcal{P}$ des programmes Après la définition des domaines $\mathcal{D}$ des déclarations et des *statements* $\mathcal{S}$, nous pouvons à présent définir le domaine des programmes $\mathcal{P}$.

$$\begin{array}{rcl} P & : & \mathcal{D} \times \mathcal{D} \times \mathcal{S} \\ p \in \mathcal{P} & ::= & \text{Prog}(\delta_p, \delta_l, s) \end{array}$$

où δ_p représente les déclarations des paramètres formels et δ_l les déclarations des variables locales au niveau du corps de la fonction. Les domaines $\mathcal{D}_{\delta_p}$ et $\mathcal{D}_{\delta_l}$ de définition des fonctions δ_p et δ_l sont nécessairement disjoints, un identificateur ne pouvant pas être en même temps un paramètre et une variable locale.

$$D_{\delta_p} \cap D_{\delta_l} = \emptyset$$

De manière plus générale, on utilisera éventuellement δ la fonction de déclaration qui est l'*union* des deux fonctions précédentes :

$$\delta(\text{Id}) = \text{si Id} \in D_{\delta_p} \text{ alors } \delta_p(\text{Id}) \text{ sinon } \delta_l(\text{Id})$$

4.2.2 Syntaxe du langage $\boldsymbol{\mathcal{L}}_0$

En conclusion nous reprenons la syntaxe du langage C, donnée précédemment par la figure 4.1, pour mettre en gras les instructions du langage $\boldsymbol{\mathcal{L}}_0$.

3. Un bloc de base d'instructions.

4. Pour alléger les notations, les informations t de type dans e_t ne sont données qu'en cas de besoin. On définit $type(e_t) = t$.

$<$ *statement* $>$	$\mathcal{S}$: :=	$<$ *expression* $>$ = $<$ *expression* $>$
		$\mid$ `if` $<$ *expression* $>$ $< s1 >$ `else` $< s2 >$
		$\mid$ `sequence` $< s >^*$
$<$ *expression* $>$	$\mathcal{E}$: :=	$<$ *constant* $>$
		$\mid$ $<$ *reference* $>$
		$\mid$ ($<$ *expression* $>$)
		$\mid$ $< u_op >$ $<$ *expression* $>$
		$\mid$ $<$ *expression* $>$ $< b_op >$ $<$ *expression* $>$
$<$ *unary_op* $>$	u_op : :=	$\mid$ `&` $\mid$ `*`
$<$ *binary_op* $>$	b_op : :=	=
$<$ *reference* $>$	$\mathcal{R}$: :=	$<$ *name* $>$ $\mid$ $<$ *name* $>$[$<$ *expression* $>^*$]
$<$ *type* $>$	$\mathcal{T}$: :=	`int` $\mid$ `float` $\mid$ $\mid$ `pointer`(t) $\mid$ `struct` $\mid$ `overloded`

FIGURE 4.2 – Syntaxe abstraite de l’ensemble du langage $\mathcal{L}_0$

4.2.3 Typage des expressions

La détermination des types des expressions est la première étape de l’analyse de pointeurs. En effet, l’analyse s’effectue principalement sur les affectations dont la partie gauche est un pointeur. Il est donc nécessaire de développer une fonction qui renvoie le type de l’expression passée en argument.

4.2.3.1 Règles du typage

Les règles strictes de typage sont détaillées par la suite, en particulier en ce qui concerne la cohérence des types entre les pointeurs et les emplacements mémoire vers lesquels ils pointent. D’autres règles sont mentionnées mais elles sont supposées être vérifiées par la phase de vérification des types du compilateur (« type-checker »), comme par exemple le fait que l’index d’un élément de tableau doit être un entier. Comme le but est d’analyser un sous-ensemble du langage C, nous commençons par introduire l’équivalence des types entre les types pointeur et pointeur sur tableau. Ceci permet de savoir si deux types sont compatibles entre eux ou non, au delà de l’égalité syntaxique :

$$\begin{aligned}\text{pointer}(t) &\sim \text{pointer}(\text{array}(t, \text{cte}))\\ \text{type_eq}(t_1, t_2) &\Leftrightarrow t_1 = t_2 \vee t_1 \sim t_2\end{aligned}$$

Cette équivalence est utilisée par la suite pour tester la cohérence des arcs *points-to*.

4.2.3.2 Détermination du type d’une expression

Une des étapes de l’analyse des pointeurs est d’identifier qu’en partie gauche d’une assignation apparaît un pointeur. Comme les expressions peuvent vite devenir complexes, il faut concevoir une fonction qui itère sur la partie gauche jusqu’à déterminer son type final. La fonction qui détermine le type est appelée type_of. Elle prend en argument une expression typée ou non. La fonction est une disjonction sur les différents cas d’expressions. Elle renvoie le type des éléments de $\mathcal{E}$. Au préalable nous avons besoin de définir la fonction γ qui renvoie le type associé à une constante ; sa signature est la suivante :

$$\begin{aligned}\gamma : \mathcal{C} &\rightarrow \mathcal{T}\\ \text{cte} &\mapsto t\end{aligned}$$

Moyennant l'ajout prématuré [5] du type abstrait `overloaded`, elle est définie par Fonction 1.

Fonction 1 : γ

```
γ(cte : C)
switch cte do
    case ℕ
        return int
    case *
        return int
    case NULL
        return overloaded
    case undefined
        return overloaded
```

La fonction type_of est ensuite définie par induction sur la structure des expressions (voir fonction 2). Bien évidemment pour e typée, on a type_of$(e_t, \delta) = t$

4.3 Le schéma général de l'analyse *points-to*

Dans le but de définir le schéma de l'analyse, nous commençons par présenter un graphe d'appels concis des principales fonctions qui interviennent dans l'analyse. Par la suite nous détaillons les étapes qui permettent le calcul des arcs *points-to* d'un programme défini dans notre premier langage $\mathcal{L}_0$.

4.3.1 Graphe des appels de l'analyseur et résultat de l'analyse

Dans les prochaines sous-sections nous présentons d'abord le graphe des appels des fonctions nécessaires au traitement du langage $\mathcal{L}_0$. Ensuite nous détaillons les étapes de notre analyse de pointeurs.

4.3.1.1 Le graphe des appels

Après la définition des domaines nécessaires à la définition de notre langage $\mathcal{L}_0$ qui est un sous-ensemble de C, nous définissons les fonctions qui manipulent leurs éléments pour calculer l'information *points-to*. La figure 4.3 montre le graphe des appels des principales fonctions qui permettent la génération des arcs *points-to* pour un programme.

Le graphe d'appels montre que l'analyse commence par l'initialisation des variables locales via la fonction local$_{\mathcal{PT}}$. Ensuite, c'est le transformeur, $T_{\mathcal{PT}}$, qui est appelé avec en argument le « statement » en cours et l'information *points-to* précédemment calculée. Il appelle à son tour les fonctions Gen et Kill qui déterminent les arcs à ajouter et à supprimer. Gen et Kill utilisent les fonctions eta (expression_to_adress) et etv (expression_to_value) pour évaluer la partie gauche et la partie droite de l'affectation ainsi que alp (abstract_location_p) qui renvoie vrai si l'emplacement est abstrait, faux sinon et la fonction sti qui transforme un chemin dépendant de l'état mémoire courant en chemin indépendant. Les détails des fonctions eval, qui permet d'évaluer une expression en renvoyant l'ensemble des cases mémoire vers lesquelles elle pointe, et stub_cp(), qui

5. Le treillis des types n'est introduit qu'au chapitre suivant où la notion de treillis est essentielle pour pouvoir traiter les boucles.

Fonction 2 : type_of

```
type_of : $\mathcal{E} \times \mathcal{D} \to \mathcal{T}$
type_of($e : \mathcal{E}, \delta : \mathcal{D}$)
switch $e$ do
    case $\mathrm{Id}$
        return $\delta(\mathrm{Id})$
    case $e_1.\mathrm{Id}$
        $t_1$ = type_of($e_1, \delta$)
        if $t_1 \notin$ struct then
            return error
        else
            return $t_1(\mathrm{Id})$
    case $*e_s$
        $t$ = type_of($e_s, \delta$)
        if $t \neq$ pointer($t'$) then
            return error
        else
            return $t'$
    case $\&e_a$
        $t$ = type_of($e_a, \delta$)
        return pointer($t$)
    case $e_1[e_2]$
        $t_1$ = type_of($e_1, \delta$)
        $t_2$ = type_of($e_2, \delta$)
        if $t_1$ = array($t$, cte) $\wedge$ $t_2$ = int then
            return $t$
        else
            return error
    case cte
        return $\gamma(e)$
```

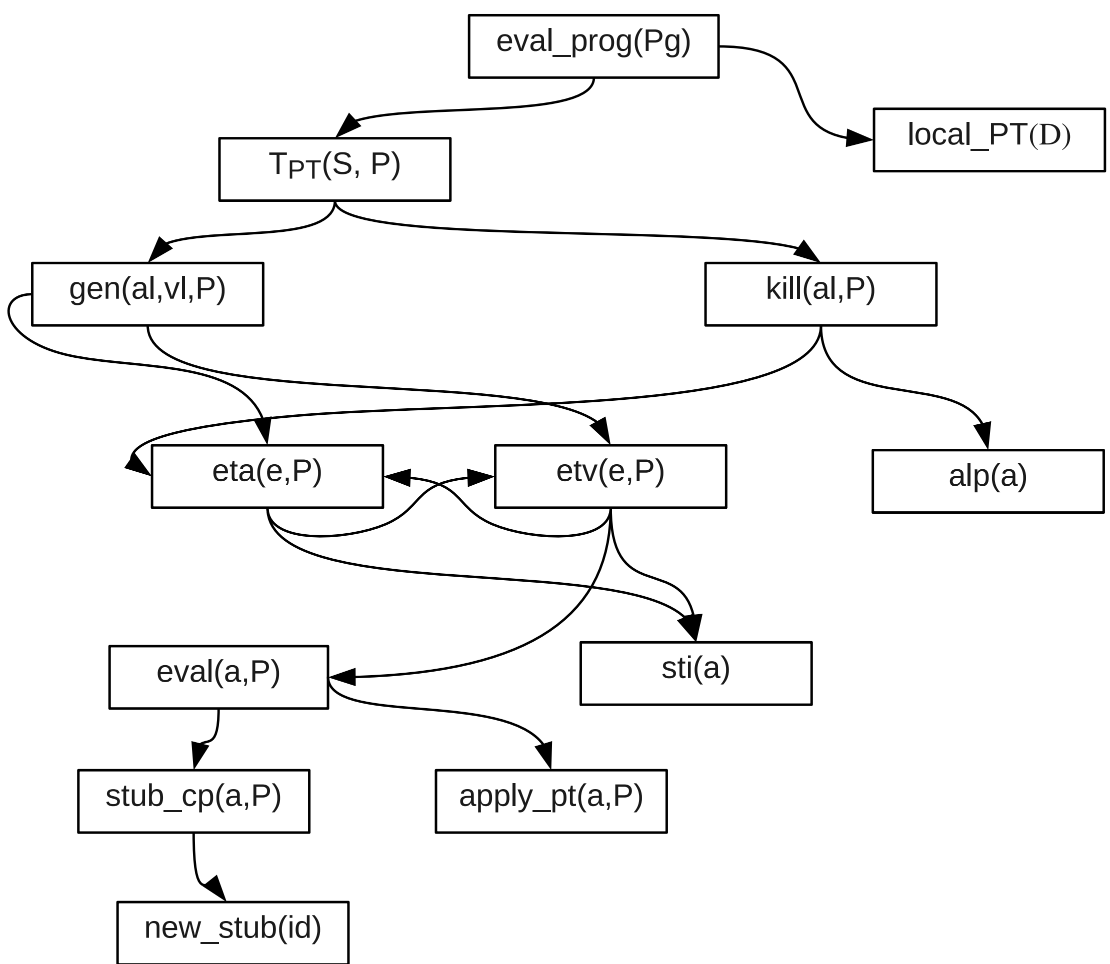


FIGURE 4.3 – le graphe des appels

crée pour un paramètre formel de type pointeur une cible vers laquelle il pointe, sont donnés aux sections 4.3.3.1 et 4.3.3.2 respectivement. Ces fonctions utilisent aussi le résultat de la fonction apply_ptqui pour un chemin constant renvoie la liste des cases mémoire pointées.

4.3.1.2 Le résultat de l'analyse *points-to*

L'analyse de pointeur est définie par la relation qui associe à chaque chemin de mémoire constant, de type pointeur, une ou plusieurs cases mémoire vers lesquelles il pointe potentiellement. Plus formellement, la relation *points-to* relative à un point de contrôle est définie comme suit :

$$P \in \mathcal{PT} : \mathcal{P}(\mathcal{A} \times \mathcal{A}) \cup \{\bot\} \tag{4.6}$$

Toutes les expressions sont traduites en chemin constants ; ainsi tous les déréférencements de pointeurs ont été remplacés par leurs cibles. L'élément $\bot$ correspond à une valeur particulière qui représente un cas d'erreur. Par exemple, si la constante `NULL` est trouvée en partie gauche d'une assignation, $\bot$ est généré. La dimension *type* des arcs *points-to* est très importante pour l'analyse des pointeurs et est utilisée pour décider de la cohérence et de la correction des arcs. Nous définissons une condition qui doit être vérifiée avant la création de chaque arc bien typé :

$$(a_1, a_2) \in P \text{ est bien typé ssi } \textsf{pointer}(\textsf{type_of}(a_2)) \sim pointed_type(\textsf{type_of}(a_1)) \tag{4.7}$$

La relation $\mathcal{PT}$ caractérise un graphe orienté dont les nœuds représentent des emplacements mémoire a référence constante, dont le type final, est un pointeur et les arcs la relation « pointe vers ». Elle associe à chaque couple (origine,destination) [6] un arc qui signifie qu'une source peut pointer vers une destination qui lui ait associée. Le graphe $\mathcal{PT}$ est un graphe simple qui ne contient pas d'arcs multiples reliant le même couple de nœuds ni de cycles.

4.3.2 Étapes de l'analyse *points-to*

Dans ce qui suit, nous présentons les différentes étapes nécessaires à l'analyse d'un programme écrit en `C`. L'analyse comporte une phase implicite d'initialisations des variables locales au niveau des déclarations (section 4.3.2.1), la construction du contexte formel et le traitement des instructions où en partie gauche apparaît un pointeur. La dernière sous-section est dédiée à la fonction eval() qui prend en argument un chemin d'accès constant pour renvoyer la liste des cases mémoire pointées par ce dernier, si la liste est vide et que le chemin est un paramètre formel alors la fonction appelle stub_cp, sous-section 4.3.3.2 pour créer à la volée les cibles. Les fonctions eta et etv sont quant à elles définies au niveau des sous-sections 4.3.3.3 et 4.3.3.4.

4.3.2.1 Initialisation des variables locales

Comme le montre la figure 4.3, l'analyse des pointeurs consiste à évaluer un programme en faisant appel à la fonction eval_prog. Cette fonction calcule en premier lieu les arcs *points-to* relatifs aux déclarations locales. Cette initialisation est effectuée via la fonction eval_local.

$$\begin{array}{rcl} \textsf{eval_prog} : \mathcal{P} & \rightarrow & \mathcal{PT} \\ \textsf{eval_prog}(\textsf{Prog}(\delta_p, \delta_l, s)) & = & \mathcal{T}_{\mathcal{PT}}[\![s]\!](\textsf{eval_local}(\delta_l), \delta_p) \end{array}$$

Cette dernière permet l'initialisation des variables locales de type pointeur. Elle crée un arc de la variable déclarée Id de type pointeur vers l'emplacement undefined du type pointé. Elle est définie comme suit :

6. Dans ce manuscrit nous utilisons aussi les termes (source, sink).

Fonction 3 : eval_local

eval_local : $\mathcal{A} \times \mathcal{D} \rightarrow \mathcal{PT}$
eval_local$(a : \mathcal{A}, \delta : \mathcal{D})$
$t =$ type_of(a, δ)
if $t =$ *pointer*(t') **then**
  return $\{(a, \text{undefined}'_t)\}$
else if $t \in$ *struct* **then**
  return $\bigcup_{f \in D_t}$ eval_local$(a.f, \delta)$
else if $t =$ *array*(t', N) **then**
  return $\bigcup_{j \in \{0,\ldots,N-1\}}$ eval_local$(a.[j], \delta)$
else
  return $\emptyset$

La fonction eval_local itère sur les déclarations. À chaque identifiant de type pointeur, elle associe un emplacement undefined. Ces arcs initiaux sont l'information en entrée pour l'analyse du programme.

4.3.2.2 Contexte formel

Comme l'initialisation des variables locales, une initialisation des paramètres formels de type pointeur est aussi effectuée. Seulement cette initialisation se fait à la demande ; c'est-à-dire que les paramètres formels ne sont initialisés que lorsqu'ils sont utilisés. Ceci inclut principalement les instructions de déréférencement. Le paramètre est initialisé à un stub, qui est une référence créée par l'analyse et qui représente un ou plusieurs emplacement mémoire connus des fonctions appelantes. Quand le stub est aussi un pointeur déréférencé, comme c'est le cas des paramètres formels de type multiple pointeur, l'analyse le fait pointer vers un nouveau stub et ainsi de suite. La création du contexte à la demande évite la création, ainsi que la propagation, d'une longue liste d'initialisation comme cela peut être le cas des structures de données récursives. Cette initialisation est effectuée par la fonction stub_cp (voir fonction 5).

4.3.3 Traduction des expressions d'adresses en chemins d'accès mémoire constants (CP)

Une des originalités de l'analyse *points-to* implémentée dans PIPS est de ne pas passer par une simplification des lhs via du code trois adresses, afin de pouvoir restituer le code source aussi fidèlement que possible. Les expressions complexes sont tout fois traduites sous la forme d'une représentation plus facile à analyser. Prenons par exemple les déclarations suivantes :

```
struct stuff {int val; struct stuff * next;} s1;
struct stuff *q = &s1;
```

Prog 4.2 – Structure de données récursive

Les *lhs* sont traduits en chemins d'accès dans $\mathcal{E}$. Ces derniers n'utilisent que la construction *subscript* pour convertir les notations usuelles de C de la manière suivante :

Pour pouvoir calculer de manière unique les ensembles de base au niveau des expressions d'adresse, on a besoin de les traduire en des chemins constants. Prenons cette fois-ci comme exemple l'expression `**p`, qui est d'abord traduite en une pseudo référence `p[0][0]` par norma-

```
q->next      -> q[0][2]        // next est le second champ
(*q).next    -> q[0][2]
q->next->next -> q[0][2][0][2]
```

Prog 4.3 – Traduction des chemins constants

lisation ; puis chaque déréférencement est remplacé par un ensemble contenant les emplacements mémoire qu'il représente en utilisant l'information *points-to* disponible.

La traduction d'une expression d'adresse en un ensemble de chemin d'accès constants s'effectue en trois étapes :

1. normaliser l'expression d'adresse ; par exemple, `**p` en `p[0][0]`, `my_str.p` en `my_str[p]`, `my_str->q` en `my_str[0][q]`.
2. évaluer les chemins en utilisant l'information *points-to* disponible initialement ; par exemple, `p[0][0][0]` est modifié en remplaçant `p[0]` par sa cible s'il apparaît dans un arc *points-to* comme `(p,q)` ce qui donnerait `q[0][0]`.
3. en fonction de l'évaluation précédente construire un ensemble de chemins constants si la substitution de `p[0]` par sa cible est possible de multiples manières à cause de l'imprécision inévitable d'une analyse statique. Renvoyer le cas d'erreur s'il y a déréférencement d'un pointeur non initialisé.

Pour illustrer le processus de traduction considérons le programme 4.4 qui contient des pointeurs multiples initialisés au niveau de la déclaration. Le but est d'abord de traduire l'expression `***p` sous la forme d'un accès à un tableau puis de traduire ce dernier en utilisant l'information *points-to*.

```
int i = 0, *p2 = &i, **p1 = &p2, ***p = &p1 ;
***p = 2;
```

Prog 4.4 – Exemple d'expression complexe

La figure suivante montre le processus de traduction pour l'expression `***p` du programme 4.4, avec comme pour information *points-to* les arcs $\{(p, p1), (p1, p2), (p2, i)\}$.

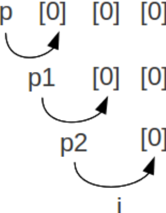


FIGURE 4.4 – Traduction de `***p`

D'une manière plus fonctionnelle, une première fonction eta est définie pour calculer l'ensemble des adresses L correspondant à une expression l :

$$(L, P') = \text{eta}([\![l]\!], P) \tag{4.8}$$

eta prend en entrée le terme gauche d'une assignation de type pointeur, ainsi qu'un ensemble d'arcs *points-to*. Elle évalue l'expression *lhs* en utilisant la relation P dans le but de la changer en un ensemble de chemins constants pouvant apparaître comme une source d'un arc *points-to*. Par exemple, pour une expression comme $*x$, le graphe de la relation P est parcouru en cherchant les arcs où x apparaît comme source. La fonction aura comme résultat l'ensemble des emplacements pointés par x (c'est-à-dire les « sinks »).

Quant à la partie droite de l'assignation, l'évaluation utilise les arcs *points-to* courants et effectue un déréférencement de plus que pour la partie gauche. L'équation générale pour la partie droite a la même signature que celle de la partie gauche et est définie comme suit :

$$(R, P') = \text{etv}([\![r]\!], P) \tag{4.9}$$

Les définitions des fonctions eta et etv, plus générales que la simple conversion d'un chemin d'accès en un chemin d'accès constant, sont fournies sous-sections 4.3.3.3 et 4.3.3.4. Il faut noter que la relation *points-to* P peut être modifiée en une relation P' par ces deux fonctions quand le contexte formel est augmenté suite à un déréférencement.

4.3.3.1 Définition de la fonction `eval`

La fonction eval décrite à la Fonction 4 reçoit comme argument un chemin constant a et un ensemble d'arcs *points-to* P. Elle évalue une référence en utilisant l'information *points-to*.

Fonction 4 : eval

eval : $\mathcal{A} \times \mathcal{PT} \to \mathcal{P}(\mathcal{A}) \times \mathcal{PT}$
eval$(a : \mathcal{A}, P : \mathcal{PT}) \mapsto (A', P')$
if $a \in \{$*undefined*, *NULL*$\}$ **then**
  return $(\emptyset, \bot)$
else
  let $e = apply_pt(a, P)$
  if $e \neq \emptyset$ **then**
    return (e, P)
  else
    return eval$(a,$ stub_cp$(a, P))$

Le chemin constant est supposé être de type pointeur. En premier lieu, si un pointeur est évalué à undefined ou `NULL` l'évaluation retourne une liste vide de chemins d'accès ainsi qu'un ensemble d'arcs *points-to* qui vaut $\bot$. En second lieu, l'ensemble des arcs *points-to* est utilisé pour chercher la ou les cibles de a. C'est le rôle de la fonction $apply_pt$. Si aucune cible n'est trouvée et que a est un paramètre formel ou un *stub* (voir section 4.3.3.2), alors P est augmenté par la nouvelle cible créée par stub_cp pour construire le contexte formel représentant un site d'appel générique sans aliasing. La fonction renvoie la cible ainsi que l'ensemble P augmenté.

La fonction est appelée principalement pour déterminer quelle variable est lue ou écrite par une instruction de déreférencement de pointeur. Le déréférencement d'un pointeur non-initialisé ou qui vaut `NULL` aboutit à un ensemble d'arcs non valide. Nous pouvons alors conclure que le programme comporte des erreurs.

4.3.3.2 Définition des fonctions `stub_cp` et `new_stub`

Cette fonction, présentée à la figure 5, prend en entrée un chemin d'accès constant a, ainsi qu'un ensemble P d'arcs *points-to*.

Fonction 5 : stub_cp

```
stub_cp : A × PT → PT
stub_cp(a : A, P : PT)
if a ∈ sources(P) then
    return P
else if type(a) = pointer(t) then
    n = new_stub(a)
    type(n) = t
    return stub_cp(n, P ∪ {(a, n)})
else if type(a) = struct then
    P′ = P
    foreach f ∈ D_type(a) do
        P′ = stub_cp(a.f, P′)
    return P′
else if type(a) = array(t′, N) then
    P′ = P
    for j ∈ {0, ..., N − 1} do
        P′ = stub_cp(a[j], P′)
    return P′
else
    return P
```

Selon le type de a, la fonction crée un nouvelle variable dont le nom est dérivé de a et dont le type correspond au type pointé par a. Elle retourne comme résultat un nouvel ensemble qui est l'union de P et des nouveaux arcs créés. Lors de la création des arcs, la fonction teste le type de la nouvelle cible créée à partir de a, et, tant que c'est de type pointeur, elle continue à créer des cibles jusqu'à tomber sur un type basique (différent des types pointeurs et struct).

La fonction appelle à son tour la fonction new_stub, qui permet de générer un nouveau nom à partir d'un chemin constant Elle est défini comme suit :

$$\text{new_stub} : \mathcal{A} \rightarrow \mathcal{I}$$

et la fonction sources(P), qui renvoie les chemins d'accès qui apparaissent comme source au niveau des arcs *points-to*, est définie par :

$$\begin{array}{rcl} \text{sources} : \mathcal{PT} & \rightarrow & \mathcal{P}(\mathcal{A}) \\ P & \mapsto & \{a \in \mathcal{A} | \exists s \in \mathcal{A}, (a, s) \in P\} \end{array}$$

4.3.3.3 Définition de la fonction `eta`, `expression_to_address()`

La fonction eta est défini à la fonction 6. Elle prend en entrée le terme gauche d'une assignation de type pointeur, ainsi qu'un ensemble d'arcs *points-to*. Elle évalue l'expression e_t dans le but de la changer en un ensemble de chemins constants pouvant apparaître comme une source [7] d'un arc *points-to*, c'est-à-dire comme la partie gauche d'une assignation de pointeurs.

Par exemple, pour une expression de type $*x$, l'ensemble P est parcouru en cherchant les relations où x apparaît comme source.

La fonction a comme résultat l'ensemble des emplacements pointés par x ; c'est-à-dire ses destinations(voir ci-dessous pour la définition de sti). La fonction renvoie `error` dans le cas où

7. Le terme origine est aussi utilisé.

Fonction 6 : eta

eta : $\mathcal{E} \times \mathcal{PT} \rightarrow \mathcal{P}(\mathcal{A}) \times \mathcal{PT}$
eta$(e : \mathcal{E}, P : \mathcal{PT}) \mapsto (A, P')$
switch e **do**
  case Id
    return $(\{\mathrm{Id}\}, P)$
  case $e'.f$
    $(A', P') =$ eta(e', P)
    return $(\bigcup_{a' \in A'} \{a'.f\}, P')$
  case $*e$
    return etv(e, P)
  case $\&e$
    return $(\{\mathtt{error}\}, P)$
  case $e_1[e_2]$
    $(A', P') =$ eta(e_1, P)
    return $(\bigcup_{a' \in A'} \{a'[\text{sti}(e_2)]\}, P')$
  case *cte*
    return $(\{\mathtt{error}\}, P)$

cas	typage	syntaxe	résultat $\mathcal{P}(\mathcal{A})$	résultat $\mathcal{PT}$
Id	pointer	`p`	$\{p\}$	P
$e'.f$	pointer	`s.f`	$\{s.f\}$	P
	pointer	`(*s).f`	$\{_s_1.f\}$	$P \cup \{(s, _s_1)\}$
	pointer	`a[1].f`	$\{a[1].f\}$	P
	pointer	`a[i].f`	$\{a[*].f\}$	P
$\&e$	int	`&p`	$\{\mathtt{error}\}$	P
$*e$	pointer	`*p`	$\{_p_1\}$	$P \cup \{(p, _p_1)\}$
	pointer	`**p`	$\{_p_1_1\}$	$P \cup \{(p, _p_1), (_p_1, _p_1_1)\}$
$e1[e2]$	pointer	`a[1]`	$\{a[1]\}$	P
	pointer	`a[*p]`	$\{a[*]\}$	P
cte	int	`1`	$\{\mathtt{error}\}$	P

Tableau 4.2 – Exemples de calcul d'adresses abstraites correspondant à des expressions

l'expression ne peut apparaître comme terme gauche comme par exemple la constante `NULL` ou l'opérateur d'adressage $\&$. D'autres exemples sont traités dans le tableau 4.2. Il faut noter que, lorsque l'expression comprend un paramètre formel qui n'a pas été initialisé, l'ensemble des arcs *points-to* est augmenté avec l'initialisation à la demande du paramètre.

4.3.3.4 Définition de la fonction `etv`, `expression_to_values()`

La fonction etv (Fonction 7) prend en entrée le terme droit d'une assignation ainsi qu'un ensemble d'arcs *points-to*. Elle évalue l'expression e dans le but de la changer en un ensemble de chemins constants pouvant apparaître comme une destination d'un arc *points-to*[8]. La fonction effectue une évaluation de plus que la fonction eta ; dans le cas où e est un identificateur, un test sur le type t de e est effectué. Si t est un pointeur, alors il est évalué pour obtenir la case mémoire vers laquelle il pointe. L'évaluation de l'expression est faite par la fonction `eval` (définie au niveau de la section 4.3.3.1). Pour les expressions complexes qui comportent un chemin d'accès composé de sous chemins qui doivent être évalués de gauche à droite, comme par exemple `(*s).p`, la fonction commence par récupérer le préfixe du chemin qui correspond à la partie structure `*s`. Le préfixe est traduit via la fonction eta. Cette dernière renvoie une liste de chemins d'accès constants a_1 sur lesquels la fonction itère en concaténant a_1 au suffixe de l'expression e et en fournissant le résultat comme argument à la fonction eval. La fonction couvre tous les cas possibles en effectuant une disjonction sur tous les types d'expressions. Dans le cas où e_t est un élément de tableau, l'expression qui représente l'indice est passée comme argument à la fonction sti[9]. La fonction teste si l'expression est dépendante de l'état
mémoire[10] *via* le prédicat sdp[11].

Si ce dernier est évalué à vrai, alors l'expression est changée en $*$ qui représente n'importe quel élément du tableau[12]. La pseudo-expression $*$ représente un entier compris entre 0 et l'indice du dernier élément du tableau. Sinon, il s'agit d'une valeur constante et l'expression est renvoyée comme résultat.

$$\begin{aligned} \text{sti} : \mathcal{E} &\rightarrow \mathcal{C} \\ e &\mapsto \text{si } \mathsf{sdp}(e) \text{ alors } * \text{ sinon } e \end{aligned}$$

Le tableau 4.3 montre des exemples de calcul de destination à partir des expressions qui apparaissent en partie droite. L'ensemble des arcs P est augmenté par les nouvelles cibles des paramètres formels créées par la fonction `stub_cp`.

4.4 Définition de l'analyse *points-to* pour le langage $\mathcal{L}_0$

Après la définition des fonctions eta et etv qui constituent les premières étapes de l'analyse des pointeurs, nous définissions les fonctions Kill et Gen qui permettent de calculer l'information *points-to* finale pour les instructions du type affectation et test.

4.4.1 L'opérateur d'assignation d'un opérateur

Les instructions « génératrices » de arcs *points-to* (par exemple 4.6) sont principalement les affectations dont la partie gauche est un pointeur. Cette dernière est traduite en un ou plusieurs

8. appelée aussi sink
9. sti : abréviation de *store independent index*.
10. L'état mémoire est appelé « store »
11. sdp : abréviation d prédicat *store dependent*
12. Ceci correspond a l'utilisation très simple d'un treillis, le treillis des constantes, qui est défini en détails dans le chapitre 5.

Fonction 7 : etv

etv : $\mathcal{E} \times \mathcal{PT} \to \mathcal{P}(\mathcal{A}) \times \mathcal{PT}$
etv$(e, P) \mapsto (A, P')$
switch e **do**
  case Id
    if *type*(Id) = *pointer*(t) **then**
      return eval(Id, P)
    else
      return $(\{*\}, P)$
  case $e'.f$
    $(A', P') =$ eta(e', P)
    $A = \emptyset$
    foreach $a \in A' \wedge$ *type*$(a.f) = pointer(t')$ **do**
      $(A'', P') =$ eval$(a.f, P')$
      $A = A \cup A''$
    return (A, P')
  case $*e'$
    $(A', P') =$ etv(e', P)
    $A = \emptyset$
    foreach $v \in A' \wedge$ *type*$(v) = pointer(t')$ **do**
      $(A'', P') =$ eval(v, P')
      $A = A \cup A''$
    return (A, P')
  case $\&e'$
    return eta(e', P)
  case $e_1[e_2]$
    $(A', P') =$ eta(e_1, P)
    $A = \emptyset$
    foreach $a \in A' \wedge$ *type*$(a[e_2]) = pointer(t')$ **do**
      $(A'', P') =$ eval$(a[$sti$(e_2)], P')$
      $A = A \cup A''$
    return (A, P')
  case *cte*
    return $(\{$cte$\}, P)$

cas	**typage**	**lhs**	**résultat** $\mathcal{P}(\mathcal{A})$	**résultat** $\mathcal{PT}$
Id	pointer	`p`	$\{_p_1\}$	$P \cup \{(p, _p_1)\}$
$e'.f$	pointer	`s.f`	$\{s._f_1\}$	$P \cup \{(s.f, s._f_1)$
	pointer	`(*s).f`	$\{_s_1._f_1\}$	$P \cup \{(s, _s_1), (_s_1.f, _s_1._f_1)\}$
	pointer	`a[1].f`	$\{a[1]._f_1\}$	$P \cup \{(a[1].f, a[1]._f_1)\}$
	pointer	`a[i].f`	$\{a[*]._f_1\}$	$P \cup \{(a[*].f, a[*]._f_1)\}$
$\&e$	int	`&p`	$\{\mathtt{error}\}$	P
$*e$	pointeur	`*p`	$\{_p_1_1\}$	$P \cup \{(_p_1, _p_1_1)\}$
	pointeur	`**p`	$\{_p_1_1_1\}$	$P \cup \{(_p_1, _p_1_1), (_p_1_1, _p_1_1_1)\}$
$e1[e2]$	pointer	`a[1]`	$\{_a_1[1]\}$	$P \cup \{(a[1], _a_1[1])\}$
	pointer	`a[*p]`	$\{_a_1[*]\}$	$P \cup \{(a[*], _a_1[*])\}$
cte	int	`1`	$\{1\}$	P

Tableau 4.3 – Exemples de calcul des valeurs correspondantes à des expressions

chemins d'accès mémoire constants (voir la section 4.3.3). Si les expressions droite et gauche ont pu être traduites toutes les deux en ensembles de chemins d'accès constants, L et R, on détermine alors les ensembles de base nécessaires au calcul des arcs *points-to*. Comme pour toute analyse de flot de données, ces ensembles sont Kill et Gen et ils sont calculés en fonction de L, R et In, la valeur courante des *points-to*.

4.4.1.1 Définition de l'ensemble Gen

L'ensemble Gen correspond aux nouveaux arcs générés par l'instruction en cours d'analyse. Il se calcule généralement pour une affectation de la forme

$$\boxed{\textit{lhs} = \textit{rhs}}$$

Il correspond aux arcs *points-to* générés par une affectation. Le calcul de ces arcs suppose que, au préalable, la partie gauche de l'assignation a été transformée en une liste de chemins d'accès mémoire constants qui peuvent apparaître comme source de l'arc. La partie droite est aussi pré-traduite en une liste de chemins d'accès mémoire constants qui peuvent apparaître comme destination d'un arc *points-to*. Ces traductions sont effectuées par les fonctions eta et etv et leurs résultats A et V sont fournis à l'ensemble Gen qui itère sur leurs éléments en associant à chaque source une des destinations possibles pour créer l'arc en question.

$$\begin{array}{rcl} \text{Gen} : \mathcal{P}(\mathcal{A}) \times \mathcal{P}(\mathcal{A}) & \rightarrow & \mathcal{PT} \\ \text{Gen}(A, V) & = & \text{si } A = \emptyset \vee V = \emptyset \text{ alors } \bot \\ & & \text{sinon } \; \{(a, v) | a \in A \wedge v \in V\} \end{array} \quad (4.10)$$

4.4.1.2 Définition de l'ensemble Kill

Cet ensemble représente les arcs *points-to* qui doivent être supprimés de l'ensemble courant parce que l'instruction en cours fait pointer la source vers un nouvel emplacement. Le précédent arc n'est plus valable. L'ensemble Kill récupère à son tour le résultat de la fonction `eta` et itère sur ses éléments en testant chaque fois si l'élément apparaît comme une source d'un arc *points-to*

dans l'ensemble P. Tous les arcs identifiés sont retournés comme résultat.

$$\begin{aligned} \text{Kill} : \mathcal{P}(\mathcal{A}) \times \mathcal{PT} &\rightarrow \mathcal{PT} \\ \text{Kill}(A, P) &= \text{si } A = \emptyset \text{ alors } \bot \\ &\quad \text{sinon } \{(l, r) \in P | l \in A \wedge |A| = 1\} \end{aligned} \tag{4.11}$$

4.4.1.3 Définition de la fonction transformeur $T_{\mathcal{PT}}$

La fonction $T_{\mathcal{PT}}$ correspond au transformeur d'arcs *points-to*. Il prend en entrée un *statement* ainsi qu'un ensemble initial d'arcs *points-to*. Selon le type du *statement* en cours un traitement spécifique est appliqué. Notre sous-ensemble contient trois types de *statements* :

1. la séquence : le *statement* en cours est analysé en utilisant l'information *points-to* calculée par le *statement* précédent ;
2. le test : une union [13] des résultats des deux branches est effectuée ;
3. l'opérateur d'assignation : c'est l'instruction génératrice des arcs *points-to*.

Nous définissons par la suite les opérateurs sur les transformeurs $T_{\mathcal{PT}}$ ainsi que les propriétés de l'élément $\bot$ introduit au niveau de la sous section 4.3.3.1. Quand une instruction génère un arc *points-to* faux, comme par exemple la constante `NULL` en partie gauche, nous avons besoin de modéliser cette valeur particulière que peut prendre $\mathcal{PT}$. Cette valeur particulière est $\bot$; cet élément est considéré comme l'élément absorbant pour l'opération $\circ$ utilisée pour le calcul des arcs *points-to* pour une séquence $s_1; s_2$. Ainsi, si l'on combine deux transformeurs :

$$\begin{aligned} T_{\mathcal{PT}Seq}[\![s_1; s_2]\!] &= T_{\mathcal{PT}2} \circ T_{\mathcal{PT}1} = T_{\mathcal{PT}}[\![s_2]\!](T_{\mathcal{PT}}[\![s_1]\!]) \\ T_{\mathcal{PT}}[\![s]\!](\bot) &= \bot \end{aligned} \tag{4.12}$$

L'élément $\bot$ est par contre considéré comme l'élément neutre pour l'opération $\sqcup$ ainsi :

$$\begin{aligned} \mathcal{PT}_1 \sqcup \mathcal{PT}_2 &= \text{si } \mathcal{PT}_1 = \bot \text{ alors } \mathcal{PT}_2 \\ &\quad \text{sinon si } \mathcal{PT}_2 = \bot \text{ alors } \mathcal{PT}_1 \\ &\quad \text{sinon } \mathcal{PT}_1 \cup \mathcal{PT}_1 \end{aligned} \tag{4.13}$$

Après la définition des opérateurs, nous pouvons à présent définir le transformeur $T_{\mathcal{PT}}$, voir la Fonction 8. Quant aux transformeurs $T_{\mathcal{PT}ass}$ et $T_{\mathcal{PT}test}$ ils sont définis par les fonctions 9 et 10.

4.4.1.4 Traitement des affectations, $T_{\mathcal{PT}ass}$

Avant de créer un arc *points-to*, la partie gauche et la partie droite de l'affectation sont évaluées *via* les fonctions `eta` et `etv`. Cette évaluation permet de se rendre compte des éventuelles erreurs que le programme peut contenir. En effet si en partie droite l'évaluation retourne undefined cela veut dire que le pointeur n'a pas été initialisé mais qu'il est quand même utilisé. Cette erreur se traduit par un ensemble d'arcs qui vaut $\bot$. Après l'évaluation, un test sur le type de la partie gauche est effectué. Si ce dernier est de type pointeur alors la création de l'arc peut commencer. La première étape consiste à filtrer les résultats des précédentes évaluations ; de la liste des chemins constants de la partie gauche les chemins `NULL` et undefined doivent être supprimés. En effet, un arc *points-to* ne peut pas avoir une source qui vaut ces valeurs, cela veut

13. Un opérateur d'union spécifique aux ensembles d'arcs *points-to*, $\sqcup$.

Fonction 8 : $T_{\mathcal{PT}}$

$T_{\mathcal{PT}} : \mathcal{S} \times \mathcal{PT} \rightarrow \mathcal{PT}$
$T_{\mathcal{PT}}[\![s]\!](P) \mapsto P'$
if $s =$ *Sequence*(s_1, s_2) **then**
  return $T_{\mathcal{PT}Seq}[\![s]\!](P)$
else if $s =$ *Test*(c, s_1, s_2) **then**
  return $T_{\mathcal{PT}test}[\![s]\!]$
else if $s =$ *Assign*(e_1, e_2) **then**
  return $T_{\mathcal{PT}ass}[\![s]\!]$

Fonction 9 : $T_{\mathcal{PT}ass}$

$T_{\mathcal{PT}ass} : \mathcal{S} \times \mathcal{PT} \rightarrow \mathcal{PT}$
$T_{\mathcal{PT}ass}(P) \mapsto P'$
$(A_1, P') = \text{eta}(e_1, P)$
$(A_2, P'') = \text{eta}(e_2, P')$
if *type*$(e_1) =$ *pointer*(t') **then**
  $A' = A_1 - \{\text{undefined}, \texttt{NULL}\}$
  $(V, P''') = \text{etv}(e_2, P'')$
  $V' = V - \{\text{undefined}\}$
  return $(P''' - \text{Kill}(A', P''')) \cup \text{Gen}(A', V')$
else
  return P''

dire que le pointeur déréférencé n'a pas été initialisé. De la partie droite seul undefined doit être supprimé. Après la suppression des chemins d'accès non valides, l'équation de flot de données classique est appliquée. Si la partie gauche n'est pas un pointeur alors l'information *points-to* est propagée au *statement* suivant.

4.4.1.5 Traitement des tests, fonction $T_{\mathcal{PT}test}$

La fonction analysant les tests est la Fonction 10. Les difficultés liées à la mise à jour du contexte formel et à la précision des arcs sont cachées dans l'opérateur $\sqcup$.

Fonction 10 : $T_{\mathcal{PT}test}$

$T_{\mathcal{PT}test} : \mathcal{S} \times \mathcal{PT} \rightarrow \mathcal{PT}$
$T_{\mathcal{PT}test}[\![\text{if } (c)\ s_1 \text{else } s_2]\!](P) \mapsto P'$
return $T_{\mathcal{PT}}[\![s_1]\!](\text{ntp}(c, \text{true}, P)) \sqcup T_{\mathcal{PT}}[\![s_2]\!](\text{ntp}(c, \text{false}, P))$

La fonction ntp, qui à ce stade du manuscrit vaut la fonction identité, est étendue par la suite pour évaluer les conditions et leurs impacts sur la relation *points-to* ; par exemple, elle peut tester si un pointeur est différent du pointeur `NULL`.

4.4.2 Exemples

Nous illustrons les fonctions de traitement des expressions `C`, `eta`, `etv` ainsi que le transformeur *points-to* définis ci-dessous par deux exemples. Ces derniers comportent des chemins

d'accès complexes qu'il faut transformer en chemins constants.

4.4.2.1 Exemple 1

Le premier exemple illustre une initialisation d'un pointeur à `NULL`. Le déréferencement par la suite de ce pointeur devrait aboutir à un arrêt du programme, nous verrons par la suite comment l'information *points-to* permet de détecter certaines erreurs de programmation.

```
int *p;    // S1
p = NULL;  // S2
*p = 1;    // S3
```

Prog 4.5 – Exemple 1

La fonction local$_{\mathcal{PT}}$ calcule les arcs *points-to* des déclarations et fournit le résultat suivant :

$$\{(p, \text{undefined})\}$$

Le *statement* `S2` a en partie gauche un pointeur ; nous appliquons `eta` sur la partie gauche et `etv` sur la partie droite.

$$\text{eta}(p, \{(p, \text{undefined})\}) = (\{p\}, \{(p, \text{undefined})\})$$

$$\text{etv}(\texttt{NULL}, \{(p, \text{undefined})\}) = (\{\texttt{NULL}\}, \{(p, \text{undefined})\})$$

A partir de eta, nous obtenons l'ensemble Kill suivant :

$$\text{Kill} = \{(p, \text{undefined})\}$$

et à partir de eta et etv, nous obtenons l'ensemble Gen suivant :

$$\text{Gen} = \{(p, \texttt{NULL})\}$$

Les arcs obtenus après l'analyse du *statement* 2 sont :

$$\text{Out} = \{(p, \texttt{NULL})\}$$

L'analyse du dernier *statement* $S3$ fournit les résultats suivants :

$$\text{eta}(*p, \{(p, \texttt{NULL})\}) = (\{\texttt{NULL}\}, \{(p, \texttt{NULL})\})$$

et :

$$\text{eta}(1, \{(p, \texttt{NULL})\}) = (\{error\}, \{(p, \texttt{NULL})\})$$

Comme le type de l'expression en terme gauche n'est pas un pointeur, l'analyse s'arrête avec le résultat du *statement* précédent.

4.4.2.2 Exemple 2

Le deuxième exemple(Prog 4.6) permet l'initialisation des champs pointeurs d'un struct. L'argument de la fonction est un pointeur vers un struct, d'où les initialisation à des stubs. Les pointeurs de la structure sont des doubles pointeurs qui nécessitent une évaluation pour être transformés en chemins d'accès constants.

La fonction local$_{\mathcal{PT}}$ calcule les arcs *points-to* générés par des déclarations. Comme ces dernières ne comportent pas de pointeurs dans cet exemple, la relation *points-to* obtenue pour `S1` est vide.

```
struct stuff { int ** pp1; int **pp2;};
void foo(struct stuff *ps)
{
  int i = 1, j = 2;      // S1
  *((*ps).pp1) = &i;     // S2
  *((*ps).pp2) = &j;     // S3
  (*ps).pp1 = (*ps).pp2; // S4
}
```

Prog 4.6 – Exemple 2

Le *statement* S2 a en partie gauche un pointeur. Nous appliquons eta sur la partie gauche et etv sur la partie droite.

$$\text{eta}(*((*ps).pp1), \emptyset) = (\{_ps_1._pp1_1\}, \{(ps \rightarrow _ps_1), (_ps_1.pp1 \rightarrow _ps_1._pp1_1)\})$$

$$\begin{aligned} P_1 &= \{(ps, _ps_1), (_ps_1.pp1, _ps_1._pp1_1)\}, \\ \text{etv}(\&i, P_1) &= (\{i\}, P_1) \end{aligned}$$

A partir de eta nous obtenons l'ensemble Kill suivant :

$$\text{Kill} = \emptyset$$

et à partir de eta et etv, nous obtenons l'ensemble Gen suivant :

$$\text{Gen} = \{(_ps_1._pp1_1, i)\}$$

Les arcs obtenus après le *statement* S2 du programme sont :

$$\text{Out} = \{(_ps_1._pp1_1, i), (ps, _ps_1), (_ps_1.pp1, _ps_1._pp1_1)\}$$

Le *statement* S3 a en partie gauche un pointeur. Nous appliquons eta sur la partie gauche et etv sur la partie droite.

$$\begin{aligned} \text{eta}(*((*ps).pp2), \text{Out}) &= (\{_ps_1._pp2_2\}, \\ &\quad \{(ps, _ps_1), (_ps_1.pp2, _ps_1._pp2_2), \\ &\quad (_ps_1._pp1_1, i), (_ps_1.pp1, _ps_1._pp1_1)\}) \\ \text{etv}(\&j, \text{Out} \cup \{(_ps_1.pp2, _ps_1._pp2_2)\}) &= (\{j\}, \text{Out} \cup \{(_ps_1.pp2, _ps_1._pp2_2)\}) \end{aligned}$$

A partir de eta nous obtenons l'ensemble Kill suivant :

$$\text{Kill} = \emptyset$$

et à partir de eta et etv nous obtenons l'ensemble Gen suivant :

$$\text{Gen} = \{(_ps_1._pp1_2, j)\}$$

Les arcs obtenus après le *statement* S3 du programme sont :

$$\begin{aligned} \text{Out} &= \{(_ps_1._pp1_1, i), (ps, _ps_1), (_ps_1.pp1, _ps_1._pp1_1), \\ &\quad (_ps_1._pp1_2, j), (_ps_1.pp2, _ps_1._pp2_2)\} \end{aligned}$$

Le *statement* S4 a en partie gauche un pointeur. Nous appliquons eta sur la partie gauche et etv sur la partie droite.

$$\text{eta}((*ps).pp1, \text{Out}) = (\{_ps_1.pp1\}, \text{Out})$$

$$\text{etv}((*ps).pp2, \text{Out}) \quad = \quad (\{_ps_1._pp2_2\}, \text{Out})$$

À partir de eta nous obtenons l'ensemble Kill suivant :

$$\text{Kill} = \{(_ps_1.pp1, _ps_1._pp1_1)\}$$

Et à partir de eta et etv, nous obtenons l'ensemble Gen suivant :

$$\text{Gen} = \{(_ps_1.pp1, _ps_1._pp2_2)\}$$

Les arcs obtenus après le *statement* S4 du programme sont :

$$\begin{aligned} \text{Out} \quad = \quad & \{(_ps_1._pp1_1, i), (ps, _ps_1), (_ps_1._pp1_2, j), \\ & (_ps_1.pp2, _ps_1._pp2_2), (_ps_1.pp1, _ps_1._pp2_2)\} \end{aligned}$$

Cet exemple, qui comprend des chemins d'accès complexes à des champs d'une structure, montre comment sont traduits ces chemins en appliquant les fonctions eta et etv.

4.5 Définition du langage $\mathcal{L}_1$

Après avoir défini l'analyse pour un sous-ensemble du langage C, nous introduisons un langage plus riche très proche du C. Au niveau de ce langage, appelé $\mathcal{L}_1$, les chemins d'accès non constants sont plus complexes. Ils sont construits à partir d'une expression, le domaine $\mathcal{E}$, en utilisant en plus le déréférencement de pointeur, qui ressemble à une indexation comme `p[3]`, mais qui n'en est pas une à cause du type pointeur de la variable `p`.

4.5.1 Langage $\mathcal{L}_1$ et notations

Dans ce qui suit nous commençons par enrichir le langage $\mathcal{L}_0$ et le faire évoluer vers un langage $\mathcal{L}_1$ plus riche en introduisant les expressions. Ensuite nous présentons les nouvelles notations associées à ce langage.

4.5.1.1 Définition du langage $\mathcal{L}_1$

Nous allons définir l'analyse *points-to* pour les structures de contrôle ainsi que pour les expressions au niveau du langage C. Pour cela nous introduisons la syntaxe d'un sous-langage $\mathcal{L}_1$ qui correspond à des instructions C représentés dans la représentation intermédiaire [RI] de PIPS [pip]. Nous nous intéressons aux instructions qui peuvent être une séquence ou une structure de contrôle ou encore une expression. Le traitement des séquences va assurer l'aspect sensible au flot de contrôle, c'est-à-dire que les arcs *points-to* d'une instruction vont dépendre des arcs de l'instruction qui la précède et être calculés à partir de ceux-ci.

Les structures de contrôle comme les tests ou les boucles nécessitent la notion d'approximation sur les arcs *points-to*. Ceci est dû au fait qu'en général les conditions de tests ou de boucles ne peuvent pas être évaluées statiquement. On est donc dans l'obligation de prendre en compte les chemins possibles d'exécution et de sur-approximer le résultat en prenant l'union des arcs générés au niveau de chaque chemin. La figure 4.5 illustre la syntaxe du sous-langage $\mathcal{L}_1$. Le cas des expressions sera détaillé par la suite au niveau de la section 4.6.

Pour résumer, un *statement*, au niveau de la représentation interne PIPS, est une instruction. Cette instruction peut être un appel à une fonction, une séquence d'instruction ou encore une expression. Plus précisément une instruction peut être :

- une sequence ;
- un test ;

```
< statement >   S ::=     exit
                        | < expression > = < expression >
                        | if < expression > < s1 > else < s2 >
                        | return (< expression >)
                        | sequence < s >*

< expression >  E ::=     < constant >
                        | < reference >
                        | (< expression >)
                        | < u_op > < expression >
                        | < expression > < b_op > < expression >
                        | < sizeofexpression >
                        | < call >
                        | < va_args >

< unary_op >    u_op ::=  - | ! | &| *| ++| -

< binary_op >   b_op ::=  = | + | - | * | / | < | > | <= | >= | == | !=

< reference >   R ::=     < name > | < name >[< expression >*]

< type >        T ::=     int | float | pointer(t) | struct
```

FIGURE 4.5 – Syntaxe du sous-langage $\mathcal{L}_1$

- une expression ;
- une instruction « return » ;
- ou une instruction « exit ».

Notons que, par rapport au langage $\mathcal{L}_0$, le nombre d'instructions a légèrement augmenté : les instructions de « return » ou « exit ») sont apparues. A noter aussi l'apparition de l'expression qui englobe plusieurs sous-cas d'instructions comme par exemple l'opérateur d'affectation. Pour le reste du chapitre nous allons traiter les instructions « return » et « exit ».

4.5.1.2 Notations

Les notations utilisées dans la suite du chapitre sont rassemblées dans le tableau 4.4 ci-après.

Notation	Signification
$\mathcal{S}$	statement
$\mathcal{E}$	expression
Soe	size of expression
Ret	return
Ext	exit
u_op	unary operator
b_op	binary operator
C	call
$Varg$	variable argument list
Ir	intrinsic
UC	user function call
Ass	assign operator

Tableau 4.4 – Les notations du chapitre intraprocédural

À partir de la syntaxe du langage $\boldsymbol{\mathcal{L}_1}$ et de la première fonction de transformeur (Fonction 8), nous pouvons déduire une version plus générale décrite à la fonction 11 qui calcule l'ensemble des arcs *points-to* générés au niveau du *statement* s avec comme paramètres d'entrée un ensemble d'arcs P pour l'ensemble des instructions du langage $\boldsymbol{\mathcal{L}_1}$.

Fonction 11 : $T_{\mathcal{PT}}$

$T_{\mathcal{PT}} : \mathcal{S} \times \mathcal{PT} \to \mathcal{PT}$
$T_{\mathcal{PT}}[\![s]\!](P) \mapsto P'$
switch s **do**
  case *sequence*
    return $T_{\mathcal{PT}seq}[\![s]\!](P)$
  case *test*
    return $T_{\mathcal{PT}test}[\![s]\!](P)$
  case *call*
    return $T_{\mathcal{PT}call}[\![s]\!](P)$
  case *return*
    return $T_{\mathcal{PT}return}[\![s]\!](P)$
  case *exit*
    return $T_{\mathcal{PT}exit}[\![s]\!](P)$

Dans le reste du chapitre, nous détaillons, pour chaque type de *statement*, les arcs *points-to* générés. Certaines instructions comme l'instruction « exit » arrête l'exécution du programme et par conséquent l'analyse de pointeurs ne prend plus en compte les instructions qui peuvent apparaître par la suite. D'autres instructions sont plus difficiles à traiter et permettent de déterminer les dimensions de l'analyse [14]. Parmi ces instructions on trouve l'instruction *call* : selon la manière dont elle est traitée, elle va influencer la précision de l'analyse. Si l'effet de l'appel est pris en compte, on assurera l'aspect interprocédural de l'analyse, sinon on fournira des résultats moins précis.

4.5.2 Les points de séquence

Au niveau de l'évaluation d'une expression avec effets de bord, la norme C définit des points spécifiques, appelés points de séquence. Tous les effets de bord des évaluations précédentes ont été calculés et les effets de bord des évaluations ultérieures n'ont pas encore eu lieu.

Pour pouvoir calculer les arcs *points-to* en fonction du type de l'expression droite, on doit déterminer l'ordre d'évaluation de ses sous-expressions. Pour cela on doit prendre en compte les emplacements des points de séquence, spécifiés par la norme C99 :

- à la fin du premier opérande de l'opérateur ET logique, « `&&` »,
- à la fin du premier opérande de l'opérateur OU logique, « `||` »,
- à la fin du premier opérande de l'opérateur conditionnel, « `?:` »,
- à la fin de chaque opérande de l'opérateur virgule « `,` »,
- au point d'appel d'une fonction après évaluation de ses arguments.

Comme l'opérateur d'assignation n'est pas un point de séquence, on peut effectuer une descente récursive sur l'expression à droite et l'expression à gauche, par le biais de la fonction eta(e, P) pour la partie gauche et etv(e, P) pour la partie droite. En ce qui concerne l'opérateur virgule « `,` », on précise les différents comportements de cet opérateur qui a la plus faible priorité en C.

14. Les dimensions d'une analyse *points-to* sont citées dans le chapitre bibliographique 3.

```
int main (){
  int x1 = 0, y1 = 1, z1 = 2, t1 = 3, *x, *y, *z, *t;
  z = &z1;
  y = &y1;
  t = &t1;
  x = z, y, t;           // x pointe vers z1
  x = (z, y, t);         // x pointe vers t1
  x = z, z = t;          // x pointe vers z1, z pointe vers t1
  x = (z = &y1, z = t); // x et z pointent vers t1
  return 0;
}
```

Prog 4.7 – L'opérateur « virgule »

La différence entre la sixième et la septième affectation est que l'opérateur d'affectation « = » a une plus grande priorité que la virgule « , ». L'instruction `x = z, y, t` est équivalente à `(x = z), y, t`.

4.5.3 Rupture de contrôle : `exit()`

Le flot de contrôle peut être interrompu par un intrinsèque du langage comme `exit`. Il n'existe donc aucune relation *points-to* quand l'exécution d'`exit` semble se terminer. La postcondition n'est pas vide ; elle n'est pas définie. La valeur spéciale $\perp$ est utilisée.

4.5.4 Rupture de contrôle : `return`

Dans le cas où une valeur est retournée, un identificateur spécial, appelé *return value*, est introduit dans l'ensemble des identificateurs. L'intérêt de ce mécanisme très simple n'apparaît qu'au chapitre 6.

4.6 Les expressions en langage $\mathcal{L}_1$

L'expression est l'élément le plus fondamental du langage `C` et on peut en distinguer différentes catégories. Généralement on considère comme expression tout opérateur appliqué à des opérandes, par exemple `a+b+4` ou `p++` ou bien `s=5`. Comme le langage $\mathcal{L}_1$ est un sous-langage de `C` il supporte une grande variété de types et d'opérations ; tout peut être considéré comme expression y compris une constante ou une variable. Les expressions sont très développées et sophistiquées et permettent d'effectuer la programmation avec beaucoup de facilité et de flexibilité. Mais cet avantage peut être aussi une difficulté quand il s'agit d'analyser des expressions compliquées avec des effets de bord, par exemple `tab[*(p = &i)] = foo(*q)`. Plus généralement, une expression en langage `C` peut être :

- une référence ;
- un appel à une fonction utilisateur (chapitre 6) ;
- un appel à une fonction intrinsèque ; on y trouve, entre autres, l'affectation ;
- un cast (chapitre 5) ;
- ou encore un opérateur comme la virgule ou le point d'interrogation.

D'une façon plus formelle, les expressions sont définies selon la figure 4.6.

Compte tenu de la complexité des expressions, il n'est pas toujours possible de calculer les arcs *points-to*. Il faut, en premier lieu, identifier le type de l'expression analysée et ensuite lui appliquer un traitement spécifique. Dans la suite de chapitre, on traite dans chaque sous-section un type d'expression. Nous nous intéressons plus particulièrement à l'appel à l'opérateur

$<$ *expression* $>$	$E ::=$	$<$ *constant* $>$
		$\mid <$ *reference* $>$
		$\mid (<$ *expression* $>)$
		$\mid < u_op > <$ *expression* $>$
		$\mid <$ *expression* $> < b_op > <$ *expression* $>$
		$\mid <$ *sizeofexpression* $>$
		$\mid <$ *subscript* $>$
		$\mid <$ *call* $>$
		$\mid <$ *va_args* $>$
$<$ *subscript* $>$	$Sub ::=$	$[<$ *expression* $>^*]$
$<$ *sizeofexpression* $>$	$Soe ::=$	$<$ *type* $>$
		$\mid <$ *expression* $>$
$<$ *call* $>$	$C ::=$	$<$ *entity* $> (<$ *expression* $>^*)$
$<$ *v_args* $>$	$Varg ::=$	$(<$ *sizeofexpression* $>^*)$

FIGURE 4.6 – Les expressions en $\mathcal{L}_1$

intrinsèque d'affectation (déjà traité sous-section 4.4.1.3), car c'est la principale opération qui génère les arcs *points-to*. Au niveau de la représentation intermédiaire de PIPS, une expression est aussi un sous-type de *statement* (un *statement* est une suite d'instructions). On s'intéresse en particulier au cas de l'expression qui est modélisée, au niveau structure de données `NewGen`, selon le programme 4.8.

```
expression = syntax x normalized ;
syntax = reference + range + call + cast + sizeofexpression + subscript +
    application + va_arg:sizeofexpression* ;
```

Prog 4.8 – Structure de données d'une expression

À partir de cette disjonction, nous pouvons en déduire la fonction 12 qui est une surcharge du transfoeur pour *statement*.

Les sous-cas d'expression peuvent être eux aussi des expressions (par exemple « sizeofexpression ») ou avoir des arguments de type expression. Une descente récursive sur chaque sous-cas est nécessaire. Certains sous-cas vont être égaux à l'ensemble vide car ils ne peuvent pas générer des arcs *points-to*, comme par exemple une référence à un scalaire sans liste d'indices ou une constante. D'autres, comme $T_{\mathcal{PT}C}[\![C]\!]$, plus précisément l'appel à l'opérateur intrinsèque d'affectation, génèrent tous les arcs. Si nous définissons la fonction $T_{\mathcal{PT}}[\![E]\!]$, nous pourrons définir par la suite les fonctions $T_{\mathcal{PT}}$ pour les autres instructions et pour les différentes structures de contrôle.

4.6.1 Définition de l'analyse *points-to* pour le langage $\mathcal{L}_1$

Pour analyser le langage $\mathcal{L}_1$, les fonctions `eta` et `etv` sont étendus pour pouvoir traiter tous les cas d'expressions pouvant apparaître comme partie gauche ou droite d'une affectation. Comme il n'est pas utile de détailler tous les cas, nous définissons dans cette section les équations associées à quelques cas d'expression comme les références, les appels aux fonctions ou encore certains intrinsèques.

Fonction 12 : $T_{\mathcal{PT}}$ d'une expression

$T_{\mathcal{PT}} : \mathcal{E} \times \mathcal{PT} \rightarrow \mathcal{PT}$
$T_{\mathcal{PT}}[\![e]\!](P) \mapsto P'$
switch e **do**
  case R
    `Cas de la référence`
    return $T_{\mathcal{PT}R}[\![e]\!](P)$
  case C
    `Cas du call`
    return $T_{\mathcal{PT}C}[\![e]\!](P)$
  case Soe
    `Cas size of expression`
    return $T_{\mathcal{PT}Soe}[\![e]\!](P)$
  case Sub
    `Cas du subscript`
    return $T_{\mathcal{PT}Sub}[\![e]\!](P)$
  case $Varg$
    `Cas du v_args`
    return $T_{\mathcal{PT}Varg}[\![e]\!](P)$

4.6.1.1 Cas de la référence

Comme nous l'avons introduit précédemment, une référence est une variable avec une liste d'indices de type expression. C'est pour cette raison que nous effectuons une descente récursive sur les éventuelles expressions présentes au niveau des indices, ce qui permet aussi de prendre en compte les effets de bord. Par exemple une référence de type `a[*(p = q)]`, où `p` et `q` sont des pointeurs, génère une arc *points-to* p, s. Nous commençons par une référence à un scalaire, qui correspond à une variable du programme sans indices et qui, par conséquent, ne peut générer de arcs *points-to*. Nous faisons donc l'hypothèse suivante :

$$\mathcal{PT}[\![r]\!](P) = (P) \tag{4.14}$$

Dans le cas où la référence contient une liste d'indices, l'équation (4.14) devient :

$$T_{\mathcal{PT}}[\![r[e_0]...[e_n]]\!](P) \quad = \quad T_{\mathcal{PT}}[\![e_n]\!] \circ \cdots \circ T_{\mathcal{PT}}[\![e_0]\!] \circ T_{\mathcal{PT}}[\![r]\!](P) \tag{4.15}$$

En effet, en cas de liste d'indices, on commence par calculer l'ensemble de *points-to* en fonction de l'ensemble initial P. Par la suite, on calcule pour chaque expression en fonction de l'ensemble de *points-to* de l'expression précédente.

4.6.1.2 Cas de l'appel de fonction

Deux cas sont à distinguer : les appels de fonctions intrinsèques (*i.e.* les fonctions de la bibliothèque C) et les appels de fonctions définies par l'utilisateur. Dans chacun des cas, une descente récursive sur les arguments doit être effectuée au préalable. Comme il y a deux types d'appels qui nécessitent chacun un traitement différent on peut formuler les arcs *points-to* pour les appels sous la forme d'une disjonction à la fonction 13.

Ir et *UC* définissent respectivement les opérateurs intrinsèques et les appels aux fonctions utilisateur. Pour les opérateurs intrinsèques, on s'intéresse particulièrement à l'opérateur d'affec-

Fonction 13 : transformeur_pts_to_call

$T_{\mathcal{PT}}$: C $\times \mathcal{PT} \rightarrow \mathcal{PT}$
$T_{\mathcal{PT}}(c, P) \mapsto P'$
if *c*$\in$ *Ir* **then**
  return $T_{\mathcal{PT}}$Ir$[\![c]\!](P)$
else
  return $T_{\mathcal{PT}}$UC$[\![c]\!](P)$

tation qui génère les arcs *points-to* et aux opérateurs qui nécessitent une mise à jour sur ces arcs comme les opérateurs d'arithmétique sur pointeurs.

Appel à des fonctions utilisateur Les appels aux fonctions utilisateurs sont étudiés, dans un premier temps, dans le cadre de l'analyse intraprocédurale et dans un deuxième temps dans le cadre de l'analyse interprocédurale (chapitre 6). Dans le cadre de l'analyse intraprocédurale, nous nous contentons de rappeler la fonction sur les arguments de la fonction, qui sont de type expression. L'équation générale a la même forme que celle de la référence 4.15, le même calcul est appliqué.

Les arcs *points to* sont générés par effet de bord des expressions contenues au niveau des arguments. Pour chaque argument l'ensemble *points to* est calculé en fonction de l'ensemble de l'argument précédent.

4.6.1.3 Cas de l'arithmétique sur pointeurs

Un pointeur contient l'adresse d'une autre variable, en pratique sous la forme d'un entier (la taille dépend de l'architecture). On peut donc appliquer un certain nombre d'opérateurs arithmétiques sur les pointeurs. Les opérations arithmétiques valides sur les pointeurs sont [15] :

- l'addition d'un entier à un pointeur ; le résultat est un pointeur de même type que le pointeur de départ ;
- la soustraction d'un entier à un pointeur ; le résultat est un pointeur de même type que le pointeur de départ ;
- les pré- et post-incrémentations ou décrémentations qui sont des cas particuliers des deux items précédents ;
- la différence de deux pointeurs pointant tous deux vers le même type d'objet. Le résultat est un entier ;
- la somme de deux pointeurs n'est pas autorisée.

Un traitement spécifique est appliqué à l'opérateur « += » qui s'inspire de la norme du langage C. En effet, si le pointeur pointe vers le i^{me} élément d'un tableau, alors l'expression $(lhs)+N$ (respectivement, $N + (lhs)$) et $(lhs) - N$ pointe, respectivement vers le $i + n$ et le $i - n$ élément du même tableau. Un exemple d'arithmétique sur pointeur est donné par le programme 4.9 [16].

4.6.1.4 Cas de «sizeofexpression »

Le traitement de «sizeofexpression » se fait en appliquant l'équation *points to* au cas expression, noté par Soe_{expr} :

$$T_{\mathcal{PT}}[\![Soe]\!](P) = T_{\mathcal{PT}}[\![Soe_{expr}]\!](P) \qquad (4.16)$$

15. La section 6 de la norme C
16. Nous avons mis en place une propriété pour vérifier la conformité à la norme C lors des opérations arithmétique

```
int main(){
// Points To: none
   int n = 4, m = 3;

// Points To: none
   int a[n][m];

// Points To: none
   int (*p)[m] = a;

// Points To:
// p -> a[0]
   p += 1;

// Points To:
// p -> a[1]
   return 0;
}
```

Prog 4.9 – arcs *points-to* pour l'arithmétique des pointeurs

4.6.1.5 Cas de l'opérateur «subscript »

Comme cela a été détaillé à la figure 4.6, un opérateur de tableau est noté, dans la représentation interne de PIPS :

```
subscript = array:expression x indices:expression* ;
```

Pour calculer les arcs *points to*, on doit commencer par les calculer pour le champ $array$ qui est de type expression, puis utiliser cet ensemble pour calculer les *points to* au niveau de la liste d'indices, qui sont aussi de type expression, ceci nous ramène à l'équation (4.17) :

$$T_{\mathcal{PT}}[\![Sub(array, ind_0...ind_n]\!](P) \quad = T_{\mathcal{PT}}[\![ind_{k+1}]\!] \circ ... \circ T_{\mathcal{PT}}[\![ind_0]\!](T_{\mathcal{PT}}[\![array]\!](P))$$

4.6.1.6 Cas du « va_args »

Comme le montre la figure 4.6, « va_args » est une liste de « sizeofexpression ». On utilise donc l'équation (4.16) sur chaque élément de la liste.

$$T_{\mathcal{PT}}[\![Va(Soe_0, ..., Soe_n)]\!](P) \quad = \quad T_{\mathcal{PT}}[\![Soe_n]\!] \circ \cdots \circ T_{\mathcal{PT}}[\![Soe_0]\!](P) \tag{4.17}$$

4.6.1.7 L'algorithme pour l'instruction d'affectation

Le cas de l'affectation, pour le langage $\mathcal{L}_0$, a été traité dans la sous-section 4.4.1. Mais d'autres instructions peuvent mettre à jour ces arcs par effet de bord. Nous décrivons à présent l'algorithme qui effectue le traitement des affectations de pointeurs, avec une partie droite pouvant être n'importe quelle expression du langage $\mathcal{L}_1$.

Nous rappelons d'abord que l'analyse intraprocédurale d'Emami [Ema93] analyse les instructions de type affectation où la partie gauche est de type pointeur. Elle classe au préalable les affectations selon 15 modèles et associe à chaque modèle une règle de calcul des arcs *points-to*. Les arcs *points-to* ne sont calculés qu'au niveau de la pile et ne prennent en compte que les variables scalaires et les tableaux, les tableaux étant représentés par leur élément 0 ou par le reste de l'ensemble de leurs éléments.

Les déréférencements à niveaux multiples comme `***x` sont simplifiés par l'introduction de variables temporaires par une passe préalable. Finalement, l'analyse ne met pas à jour les arcs *points-to* en cas d'arithmétique sur pointeurs.

La fonction que nous avons définie est une généralisation de l'algorithme d'Emami [Ema93]. Elle calcule les arcs *points-to* au niveau des instructions d'affectation quand la valeur affectée est de type pointeur. Il n'y a aucune restriction syntaxique : le code source n'est pas modifié par une passe de simplification préalable. L'algorithme commence par calculer les effets de bords des expressions gauche et droite sur les arcs *points-to*, en respectant les points de séquence 4.5.2.

Puis, s'il s'agit d'un pointeur, la partie gauche est traduite, *lhs* ou *lvalue*, en un chemin d'accès mémoire constant (voir la section 4.3.3). Ensuite une disjonction selon le type de la partie droite, *rhs*, est faite afin de la transformer aussi en un ensemble de valeurs qui sont aussi des chemins d'accès mémoire constants.

En $\mathcal{L}_l 1$, la partie droite peut être soit :

- une référence à un scalaire ;
- la valeur « `NULL` » ou zéro ;
- une opération adresse, *address-of*, dénotée « `&` » ;
- une référence à un élément de tableau ;
- une opération d'arithmétique sur pointeur ;
- une des opérations suivantes : l'affectation « `=` », l'expression conditionnelle « `?` », la séquence « `,` », le et logique « `&&` », le ou logique « `||` », ou bien l'indexage « `[ ]` » ;
- un appel à une fonction intrinsèque comme « `malloc` » ou « `calloc` » ou encore « `free` ». ;
- une constante entière ;

Si les expressions droite et gauche ont pu être traduites toutes les deux en ensembles de chemins d'accès constants, L et R, on détermine alors les ensembles de base nécessaires au calcul des arcs *points-to*. Comme pour toute analyse de flot de données, ces ensembles sont Kill, Gen et Out, et ils sont calculés en fonction de L, R et In, la valeur courante des *points-to*.

La fonction 14 décrit d'une façon générale comment les arcs *points-to* sont calculés pour une instruction d'affectation avec comme terme gauche une expression de type pointeur. C'est le même algorithme que la fonction 8, mais augmenté par le calcul des effets de bord des parties gauche et droite.

Fonction 14 : $T_{\mathcal{PT}}$

$T_{\mathcal{PT}}(s : v, lhs : \mathcal{E}, rhs : \mathcal{E}, In : \mathcal{PT})$s;
$cur, incur, Kill, Gen, Out : \mathcal{PT}$;
$L, R : \mathcal{A}$;
$c : C$;
$nrhs : \mathcal{E}$;
`// Calcul des effets de bord de lhs et rhs ;`
$cur = T_{\mathcal{PT}}[\![rhs]\!](In)$;
$incur = T_{\mathcal{PT}}[\![lhs]\!](cur)$;
$Out = T_{\mathcal{PT}ass}[\![s]\!](incur)$;

4.6.2 Cas des branchements conditionnels «if then...else »

Cette sous-section est une extension de la section 4.4.1.5 qui traite le cas des branchements conditionnels du langage C en ignorant la sémantique des conditions. L'analyse des pointeurs peut être affinée quand la condition du test peut être évaluée grâce à l'information *points-to*. Quand la condition porte sur la valeur d'un pointeur, l'ensemble *points-to* permet de l'évaluer et par conséquent de déterminer le chemin d'exécution.

4.6.2.1 Les équations exactes

L'analyse des structures de contrôle comme les tests et les boucles se fait en analysant les composants de ces structures instruction par instruction, puis en identifiant et en appliquant l'équation qui lui est associée. Comme au niveau des branchements conditionnels, il n'est pas possible de savoir quelle branche sera exécutée, car l'évaluation de la condition n'est pas toujours possible statiquement, et nous devons analyser les deux branches. Le résultat globale pour le test est l'union des deux ensembles *points to* des branchements. L'opérateur de l'union sera détaillé par la suite (voir la fonction 20).

```
int main() {
  int *p, i = 1, j = 2;
  if(cond())
    p = &i;
  else
    p = &j;
  return 0;
}
```

Prog 4.10 – Exemple C avec une instruction `if then...else`

Prenons comme exemple le programme 4.10. La valeur de retour de `cond()` n'est pas connue au moment de l'analyse de pointeurs ; cela revient à exécuter la branche vraie du test `p = &i` ainsi que la branche faux `p = &j`. En conclusion, `p` pointe soit vers `i` soit vers `j`. Ceci se traduit par le résultat de l'analyse au niveau du programme 4.11.

```
int main(){
  int *p, i = 1, j = 2;
  if (cond())
    p = &i;
  else
    p = &j;
// Points To:
// p -> i
// p -> j
  return 0;
}
```

Prog 4.11 – Les arcs *points-to* pour une instruction `if then...else`

Ainsi on retrouve au point programme suivant le test, les arcs générés par l'analyse des deux branches plus les arcs qui existaient avant le test, et qui n'ont pas été mis à jour.

Nous rappelons l'équation déjà définie pour le langage $\mathcal{L}_0$:

$$T_{\mathcal{PT}}[\![\text{Test}(c, s_1, s_2)]\!](P) = T_{\mathcal{PT}}[\![s_1]\!](\text{ntp}(c, \text{true}, P)) \sqcup T_{\mathcal{PT}}[\![s_2]\!](\text{ntp}(c, \text{false}, P)) \tag{4.18}$$

Pour le langage $\mathcal{L}_0$ nous avons défini ntp comme la fonction identité sans effet sur le calcul des arcs *points-to*. Pour le langage $\mathcal{L}_1$ nous donnons une nouvelle définition de cette fonction qui va permettre de filtrer les arcs *points-to* et d'évaluer, quand c'est possible, la condition à l'entrée du test. Par exemple, si on a comme condition `p!=NULL` et que l'ensemble des arcs *points-to* contient comme seul arc partant de `p` l'arc `(p,NULL)`, on peut déduire que la condition est évaluée à faux et que l'instruction gardée par la condition ne sera pas exécutée.

4.7 Conclusion

Tout au long de ce chapitre, nous avons présenté les grandes lignes de l'analyse intraprocédurale que nous avons développée dans le compilateur PIPS [pip]. Après une introduction aux spécificités de cette analyse, la définition d'un premier langage simple, $\mathcal{L}_0$, nous a permis de détailler l'algorithme qui permet de traiter la principale instruction génératrice d'arcs *points-to*, l'affectation avec en partie gauche un pointeur, ainsi que la séquence et le test.

Nous avons augmenté le langage $\mathcal{L}_0$ avec de nouvelles instructions pour aboutir à un langage $\mathcal{L}_1$ pour lequel nous avons étendu ou défini les fonctions correspondantes.

Dans le prochain chapitre, nous allons formaliser l'abstraction des emplacements mémoire nécessaire pour le traitement des appels de fonctions, des boucles et des structures de données récursives allouées au niveau du tas. Ces instructions étendent le langage $\mathcal{L}_1$ à un nouveau langage $\mathcal{L}_2$ qui nécessite la prise en compte d'approximations sur les arcs *points-to*.

CHAPITRE 5

L'analyse intraprocédurale étendue

La première partie de l'analyse intraprocédurale des pointeurs a permis la définition des équations sémantiques pour un premier simple langage $\mathcal{L}_0$. Ces équations ont été par la suite étendues à un langage $\mathcal{L}_1$ plus riche, avec notamment l'introduction de l'élément fondamental du langage `C` qu'est l'expression. La deuxième partie, définie dans ce chapitre, étend l'analyse à un langage $\mathcal{L}_2$ qui est enrichi par rapport à $\mathcal{L}_1$ avec :

1. l'allocation dynamique ;
2. le cast ;
3. et le code non structuré, i.e. les graphes de flot de contrôle (« unstructured »).

Ces extensions couvrent pratiquement tout le langage `C` à l'exception des appels de procédures et nécessitent un nouvel effort d'abstraction de la mémoire pour modéliser des emplacements mémoire autres que ceux des variables du programme. Le chapitre comporte essentiellement trois parties :

1. une partie consacrée aux emplacements abstraits particuliers comme le tas ou le pointeur `NULL` (sections 5.1, 5.2) ;
2. une partie consacrée à la modélisation de la mémoire sous forme de treillis où sont détaillés les points suivants :
   (a) les sous-treillis qui composent le treillis CP, « constant path » (section 5.3) ;
   (b) les choix de conception du traillis CP (section 5.4)
   (c) le treillis PT des relations *points-to* et le traitement des casts (section 5.5) ;
3. et une partie consacrée à l'utilisation des treillis CP et PT pour résoudre le problème de calcul de points fixes lors du calcul des arcs *points-to* dans les boucles et les graphes de contrôle (sections 5.6 et 5.7).

5.1 Cas particuliers d'emplacements mémoire abstraits

L'analyse des pointeurs ne peut pas toujours donner un résultat exact sur l'adresse mémoire pointée. Un certain nombre d'adresses abstraites ont été définies pour représenter des approximations en s'appuyant sur les zones d'allocation des variables. Parmi ces zones définies figure l'élément « `anywhere` » qui représente n'importe quelle adresse définie d'un module ou de n'importe quel module, d'un type particulier ou de n'importe quel type. Les zones comptent aussi les abstractions STATIC, DYNAMIC, STACK, HEAP et FORMAL. Nous précisons par la suite des cas particuliers d'emplacements abstraits qui sont le pointeur `NULL`, les pointeurs non initialisés ainsi que les éléments de tableaux de pointeurs.

5.1.1 Modélisation de `NULL`

L'affectation de la valeur conventionnelle `NULL` doit être modélisée parce que cette valeur indique que le pointeur est bien initialisé, même si elle interdit son déréférencement. Seul le test est possible. Dans une analyse *points-to*, nous devons définir une cible (*sink*) particulière pour

modéliser cet emplacement particulier. Dans la section 5.3.1 où sera introduit le treillis des emplacements mémoire nous verrons à quel niveau se situe l'emplacement `NULL`. Pour le moment il faut retenir que c'est un emplacement unique par programme vers lequel pointent tous les pointeurs initialisés à `NULL` ou à zéro. Et que les tests dans les conditions comme `if(p==NULL)` peuvent être évalués grâce à la modélisation de la valeur `NULL` et à la fonction ntp (voir section 4.4.1.3).

5.1.2 Modélisation des pointeurs non initialisés

Nous pourrions convenir qu'un pointeur non-initialisé n'a pas d'image dans la relation *points-to*, mais ceci ne permettrait pas de modéliser l'impact des tests après lesquels le pointeur peut être initialisé ou non, et cela pourrait aussi créer des risques sur la nature de l'information, exacte ou non, lors de l'utilisation de l'information *points-to*. Il n'est donc pas possible d'utiliser une représentation implicite de la non-initialisation et la modélisation de cette cible particulière doit être effectuée. Dans notre spécification, nous désignons cette valeur par $undefined$[1]. Cette valeur particulière ne peut pas être la source d'un arc *points-to*.

5.1.3 Modélisation des tableaux de pointeurs

Au niveau des paramètres formels comme des variables en général, une des difficultés est de modéliser un tableau de pointeurs et les emplacement(s) vers lesquels les éléments pointent. Contrairement à ce qui a été fait pour $\mathcal{L}_0$ et $\mathcal{L}_1$, nous ne pouvons pas représenter tous les éléments de tableaux par des indices constants, `a[0], a[1]...`, ne serait-ce qu'à cause des tableaux de types dépendants comme `double a[n]` qui nécessiteraient un ensemble non borné d'indices constants.

Pour modéliser un ensemble d'éléments tableaux de pointeurs, nous avons choisi de le représenter par le nom du tableau indexé par « * » symbole que nous avons introduit dans le chapitre 4, plus précisément au niveau du domaine des constantes $\mathcal{C}$ (voir section 4.2.1). La constante « * » représente n'importe quel élément du tableau. Les choix de l'initialisation sont les suivants :

1. `q[*]-> _q_4` : n'importe quel élément du tableau `q` pointe vers une cible de type « stub » (voir section 4.3.2.2) qui représente un ou plusieurs emplacements abstraits ;
2. `q[*]-> _q_4[0]` : n'importe quel élément du tableau pointe vers le premier élément d'un tableau de « stubs » ;
3. `q[*]-> _q_4[*]` : n'importe quel élément du tableau pointe vers n'importe quel élément d'un tableau de « stubs » ;

Quand les indices d'un élément de tableau sont connus numériquement, ils peuvent être utilisés à condition que leur magnitude soit bornée. La borne est définie par une propriété de PIPS (voir section 5.7.3).

Ce problème d'initialisation est résolu en partie grâce à l'initialisation à la demande. En effet, ce n'est que lorsqu'un des éléments du tableau est lu ou écrit que l'initialisation est effectuée, et pas nécessairement vers un emplacement abstrait.

5.1.4 Modélisation des constantes entières

Moyennant un forçage de type, n'importe quelle constante entière peut être affectée à un pointeur. Ce mécanisme est utilisée dans les systèmes embarqués pour accéder à des registres matériels particuliers, mais n'est en général pas utilisé en calcul scientifique[2]. Notre analyse ne

1. La valeur $undefined$ est définie par la norme C sous le nom de $indeterminate$, l'adjectif $undefined$ étant réservé au comportement du programme. Au niveau de l'implémentation cette valeur est désignée par $undefined$.

2. Il est amusant de constater que le logiciel Newgen utilisé pour développer PIPS utilise des valeurs de pointeurs particulières pour signaler des pointeurs non-initialisés.

prend pas en compte cette possibilité quand la constante est non nulle. Une abstraction de plus haut niveau est renvoyée.

5.1.5 Conclusion

Cette section a permis de donner une première idée de la construction d'une abstraction de la mémoire avant d'effectuer une analyse de pointeurs. Elle aborde aussi le traitement de plusieurs cas particulier de pointeurs. Parmi ces cas figurent :

- la modélisation du pointeur `NULL` et son impact sur l'analyse de pointeurs ;
- la modélisation d'un pointeur non initialisé ;
- la modélisation des tableaux de pointeurs ;
- la modélisation des constantes entières différentes de 0.

5.2 Modélisation de l'allocation dynamique

La modélisation de l'allocation dynamique est une dimension très importante pour une analyse de pointeurs du langage `C`. En effet, le programmeur peut allouer un espace mémoire au niveau du tas à n'importe quel point du programme. Cet espace mémoire doit être par la suite libéré si nous voulons éviter une dégradation des perfermonces. Nous avons besoin de modéliser cette allocation ainsi que la libération pour avoir des informations précises et utiles sur les cibles de pointeurs qui peuvent être utilisées pour détecter des erreurs de programmation ou pour permettre de tester la légalité de transformations de programme.

5.2.1 Modélisation du tas

Si une allocation dynamique est effectuée via des routines comme `malloc` ou `calloc`, c'est au niveau du tas que se situe la cible du pointeur retourné et non pas au niveau de la pile. Nous avons besoin de savoir modéliser cet emplacement. Aussi, pour pouvoir fournir une analyse sensible au contexte, nous aurions besoin de recueillir des informations sur la pile d'appel, le module, la zone mémoire, l'état courant, et le *statement* auquel est rattaché le pointeur. Cette modélisation dépend aussi des besoins de l'application. Si cette dernière ne contient que peu d'allocation dynamique et que les principaux arcs *points-to* sont au niveau de la pile, alors la modélisation du tas sous la forme d'un unique emplacement abstrait peut suffire. Si, au contraire, plusieurs pointeurs sont alloués au niveau du tas, alors nous avons besoin de plus de précision sur le site d'appel. Nous avons conçu une modélisation paramétrée pour pouvoir s'adapter aux profils des applications[3]. D'une façon générale, un ensemble de cellules mémoire du tas, allouées à une même ligne d'une fonction, est modélisé sous la forme `module:*HEAP*_NumStatement`.

5.2.1.1 Les options de la modélisation du tas

Ces options, appelées aussi propriétés dans PIPS, sont *unique*, *insensitive* et *flow-sensitive*. Chacune d'elle permet l'activation d'une analyse *points-to* avec une modélisation différente du tas et par conséquent une analyse avec des résultats différents.

- *unique* : le tas est modélisé comme un seul tableau nommé `*HEAP*` ; cette représentation même si elle est concise permet néanmoins la parallélisation des boucles simples ;
- *insensitive* : c'est le cas insensible au contexte ; cela veut dire que il n'y a pas d'information sur la ligne d'allocation mais seulement sur la fonction où a lieu l'appel à `malloc` ;
- *flow-sensitive* : le tableau qui représente le tas est identifié par le numéro de l'instruction `malloc` ainsi que le nom de la fonction où a lieu l'appel.

3. Cette modélisation est paramétrée via des propriétés PIPS, qui permettent de contrôler son degré de précision.

L’option *context-sensitive* où toute la pile d’appel est sauvegardée n’est pas implantée.

5.2.1.2 Les problèmes de l’allocation dynamique

La modélisation de l’allocation dynamique présente beaucoup de difficultés pour l’analyse de pointeurs. Parmi celles-ci, nous citons :

1. la traduction du site d’appel quand la chaîne d’appel est profonde et qu’il faut dépiler la pile des appels ;
2. l’allocation dynamique d’un tableau de pointeurs ; vers quels emplacements faut-il faire pointer les éléments du tableau ? comment représenter ces derniers ?
3. l’allocation dynamique multiple qui peut se produire quand un appel à `malloc` est fait dans le corps d’une boucle,
4. ainsi que les appels récursifs.

5.2.2 Modélisation d’un appel à « `malloc` »

La cible créé par un appel à `malloc` correspond, par défaut, au premier élément d’un tableau de taille à priori inconnue. Le type de ce dernier est calculé en analysant avec une heuristique l’argument de `malloc` qui est codé sous la forme d’une expression. Généralement cette expression est de la forme :

1. `sizeof(type)` ;
2. `NB*sizeof(type)` où `NB` représente le nombre d’éléments du tableau dynamique que nous voulons créer.

Si la cellule allouée contient des pointeurs, directement ou indirectement, ils doivent être initialisés à la valeur $indeterminate$ d’après la norme C. Sont récursivement concernés, outre les pointeurs scalaires, les tableaux de pointeurs ou de structures et les structures.

La norme prévoit aussi que `malloc` puisse retourner `NULL` quand la réserve de mémoire dynamique est épuisée. Cette possibilité peu fréquente est signe, en général, d’une erreur de programmation n’a pas été implantée afin de ne pas ajouter du bruit dans l’information *points-to*[4].

5.2.3 Modélisation d’un appel à `free`

Après la modélisation du tas et des appels à `malloc`, nous nous intéressons naturellement à l’intrinsèque `free`. Cette fonction permet de libérer l’espace mémoire alloué précédemment par la fonction `malloc`. Elle prend comme argument un pointeur qui doit obligatoirement pointer vers une zone allouée via `malloc`. Prenons le programme 5.1 comme exemple.

```
int main()
{
  int *pi, *qi;
  pi = (int *) malloc(sizeof(int));
  qi = pi;
  free(pi); // S
  return 0;
}
```

Prog 5.1 – Exemple de libération de mémoire

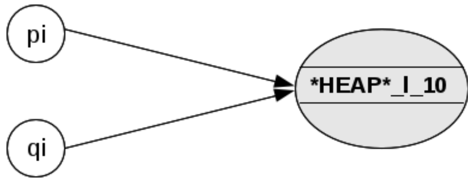


FIGURE 5.1 – Graphe des arcs *points-to* avant l’appel à `free`

4. Pour être strict, cette valeur de retour doit être implantée, mais elle pourrait être contrôlée par une propriété de PIPS.

En termes de graphe *points-to*, au niveau du point programme `S` avant l'appel à `free`, les arcs entre pointeurs sont représentés par le graphe 5.1.

L'effet de la fonction `free` sur les arcs *points-to* est de modifier la valeur du pointeur passé en argument sans l'écrire. La norme `C` spécifie que le pointeur ne pointe plus vers une zone mémoire valide et, par conséquent, une destination *indeterminate* doit lui être affectée implicitement. Ceci implique aussi qu'il ne peut plus être déréférencé.

Une autre conséquence à prendre en compte concerne les pointeurs en alias avec l'argument de `free`. Une fois la zone mémoire libérée, ces alias sont eux aussi réinitialisés et ne peuvent plus être déréférencés. Réinitialiser veut dire ici que les pointeurs se retrouvent assimilés à des pointeurs déclarés pour la première fois ; il faut les faire pointer vers l'élément du treillis représentant la valeur *indeterminate* de la norme, `undefined`.

Le résultat de l'appel à `free` sur l'état des arcs *points-to* est montré sur le programme 5.2.

5.2.3.1 Les équations de calcul d'arcs *points-to* après un `free`

Pour pouvoir supprimer les arcs vers la zone libérée après l'appel à `free`, nous devons établir des équations spécifiques pour ce cas particulier. Prenons en compte les équations précédemment définies qui permettent d'évaluer les adresses correspondantes à une expression e dans un contexte *points-to* In.

$$(L, \text{In}') = eta(e, \text{In}) \quad (5.1)$$
$$(R_2, \text{In}') = etv(e, \text{In}) \quad (5.2)$$

L'ensemble R_2 doit être restreint parce que tous les chemins constants ne peuvent pas faire l'objet d'un `free` : seules certaines adresses du tas et la valeur `NULL` sont autorisées par la norme. On calcule donc un nouvel ensemble R, où ne peuvent apparaître que `NULL`, les éléments du tas, les éléments du contexte formel puisqu'on n'en connaît pas l'allocation, et toutes les valeurs abstraites qui contiennent des éléments du tas.

$$R_1 = \{cp \in R_2 | cp \cap (\text{HEAP} \cup \text{FORMAL} \cup \{anywhere\}) \neq \emptyset\} \quad (5.3)$$

Il faut noter que les éléments de R, sont eux-mêmes des ensembles, puisqu'il s'agit d'adresses abstraites et non d'adresses concrètes.

Si R_1 ne contient qu'un élément et que cet élément est `NULL`, la norme spécifie que l'appel à `free` n'a aucun effet sur l'exécution et que l'analyse est terminée. Sinon, il est possible de retirer l'élément `NULL`pour obtenir R final :

$$R = R_1 - \{\texttt{NULL}\} \quad (5.4)$$

Si R est vide, une erreur d'exécution doit survenir. C'est par exemple le cas avec les séquences `{int i, *p = &i; free(p);}` et `{int *p; free(p);}`. Et l'analyse est terminée. Il reste donc à traiter les cas où les ensembles L et R sont non vides.

Comme l'appel à `free(e)` génère de nouveaux arcs *points-to* et en supprime d'autres, il est évident qu'il faut déterminer de nouvelles équations pour calculer les ensembles Gen_1, Gen_2, Kill_1 et Kill_2 correspondants. Maintenant, toutes les sources qui appartiennent à l'ensemble L doivent ou peuvent pointer vers l'emplacement abstrait `undefined`, d'où l'équation (5.5).

$$\text{Gen}_1(L) = \{(l, undefined) | l \in L\} \quad (5.5)$$

Dans le cas où le cardinal de L vaut 1 et où son unique élément l est un atome du treillis, *i.e.* un singleton, l'unique arc généré est un arc exact qui traduit le fait que selon la norme la valeur du pointeur l, $M[l]$, est $indeterminate$.

```c
int main() {
   int *pi, *qi;

// Points To:
// pi -> undefined , EXACT
// qi -> undefined , EXACT
   pi = (int *) malloc(sizeof(int));

// Points To:
// pi -> *HEAP*_l_10 , MAY
// qi -> undefined , MAY

   qi = pi;

// Points To:
// pi -> *HEAP*_l_10 , MAY
// qi -> *HEAP*_l_10 , MAY


   free(pi);
// Points To:
// pi -> undefined , EXACT
// qi -> undefined , MAY
// qi -> *HEAP*_l_10 , MAY

   return 0;
}
```

Prog 5.2 – Le résultat de l'analyse après l'appel à `free`

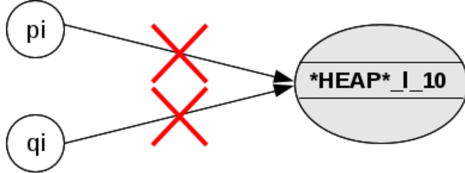


FIGURE 5.2 – Graphe des arcs *points-to* après l'appel à `free`

Comme l'appel à `free` a des effets sur les pointeurs en alias avec son argument, il est évident qu'il faut définir deux ensembles Gen_1 et Gen_2. L'équation de Gen_2 est la suivante :

$$\text{Gen}_2(R, \text{In}) = \{(source, undefined) | \exists r \in R \ \wedge \ \exists (source, r) \in \text{In}\} \tag{5.6}$$

Elle définit les nouveaux arcs impliquant les pointeurs qui pointaient vers la même zone au niveau du tas. Les sources sont récupérées et elles pointent maintenant vers `undefined`. Dans le cas où R est un singleton et où son unique élément est atomique, les arcs générés sont exacts. Mais comme les emplacements alloués au niveau du tas ne sont pas atomiques leur approximations vaut MAY. En effet, une allocation au niveau du tas représente un ou plusieurs emplacements abstraits. Et Comme la modélisation ne comporte que des chemins d'accès constants abstraits, ceci n'est possible que lorsque l'objet de l'appel à `free` est un élément du contexte formel.

Comme nous avons défini les deux ensembles Gen, nous définissons maintenant les ensembles Kill, c'est-à-dire les arcs *points-to* qui ne sont plus valables et dont les sources sont l'argument de la fonction `free`, ainsi que les pointeurs en alias avec cet argument. Le premier ensemble Kill correspond aux pointeurs définis par l'expression e. Il est défini par l'équation suivante :

$$\text{Kill}_1(L, \text{In}) = \{(l, r) \in \text{In} | l \in L \ \wedge \ |L| \ = 1 \ \wedge \text{atomic}(l)\} \tag{5.7}$$

où atomic est un prédicat sur CP qui est précisé sous-section 5.3.6 et qui signifie que l'adresse mémoire concrète est unique.

Quant aux alias de ces pointeurs, leur ensemble Kill est défini par l'équation (5.8).

$$\text{Kill}_2(R, \text{In}) = \{(l, r) \in \text{In} | r \in R \ \wedge \ |R| \ = 1 \ \wedge \text{atomic}(r)\} \tag{5.8}$$

Pour garantir la correction de l'analyse, tous les arcs pointant vers cette zone doivent être supprimés.

L'équation finale qui nous permet de calculer l'ensemble Out après un appel à la fonction `free` est :

$$\text{Out} = (\text{In} - \text{Kill}_1 - \text{Kill}_2) \cup \text{Gen}_1 \cup \text{Gen}_2 \tag{5.9}$$

La libération d'une zone peut générer des pointeurs de valeur *indeterminate*, appelés *dangling pointers*, et des fuites mémoire.

5.2.3.2 L'algorithme de traitement de l'appel à `free`

Après avoir défini les équations liées au traitement de l'appel à `free`, nous présentons ci-dessous l'algorithme qui permet de traiter cette instruction de libération de mémoire.

Fonction 15 : $points_to_free$

$points_to_free : \mathcal{E} \times \mathcal{PT} \rightarrow \mathcal{PT}$
$points_to_free(e : \mathcal{E}, \text{In} : \mathcal{PT})$
$(L, \text{In}') = eta(e, \text{In})$
$(R_2, \text{In}') = etv(e, \text{In})$
$R_1 = \{cp \in R_2 | cp \cap (\text{HEAP} \cup \text{FORMAL} \cup \{anywhere\}) \neq \emptyset\}$
$R = R_1 - \{\text{NULL}\}$
$\text{Gen}_1(L) = \{(l, undefined) | l \in L\}$
$\text{Gen}_2(R, \text{In}) = \{(source, undefined) | \exists r \in R \ \wedge \ \exists (source, r) \in \text{In}\}$
$\text{Kill}_1(L, \text{In}) = \{(l, r) \in \text{In} | l \in L \ \wedge \ |L| \ = 1 \ \wedge \text{atomic}(l)\}$
$\text{Kill}_2(R, \text{In}) = \{(l, r) \in \text{In} | r \in R \ \wedge \ |R| \ = 1 \ \wedge \text{atomic}(r)\}$
$\text{Out} = (\text{In} - \text{Kill}_1 - \text{Kill}_2) \cup \text{Gen}_1 \cup \text{Gen}_2$

5.2.4 Détection des erreurs liées à l'allocation mémoire

Une mauvaise manipulation de la mémoire dynamique peut être dangereuse pour le programme. Le programmeur peut effectuer plusieurs allocations, par exemple un appel à `malloc` dans une boucle, et dépasser la taille du tas. L'analyse de pointeurs peut détecter des erreurs liées à la mémoire dynamique, comme la présence de pointeurs pendants et les fuites mémoires.

5.2.4.1 Les pointeurs pendants

La liberté qu'offre le langage `C` en matière de gestion de mémoire n'a pas que des avantages. En effet, l'opération de libération peut être dangereuse si par exemple un pointeur libéré implicitement est réutilisé plus loin dans le programme. Le pointeur est dit alors pointeur pendant [5]. Comme le montre la figure 5.3, il faudrait prendre en compte le fait que le pointeur `p1` pointe vers une zone libérée ainsi que le pointeur `p2` qui est en alias avec lui.
Notre analyse nous permet de détecter les pointeurs qui peuvent être devenus pendants *via* l'équation (5.10) :

$$DP(\text{Gen}_2(R, \text{In})) = \{r | \exists (r, l) \in \text{Gen}_2(R, \text{In})\} \tag{5.10}$$

Il y a une erreur de pointeur pendant dans un programme si des pointeurs pointent à coup sûr vers une zone mémoire libérée unique et connue exactement. Il faut donc utiliser une version beaucoup plus restrictive de la définition de Gen_2.

$$DP(L, \text{In}) = \{r | \exists ! \ (r, l) \in \text{In} \ \wedge \ L \ = \{l\} \ \wedge \ \text{atomic}(l)\} \tag{5.11}$$

5. « Dangling pointer » en anglais.

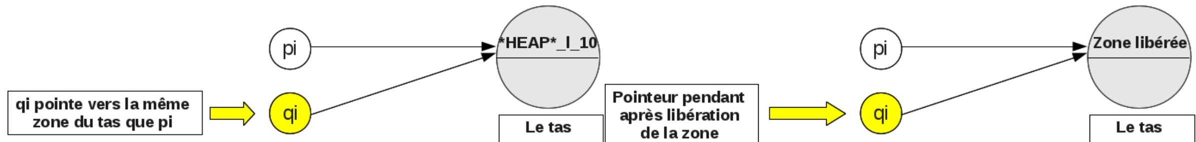


FIGURE 5.3 – État du tas avant l'appel à `free`

FIGURE 5.4 – État du tas après l'appel à `free`

Chaque fois que $DP(L, \text{In})$ est non vide, ce qui implique que l est un élément du contexte formel et non directement un élément du tas, un avertissement est affiché par l'analyseur.

5.2.4.2 Les fuites mémoire

Une fuite mémoire est le résultat de la perte de la référence sur une zone mémoire. L'espace mémoire n'est plus référencé ; il ne peut plus être accédé ni libéré. Ceci peut provoquer une augmentation considérable de l'espace mémoire utilisé jusqu'à dépasser la zone disponible et provoquer l'arrêt du programme. Ce type d'erreur est difficile à détecter et à corriger et s'appelle fuite mémoire [6]. Prenons comme exemple le programme 5.3.

```c
#include<stdlib.h>
#include<stdio.h>
 struct foo {
  struct foo* next;
  int val;
};







typedef struct foo MyStruct;
int main()
{
  MyStruct *root;
  root = malloc(sizeof(MyStruct));
  root->next = malloc(sizeof(MyStruct));
  free(root);
  return 0;
}
```

Prog 5.3 – Programme C contenant une fuite mémoire

```c
int main()
{
   MyStruct *root;
// Points To:
// root -> undefined , EXACT
   root = malloc(sizeof(MyStruct));

// Points To:
// root -> *HEAP*_l_11 , MAY
   root->next = malloc(sizeof(MyStruct));

// Points To:
// *HEAP*_l_11.next -> *HEAP*_l_12 , MAY
// root -> *HEAP*_l_11 , MAY
   free(root);

// Points To:
// root -> undefined , EXACT
   return 0;
}
```

Prog 5.4 – Les arcs *points-to* pour le programme contenant la fuite mémoire

Pour mieux comprendre cette erreur, nous avons illustré l'état du tas, avant l'appel à `free` pour libérer la zone pointé par `root`, par la figure 5.5. La figure 5.6 montre qu'après la libération du pointeur `root` la zone `*HEAP*_l_12` pointée par `*HEAP*_l_11.next` n'est plus référencée. Quand l'instruction `free` est traitée, nous sommes capable de détecter cette erreur. En effet, après le calcul de R (voir équation (5.1)) nous devons vérifier s'il existe des arcs *points-to* (source, destination) dont la source appartient à R et dont la destination est dans le tas. L'existence de cet arc indique l'existence d'une possible fuite mémoire. Pour cela nous calculons l'ensemble PML (« potential memory leaks ») :

$$PML(R, \text{In}) = \{l | \exists r \in R \ \wedge \ l \in apply_pt(r, \text{In})\} \tag{5.12}$$

6. « memory leak » en anglais.

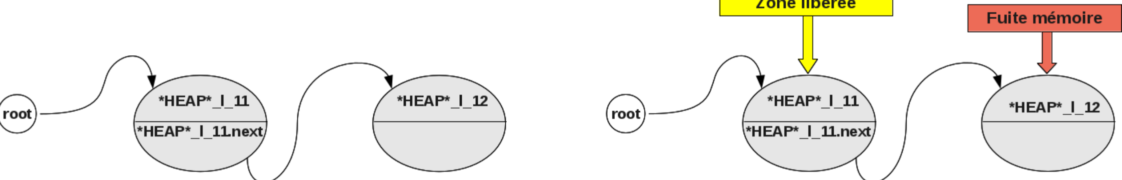


FIGURE 5.5 – État du tas pour le programme 5.4

FIGURE 5.6 – État du tas pour le programme 5.4 après l'appel à `free`

avec $apply_pt$ définie à la sous-section 5.5.1. Puis, à partir de cet ensemble, nous pouvons détecter les fuites mémoire si une de ces cellules est devenue à coup sûr inatteignable, avec $P = PML(R, \text{In})$:

$$ML(P, \text{In}) = \{m | m \in P \ \wedge \ \text{HEAP}(m) \ \wedge \ \neg reachable_p(m, \text{In})\} \tag{5.13}$$

Un avertissement est adressé à l'utilisateur si cette erreur est détectée et l'arc *points-to* en question est supprimé.

5.2.5 Conclusion

L'allocation dynamique est une dimension très importante de l'analyse des pointeurs, surtout si l'application scientifique analysée utilise essentiellement des tableaux dynamiques alloués au niveau du tas. Notre modélisation du tas est paramétrable et laisse le choix du degré de précision à l'utilisateur, mais elle est fondamentalement imprécise parce qu'incapable de désigner de manière sûre une cellule particulière du tas. Des équations permettant de détecter certaines erreurs liées à l'allocation dynamique ont néanmoins été formulées. Elles permettent à l'utilisateur de se rendre compte d'erreurs de programmation difficiles à détecter.

5.3 Abstraction de l'ensemble des adresses sous forme de treillis

La source et la destination d'un arc *points-to* sont des chemins d'accès mémoire constants (voir la section 4.2.1.3) aussi appelés cellules dans l'implantation, le chemin étant une dénotation et la cellule l'ensemble de cases mémoire correspondant. Ces chemins représentent des références généralisées, fondées sur les variables du programme ainsi que sur les emplacements abstraits que nous avons ajoutés pour assurer la finitude de la représentation. Le nombre de ces emplacements peut vite devenir très grand. C'est pour cette raison que nous avons choisi de le modéliser sous la forme d'un treillis(figure 5.7) afin de pouvoir comparer les éléments, trouver des approximations supérieures et assurer la terminaison de l'analyse.

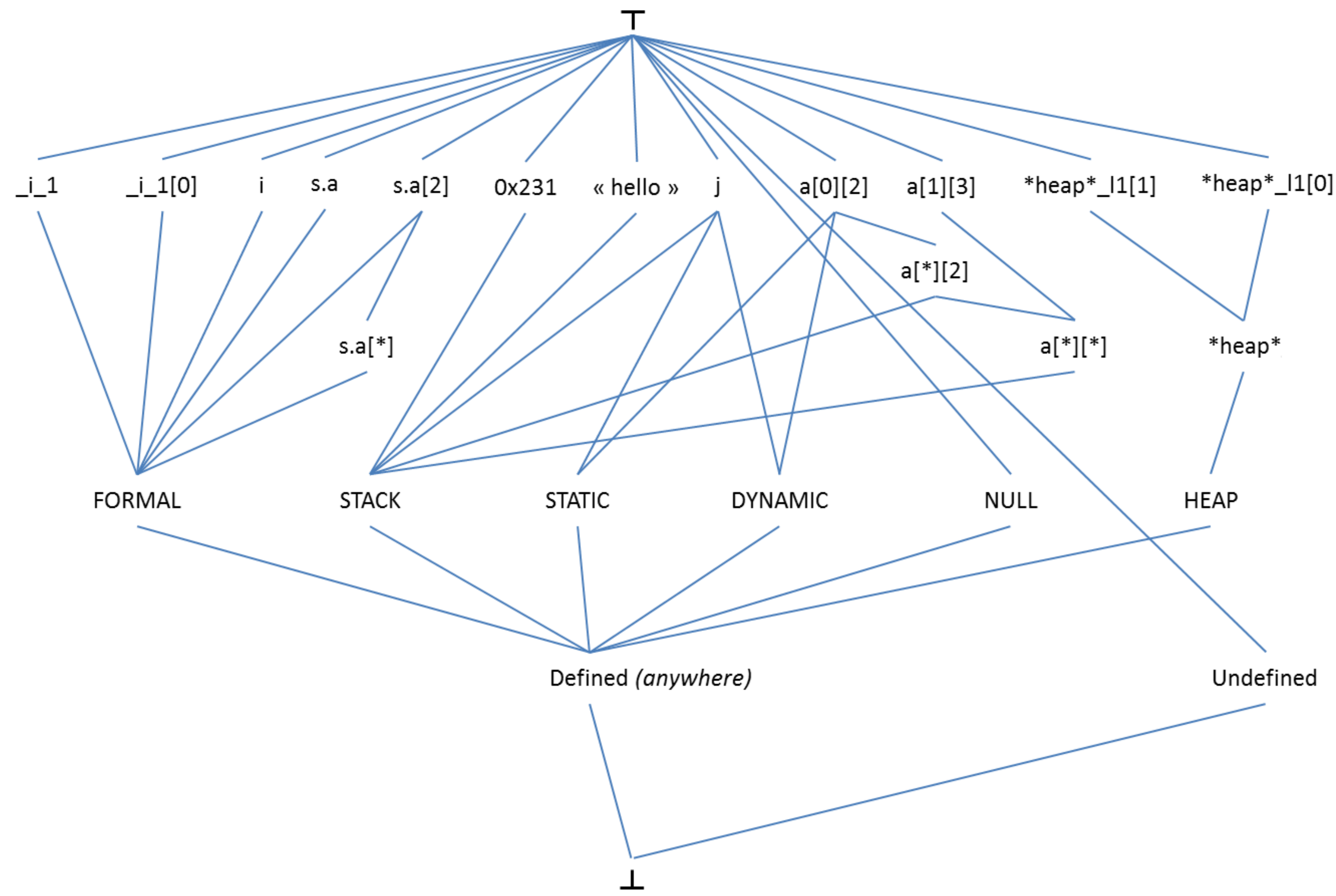


FIGURE 5.7 – Treillis des emplacements mémoire

5.3.1 Opérateurs et structure du treillis des chemins constants CP

L'écriture d'équations de flot de données pour traiter les affectations de pointeurs a fait apparaître divers opérateurs qui n'ont pu être définis de manière approfondie lors du chapitre de l'analyse intraprocédurale simple. Ce chapitre vise à en fournir des définitions précises, justifiées par l'analyse cas par cas.

Nous rappelons que pour définir la cible ou la source d'un arc *points-to* nous spécifions son nom, le module qui définit sa portée, son type et finalement la liste d'indices qu'elle contient. En conséquence le treillis CP est défini comme le produit cartésien de quatre sous-treillis :

$$\boxed{CP = Module \times Name \times Type \times Vref}$$

Le treillis CP[7] est formé de quatre composants qu'il convient de définir séparément pour réduire la complexité de l'analyse. Chaque composant lui aussi correspond à un treillis, pour lequel nous définissons le même ensemble d'opérateurs découlant des équations du chapitre 4, ce qui explique leurs noms inhabituels pour les opérations et la relation d'ordre. L'application des opérateurs sur les quatre composants et la combinaison de leur résultats constitue le résultat pour le treillis CP.

Ainsi pour deux éléments quelconques de CP :

- $cp_1 = (m_1, n_1, t_1, v_1)$
- $cp_2 = (m_2, n_2, t_2, v_2)$

nous décomposons les opérateurs max, qui correspond à sup et qui renvoie le plus petit élément de CP contenant toutes les cellules mémoire de cp_1 ou de cp_2, et $inter$, qui correspond à inf et qui renvoie le plus grand élément de CP contenant l'ensemble des cellules mémoires de cp_1 et de cp_2, *i.e.* leur intersection approchée, ainsi que le prédicat booléen $killable$, qui correspond à la relation d'ordre du treillis, à savoir l'inclusion.

Ainsi, nous définissions deux opérateurs et deux prédicats sur le treillis CP en combinant ceux des sous-treillis de la manière suivante :

$$\begin{aligned}
max_{cp}(cp_1, cp_2) &= (max_M(m_1, m_2), max_N(n_1, n_1), max_T(t_1, t_2), max_{Vref}(v_1, v_2)) \\
inter(cp_1, cp_2) &= (inter_M(m_1, m_2), inter_N(n_1, n_2), inter_T(t_1, t_2), inter_{Vref}(v_1, v_2)) \\
killable(cp_1, cp_2) &= (killable_M(m_1, m_2) \wedge killable_N(n_1, n_2) \wedge killable_T(t_1, t_2) \wedge killable_{Vref}(v_1, v_2)) \\
atomic(cp) &
\end{aligned}$$

Ces divers treillis sont complets. Ils ont tous un plus grand et un plus petit éléments, notés par défaut $\top$ et $\bot$.

Ces deux opérateurs et prédicats sont utilisés pour le calcul des équations de flot de données et ils sont détaillés pour chaque sous-treillis. L'opérateur max n'est pas utilisé fréquemment parce que nous avons choisi de conserver plus d'information en utilisant des listes finies au niveau du treillis CP et plus encore au niveau du treillis des relations *points-to*, $\mathcal{PT}$. Il assure néanmoins dans certains cas la finitude des listes. L'opérateur $inter$ n'est pas utilisé par notre analyse qui recherche des sur-approximations. Par contre il est indispensable aux analyses clients, à commencer par le calcul des « use-def chains ». Il permet de décider si un chemin constant est en conflit avec un autre ou non ; ceci se traduit par une intersection non vide d'où l'intérêt d'utiliser le treillis CP.

Le prédicat $killable$ est utilisé fréquemment pour décider si une affectation peut ou doit détruire un arc préexistant. Il faut vérifier avec l'opérateur $killable$ que la cellule cp_2 peut « tuer » la cellule cp_1, c'est-à-dire que $cp_1 \subseteq cp_2$, *i.e.* $killable(cp_1, cp_2)$. Pour cela nous appelons l'opérateur sur la source et la destination. Chacune d'entre elles est un chemin d'accès mémoire constant composé des quatre treillis cité ci-dessus. Le prédicat $killable$ du chemin constant est un « et logique » des $killable$ des quatre sous-treillis.

7. CP = *constant path*, *e.g.* `p`, `a[1]` ou `s.next`.

En dernier lieu nous définissons un prédicat sur l'atomicité d'un chemin constant cp qui ne nécessite pas la décomposition en quatre sous-treillis. Ce prédicat est indispensable pour distinguer les arcs qui existent certainement des arcs qui peuvent exister. Pour certains de ces opérateurs et prédicats cités ci-dessus nous allons définir des versions EXACT et des versions MAY. Ceci s'explique par l'incertitude introduite par certains éléments du treillis CP comme « * » qui représente l'un des éléments d'un tableau sans savoir exactement lequel.

5.3.2 Le sous-treillis $Module$

Le sous-treillis $Module$ représente le nom du module qui est en cours d'analyse, il est illustré par la figure 5.8.

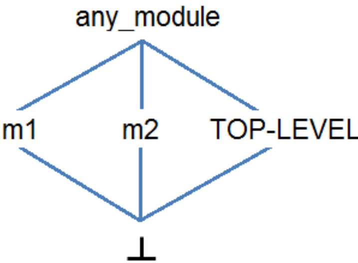


FIGURE 5.8 – treillis Module

La dimension $Module$ est aussi représentée au niveau de la représentation interne de PIPS à l'intérieur du nom d'une «entity» :

```
tabulated entity = name:string x type x storage x initial:value ;
```

La chaîne `name` se décompose en un nom de module qui est le nom de la fonction où est déclarée l'entité, et un nom local, qui est le nom de la variable correspondant à l'entité. Le champ `name` doit être unique, ce qui complique l'implantation par rapport à la présentation théorique sous forme d'un produit cartésien. Par exemple, toute modification de type implique une modification de nom. Les mêmes opérateurs sont appliqués sur cette composante, dont l'élément $\top$ est représenté dans notre analyse par la chaîne `any_module` et un $\bot$, avec une relation d'ordre partiel qui correspond à l'ordre partiel sur des constantes. Plus concrètement, l'ensemble des module peut être défini par :

$$Module = Vmodule \cup \{TOP-LEVEL\} \cup \{any_module\}$$

avec :

$$Vmodule = [a-z_A-Z][a-zA-Z_0-9]^*$$

Nous définissons ci-dessous les opérateurs appliqués au sous-treillis $Module$.

5.3.2.1 L'opérateur max pour le sous-treillis $Module$

L'opérateur max_M permet de déterminer entre deux modules lequel est le plus grand emplacement abstrait. Il est défini comme suit :

$$\begin{aligned} Module \times Module &\rightarrow Module \\ max_M(m_1, m_2) &= \text{si } m_1 \in Vmodule \wedge m_1 = m_2 \text{ alors } m_1 \text{ sinon } any_module \end{aligned} \tag{5.14}$$

Quand nous voulons déterminer le maximum entre deux modules différents c'est le sommet du treillis, l'élément any_module, qui est renvoyé.

5.3.2.2 L'opérateur intersection au niveau du treillis Module

Nous définissons l'opérateur $inter_M$ par le tableau 5.1.

$\mathbf{inter_M(m_1, m_2)}$	$m1 \neq any_module$	any_module
$m_2 \neq any_module$	(5.15)	$\emptyset$
any_module	$\emptyset$	any_module

Tableau 5.1 – Opérateur $inter_M$

avec :

$$Module \times Module \to Module \ : \ \text{si } m_1 = m_2 \text{ alors } m_1 \text{ sinon } \emptyset \qquad (5.15)$$

L'intersection de deux modules différents est l'ensemble vide.

5.3.2.3 Le prédicat $killable$ au niveau Module

Le prédicat booléen $killable$ est utilisé pour définir l'ensemble Kill défini dans la section 4.4.1.2. C'est un prédicat de type :

$$Module \ \times \ Module \ \to \ Bool$$

L'ordre des deux opérandes n'a pas d'importance. Deux versions sont définies, la version « MAY » :

$$killable_{MAYM}(m_1, m_2) = \text{si}((m_1 = any_module) \vee (m_2 = any_module)) \text{ alors true sinon } m_1 = m_2$$

Si les deux modules sont égaux alors le module m_1 (respectivement le module m_2) « tue » le module m_2 (respectivement le module m_1). Si l'un des modules est égal au somment du treillis (condition vérifiée par le prédicat any_module_p) alors $killable_{MAYM}$ renvoie vrai. La version exacte est définie comme suit :

$$killable_{EXACTM}(m_1, m_2) = \text{si}((m_1 = any_module) \vee (m_2 = any_module)) \text{ alors false sinon } m_1 = m_2$$

qui se calcule de la même que la version « MAY » sauf dans le cas où l'un des modules est égal au somment du treillis. Elle renvoie alors faux.

5.3.3 Le sous-treillis des noms de variables $Name$

Le treillis $Name$ et les emplacements abstraits introduits dans la section 5.1 sont rappelés dans la figure 5.9. Il comporte tout d'abord les noms de variables comme v_1 et v_2 qui appartiennent à l'ensemble $VName$ des noms de variables conformes à la norme `C`. Il s'y ajoute la constante `NULL`et un ensemble de variables implicites utilisées pour modéliser le tas. Cet ensemble est appelé HEAP et est décrit dans la section 5.2.1.

Plusieurs éléments extrêmes sont ajoutés pour compléter le treillis, $undefined$ qui représente les pointeurs non-initialisés et qui s'oppose à $defined$ aussi appelé $anywhere$, le minimum du treillis, $\perp$, et $\top$ qui constitue le sommet du treillis, . Plusieurs couches intermédiaires permettent de retarder la suppression d'information quand plusieurs chemins constants doivent être combinés.

Elle comprend les adresses abstraites FORMAL(F), DYNAMIC(D), STATIC(ST), STACK(SK) et HEAP(H) qui regroupent chacune l'ensemble des variables déclarées dans ces zones pour un module donné. Comme PIPS doit traiter l'aliasing statique pour Fortran, une distinction est faite entre les variables de types statiques qui sont allouées dans DYNAMIC et pour lesquelles on peut calculer statiquement une adresse ou un décalage par rapport au début de la zone allouée, et variables de types dépendants qui sont allouées dans STACK. Dans le premier cas, une adresse d'allocation peut être calculée, alors que c'est impossible dans le second.

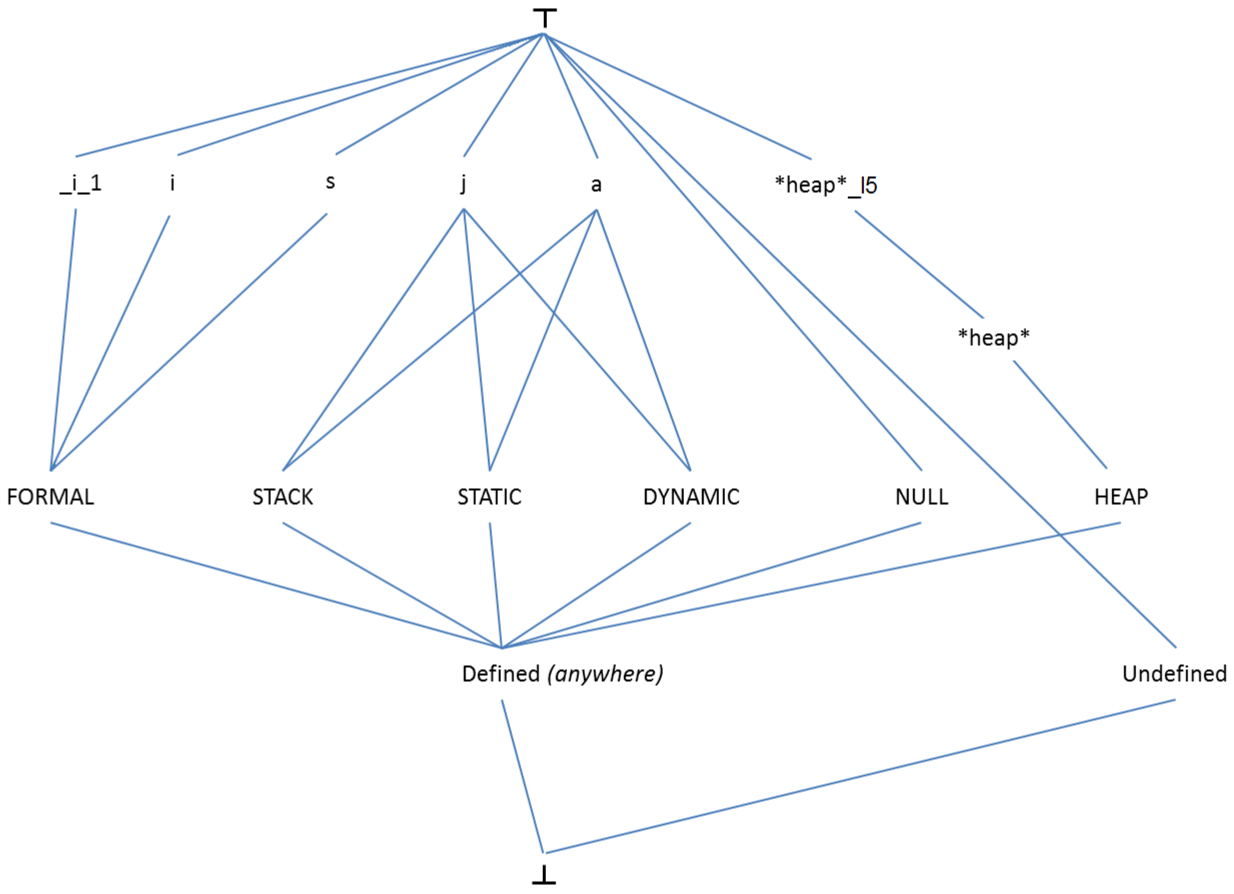


FIGURE 5.9 – Le treillis Name

Les éléments de HEAP sont des adresses abstraites. Chacun d'eux peut représenter plusieurs adresses du tas. En résumé, nous utilisons les ensembles et les singletons suivants :

- $Name = VName \cup \{undefined\} \cup \{anywhere\} \cup \{\texttt{NULL}\} \cup$ HEAP $\cup$ STACK $\cup$ STATIC $\cup$ DYNAMIC $\cup$ FORMAL ;
- $VName = [A-Za-z][A-Za-z_0-9]^*$: identificateurs ou noms de variables ;
- HEAP : modélisation du tas pouvant prendre en compte le module en cours ainsi que le numéro de ligne d'appel de `malloc`.

Ce treillis $Name$ (figure 5.9) admet comme :

- borne inférieure l'emplacement mémoire abstrait $\perp$, qui est différent de $undefined$ qui désigne en particulier un pointeur non initialisé ;
- borne supérieure $\top$, différent de $anywhere$ qui désigne une incohérence.

5.3.3.1 L'opérateur max sur la composante $Name$

Nous définissons l'opérateur max_N pour le treillis $Name$ par le tableau 5.2. Il dénote le plus petit emplacement mémoire contenant tous les emplacements contenus par chacun des deux arguments. L'ordre des arguments est donc sans importance. Les symboles $*heap*_1$ et $*heap*_2$ représentent des éléments de HEAP.

$\mathbf{max_N(n_1, n_2)}$	$\bot$	n_1	$*heap*_1$	NULL	SK	ST	D	H	F	$\top$
$\bot$	$\bot$	n_1	$*heap*_1$	NULL	SK	ST	D	H	F	$\top$
n_2	n_2	(5.16)	$\top$	n_2	(5.17)	(5.17)	(5.17)	(5.17)	(5.17)	$\top$
$*heap*_2$	$*heap*_2$	$\top$	(5.18)	$*heap*_2$	$\top$	$\top$	$\top$	H	$\top$	$\top$
NULL	NULL	(5.16)	$*heap*_1$	NULL	SK	ST	D	H	$\top$	$\top$
SK	SK	(5.17)	SK	SK	SK	$\top$	$\top$	$\top$	$\top$	$\top$
ST	ST	(5.17)	ST	ST	$\top$	ST	$\top$	$\top$	$\top$	$\top$
D	D	(5.17)	D	D	$\top$	$\top$	D	$\top$	$\top$	$\top$
H	H	(5.17)	H	H	$\top$	$\top$	$\top$	H	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

Tableau 5.2 – L'opérateur max_N

avec :

$$\text{Si } n_1 = n_2 \text{ alors } n_1 \text{ Sinon } max_N(v_to_al(n_1), v_to_al(n_2)) \quad (5.16)$$

$$max_N(v_to_al(n_1), v) \quad (5.17)$$

$$\text{Si } n_1 = n_2 \text{ alors } n1 \text{ Sinon HEAP} \quad (5.18)$$

La fonction $v_to_al(n)$ qui renvoie le plus petit emplacement abstrait incluant son argument c'est-à-dire l'emplacement abstrait où la variable n est allouée : SK, ST, D, H ou F. C'est une simplification d'écriture puisque la zone d'allocation ne dépend pas que de n, mais aussi de l'entité définie par m, le nom de module, et n.

5.3.3.2 L'opérateur intersection au niveau du treillis $Name$

Nous définissons l'opérateur $inter_N$ par le tableau 5.3.

$\mathbf{inter_N(n_1, n_2)}$	$n1 \neq \top$	$\top$
$n_2 \neq \top$	(5.19)	n_2

Tableau 5.3 – Opérateur $inter_N$

avec

$$Name \times Name \rightarrow Name \ : \ \text{si } n_1 = n_2 \text{ alors } n_1 \text{ sinon } max_N(n_1, n_2) \quad (5.19)$$

L'intersection de deux noms différents est l'ensemble vide.

5.3.3.3 Le prédicat $killable$ sur la composante $Name$

Le prédicat booléen $killable_{MAYN}$ au niveau du treillis $Name$ est défini par le tableau 5.4. L'ordre des arguments est important ; n_2 peut tuer n_1 mais pas l'inverse. Quand les deux opérandes sont des variables du programme une comparaison des noms suffit pour décider. Quand l'un des opérandes est une variable du programme et l'autre une des zones mémoire du treillis (STATIC, DYNAMIC, STACK ou FORMAL) la première est transformée en une zone mémoire via la fonction $v_to_al()$ (variable_to_abstract_location()).

Quand l'opérande n_1 n'est pas un lhs valide, c'est-à-dire ne peut être une source ou une destination d'un arc *points-to*, une exception est levée (notée E dans le tableau 5.4). En effet, nous ne pouvons pas avoir $\bot$ ou NULL ou $undefined$ comme valeur de terme gauche d'une affectation.

$\mathbf{killable_{MAYN}(n_1, n_2)}$	$\bot$	n_1	$*heap*_1$	NULL	SK	ST	D	H	F	$\top$
$\bot$	E	E	E	E	E	E	E	E	E	E
n_2	E	$n_1 = n_2$	T	E	(5.20)	(5.20)	(5.20)	F	(5.20)	T
$*heap*_2$	E	F	$n_1 = n_2$	E	F	F	F	T	F	T
NULL	E	E	E	E	E	E	E	E	E	T
SK	E	(5.21)	F	E	T	F	F	F	F	T
ST	E	(5.21)	F	E	F	T	F	F	F	T
D	E	(5.21)	F	E	F	F	T	F	F	T
H	E	F	T	E	F	F	F	T	F	T
F	E	(5.21)	F	E	F	F	F	F	T	T
$\top$	E	E	E	E	E	E	E	E	E	E

Tableau 5.4 – L'opérateur $killable_{MAYN}$

avec :

$$v_2 = v_to_al(n_2) \quad (5.20)$$
$$killable_{MAYN}(n_1, v_2)$$

et :

$$v_1 = v_to_al(n_1) \quad (5.21)$$
$$killable_{MAYN}(v_1, n_2)$$

La version exacte de ce prédicat est définie par le tableau 5.5 où nous faisons à nouveau appel à la fonction v_to_al pour permettre qu'une écriture sur une adresse abstraite puisse tuer une information portant sur une variable allouée à cette adresse abstraite et réciproquement.

$\mathbf{killable_{EXACTN}(n_1, n_2)}$	$\bot$	n_1	$*heap*_1$	NULL	SK	ST	D	H	F	$\top$
$\bot$	E	E	E	E	E	E	E	E	E	E
n_2	E	$n_1 = n_2$	T	E	(5.22)	(5.22)	(5.22)	(5.22)	(5.22)	F
$*heap*_2$	E	F	$n_1 = n_2$	E	F	F	F	T	F	F
NULL	E	E	E	E	E	E	E	E	E	E
SK	E	(5.24)	F	E	T	F	F	F	F	F
ST	E	(5.24)	F	E	F	T	F	F	F	F
D	E	(5.24)	F	E	F	F	T	F	F	F
H	E	(5.24)	T	E	F	F	F	T	F	F
F	E	(5.24)	F	E	F	F	F	F	T	F
$\top$	E	E	E	E	E	E	E	E	E	E

Tableau 5.5 – L'opérateur $killable_{EXACTN}$

avec :

$$v_2 = v_to_al(n_2) \quad (5.22)$$
$$killable_{EXACTN}(n_1, v_2)$$
$$(5.23)$$

et :

$$v_1 = v_to_al(n_1) \quad (5.24)$$
$$killable_{EXACTN}(v_1, n_2)$$

5.3.4 Le treillis $Type$

Le treillis des types que nous avons retenu est un treillis de constantes. Il regroupe les types définis par la norme C ainsi que les types spéciaux de la représentation interne de PIPS qui peuvent poser problème ou constituer des solutions, comme $type_unknown$ et $overloaded$. Chacun d'eux pourrait servir d'élément minimal ($type_unknown$) ou maximal ($overloaded$) pour le treillis. Pour la spécification, $overloaded$ est le sommet du treillis et $type_unknown$ est l'élément le plus petit (voir figure 5.10).

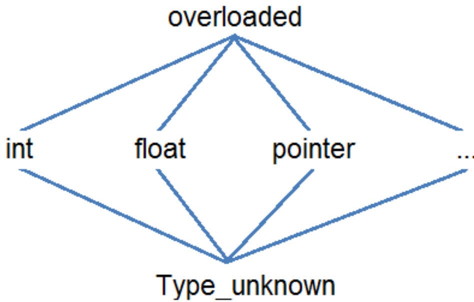


FIGURE 5.10 – Le treillis type

Nous avons introduit une propriété pour contrôler le treillis $Type$, la *property* ALIAS_ACROSS_TYPES qui doit être prise en compte lors de l'analyse des arcs *points-to*. Elle stipule que deux pointeurs de types différents peuvent ou non être en alias. Les types de C présentent une difficulté supplémentaire : l'équivalence de type. Elle a été introduite par l'équation 4.6 à la section 4.2.3.1.

5.3.4.1 L'opérateur max au niveau $Type$

Pour la composante Type, nous définissons un élément maximal $\top = overloaded$. Ainsi nous pouvons définir l'opérateur max_T le tableau 5.6.

$\mathbf{max_T(t_1, t_2)}$	$t_1 \neq type_unknown$	$type_unknown$
$t_2 \neq type_unknown$	(5.25)	t_2

Tableau 5.6 – Opérateur max_T

avec :

$$Type \times Type \longrightarrow Type : \text{si } t_1 = t_2 \text{ alors } t_1 \text{ sinon } overloaded \tag{5.25}$$

5.3.4.2 L'opérateur intersection au niveau $Type$

L'opérateur $inter_T$ est défini par le tableau 5.7 :

$\mathbf{inter_T(t_1, t_2)}$	$t_1 \neq overloaded$	$overloaded$
$t_2 \neq overloaded$	(5.26)	t_2

Tableau 5.7 – Opérateur $inter_T$

avec :

$$Type \times Type \longrightarrow Type : \text{si } t_1 = t_2 \text{ alors } t_1 \text{ sinon } type_unknown \tag{5.26}$$

5.3.4.3 Le prédicat Kill au niveau du treillis $Type$

L'opérateur $killable_{MAYT}$ a pour signature :

$$killable_{MAYT}(t_1, t_2) : Type \times Type \longrightarrow Bool$$

et il est défini par le tableau 5.8 :

$\mathbf{killable_{MAYT}(t_1, t_2)}$	$t_1 \neq overloaded$	$overloaded$
$t_2 \neq overloaded$	$t_1 = t_2$	F

Tableau 5.8 – Opérateur $killable_{MAYT}$

Sa version exacte a la même signature, mais a une autre définition donnée par le tableau 5.9 :

$\mathbf{killable_{EXACTT}(t_1, t_2)}$	$t_1 \neq overloaded$	$overloaded$
$m_2 \neq overloaded$	$t_1 = t_2$	T

Tableau 5.9 – Opérateur $killable_{EXACTT}$

Comme le type $overloaded$ est le sommet du treillis, il peut par conséquent « tuer » n'importe quel autre type.

5.3.5 Le treillis des indices $Vref$

Ce treillis correspond aux indices des références représentant les emplacement mémoire. Les références peuvent avoir un vecteur d'indices qui peut être de dimension nulle en cas de variables scalaires simples. Une référence avec des indices peut représenter un élément de tableau ou le déréférencement d'un pointeur ou bien encore un accès à un champ de structure. Le treillis est fondé sur l'ensemble des constantes qui peuvent apparaître comme indices de tableaux, à savoir les entiers positifs. Le sommet du treillis est l'élément « * », défini au niveau du domaine des constantes $\mathcal{C}$ (voir section 4.2.1), qui représente n'importe quel élément de tableau.

5.3.5.1 L'opérateur max pour le treillis $Vref$

Cet opérateur renvoie le sommet du treillis $Vref$, le symbole « * » si les indices ne sont pas des constantes. Sinon il renvoie le maximum des deux indices. Il est défini comme suit :

$$\begin{aligned} max_{Vref} : Vref \times Vref \quad &\rightarrow \quad Vref \\ \text{if } v_1 = * \vee v_2 = * \quad &\text{alors} \quad \{*\} \text{ sinon } max(v_1, v_2) \end{aligned} \tag{5.27}$$

5.3.5.2 L'opérateur d'intersection pour le treillis $Vref$

Nous pouvons définir le prédicat d'intersection $inter_{Vref}$, qui renvoie un tableau d'indices si les deux vecteurs ont des éléments en commun.

Fonction 16 : $inter_{Vref}$

```
inter_Vref(v1[0,n[, v2[0,m[)
res[]
for i = 0; i < n&&i < m; i++ do
    if v1[i] == v2[i] ∨ v1 == * then
        res[i] = v2[i]
    else if v2 == * then
        res[i] = v1[i]
return res
```

Fonction 17 : $killable_{killMAYV}$

```
killable_killMAYV(v1[0,n[, v2[0,m[)
if n == 0 then
    false
if m == 0 then
    true
else if v1_0 == v2_0 ∨ v1_0 == * ∨ v2_0 == * then
    killable_killMAYV(v1[1,n[, v2[1,n[)
else
    false
```

5.3.5.3 L'opérateur $Killable$ pour le treillis $Vref$

Nous définissons l'opérateur $killable_{killMAYV}$ et l'opérateur $killable_{killEXACTV}$ par les fonctions 17 et 18. Dans la version `MAY`, nous considérons que l'élément « * » peut « tuer » tout autre élément du treillis.

Dans la version exacte, voir la fonction 18, l'élément « * » ne peut pas « tuer » un élément du treillis. Seul deux indices égaux peuvent aboutir à la suppression d'un arc *points-to*, et uniquement si tous les prédicats des autres treillis sont évalués à vrai.

Fonction 18 : $killable_{killEXACTV}$

```
killable_killEXACTV(v1[0,n[, v2[0,m[)
if n == 0 then
    false
if m == 0 then
    true
else if v1_0 == v2_0 then
    killable_killEXACTV(v1[1,n[, v2[1,n[)
else
    false
```

5.3.6 Le prédicat d'atomicité sur le treillis CP

Le prédicat d'atomicité permet de vérifier si le chemin d'accès représente un ou plusieurs emplacements concrets pouvant être mis à jour globalement par une affectation.

Dans le but de déterminer l'atomicité d'un emplacement, nous devons d'abord vérifier s'il s'agit d'une abstraction ou non. Par principe les emplacements abstraits comme les éléments du tas ou `*anywhere*` représentent un ou plusieurs emplacements réels. Les « stubs », les éléments du contexte formel F, constituent un cas particulier. À l'intérieur d'une procédure, un stub peut être atomique si son type le permet. Au niveau d'un site d'appel, un stub peut être associé à plusieurs chemins d'accès constant et n'est donc plus atomique par défaut, quel que soit son type. Une autre construction qui implique l'existence de plusieurs emplacements est l'indice « * » qui apparaît dans la liste d'indice d'une référence. Cela est traduit par n'importe quel élément du tableau et par conséquent un ou plusieurs emplacements réels.

Fonction 19 : atomic_location_p()

atomic_location_p : $\mathcal{A} \rightarrow \mathcal{B}$
if $abstract_location_p(a) \;||\; sdp(inds)$ **then**
  true
else
  false

Le prédicat $abstract_location_p(a)$ teste si un emplacement est un emplacement abstrait et sdp teste si la liste des indices d'un chemin constant contient l'élément abstrait « * » (déjà vu à la section 4.3.3.4).

5.3.7 Conclusion

Dans cette section nous définissons le treillis CP qui contient les nœuds des arcs *points-to*. Nous avons décomposé le treillis en quatre sous-treillis pour faciliter sa définition. Ainsi un chemin constant est représenté par une référence étendue dont la variable est déclarée dans une fonction, a un nom (un identifiant), un type et éventuellement une liste d'indices prenant leurs valeurs dans l'union des entiers, de * et des noms de champs. Pour les équations sémantiques *points-to* nous avons défini les opérateurs et prédicats suivants :

1. un opérateur max pour calculer le plus petit emplacement abstrait contenant les deux opérandes ;
2. un opérateur $inter$ pour calculer l'emplacement abstrait ; il correspond à l'intersection entre les deux opérandes ;
3. un prédicat $killable$ pour décider si un arc peut être supprimé du graphe *points-to* ou non ;
4. un prédicat $atomic_location_p$ pour déterminer si un emplacement abstrait est atomique ou non.

5.4 Les choix de conception

Nous présentons dans cette section les choix que nous avons effectués lors de la conception du treillis CP et leurs impacts sur l'analyse *points-to*.

5.4.1 L'initialisation des paramètres formels à NULL

Le treillis des chemins constants CP tel qu'il est décrit par la figure 5.7 place l'emplacement NULL sur le même rang que les emplacements STATIC, HEAP et les autres zones d'allocation. Par conséquent les stubs qui sont générés au niveau de la zone FORMAL ne peuvent pas avoir la valeur NULL.

Une autre conception aurait pu être adoptée ; elle est illustrée par la figure 5.11. L'emplacement NULL est supérieur aux emplacements des stubs et par conséquent un stub aurait pu désigner le pointeur NULL. Nous expliquons ci-dessous l'impact de notre conception sur l'analyse *points-to* et en particulier sur l'analyse des conditions impliquant des pointeurs.

5.4.2 Impact de l'initialisation à NULL sur l'interprétation des conditions

Si notre conception stipule que les stubs ne peuvent pas avoir la valeur NULL, mais une propriété PIPS POINTS_TO_NULL_POINTER_INITIALIZATION que nous avons définie permet ou non une initialisation des stubs à l'emplacement NULL. Pour montrer l'impact de cette propriété, nous analysons le programme 5.5 en premier lieu en désactivant la propriété en la mettant à faux.

```
 int foo(int *q)
{
  int i = 0, *p;
  p = q;
  if(p==NULL)
    printf("i␣=␣%d", i);
  return i;
}
```

Prog 5.5 – Modélisation des paramètres formels

Nous obtenons les arcs *points-to* illustrés par le programme 5.6. Lorsque la condition p==NULL est évaluée nous ne pouvons pas savoir si p est NULL ou non. Si le point de contrôle est au niveau du corps du test alors le pointeur vaut exactement NULL. À la sortie du test, l'arc exact est maintenant un arc MAY et il est ajouté à l'ensemble *points-to*. Nous remarquons cependant que le pointeur q, qui est en alias avec le paramètre formel p, n'apparaît pas comme une source d'un arc *points-to* vers l'emplacement NULL.

Pour la deuxième exécution la propriété POINTS_TO_NULL_POINTER_INITIALIZATION est activée, nous obtenons le résultat illustré par le programme 5.7. L'initialisation du paramètre formel génère deux arcs *points-to* ; un vers un stub _p_1 et un vers l'emplacement NULL.

À la sortie du test les deux pointeurs p et q pointent vers le stub et l'emplacement NULL. Pour résumer la propriété POINTS_TO_NULL_POINTER_INITIALIZATION permet de tenir compte de l'emplacement NULL sans pour autant changer notre conception du treillis et tout en offrant le choix à l'utilisateur d'activer ou non cette propriété. En effet, pour l'analyse interprocédurale et quand le contexte d'appel est connu sans paramètres effectifs qui valent le pointeur NULL, la propriété peut être désactivée.

```
int foo(int *q)
{
   int i = 0, *p;
// Points To:
// p -> undefined , EXACT
// q -> _q_1 , EXACT

   p = q;
// Points To:
// p -> _q_1 , EXACT
// q -> _q_1 , EXACT

   if (p==(void *) 0)
// Points To:
// p -> *NULL_POINTER* , EXACT
// q -> _q_1 , EXACT

      printf("i␣=␣%d", i);
// Points To:
// p -> *NULL_POINTER* , MAY
// p -> _q_1 , MAY
// q -> _q_1 , EXACT

   return i;
}
```

Prog 5.6 – Analyse sans une initialisation des paramètres formels à `NULL`

```
int foo(int *q)
{
   int i = 0, *p;
// Points To:
// p -> undefined , EXACT
// q -> *NULL_POINTER* , MAY
// q -> _q_1 , MAY

   p = q;
// Points To:
// p -> *NULL_POINTER* , MAY
// p -> _q_1 , MAY
// q -> *NULL_POINTER* , MAY
// q -> _q_1 , MAY

   if (p==(void *) 0)
// Points To:
// p -> *NULL_POINTER* , EXACT
// q -> *NULL_POINTER* , MAY
// q -> _q_1 , MAY

      printf("i = %d", i);
// Points To:
// p -> *NULL_POINTER* , MAY
// p -> _q_1 , MAY
// q -> *NULL_POINTER* , MAY
// q -> _q_1 , MAY

   return i;
}
```

Prog 5.7 – Analyse avec une initialisation des paramètres formels à `NULL`

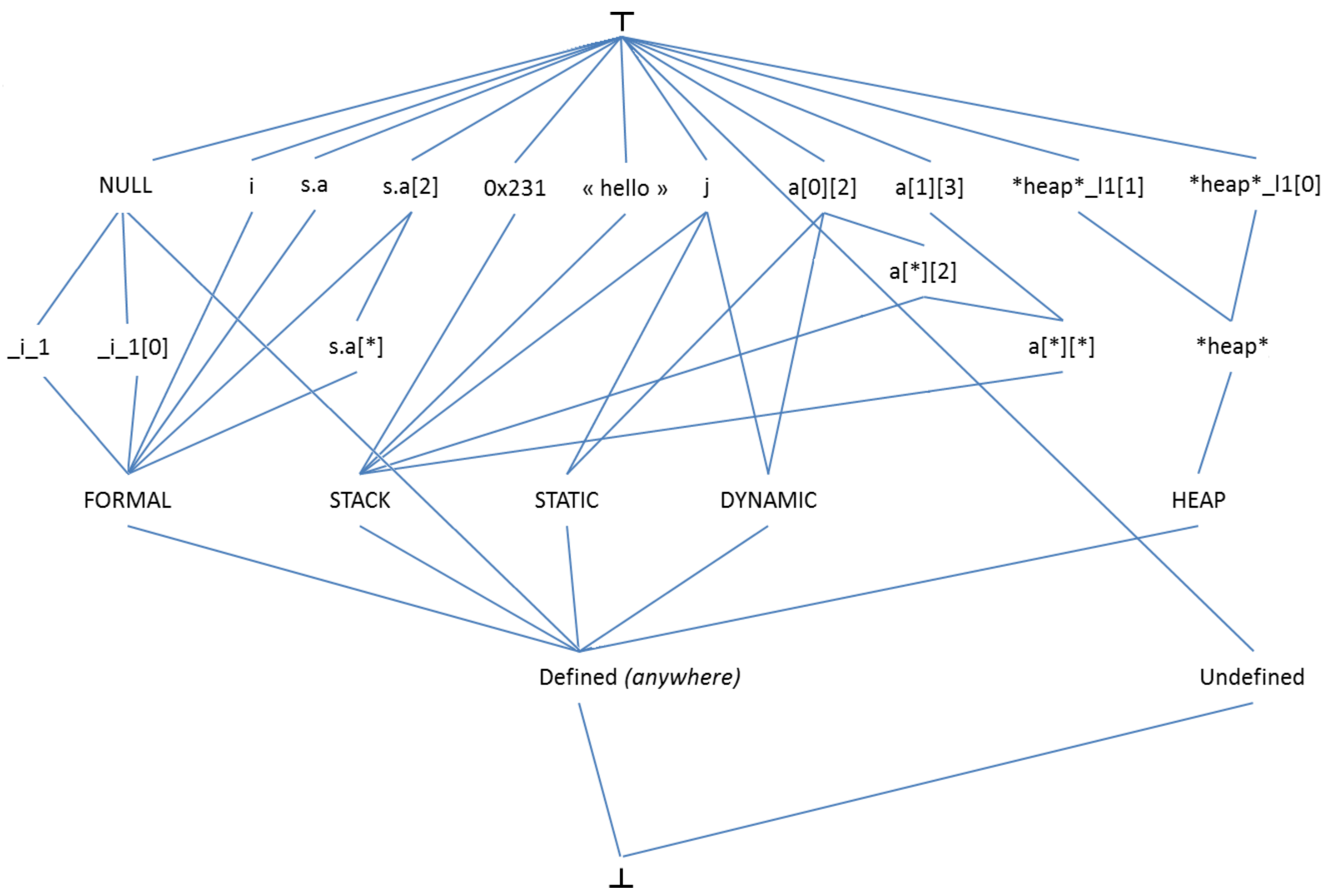


FIGURE 5.11 – Un autre choix pour le treillis des emplacements mémoire

5.4.3 L'utilisation des alias

Nous avons opté pour une analyse de pointeurs plutôt que pour une analyse d'alias (voir la sous-section 1.3.2). Cependant notre choix ne nous permet pas d'exploiter parfaitement toutes les instructions, particulièrement les conditions. En effet, si, comme pour le programme 5.6, la propriété `POINTS_TO_NULL_POINTER_INITIALIZATION` est désactivée, une analyse des alias aurait permis de conclure que le pointeur `q` pointe lui aussi vers `NULL` puisqu'il est en alias avec `p`.

5.4.4 Conclusion

Notre conception du treillis nous permet de bien séparer les emplacements possibles vers lesquels peut pointer un pointeur. Cependant il faut l'utiliser en considérant chaque partie séparément. Ainsi, lors de la création à la demande du contexte formel, il faut considérer en particulier la zone d'allocation FORMAL, les paramètres formels ainsi que les stubs. La propriété `POINTS_TO_NULL_POINTER_INITIALIZATION` nous permet d'inclure le pointeur `NULL` à l'ensemble des adresses que peut contenir un paramètre formel.

5.5 Treillis des relations *points-to* $\mathcal{PT}$

Le treillis des relations *points-to* est construit à partir du treillis des chemins constants, CP. Une relation *points-to* est vue comme un ensemble d'arcs dont les sources sont des éléments de CP de type pointeur ou d'un type contenant le type pointeur et dont les destinations sont d'un type quelconque, mais correspondant ou équivalent au type pointé par la source. Deux relations *points-to* sont en fait calculées simultanément. La première est une fonction partielle, et donc une relation, et ses arcs sont étiquetés EXACT. La seconde peut contenir des arcs qui n'existent pas dans le l'état mémoire courant, σ, et qui sont étiquetés *MAY*.

L'union de ces deux relations est une sur-approximation de la relation exacte liant, à un état mémoire donné, les pointeurs à leurs valeurs, *i.e.* les adresses qu'ils pointent. Une relation *points-to* P se décompose donc en une fonction notée P_{EXACT} et une relation P_{MAY}. Un arc de P appartient à l'un ou l'autre en fonction de son approximation dans leur union disjointe :

$$P = \{(s,d,a)|(s,d) \in P_{\text{EXACT}} \wedge a = \text{EXACT}\} \cup \{(s,d,a)|(s,d) \in P_{\text{MAY}} \wedge a = \text{MAY}\}$$

Comme notre analyse génère le contexte formel à la demande, les relations utilisées ne sont que des projections des relations *points-to* qu'on obtiendrait par une analyse descendante sur le graphe des appels. Il faut donc faire attention aux extensions du contexte formel qui peuvent avoir lieu lors de l'analyse, en particulier quand on fait l'union de deux relations avec l'opérateur $merge$.

Nous présentons successivement la concrétisation d'une relation *points-to* (section 5.5.1), le domaine des relations (section 5.5.2), le typage des arcs (section 5.5.3), l'ordre sur les relations (section 5.5.4 et enfin des remarques sur la finitude de ces relations (section 5.5.5).

5.5.1 Abstraction de la valeur d'un pointeur avec une relation *points-to*

La valeur abstraite d'un chemin de type pointeur, cp, pour une relation *points-to* donnée, P, est un ensemble de chemins d'accès constant défini par :

$$\begin{aligned} apply_pt(cp, P) &= si\ \exists d \in CP\ t.q.\ (cp, d) \in P_{\mathsf{EXACT}}\ alors\ \{d\} \\ &\quad sinon\ si\ \exists d \in CP\ t.q.\ (cp, d) \in P_{\mathsf{MAY}}\ alors\ \{d | (cp, d) \in P_{\mathsf{MAY}}\} \\ &\quad sinon\ \{d | \exists cp' \in CP\ t.q.\ killable(cp, cp')\ \wedge\ (cp', d) \in P_{\mathsf{MAY}}\} \end{aligned}$$

5.5.2 Domaine des relations *points-to*

Une relation *points-to* retourne toujours au moins une valeur pour n'importe quelle expression de type pointeur scalaire. Mais, comme la relation est calculée à la demande, les adresses associées directement ou indirectement à un paramètre formel ou à une variable globale peuvent avoir à être générées. Cette génération pose deux problèmes liés. Quel est le type de la nouvelle entité, du nouveau *stub*, qu'il faut créer ? Quel sont les indices qu'il faut lui associer pour constituer un chemin constant ? Enfin, ce nouveau chemin constant doit vérifier les conditions de typage des arcs *points-to*.

Cette génération pose aussi le problème de la finitude du nombre de nœuds du graphe qui est traité section 5.5.5.

Les tableaux de pointeurs, combinés à l'hypothèse générale de non aliasing et à l'objectif de calcul des *use-def chains*, complique le typage du *stub*. Si le chemin d'accès source est un élément de tableau, il faut que le type du *stub* destination ait au moins autant de dimensions pour ne pas créer d'aliasing entre chemin d'accès indépendants. Il faut ensuite ajuster le type du chemin constant généré à partir du *stub* afin qu'il corresponde bien au type pointé par la source. Ce problème n'avait pas d'importance avec les langages $\mathcal{L}_0$ et $\mathcal{L}_1$ parce qu'ils ne comportent ni boucles ni appels de procédures ce qui permet, théoriquement, de calculer les valeurs des indices.

Les structures pouvant comprendre, directement ou indirectement, plusieurs pointeurs créent un problème de nommage des *stubs*. Le rang du champ est utilisé pour lever l'ambiguïté sur les cibles des divers chemins d'accès.

5.5.3 Typage d'un arc *points-to*

Un arc est bien typé si le type de la source est un pointeur scalaire et si le type de la destination est un type compatible avec le type pointé.

La compatibilité entre types dépend de la norme C et de l'implémentation pour les constantes chaines de caractères.

Le type utilisé par défaut pour la cible d'un pointeur de type `void *` est de type `char []` ou bien `char`, parce qu'il n'est pas possible de déclarer un scalaire ou un tableau de type `void`.

5.5.4 Ordre sur les relations *points-to*

Les relations sont ordonnées par inclusion de l'union de leur graphes *EXACT* et *MAY*. Une relation moins précise associe à n'importe quel chemin d'accès de type pointeur un ensemble d'adresses constantes plus grand qu'une relation plus précise.

5.5.5 Finitude des relations

L'ensemble des arcs *points-to* est toujours conservé fini parce que l'ensemble des nœuds est fini et qu'il ne peut exister qu'un arc entre deux nœuds donnés. Si le nombre de nœuds liés aux éléments d'un tableau dépasse une constante prédéterminée, ces nœuds sont complétés par

un unique élément plus grand dans le treillis CP. Si le nombre de nœuds sortant d'un nœud dépasse une constante, les nœuds destinations sont remplacés par leur borne supérieure dans le treillis CP. Si le nombre de nœuds formels associés à un paramètre formel ou à une variable globale dépasse une constante, un cycle est créé dans la relation *points-to* afin de représenter les chemins non bornés. Ces questions sont reprises plus en détail section 5.7.

5.5.6 Typage des arcs *points-to*

Il est naturel de considérer que la source et la destination d'un arc *points-to* doivent avoir des types compatibles. Le type du chemin constant correspondant à la source doit être un type pointeur et le type du chemin constant correspondant à la destination doit être compatible avec le type pointé, la compatibilité de type étant définie par la norme C.

Ce principe de typage des arcs *points-to* pose problème en présence de *cast*. Par exemple, la séquence :

```
p = &x;
q = (foo) p;
```

Prog 5.8 – Exemple de *cast*

conduit à un problème si le type *foo* n'est pas compatible avec le type de p. Comme le langage C autorise de nombreux *cast*, on a donc un choix à effectuer entre relâcher la contrainte de typage des arcs, relâcher l'aliasing entre objets ou utiliser le treillis CP.

D'un point de vue pratique, il est difficile de relâcher le typage des arcs. En effet, ils sont presque toujours correctement typés et ce typage est extrêmement utile pour la mise au point de l'implémentation de l'analyse *points-to*. Pour la compatibilité avec les analyses utilisatrices, il est essentiel de ne pas ajouter d'aliasing statique entre objets ou entre chemins constants. Nous avons donc choisi d'utiliser le treillis CP, et particulièrement sa composante type, pour modéliser l'impact d'un *cast*.

L'idée consiste donc, pour chaque destination possible, à choisir un plus grand élément ayant un type compatible avec celui de la destination et celui du *cast*. L'utilisation d'un plus grand élément permet simultanément de conserver le typage des arcs et la détection des conflits mémoire. Dans l'exemple ci-dessus, on veut toujours pouvoir détecter que les accès à *p et à *q sont en conflit.

5.5.6.1 Conclusion sur les relations *points-to* abstraites

Ces définitions et propriétés viennent compléter celles introduites au chapitre 4 mais qui ne prenaient pas en compte en compte le domaine $\mathcal{PT}$ bien défini dont nous devons assurer la finitude et la non explosion de nombre d'arcs. La décomposition MAY/EXACT permet, quant à elle, de modulariser un problème trop complexe pour être traité en une seule fois.

5.6 Utilisation des treillis CP et $\mathcal{PT}$

L'utilisation d'un treillis pour la représentation des emplacements mémoire permet non seulement d'abstraire l'ensemble des cellules mémoire d'un programme mais aussi de résoudre les problèmes de chemins d'accès de longueur infinie. En effet, l'allocation dynamique d'un champ d'une structure de données récursive dans une boucle peut augmenter considérablement la longueur des chemins d'accès et aboutir à la divergence de l'algorithme d'analyse. Nous verrons dans ce qui suit comment nous pouvons utiliser les treillis pour assurer la convergence de notre analyse.

5.6.1 Calcul des ensembles Kill et Gen : affectation d'un pointeur

Nous avons déjà défini les équations qui permettent le calcul des ensembles Kill et Gen pour le langage $\mathcal{L}_1$ (voir les sections 4.4.1.1 et 4.4.1.2). Ces dernières ne prennent pas en compte le treillis CP qui permet d'ordonner les emplacements mémoire et de mieux les comparer. Dans cette sous-section, nous donnons une nouvelle définition des ensembles où le treillis CP intervient dans la phase de calcul. Nous commençons par rappeler que deux ensembles sont créés pour une opération d'affectation avec en parties gauche et droite des expressions, `lhs = rhs`. Les deux fonctions `eta` et `etv` (voir les sections 4.3.3.3 et 4.3.3.4) sont appelées respectivement sur `lhs` et `rhs` comme suit :

$$(L, \mathsf{In}') = eta(lhs, \mathsf{In})$$
$$(R, \mathsf{In}'') = etv(rhs, \mathsf{In}')$$

Nous obtenons deux ensembles de chemins d'accès mémoire constants, un pour la partie gauche L et un pour la partie droite R.

5.6.1.1 Traitement des erreurs

Des erreurs peuvent ou doivent être détectées si l'ensemble L, résultat de la fonction eta, contient ou est réduit à `NULL` ou $undefined$, à moins que l'utilisateur ait décidé que de tels déréférencements devaient être acceptés, par exemple, pour poursuivre l'analyse. Pour cela deux propriétés :

- `POINTS_TO_NULL_POINTER_DEREFERENCING` ;
- `POINTS_TO_UNINITIALIZED_POINTER_DEREFERENCING`

ont été définies pour permettre de forcer ce comportement.

5.6.1.2 Calcul de l'ensemble Kill

Nous commençons par définir l'ensemble Kill qui se décompose en fait en deux ensembles Kill_1, aussi appelé $\mathsf{Kill}_{\mathsf{EXACT}}$, et Kill_2, aussi appelé $\mathsf{Kill}_{\mathsf{MAY}}$. Le premier, Kill_1, correspond aux arcs *points-to* qui vont être supprimés définitivement du graphe *points-to* parce que l'on sait que leur origine est certainement écrite. Il est défini par l'équation suivante :

$$\mathsf{Kill}_1(L, \mathsf{In}) = \{(l, r) \in \mathsf{In} | l \in L \wedge \mid L \mid = 1 \ \wedge \ \mathsf{atomic}(L)\} \tag{5.28}$$

avec la fonction $source(edg)$ qui retourne la source d'un arc *points-to* passé en argument et atomic qui correspond à une abréviation du prédicat $atomic_location_p$ défini section 5.3.6, étendue d'un élément l à un ensemble L.

Le deuxième ensemble, Kill_2, correspond aux arcs exacts dont l'origine est peut-être écrite par l'affectation et dont l'approximation doit passer à MAY. Il est défini par l'équation suivante :

$$\mathsf{Kill}_2(L, \mathsf{In}_{\mathsf{EXACT}}) = \{(l', r) \in \mathsf{In}_{\mathsf{EXACT}} | \exists l \in L, killable_{\mathsf{MAY}}(l, l')\} - \mathsf{Kill}_1 \tag{5.29}$$

Les arcs exacts supprimés par Kill_2 doivent réapparaître sous forme approximative dans la relation MAY.

5.6.1.3 Calcul de l'ensemble Gen

Comme nous avons décomposé l'ensemble Kill, nous décomposons aussi Gen pour faciliter sa construction. En effet, l'ensemble Gen peut être décomposé en deux sous-ensembles, Gen_1 qui correspond aux arcs apparaissant dans Kill_2 et qui doivent changer d'approximation et Gen_2 qui correspond aux arcs générés par l'affectation.

Calcul de l'ensemble Gen_1 Cet ensemble correspond aux éléments de Kill_2 qui doivent changer d'approximation pour passer d'EXACT vers MAY. Cet ensemble correspond à l'ensemble Kill_2 et est défini par l'équation suivante :

$$\text{Gen}_1(\text{Kill}_2) = \{(l, sink, \text{MAY}) | (l, sink, a') \in \text{Kill}_2\} \tag{5.30}$$

Calcul de l'ensemble Gen_2 Cet ensemble correspond aux nouveaux arcs résultant de l'affectation ; il est défini par l'équation suivante :

$$\text{Gen}_2(L, R) = \{(l, r, a) | \exists l \in L, \exists r \in R, a = \text{EXACT if}(\mid L \mid = 1 \wedge \mid R \mid = 1 \wedge \text{atomic}(l) \wedge \text{atomic}(r)) \text{ MAY otherwise}\} \tag{5.31}$$

L' ensemble Out calculé après une affectation est donc le suivant :

$$\text{Out} = (\text{In} - \text{Kill}_1 - \text{Kill}_2) \cup (\text{Gen}_1 \cup \text{Gen}_2) \tag{5.32}$$

5.6.1.4 Exemple pour le calcul des ensembles Kill et Gen

Pour illustrer les étapes de construction des ensembles Kill et Gen avec l'utilisation du treillis CP, nous traitons un exemple avec un tableau de pointeurs (voir le programme 5.14).

```
int main()
{
  int *a[10], b = 0, c = 1;
  a[0] = &b;
  a[1] = &b;
  a[1] = &c;
  return(0);
}
```

Prog 5.9 – Exemple de tableau de pointeurs

Pour l'exemple 5.14, les ensembles In, Kill_1 , Kill_2, Gen_1, Gen_2 et Out pour la première affectation, « `a[0] = &b;` », ont les valeurs suivantes :

$$\begin{aligned}
\text{In} &= \{(a[*], undefined, \text{EXACT})\} \\
\text{Kill}_1 &= \emptyset \\
\text{Kill}_2 &= \{(a[*], undefined, \text{EXACT})\} \\
\text{Gen}_1 &= \{(a[*], undefined, \text{MAY})\} \\
\text{Gen}_2 &= \{(a[0], b, \text{EXACT})\} \\
\text{Out} &= \{(a[*], undefined, \text{MAY}), (a[0], b, \text{EXACT})\}
\end{aligned} \tag{5.33}$$

Notre analyse fournit le résultat illustré par le programme 5.10.

5.6.2 Cas des boucles

Une boucle en langage `C` peut être soit une boucle *while*, une boucle *do...while* ou une boucle *for*. Le traitement des corps des boucles est le même pour tous les types de boucles, avec des différences au niveau de l'information *points-to* en entrée. En effet, pour la boucle *while*, seule une condition à l'entrée est analysée (section 5.6.3). Pour la boucle *for* trois expressions doivent être analysées en sus du corps de la boucle (section 5.6.4). Enfin, pour la boucle *do...while*, l'information en entrée correspond à une première itération sur les instructions du corps (section 5.6.5).

```c
int main()
{
   int *a[10], b = 0, c = 1;
// Points To:
// a[*] -> undefined , EXACT

   a[0] = &b;
// Points To:
// a[*] -> undefined , MAY
// a[0] -> b , EXACT

   a[1] = &b;
// Points To:
// a[*] -> undefined , MAY
// a[0] -> b , EXACT
// a[1] -> b , EXACT

   a[1] = &c;
// Points To:
// a[*] -> undefined , MAY
// a[0] -> b , EXACT
// a[1] -> c , EXACT

   return 0;
}
```

Prog 5.10 – Exemple d'éléments d'un tableau de pointeurs

Dans ce qui suit, nous commençons par détailler le traitement de la boucle *while*, qui est le même que pour les deux autres boucles, la différence étant l'ordre et le nombre des conditions. Les cas des boucles *for* et *do...while* sont étudiés dans les sections 5.6.4 et 5.6.5. Enfin, l'algorithme général `any_loop_to_points_to()` est donné à la section 5.6.6.

5.6.3 Cas de la boucle « while »

Les instructions de type boucle sont les plus importantes du fait qu'elles dominent le temps de calcul dans les applications scientifiques et qu'elles sont sujettes à des optimisations permettant d'améliorer les performances du code. Pour déterminer les équations *points-to* correspondantes, nous commençons par l'étude d'un exemple contenant une boucle où se fait le calcul du nombre d'éléments d'une liste chaînée.

5.6.3.1 Exemple 1

Le programme 5.12 contient un compteur sur une liste chaînée d'entiers . C'est un exemple un peu difficile parce qu'il modifie le paramètre formel `p`, ce qui est une mauvaise pratique pour la maintenance.

Il faudrait donc conserver la valeur initiale de `p` pour ne pas finir avec des *stubs* inatteignables en sortie de fonction. Comme ce problème est lié à une mauvaise pratique, nous avons décidé de ne pas conserver de copie (voir sous-section 6.3.4.1).

Avant de déterminer l'équation sémantique des arcs *points-to*, nous commençons par dérouler la boucle en appliquant les équations de génération d'arcs *points-to*. Comme le pointeur `p` est un paramètre formel, son initialisation se fait à la demande. Il devrait être initialisé à un « stub » ainsi qu'à `NULL`. Mais comme la condition de la boucle est analysée en premier, tout en l'évaluant

```
typedef struct LinkedList{
  int *val;
  struct LinkedList *next;
} list;
```

Prog 5.11 – Définition de la structure de données

```
int count(list *p)
{
  int i = 0;
  while( p != NULL){
    i++; p = p->next;
  }
  return i;
}
```

Prog 5.12 – Exemple de liste chaînée : compteur d'entiers

à vrai pour pouvoir analyser les instructions de corps de boucle, l'arc de `p` vers `NULL` est supprimé. Ainsi l'ensemble des arcs *points-to* en entrée du corps de boucle vaut {`p->_p_1, MAY`}. Lors de l'analyse du corps de boucle on obtient les post-conditions Y_i suivantes où i est le numéro de l'itération :

$$
\begin{aligned}
\text{pour la condition en entrée :} &= \{(p, _p_1, \text{MAY})\}\\
\text{pour } i = 1 : \; Y_1 &= \{(p, _p_1, \text{MAY}), (p, _p_1_1, \text{MAY}), (p, \texttt{NULL}, \text{MAY}),\\
&\quad (_p_1.next, _p_1_1, \text{MAY}), (_p_1.next, \texttt{NULL}, \text{MAY})\}\\
\text{pour } i = 2 : \; Y_2 &= \{(p, _p_1, \text{MAY}), (p, _p_1_1, \text{MAY}), (p, \texttt{NULL}, \text{MAY}),\\
&\quad (p, _p_1_1_1, \text{MAY}), (_p_1.next, _p_1_1, \text{MAY}),\\
&\quad (_p_1.next, \texttt{NULL}, \text{MAY}), (_p_1_1.next, _p_1_1_1, \text{MAY}),\\
&\quad (_p_1_1.next, _p_1_1_1, \text{MAY})\}\\
\text{pour } i = n : \; Y_n &= \{(p, _p_1^n, \text{MAY}), (p, _p_1^{n-1}, \text{MAY}), (p, _p_1^{n-2}, \text{MAY}),\\
&\quad (p, \texttt{NULL}, \text{MAY}), (_p_1^n.next, _p_1_1^n, \text{MAY}),\\
&\quad (_p_1^{n-1}.next, _p_1_1^{n-1}, \text{MAY}), (_p_1^n.next, \texttt{NULL}, \text{MAY}),\\
&\quad (_p_1^{n-1}.next, \texttt{NULL}, \text{MAY})...\}
\end{aligned}
$$

Y_i accumule les informations *points-to* des itérations 0 à i et ^{n} désigne désigne la longueur du chemin à l'itération n. Dès la deuxième itération nous pouvons distinguer trois cas de figures :

1. un nouvel arc *points-to* est calculé qui appartient à l'itération i et à l'itération $i + 1$;
2. un nouveau arc *points-to* est calculé au niveau de l'itération i mais pas à l'itération $i + 1$;
3. un nouveau arc *points-to* est calculé au niveau de l'itération $i + 1$ mais pas à l'itération i

Si pour deux itérations consécutives il n'y a plus de nouveaux arcs générés ($Y_i = Y_{i-1}$) alors le calcul des arcs pour la boucle *while* est terminé et le résultat est transmis au « statement » suivant. Sinon, pour les trois cas cités précédemment, nous pouvons envisager les solutions suivantes :

1. dans le premier cas, nous gardons l'arc au niveau de l'itération $i + 1$ et il est propagé aux itérations suivantes ;
2. dans le deuxième cas, nous gardons l'arc mais son approximation est changée, d'où le besoin d'un opérateur de fusion entre les résultats des itérations (il est décrit par la suite) ;
3. si le nouvel arc généré au niveau de l'itération $i + 1$ admet au niveau de sa source ou destination un prédécesseur dans l'itération i, comme c'est le cas pour le programme 5.12 où `_p_1.next` est un prédécesseur de `_p_1_1.next`, un problème se pose car nous devons prendre la décision de garder ou non l'arc où le prédécesseur apparaît ; une autre question

se pose aussi si, comme dans l'exemple, l'analyse itère indéfiniment sur la boucle : comment faut-il procéder pour l'arrêter ? l'équation (5.34) peut être utilisée ; elle définit une longueur maximum du chemin d'accès mémoire (ou définit l'élément absorbant « `anywhere` ») ;

En résumé, le chemin d'accès de préfixe $_p_1$ s'allonge à chaque itération de la boucle. Et nous pouvons en déduire que l'algorithme risque d'itérer indéfiniment sur le corps de la boucle. On a donc besoin de déterminer un point fixe pour arrêter le calcul. D'après la littérature, nous pouvons avoir recours au « k-limiting » [Deu94] en imposant au préalable une condition sur la longueur des chemins d'accès mémoire en sortie de boucle. Une autre possibilité serait de décider de changer le chemin d'accès infini (dépassant une certaine longueur précédemment définie) en un arc *points-to* qui pointe vers un emplacement abstrait. Le choix de cet emplacement dépend du degré de précision choisi. En effet, l'emplacement peut être soit le somment `*anywhere*`, soit un autre emplacement respectant l'ordre défini par le treillis des emplacements abstraits. Il peut être typé ou non.

Une autre solution consiste à créer un cycle une fois la longueur maximale atteinte. En choisissant cette longueur maximale des chemins mémoire, nous pouvons garantir que, dès que cette valeur est atteinte, aucun nouveau nœud n'est créé. Ainsi le calcul des arcs au niveau de la boucle atteint un point fixe et la convergence de l'algorithme est assurée.

5.6.3.2 Équations

Nous commençons donc par définir une première solution qui consiste à garder la longueur du chemin d'accès mémoire égale à celle calculée lors des les premières itérations sur le corps de la boucle :

$$\{(q, pn^i, a)\} \cup \text{In} = \bigcup_{j \leq l} \{(q, pn^j, a)\} \cup \text{In avec } \; l = k \text{ si } i \geq k \text{ et } i \text{ sinon} \tag{5.34}$$

où k est la longueur maximale que nous avons déjà prédéfinie et contrôlée via une propriété PIPS « `POINTS_TO_PATH_LIMIT` » . Cette propriété garantit que le graphe *points-to* ne contient pas de chemins d'accès mémoire d'une grande longueur. D'autres propriétés, qui garantissent la finitude du graphe, *point-to* sont définies à la section 5.7. Parmi ces propriétés, nous citons celle qui permet de limiter le nombre d'arcs sortant d'un nœud (libération d'une liste) ainsi que celle qui limite le nombre de nœuds créés (arithmétique sur pointeur).

Après le déroulement de la boucle et dans le but de déterminer les équations relatives à cette instruction, nous récrivons la boucle *while* sous la forme « $if\ c\ then\ b; while\ c\ do\ b;$ » ; le transformeur *points-to* associé s'écrit alors sous la forme suivante :

$$\begin{aligned} \mathcal{T}_{\mathcal{PT}}[\![while\ c\ do\ b]\!](P) &= \mathcal{T}_{\mathcal{PT}}[\![if\ c\ then\ b; while\ c\ do\ b]\!](P) \\ &= \mathcal{T}_{\mathcal{PT}}[\![b; while\ c\ do\ b\ ]\!](\mathcal{T}_{\mathcal{PT}}[\![c]\!](P)) \sqcup \mathcal{T}_{\mathcal{PT}}[\![c]\!](P) \\ &= \mathcal{T}_{\mathcal{PT}}[\![while\ c\ do\ b]\!](\mathcal{T}_{\mathcal{PT}}[\![b]\!](\mathcal{T}_{\mathcal{PT}}[\![c]\!](P))) \sqcup \mathcal{T}_{\mathcal{PT}}[\![c]\!](P) \end{aligned} \tag{5.35}$$

Ensuite nous déroulons cette équation en remplaçant chaque fois $\mathcal{T}_{\mathcal{PT}}[\![while\ C\ do\ b]\!]$ par l'équation 5.35.

$$\begin{aligned} \mathcal{T}_{\mathcal{PT}}[\![while\ c\ do\ b]\!](P) &= \mathcal{T}_{\mathcal{PT}}[\![while\ c\ do\ b]\!](\mathcal{T}_{\mathcal{PT}}[\![b]\!](\mathcal{T}_{\mathcal{PT}}[\![c]\!](P))) \sqcup \mathcal{T}_{\mathcal{PT}}[\![c]\!](P) \\ &= [\mathcal{T}_{\mathcal{PT}}[\![while\ c\ do\ b]\!](\mathcal{T}_{\mathcal{PT}}[\![b]\!](\mathcal{T}_{\mathcal{PT}}[\![c]\!](\mathcal{T}_{\mathcal{PT}}[\![b]\!](\mathcal{T}_{\mathcal{PT}}[\![c]\!](P))))) \\ &\sqcup \mathcal{T}_{\mathcal{PT}}[\![c]\!](P)]\ (\mathcal{T}_{\mathcal{PT}}[\![b]\!](\mathcal{T}_{\mathcal{PT}}[\![c]\!](P))) \sqcup \mathcal{T}_{\mathcal{PT}}[\![c]\!](P) \end{aligned} \tag{5.36}$$

Les notations peuvent être simplifiées en remplaçant :

- $\mathcal{T}_{\mathcal{PT}}[\![while\ c\ do\ b]\!]$ par $\mathcal{W}$;

- $\mathcal{T}_{\mathcal{PT}}[\![b]\!]$ par $\mathcal{B}$;
- $\mathcal{T}_{\mathcal{PT}}[\![c]\!]$ par $\mathcal{C}$;

Pour aboutir aux équations suivantes où $\mathcal{W}$ est remplacé chaque fois par sa valeur pour aboutir à une équation récurrente.

$$\begin{aligned} \mathcal{W} &= \mathcal{W}\,\mathcal{B}\,\mathcal{C}\ \sqcup \mathcal{C} \\ &= \mathcal{W}\,\mathcal{B}\,\mathcal{C}\ \sqcup \mathcal{C}\,\mathcal{B}\,\mathcal{C}\ \sqcup \mathcal{C} \\ &= \bigsqcup_{k=0}^{\infty} (\mathcal{B}\mathcal{C})^k \mathcal{C} \end{aligned} \tag{5.37}$$

Si nous définissons la première itération comme le calcul des arcs *points-to* du corps de la boucle en fonction de la condition, nous pouvons réécrire l'équation récurrente ainsi :

$$\begin{aligned} Y_1 &= \mathcal{B}\mathcal{C} \\ Y_i &= Y_{i-1} \sqcup \mathcal{B}\mathcal{C}Y_{i-1} \end{aligned} \tag{5.38}$$

Plus concrètement, si nous notons Out l'ensemble final des arcs *points-to*, il peut aussi être écrit en fonction de la dernière itération sur le corps de la boucle(atteinte du point fixe), notée Y_{final}, et de l'analyse de la condition en l'évaluant à faux, notée $\bar{\mathcal{C}}$. Ceci est traduit par l'équation (5.39) :

$$\text{Out} = \bar{\mathcal{C}}Y_{final} \tag{5.39}$$

Le résultat final de notre algorithme, qui est détaillé à la sous-section 5.6.6, est illustré par le programme 5.13 où l'on remarque le cycle créé par l'arc `_p_1_2__1.next->_p_1_2__1`. La longueur maximale d'un chemin est fixée à 2 (ou 3 ?, voir la valeur par défaut de la propriété). Elle est atteinte par le chemin `p->_p_1,_p_1.next->_p_1_2__1,_p_1_2__1.next->_p_1_2__1`.

Nous remarquons que les arcs *points-to* obtenus en sortie de boucle contiennent un arc exact du pointeur `p` vers l'emplacement `NULL` et la suppression de tous les autres arcs dont la source était `p`. Ceci s'explique par le fait que la condition de la boucle est évaluée en utilisant l'information *points-to* ; si on quitte le corps de boucle alors la condition est évaluée à faux et le pointeur `p` vaut `NULL`. Nous remarquons aussi que tous les chemins de préfixe `_p_1` (sauf lui même) ont été fusionnés en un seul noeud `_p_1_2__1`.

5.6.3.3 Exemple 2

L'exemple 5.15 montre une évolution du terme droit, `tab[i]`, pour lequel nous allons appliquer le même traitement en déroulant la boucle.

En effet, le même problème peut se poser pour le terme gauche d'une affectation, comme nous pouvons le constater au niveau du programme 5.15 qui reprend la même structure de données que l'exemple précédent avec un champ de type pointeur sur un entier val. Le programme affecte, à chaque itération de boucle, un élément du tableau tab à val comme ci-dessous où pour une meilleure lisibilité nous ne gardons que les arcs associés au champ `val`.

Comme l'information sur les indices contenus dans un arc *points-to* est indépendante de l'état mémoire courant d'après la définition du treillis $Vref$, une modification de l'indice de boucle est effectuée. Cette représentation est propre au treillis $Vref$: chaque indice non constant est remplacé par « $*$ », où « $*$ » désigne n'importe quel entier appartenant à l'intervalle $[0, 19]$. Cette transformation est définie par (4.10). Cette abstraction est fournie et visible au niveau du treillis CP (voir figure 5.7). Elle permet de sur-approximer les indices de tableaux quand cette information n'est pas disponible. Ceci est aussi discuté section 5.6.4. Le résultat de notre algorithme est illustré par le programme 5.16.

```
int count(list *p){
   int i = 0;

// Points To:
// p -> *NULL_POINTER* , MAY
// p -> _p_1 , MAY
   while (p!=(void *) 0) {
// Points To:
// _p_1.next -> *NULL_POINTER* , MAY
// _p_1.next -> _p_1_2__1 , MAY
// _p_1_2__1.next -> *NULL_POINTER* , MAY
// _p_1_2__1.next -> _p_1_2__1 , MAY
// p -> _p_1 , MAY
// p -> _p_1_2__1 , MAY
      i++;
// Points To:
// _p_1.next -> *NULL_POINTER* , MAY
// _p_1.next -> _p_1_2__1 , MAY
// _p_1_2__1.next -> *NULL_POINTER* , MAY
// _p_1_2__1.next -> _p_1_2__1 , MAY
// p -> _p_1 , MAY
// p -> _p_1_2__1 , MAY
      p = p->next;
   }
// Points To:
// _p_1.next -> *NULL_POINTER* , MAY
// _p_1.next -> _p_1_2__1 , MAY
// _p_1_2__1.next -> *NULL_POINTER* , MAY
// _p_1_2__1.next -> _p_1_2__1 , MAY
// p -> *NULL_POINTER* , EXACT
   return i;
}
```

Prog 5.13 – Les arcs *points-to* pour le programme 5.12

```
typedef struct LinkedList{
  int *val;
  struct LinkedList *next;
} list;
```

Prog 5.14 – Définition de la structure de données

```
int tab[20];
int init(list* p)
{ int i = 0, j;
  for(j=0; j<20; j++){
    tab[j] = j
  }
  while(p != NULL){
    p->val = &tab[i];
    p = p->next; i++;
  }
  return i;
}
```

Prog 5.15 – Exemple de liste chaînée : champ valeur de type pointeur

$$\begin{aligned}
\text{pour } i = 1:\ Y_1 &= \{(p, _p_1, \mathsf{MAY}), (p, _p_1_1, \mathsf{MAY}), (p, \texttt{NULL}, \mathsf{MAY}),\\
&\quad (_p_1.val, tab[*], \mathsf{MAY}), (_p_1.val, \texttt{NULL}, \mathsf{MAY})\}\\
\text{pour } i = 2:\ Y_2 &= \{(p, _p_1, \mathsf{MAY}), (p, _p_1_1, \mathsf{MAY}), (p, \texttt{NULL}, \mathsf{MAY}), (p, _p_1_1_1, \mathsf{MAY}),\\
&\quad (_p_1.val, tab[*], \mathsf{MAY}), (_p_1_1.val, tab[*], \mathsf{MAY}), (_p_1.val, \texttt{NULL}, \mathsf{MAY}),\\
&\quad (_p_1_1.val, \texttt{NULL}, \mathsf{MAY})\}\\
\text{pour } i = n:\ Y_n &= \{(p, _p_1^n, \mathsf{MAY}), (p, _p_1^{n-1}, \mathsf{MAY}), (p, _p_1^{n-2}, \mathsf{MAY}), (p, \texttt{NULL}, \mathsf{MAY}),\\
&\quad (_p_1^n.val, tab[*], \mathsf{MAY}), (_p_1^{n-1}.val, tab[*], \mathsf{MAY})\\
&\quad (_p_1^n.val, \texttt{NULL}, \mathsf{MAY}), (_p_1^{n-1}.val, \texttt{NULL}, \mathsf{MAY})...\}
\end{aligned}$$

5.6.3.4 Exemple 3

Le troisième exemple illustre une allocation dynamique au sein d'une boucle *while* (voir programme 5.17).

Pour les approximations des arcs *points-to* au niveau des boucles, nous utilisons un opérateur $merge$ qui est implémenté par la fonction 20). Il modifie certaines approximations en « MAY », puisqu'il n'est toujours pas possible de vérifier que la boucle va être exécutée ou non. Une amélioration de l'algorithme consisterait à utiliser les préconditions pour raffiner, en itérant sur le corps de la boucle, l'analyse au niveau des structures de contrôle. Une autre solution, déjà intégrée à l'analyse, est d'utiliser l'information *points-to* pour évaluer les conditions sur les valeurs du pointeur. Par exemple une condition de type `if(p == NULL)` peut être évaluée à faux si l'information *points-to* ne contient pas d'arc de `p` vers l'emplacement `NULL`.

Fonction 20 : merge

merge : $\mathcal{PT} \times \mathcal{PT} \rightarrow \mathcal{PT}$
let $P_1 \in \mathcal{PT}$
let $P_2 \in \mathcal{PT}$
let $P_{\mathsf{EXACT}} = P_{1\mathsf{EXACT}} \sqcap P_{2\mathsf{EXACT}}$
let $P_{\mathsf{MAY}} = (P_{1\mathsf{MAY}} \sqcup P_{2\mathsf{MAY}} \sqcup P_{1\mathsf{EXACT}} \sqcup P_{2\mathsf{EXACT}}) - P_{\mathsf{EXACT}}$
return $P_{\mathsf{EXACT}} \sqcup P_{\mathsf{MAY}}$

5.6.4 Cas de la boucle *for*

Une boucle *for* est constituée d'une expression d'initialisation, d'une condition, d'un incrément et d'un corps sous la forme d'un ensemble d'instructions. Pour le calcul des arcs *points-to* au niveau d'une boucle «for », nous appelons la fonction `forloop_to_points_to()` qui commence par calculer les arcs au niveau des conditions d'entrée puis itère sur le corps de la boucle jusqu'à atteindre un point fixe, comme nous l'avons expliqué pour la boucle *while*.

Ce nouvel exemple (programme 5.19) est intéressant parce que la condition de la boucle contient une affectation de pointeurs et son corps une opération arithmétique de pointeurs. Comme expliqué à la section 4.6.1.3, cet opération d'arithmétique sur un pointeur a pour résultat de faire pointer le pointeur vers l'élément suivant. Ceci se traduit par le programme 5.20.

La propriété `POINTS_TO_SUBSCRIPT_LIMIT`, définie à la section 5.7.3 et qui vaut 4, est utilisée pour limiter le nombre de nœuds que nous créons.

```
int init(list *p)
{
  int i = 0, j, tab[20];
  for(j = 0; j <= 19; j += 1)
    tab[j] = j;
  // Points To:
  // p -> *NULL_POINTER* , MAY
  // p -> _p_1 , MAY

  while (p!=(void *) 0) {
    // Points To:
    // _p_1.next -> *NULL_POINTER* , MAY
    // _p_1.next -> _p_1_2__1 , MAY
    // _p_1.val -> tab[*] , MAY
    // _p_1_2__1.next -> *NULL_POINTER* , MAY
    // _p_1_2__1.next -> _p_1_2__1 , MAY
    // _p_1_2__1.val -> tab[*] , MAY
    // p -> _p_1 , MAY
    // p -> _p_1_2__1 , MAY

    p->val = &tab[i];
    // Points To:
    // _p_1.next -> *NULL_POINTER* , MAY
    // _p_1.next -> _p_1_2__1 , MAY
    // _p_1.val -> tab[*] , MAY
    // _p_1_2__1.next -> *NULL_POINTER* , MAY
    // _p_1_2__1.next -> _p_1_2__1 , MAY
    // _p_1_2__1.val -> tab[*] , MAY
    // p -> _p_1 , MAY
    // p -> _p_1_2__1 , MAY

    p = p->next;
    // Points To:
    // _p_1.next -> *NULL_POINTER* , MAY
    // _p_1.next -> _p_1_2__1 , MAY
    // _p_1.val -> tab[*] , MAY
    // _p_1_2__1.next -> *NULL_POINTER* , MAY
    // _p_1_2__1.next -> _p_1_2__1 , MAY
    // _p_1_2__1.val -> tab[*] , MAY
    // p -> *NULL_POINTER* , MAY
    // p -> _p_1_2__1 , MAY

    i++;
  }
  // Points To:
  // _p_1.next -> *NULL_POINTER* , MAY
  // _p_1.next -> _p_1_2__1 , MAY
  // _p_1.val -> tab[*] , MAY
  // _p_1_2__1.next -> *NULL_POINTER* , MAY
  // _p_1_2__1.next -> _p_1_2__1 , MAY
  // _p_1_2__1.val -> tab[*] , MAY
  // p -> *NULL_POINTER* , EXACT
  return i;
}
```

Prog 5.16 – Les arcs *points-to* pour le programme 5.15

```
#include<stdlib.h>
#include<stdio.h>
#include<stdbool.h>

typedef struct LinkedList{
  int val;
  struct LinkedList *next;
} list;

list * initialize()
{
  list *first = NULL, *previous = NULL;
  bool break_p = false; // added to avoid an untructured...
  while(!feof(stdin) && !break_p){
    list * nl = (list *) malloc(sizeof(list));
    nl->next = NULL;
    if(scanf("%d",&nl->val)!=1)
      break_p = true;
    if(first == NULL)
      first = nl;
    if(previous != NULL)
      previous->next = nl;
    previous = nl;
  }
  return first;
}
```

Prog 5.17 – Exemple de liste chaînée: allocation dynamique

5.6.5 La boucle *do...while*

La boucle *do...while* est différente des autres boucles parce que le corps doit être analysé au moins une fois. Le résultat de cette itération est fourni comme ensemble d'entrée aux itérations suivantes et la même fonction est utilisée pour atteindre un point fixe grâce à la finitude du treillis. En d'autres termes, la construction $do\ b\ while(C)$ est analysée sous la forme «$b; while(C)\ b$».

5.6.6 L'algorithme général `any_loop_to_points_to()`

Deux algorithmes ont été définis et testés expérimentalement. Le premier consiste à propager l'information *points-to* cumulée depuis la première itération, comme cela a été présenté sur des exemples. Le second consiste à calculer d'abord la k-ième itérée puis à l'ajouter à la précondition du corps de boucle. Le second algorithme devait être plus rapide, puisqu'il évitait les re-calculs massifs impliqués par le premier, mais il a posé des problèmes de convergence qui n'ont pas été résolus. Nous avons donc décidé d'utiliser le premier algorithme.

L'algorithme reçoit quatre arguments qui sont le corps de la boucle et trois expressions. Ces dernières sont définies par rapport à la nature de la boucle analysée ; si c'est une boucle *while* alors les expressions d'initialisation et d'incrément ne sont pas définies. Le prédicat `expression_undefined_p` permet de tester si une expression est définie.

L'algorithme commence par calculer les arcs *points-to* au niveau des expressions d'initialisation, si elles sont bien définies, pour les fournir comme information en entrée pour l'analyse du corps de boucle. Avant d'analyser ce dernier nous commençons par récupérer la nombre maximal d'itérations à effectuer pour analyser le corps de boucle. Ce nombre est fonction des différentes limites imposées à l'analyse par les propriétés, il doit leur être supérieur d'où le +10 lors de l'initialisation de k :

```
list * initialize()
{
   list *first = (void *) 0, *previous = (void *) 0;
   _Bool break_p = 0;

// Points To:
// first -> *NULL_POINTER* , EXACT
// previous -> *NULL_POINTER* , EXACT
    while (!feof(stdin)&&!break_p) {
      list *nl = (list *) malloc(sizeof(list));
      nl->next = (void *) 0;
      if (scanf("%d", &nl->val)!=1)
         break_p = 1;
      if (first==(void *) 0)
         first = nl;
      if (previous!=(void *) 0)
         previous->next = nl;

// Points To:
// *HEAP*_l_17.next -> *HEAP*_l_17 , MAY
// *HEAP*_l_17.next -> *NULL_POINTER* , MAY
// first -> *HEAP*_l_17 , MAY
// nl -> *HEAP*_l_17 , MAY
// previous -> *HEAP*_l_17 , MAY
// previous -> *NULL_POINTER* , MAY
      previous = nl;
   }

// Points To:
// *HEAP*_l_17.next -> *HEAP*_l_17 , MAY
// *HEAP*_l_17.next -> *NULL_POINTER* , MAY
// first -> *HEAP*_l_17 , MAY
// first -> *NULL_POINTER* , MAY
// previous -> *HEAP*_l_17 , MAY
// previous -> *NULL_POINTER* , MAY
   return first;
}
```

Prog 5.18 – Les arcs *points-to* pour le programme 5.17

```
int main(){
 float *a, *p;
 int i;
 a = (float *) malloc(10* sizeof(float));
 for(p = a, i=0; i<10; p++)
   *p = 1.0;
 return(0);
}
```

Prog 5.19 – Exemple boucle *for*

```
int main(){
   float *a, *p;
   int i;
   a = (float *) malloc(10*sizeof(float));
   p = a, i = 0;
// Points To:
// a -> *HEAP*_l_8[0] , MAY
// p -> *HEAP*_l_8[0] , MAY
   while (i<10) {
// Points To:
// a -> *HEAP*_l_8[0] , MAY
// p -> *HEAP*_l_8[*] , MAY
// p -> *HEAP*_l_8[0] , MAY
// p -> *HEAP*_l_8[1] , MAY
// p -> *HEAP*_l_8[2] , MAY
// p -> *HEAP*_l_8[3] , MAY
      *p = 1.0;
// Points To:
// a -> *HEAP*_l_8[0] , MAY
// p -> *HEAP*_l_8[*] , MAY
// p -> *HEAP*_l_8[0] , MAY
// p -> *HEAP*_l_8[1] , MAY
// p -> *HEAP*_l_8[2] , MAY
// p -> *HEAP*_l_8[3] , MAY
      p++;
   }
// Points To:
// a -> *HEAP*_l_8[0] , MAY
// p -> *HEAP*_l_8[*] , MAY
// p -> *HEAP*_l_8[0] , MAY
// p -> *HEAP*_l_8[1] , MAY
// p -> *HEAP*_l_8[2] , MAY
// p -> *HEAP*_l_8[3] , MAY
   return 0;
}
```

Prog 5.20 – Les arcs *points-to* pour le programme 5.19

Fonction 21 : any_loop_to_points_to($\mathcal{S} : b, \mathcal{E} : c, \mathcal{E} : init, \mathcal{E} : inc, \mathcal{PT}$: In)

```
PT : Out, In, Prev, Skip
int : i, k
B : fix_point_p
Out = In
k = get_int_property("POINTS_TO_PATH_LIMIT") +
get_int_property("POINTS_TO_SUBSCRIPT_LIMIT") +
get_int_property("POINTS_TO_OUT_DEGREE_LIMIT") + 10;
if expression_defined_p(init) then
    Out = T_PT[[init]](Out)
Skip = Out
Out = ntp(c, true, Out)
Skip = ntp(c, false, Skip)
fix_point_p = false
for i = 0; i ≤ k; i + + do
    Prev = Out
    Out = T_PT[[b]](Prev)
    if expression_defined_p(inc) then
        Out = T_PT[[inc]](Out)
    if expression_defined_p(c) then
        Out = ntp(c, true, Out)
    Out = merge(Prev, Out)
    if Out = Prev then
        fix_point_p = true
        Out = T_PT[[b]](Prev)
        if expression_defined_p(inc) then
            Out = T_PT[[inc]](Out)
        if expression_defined_p(c) then
            Out = ntp(c, false, Out)
        break
if ¬fix_point_p then
    return error
Out = merge(Out, Skip)
```

- `POINTS_TO_SUBSCRIPT_LIMIT`,
- `POINTS_TO_OUT_DEGREE_LIMIT`,
- `POINTS_TO_PATH_LIMIT`.

Ensuite, nous itérons sur le corps de boucle en calculant les arcs *points-to* générés par les instructions du corps via la fonction `statement_to_points_to()` et, si elle existe, par l'expression d'itération. Cet ensemble d'arcs est enfin filtré en recalculant à nouveau l'impact de la condition de continuation, si elle existe. Cet ensemble mis à jour est comparé à celui obtenu à l'itération précédente : si c'est le même, alors le point fixe est atteint et l'analyse traite la sortie de boucle, sinon le calcul continue. Des opérations de normalisation et de suppression d'arcs non atteignables sont effectuées sur le graphe *points-to* avant de tester l'égalité des résultats de deux itérations successives. Si l'itération k est atteinte alors que le point fixe ne l'est toujours pas, l'analyse s'arrête avec un message d'erreur. Avant de sortir de la boucle, les indices des éléments de tableaux qui dépendent de l'indice de boucle sont changés en « * » via la fonction sti définie précédemment à la section 4.3.3.4. Une fois la convergence atteinte, le corps de boucle est analysé à nouveau avec comme information *points-to* en entrée le résultat de l'itération précédente. Ce dernier passage permet d'associer les arcs *points-to* à chaque instruction du corps de boucle. Après ce dernier passage et en sortie de boucle les acrs *points-to* sont filtrés avec l'analyse de la condition évaluée à faux. Ensuite l'opérateur $merge$ (programme 20) est appliqué pour fusionner les arcs *points-to* en entrée avec ceux générés par la boucle.

Enfin, il faut prendre en compte le cas où le corps de la boucle n'est pas analysé parce que la condition d'entrée est évaluée à faux. Dans ce cas, l'information *points-to* en entrée, c'est-à-dire celles rattachée à la condition, évaluée à faux, notée $Skip$ qui est propagée sur les instructions du corps de boucle.

5.6.7 Conclusion sur les boucles

Moyennant des différences mineures, toutes les boucles structurées de C sont traitées par une unique fonction qui prend en compte la présence ou non des trois expressions définissant l'entête des boucles `for`. Seule la boucle `do...while()` nécessite un traitement préalable différent.

5.6.8 Cas des graphes de flot de contrôle, « $unstructured$ »

Au niveau de la représentation interne de PIPS, un code non structuré est représenté par un graphe de flot de contrôle et déclaré en Newgen par :

```
unstructured = entry:control x exit:control ;
control = statement x predecessors:control* x successors:control* ;
```

Prog 5.21 – Structure de données pour les graphes de contrôle

Cette structure de données est utilisée pour isoler les parties non structurées du code des parties structurées. En effet, le graphe est représenté comme un unique *statement* d'une composante structurée. Le contenu de ce *statement*, appelé *instruction*, est un graphe avec un nœud d'entrée et un unique nœud de sortie (voir l'exemple de la figure 5.12). Chaque nœud n est implémenté par une structure *control* qui a d'éventuels successeurs et prédécesseurs, les arcs sortants et entrants, ainsi qu'une étiquette, un *statement*, qu'on peut récupérer via la fonction `control_statement` qui est dénoté s_n. Un code non structuré présente plusieurs difficultés. En effet, il peut représenter un graphe acyclique ou un graphe cyclique, tout comme il peut contenir des nœuds non-atteignables.

5.6.8.1 Analyse insensible au flot de contrôle

Nous commençons par traiter le cas où le graphe contient un cycle. Pour cela nous ajoutons un arc de retour du contrôle exit $c5$ au contrôle entry $c1$ au DAG [8] de la figure 5.12. L'algorithme décrit par la fonction 22 est insensible au flot ; c'est-à-dire que les nœuds (dits aussi contrôle) sont traités sans tenir compte de leur ordre d'exécution. Pour cela nous ajoutons deux nœuds fictifs au graphe, un en entrée noté α et un de sortie noté ω, n'ayant pas d'effet sur la relation *points-to*. La première étape est de calculer les arcs *points-to* pour tous les nœuds en utilisant la même information, celle en entrée du graphe In_u. Ceci est modélisé par les arcs verts de la figure 5.13. Ensuite via les arcs bleus, tous les nœuds convergent tous vers le nœud de sortie ω et c'est l'arc rouge qui part de ce dernier pour effectuer un retour vers le nœud d'entrée α qui permet d'itérer sur les nœuds jusqu'à atteinte du point fixe en utilisant chaque fois le résultat de l'itération précédente pour calculer celui de l'itération en cours.

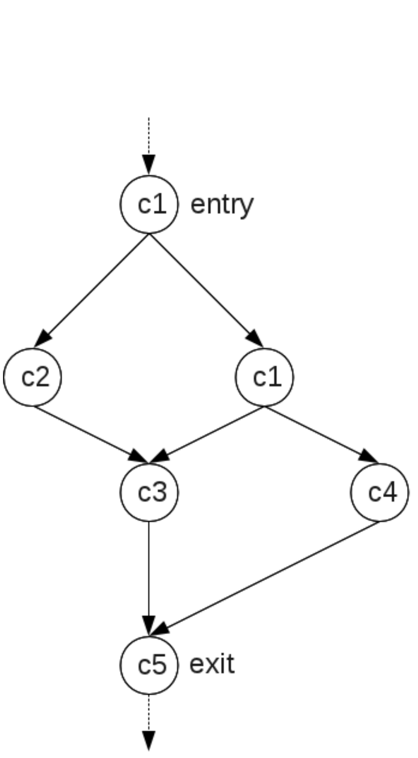


FIGURE 5.12 – Exemple de graphe non structuré sans cycle

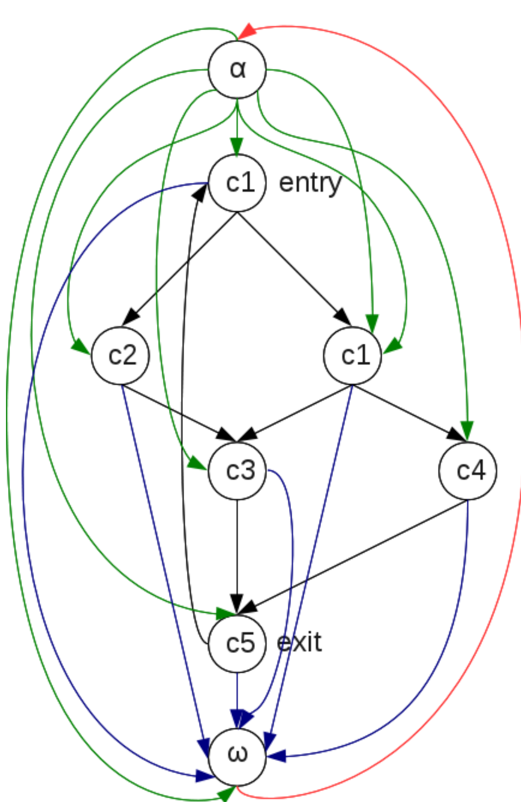


FIGURE 5.13 – Exemple de graphe non structuré complet pour la fonction 22

La fonction $pt_mapping(s_c)$ permet d'associer au niveau du statement s_c l'information *points-to* qui est la même pour tous les nœuds puisqu'ils sont tous connectés via les nouveaux arcs verts, bleus et rouges.

5.6.8.2 Analyse partiellement sensible au flot de contrôle

Un graphe de contrôle peut ne pas contenir de cycle (DAG) ou bien débuter par un DAG, comme par exemple le graphe de la figure 5.12. Il est donc intéressant de définir un algorithme

8. Directed acyclic graph.

Fonction 22 : points_to_cyclic_graph(graph G(V,E), $\mathcal{PT}$ In)

points_to_cyclic_graph(graph $G(V, E)$, $\mathcal{PT}$ In)
$\text{Out}_0 = \bigsqcup_{c\in V} T_{\mathcal{PT}}(s_c, \text{In})$
i = 0
repeat
  i++
  $\text{Out}_i = \bigsqcup_{c\in V} T_{\mathcal{PT}}(s_c, \text{Out}_{i-1})$
until $Out_i = Out_{i-1}$
$\text{Out}_u = \text{Out}_i$
$\forall c \in V pt_mapping(s_c) = \text{Out}_u$

sensible au flot pour les DAG, quitte à utiliser l'algorithme insensible au flot (fonction 22) pour le reste du graphe.

L'algorithme 23 est un algorithme général qui permet de traiter tous les cas possibles cités précédemment. Cet algorithme prend en entrée le graphe G et une relation *points-to* In_u. Il commence par récupérer tous les nœuds atteignables depuis le nœud d'entrée dans l'ensemble $Reachable$. Ensuite un test est effectué pour vérifier que le nœud d'entrée, $entry$, n'appartient pas à un cycle. Une fois que nous avons vérifié que $entry$ n'appartient pas à un cycle, il est ajouté à l'ensemble des nœuds prêt à être traités, $rtbp$ (*ready to be processed*). L'ensemble In_n, l'information *points-to* en entrée du premier nœud, $entry$, est initialisée à In_u. Ensuite, pour un nœud n prêt à être traité, l'ensemble In_n est calculé en faisant l'union de son ancienne valeur avec l'ensemble des relations *points-to* générées au préalable par l'analyse, via la fonction $u_ntp()$ de son prédécesseur p. Ce dernier doit être atteignable et la fonction $u_ntp()$ (voir la fonction 24) permet de couvrir tous les cas d'instructions. En effet, s'il s'agit d'un test comme c'est le cas du contrôle $c1$ de la figure 5.12 la fonction permet d'évaluer la condition du test en utilisant l'information *points-to*. Les arguments de la fonction sont :

1. s_p le statement du contrôle ;
2. $which_branch_p(p, n)$ qui teste s'il s'agit de la branche vrai ou faux du test ;
3. $Post(p)$ qui permet récupérer l'information *points-to* associée au nœud p.

Ensuite c'est au tour du contrôle n d'être analysé via le transformeur $T_{\mathcal{PT}}$ et l'information *points-to* est sauvegardée au niveau du statement associé via la fonction $pt_mapping()$. Le nœud est ajouté après à l'ensemble $Processed$ des nœuds de contrôles qui ont été traités. L'ensemble $rtbp$ est mis à jour en y ajoutant les successeurs de n dont les prédécesseurs atteignables ont déjà été analysés. Quand cet ensemble $rtbp$ est vide, les contrôles restants sont des nœuds appartenant à un cycle ; ils sont traités via la fonction 22 d'une façon insensible au flot, avec comme information *points-to* en entrée l'ensemble In_c. Ce dernier correspond à l'information *points-to* de tous les prédécesseurs des nœuds appartenant au cycle. La fonction se termine par l'analyse du nœud de sortie $exit$.

5.7 Finitude de la représentation/abstraction de la relation *points-to*

La relation *points-to* est représentée sous la forme d'un graphe où les arcs représentent la relation et les nœuds les emplacements mémoire. Ces derniers comportent les variables définies par le programme ainsi que des emplacements générés par notre analyse pour modéliser le contexte formel d'appel et le tas. Le nombre des emplacements peut augmenter considérable-

Fonction 23 : points_to_graph(graph G(V,E), $\mathcal{PT}$ In_u)

node $entry$ = graph_entry(G)
node $exit$ = graph_exit(G)
`Recursive definition of the set Reachable`
$Reachable = \{n \in G | n = entry \vee Pred(n) \subset Reachable\}$
`All these set are initialized to empty by default`
$rtbp, rtbp', Processed$: set
$Post : mapping$
if $Reachable \cap Pred(entry) = \emptyset$ **then**
  $rtbp = \{entry\}$
else
  $rtbp = \emptyset$
while $rtbp \neq \emptyset$ **do**
  $rtbp' = rtbp$
  for $n \in rtbp$ **do**
    if $n = entry$ **then**
      $\text{In}_n = \text{In}_u$
    else
      $\text{In}_n = \emptyset$
    for $p \in Pred(n) \cap Reachable$ **do**
      $\text{In}_n = \text{In}_n \sqcup u_ntp(s_p, which_branch_p(p, n), Post(p))$
    $Post(n) = T_{\mathcal{PT}}(s_n, \text{In}_n)$
    $Processed = Processed \cup \{n\}$
    $pt_mapping(s_n) = \text{In}_n$
    $rtbp' = rtbp' - \{n\} \cup \{m \in Succ(n) | Pred(m) \cap Reachable \subseteq Processed\}$
  $rtbp = rtbp'$
$Cycle = Reachable - Processed$
$Entries = \bigcup_{c \in Cycle} Pred(c) - Cycle$
$\text{In}_c = \bigsqcup_{c \in Entries} Post(c)$
Out = points_to_cyclic_graph($Cycle, \text{In}_c$)
return $Post(exit)$

Fonction 24 : u_ntp($s : \mathcal{S}, b : \mathcal{B}, P : \mathcal{PT}$)

$u_ntp : \mathcal{S} \times \mathcal{B} \times \mathcal{PT} \rightarrow \mathcal{PT}$
if $test_p(s)$ **then**
  $ntp(test_cond(s), b, P)$
else
  P

ment quand des boucles sont analysées [9]. La taille du graphe doit être contrôlée par le biais de l'ensemble des nœuds ou de l'ensemble des chemins pour garantir des propriétés importantes de l'analyse comme la convergence et la finitude du graphe.

Potentiellement, le nombre de noeuds peut croître *via* trois mécanismes différents : l'allocation dynamique, le déréférencement de pointeurs du contexte formel et l'arithmétique sur pointeurs. Le modèle du tas ne permet pas de générer plus de noeuds qu'il n'y a de lignes dans le programme analysé. Il reste donc à limiter la génération de nouveaux *stubs* formels, quand un déréférencement est appliqué, et la génération de nouveaux indices, quand l'arithmétique sur pointeur est utilisée (section 5.7.3). Dans le cas où l'on s'interdit de générer un nouveau *stub*, il faut néanmoins générer un nouvel arc. Deux possibilités ont été étudiées : l'utilisation d'une adresse abstraite, typée ou non, par exemple `anywhere`, ou la création d'un cycle avec un arc arrière. La première solution semble augmenter l'imprécision de l'analyse. La second a l'inconvénient de transformer certains *stubs* en emplacement abstrait, *i.e.* de leur faire perdre leur atomicité.

La détection d'une limite peut se faire soit en mesurant la longueur du chemin le plus long qui serait généré par la création d'un nouveau *stub* (section 5.7.1), soit en mesurant la longueur du chemin le plus long dont l'origine est un paramètre formel (section 5.7.2). La première solution a été retenue, mais aucun avantage clair n'a été trouvé en sa faveur.

5.7.1 Nombre de déréférencements

Le nombre de nœuds peut être borné en limitant le nombre de déréférencements effectués à partir d'un paramètre formel p.

Ce nombre peut être mesuré de deux manières différentes, soit en reconstruisant tous les chemins permettant de remonter à un paramètre formel, soit en étudiant la structure du nom du *stub* pour lequel il faut trouver un successeur. Cette deuxième solution devrait être la moins complexe puisqu'elle n'impose aucune traversée du graphe de la relation *points-to*.

Bien que cette approche ait l'avantage de l'efficacité, nous avons préféré utiliser une mesure plus intrinsèque au graphe, la longueur du chemin le plus long.

5.7.2 Longueur des chemins dans le graphe

À cause des structures de données récursives, présentes dans le langage `C` et utilisées potentiellement dans les applications scientifiques, la longueur des chemins d'accès mémoire peut augmenter considérablement. Cette augmentation de la longueur peut être due, par exemple, à un parcours de liste chaînée (voir le programme 5.12). L'inconvénient de ces listes longues, outre que le calcul des boucles peut ne pas se terminer comme expliqué section 5.6.3, est qu'elles dégradent les performances de l'analyse sans forcément en améliorer la qualité. En effet, l'espace mémoire nécessaire pour les stocker augmente ainsi que le temps nécessaire pour les parcourir chaque fois qu'il y a une mise à jour à effectuer.

Pour ces différentes raisons nous avons défini la propriété `POINTS_TO_PATH_LIMIT` qui contrôle la longueur maximum que peut atteindre un chemin du graphe *points-to* dans notre analyse. Si la longueur du chemin prolongé par un nouveau *stub* dépasse cette limite, ce *stub* est remplacé soit par un emplacement abstrait correspondant soit par un autre stub de même type se trouvant dans le préfixe du chemin. L'emplacement abstrait est soit le sommet du treillis, `*ANYWHERE*`, soit la zone mémoire `*HEAP*` quand le chemin initial a été alloué au niveau du tas.

Une seule solution est actuellement implémentée, la création d'un cycle.

9. Les appels récursifs pourraient avoir le même effet, mais les appels de procédures ne sont traités qu'au chapitre suivant. Nous n'abordons toutefois pas les appels récursifs peu utilisés dans les applications que nous traitons dans cette thèse.

5.7.3 Nombre de nœuds du graphe *points-to*

Nous avons besoin de contrôler aussi les indices utilisés dans les nœuds du graphe *points-to* pour garantir sa finitude. Prenons comme exemple un programme qui initialise tous les éléments d'un tableau de pointeurs.

```
int *a[10];
int i = 0;
a[0] = &i;
a[1] = &i;
a[2] = &i;
a[3] = &i;
a[4] = &i;
a[5] = &i;
a[6] = &i;
```

Prog 5.22 – Initialisation des éléments d'un tableau de pointeurs

En résultat de l'analyse *points-to* nous allons avoir tous les nœuds qui correspondent aux éléments du tableau avec en plus l'élément `a[*]` qui est correspond à l'abstraction de tous les autres éléments nécessaire lors de la déclaration du tableau. Dans le cas du programme 5.22, il s'agit d'un tableau de seulement dix éléments. Lors d'une opération d'arithmétique sur pointeurs, nous devons aussi contrôler le nombre de chemins constants créés (voir le programme 5.19). Pour plus de sûreté, à partir d'une certaine valeur définie par la propriété `POINTS_TO_SUBSCRIPT_LIMIT` , les nouveaux indices sont fusionnés en un seul. Ce dernier est l'élément « * » qui représente n'importe quel élément du tableau pointé par le tableau.

5.7.4 Nombre d'arcs *points-to* sortants d'un nœud

Un nœud dans le graphe *points-to* peut avoir plusieurs destinations, voire de très nombreuses destinations dans le cas ou un pointeur unique a été utilisé pour parcourir une structure de données récursive, par définition non-bornée à priori. Le but de la propriété `POINTS_TO_OUT_DEGREE_LIMIT` est de limiter le nombre d'arcs sortants d'un nœud qui peut croître indéfiniment dans le cas de la libération d'une liste. Cette limitation du graphe se fait grâce au treillis CP. En effet, le treillis permet de fusionner les nœuds cibles en un seul qui est l'emplacement abstrait le plus petit contenant tous les nœuds. Le programme 5.23 illustre l'utilisation de cette propriété. En sortie de

```
struct cons_t {
  double value;
  struct cons_t * next;
};
typedef struct cons_t * list;
const list nil = ((list) 0);
void list_free(list l)
{ list n = l, p = l;
  while (n!=nil) {
    n = n->next;
    free(p);
    p = n;
  }
  return;
}
```

Prog 5.23 – Libération d'une liste chaînée

```
// Points To:
// l -> *NULL_POINTER* , MAY
// l -> _l_1 , MAY
// n -> *NULL_POINTER* , EXACT
// nil -> *NULL_POINTER* , EXACT
// p -> *NULL_POINTER* , MAY
// p -> _l_1 , MAY
```

Prog 5.24 – Les arcs *points-to* pour le programme 5.23

la boucle *while*, avec une propriété `POINTS_TO_SUBSCRIPT_LIMIT` qui vaut 2, nous obtenons les arcs réduits du programme 5.24.

5.7.5 Conclusion

Nous avons montré dans cette section comment, grâce à des propriétés qui peuvent être définies par l'utilisateur, nous pouvons contrôler les caractéristiques suivantes du graphe *points-to* :

1. la longueur des chemins d'accès mémoire ;
2. le nombre de nœuds du graphe ;
3. le nombre d'arcs sortants d'un nœud ;
4. la convergence des algorithmes de traitement des boucles et des graphes de flot de contrôle.

Les sites d'appels constituent un autre risque d'explosion de la relation *points-to*. Il sera traité chapitre 6.

5.8 Conclusion

Dans le chapitre précédent, nous avons présenté l'idée générale de l'algorithme intraprocédurale de l'analyse *points-to* en utilisant deux langages sous-ensembles de C. Dans ce chapitre nous avons introduit le treillis des chemins mémoire constants qui assure la convergence de notre algorithme ainsi que la finitude du graphe *points-to*. Cette abstraction en treillis permet une modélisation pratique de la mémoire. Elle permet aussi de couvrir un langage plus riche que les langages $\mathcal{L}_0$ et $\mathcal{L}_1$ (voir la section 4.5.1) qui contient les boucles, l'arithmétique sur pointeurs et les graphes de flot de contrôle. Ces extensions présentent beaucoup de difficultés lors d'un calcul des arcs *points-to*. En utilisant le treillis ainsi que des propriétés qui assurent l'atteinte du point fixe, nous assurons une analyse finie des applications.

CHAPITRE 6

L'analyse interprocédurale : traitement des sites d'appels

Dans ce chapitre nous nous intéressons à l'aspect interprocédural de notre analyse. L'intérêt est de pouvoir traiter tout un programme contenant plusieurs fonctions [ASU86]. Le but final est de transmettre des informations entre les appelants et les appelés. Toutes les analyses ne sont pas forcément interprocédurales. Certaines par contre n'ont de sens que si elles le sont, comme par exemple le calcul du graphe des appels ou l'analyse de pointeurs. Pour cette dernière, après le calcul interprocédural des arcs *points-to*, il est nécessaire de propager cette information d'une procédure à une autre, dans le but de permettre des optimisations du code, principalement la parallélisation. Cette propagation inclut aussi la traduction des arcs *points-to* de l'appelé vers le site d'appel et des paramètres effectifs vers les paramètres formels, ou la traduction inverse.

Les analyses interprocédurales présentent beaucoup de difficultés car le comportement d'une fonction dépend du contexte d'appel dans lequel elle est appelée. Un choix s'offre à nous. Il consiste à développer soit :

1. une analyse insensible au contexte : elle peut être réalisée par exemple par la création d'un super-graphe de flot de contrôle où les sites d'appels et les instructions de retour sont transformés en opérations de branchements. Des instructions d'assignation sont rajoutées pour affecter les valeurs des paramètres effectifs aux paramètres formels et la valeur de retour à la variable qui doit recevoir le résultat. Le graphe ainsi créé est analysé intraprocéduralement. Cette analyse offre l'avantage d'être simple mais souffre d'imprécision due à la non prise en compte du contexte et aux chemins irréalisables [Ste96] [And94].
2. une analyse sensible au contexte. Deux méthodes sont cités dans [ASU86] pour réaliser une analyse de pointeurs sensible au contexte :
   (a) le clonage de fonctions : créer un clone pour chaque fonction et chaque contexte puis appliquer l'analyse au code obtenu (cette solution a été réutilisée dans les travaux d'Emami [EGH94] et de Lam [ADLL05]) ;
   (b) la création de résumés, i.e. de *transformers* : un résumé, qui consiste en une description concise du comportement de la fonction, est calculé. Ce résumé peut être utilisé pour plusieurs sites d'appels tant qu'ils ont le même profil d'aliasing (cette solution a été réutilisée dans les travaux de Wilson [WHI95]). Quand différentes hypothèses d'aliasing sont vérifiées par l'ensemble des sites d'appel, le résumé de la fonction devient une association des conditions d'aliasing vers des descriptions de comportements.

D'autres solutions existent comme :

1. l'expansion [1] des fonctions ;
2. la décision de ne pas prendre en compte l'appel de la fonction lorsqu'il est sûr qu'elle n'a pas d'effets de bord sur les paramètres ou les variables globales, en d'autres termes ignorer les sites d'appel ;

1. En anglais « inlining »

3. l'utilisation du résultat d'un précédent appel [Int] ;
4. ou encore la réanalyse de la fonction à chaque site d'appel.

Il faut cependant se poser la question sur les enjeux d'implémenter une analyse interprocédurale qui peut certes augmenter significativement sa précision mais risque d'être une tâche fastidieuse et très coûteuse en termes de temps d'exécution et d'espace mémoire. Dans le cadre de cette thèse nous avons choisi de développer trois traitements différents des sites d'appel. Au niveau d'un site d'appel, il faut être capable de calculer les ensembles Kill et Gen. Comme il s'agit d'une analyse abstraite et que l'équation de calcul de la nouvelles relation points-to est décroissante par rapport a Kill et croissante par rapport à Gen, il faut disposer d'une sous-approximation de Kill et d'une sur-approximation de Gen. L'analyse minimale que nous proposons sous le nom `intraprocedural_points-to` consiste à choisir l'ensemble vide comme sous-approximation de Kill et l'élément maximal du treillis induit sur les arcs points-to par le treillis des chemins d'adresse constants, `ANYWHERE -> ANYWHERE`.

La deuxième analyse, appelée `fast_interprocedural_analysis`, consiste aussi à prendre l'ensemble vide comme sous-approximation de Kill, et à calculer l'ensemble Gen à partir des chemins d'accès constant qui ont fait l'objet d'une écriture dans la procédure appelée. Cette information est donnée par la passe de calcul des effets (cf. sous-section 6.1.5.2), utilisant ou non l'information `points-to` de l'appelé [2].

Enfin, notre troisième analyse, `interprocedural_points_to_analysis`, présente des analogies avec l'analyse de Wilson [WHI95] dont nous n'avons malheureusement trouvé aucune présentation précise, y compris dans sa thèse. En l'absence d'appels récursifs [3], elle consiste à faire une analyse points-to ascendante sur le graphe des appels, pour calculer des *transformeurs* de *points-to*, les résumés de Wilson, et à vérifier lors des appels la condition de non-aliasing avant de les appliquer pour calculer les *points-to* au niveau de l'appelé. Le *transformeur* doit permettre de calculer une sous-approximation de l'ensemble Kill et une meilleure sur-approximation de l'ensemble Gen que nos deux analyses précédentes. Dans le cas où la condition d'aliasing n'est pas vérifiée, nous ne réanalysons pas la procédure appelée mais nous pouvons utiliser notre analyse intraprocédurale [4].

Ce chapitre est organisé de la manière suivante. Nous commençons par présenter plus en détail les différents types d'analyses interprocédurales possibles et leur impact sur l'ordonnancement des passes d'un compilateur, en l'occurrence PIPS (section 6.1). Nous faisons ensuite le bilan des difficultés qui doivent être traitées par un *transformeur* de *points-to* pour traiter précisément des appels de procédures (section 6.2). Nous décrivons ensuite notre schéma d'analyse interprocédurale et en particulier le mécanisme de traduction des `points-to` entre appelant et appelée (section 6.3.4), le test de non-aliasing (section 6.4) et les calculs des approximations ensembles Kill et Gen. Nous montrons ensuite la correction de l'ensemble des étapes de calcul (section 6.6). Enfin, nous montrons nos résultats sur quelques exemples (chapitre 7).

6.1 Les différentes approches pour le traitement des sites d'appels

Dans PIPS l'organisation des analyses en phases nous permet de programmer différentes approches pour le traitement des sites d'appel. Ces approches ont des rapports coût/performance variés et peuvent être sélectionnées selon les objectifs du programme ou la nature de

2. Le test de validité de l'hypothèse de non-aliasing n'est pas effectué. Les paramètres formels doivent donc porter le *qualifier restrict*. Nous interprétons *restrict* comme une garantie transitive de non-aliasing et non comme une garantie limitée au premier niveau, bien que ce soit ce qu'indique la norme C.

3. PIPS, notre cadre de développement, détecte les appels récursifs et interrompt par défaut toutes les analyses sur les parties de l'application concernée par les cycles récursifs.

4. En cas d'aliasing, notre analyse `fast_interprocedural_analysis` n'est pas correcte.

l'application. Le résultat de chacune de ces approches est décrit pour le programme 6.1. Nous

```
void swap01(int **p, int **q)
{
  int *pt = *p;
  *p = *q;
  *q = pt;
  return;
}

int main()
{
  int i = 1, j = 2, z = 3, *pi = &i, *pj = &j, *pz = &z, **ppi = &pi, **ppj = &pj;
  swap01(ppi, ppj);

  return 0;
}
```

Prog 6.1 – L'exemple interprocédural à analyser

avons choisi un programme avec des pointeurs de pointeurs afin d'illustrer l'effet de bord des fonctions sur les cases mémoire qui peuvent être visibles depuis le site d'appel. Cela illustre aussi l'importance de l'analyse interprocédurale.

6.1.1 L'approche insensible au contexte

Cette solution consiste à prendre l'ensemble vide comme Kill et à ignorer quelle fonction est appelée pour le calcul de Gen. L'ensemble Gen ne contient que deux éléments, l'élément maximal du treillis *points-to*, un arc `*ANYMODULE*:*ANYWHERE*->*ANYMODULE*:*ANYWHERE*` pour les pointeurs définis et l'arc `*ANYMODULE*:*ANYWHERE*->undefined` [5] pour les éventuelles libérations mémoire. Comme le premier arc est maximal, il contient tous les autres arcs et on peut donc choisir soit de préserver les arcs de l'ensemble In en minimisant leur approximation à *MAY*, soit de les retirer tous à cause de l'inclusion, ce qui peut donner l'illusion d'un ensemble *Kill* pris égal à l'ensemble In. L'existence de ces deux possibilités tient au fait qu'on n'a pas une simple structure de treillis, mais la combinaison d'une liste finie et d'un treillis. On voit bien qu'après un site d'appel ainsi traité, on souhaite pouvoir à nouveau générer une information *points-to* précise et qu'il ne faut donc pas considérer l'arc maximum comme absorbant. Le programme 6.2 montre le résultat d'une telle analyse.

6.1.2 Limitation de l'ensemble Gen aux paramètres et variables globales

Comme pour l'analyse précédente, l'ensemble Kill est l'ensemble vide. Mais l'ensemble Gen est calculé en fonction de la signature de la fonction, ce qui pose des problèmes au niveau des variables globales.

6.1.2.1 Le cas des paramètres formels

Le but est de calculer rapidement les arcs *points-to*. Les appels de fonctions vont remplacer les arcs impliquant soit le pointeur qui reçoit la valeur de retour, soit des pointeurs qui peuvent être modifiés par un effet de bord de la fonction ; dans le cas de notre exemple il s'agit des pointeurs `ppi`, `ppj`, `pi` et `pj`. Pour déterminer cette liste de pointeurs susceptibles d'être modifiés,

5. Pour le reste du chapitre et pour une meilleure lisibilité `undefined` remplace l'emplacement `*ANYMODULE*:undefined`

```
int main()
{
   int i = 1, j = 2, z = 3, *pi = &i, *pj = &j, *pz = &z, **ppi = &pi, **ppj = &pj;
// Points To:
// pi -> i , EXACT
// pj -> j , EXACT
// ppi -> pi , EXACT
// ppj -> pj , EXACT
// pz -> z , EXACT

   swap01(ppi, ppj);
// Points To:
// *ANY_MODULE*:*ANYWHERE* -> *ANY_MODULE*:*ANYWHERE* , MAY
// *ANYMODULE*:*ANYWHERE*-> undefined, MAY
// pz -> z , EXACT

   return 0;
}
```

Prog 6.2 – Les arcs *points-to* pour l'exemple interprocédural

les paramètres effectifs sont récupérés. Ensuite une vérification est effectuée pour tester s'il s'agit ou non de pointeurs de pointeurs qui peuvent être modifiés par le corps de la fonction. Dans l'exemple, les variables `ppi` et `ppj` sont des doubles pointeurs ; les arcs qui partent de nœuds contenants ces deux pointeurs sont remplacés par un arc maximal `*ANY_MODULE*:*ANYWHERE* -> *ANY_MODULE*:*ANYWHERE*`. Cet arc correspond au sommet du treillis des arcs $\mathcal{PT}$; comme il n'est pas possible de savoir à quel module appartient les nouvelles cibles de `pi` et `pj` alors le module est représenté par `*ANY_MODULE*`. L'effet de la fonction de libération de mémoire `free` pourrait être sur-approximé par l'élément `undefined`, mais le treillis de la section 5.3 ne le prévoit pas.

6.1.2.2 Le cas des variables globales

Il faudrait repartir de l'unité de compilation contenant la fonction et y rechercher les variables globales qui sont dans sa portée, ou bien considérer que toutes les variables globales de l'application sont potentiellement touchées. Les arcs *points-to* dont la source ou la cible est une variable globale de type pointeur doivent par conséquent être mis à jour.

6.1.2.3 Conclusion

Cette approche n'est pas satisfaisante à cause des variables globales qui conduisent à une forte perte de précision, quand les informations d'effet ne sont pas disponibles pour chaque fonction. Nous avons choisi d'utiliser les effets résumés de PIPS pour obtenir l'information nécessaire directement. C'est l'objet de la section suivante.

6.1.3 L'approche utilisant l'ensemble des adresses écrites par le site d'appel

Une autre approche, plus précise mais aussi plus coûteuse, consiste toujours à prendre l'ensemble vide comme ensemble Kill mais à utiliser les pointeurs qui ont été écrits par la fonction appelée pour limiter la taille de l'ensemble Gen. Le calcul de cet ensemble nécessite le précalcul d'une ressource de type effets cumulés, résumés au niveau d'une fonction donné par le

```
//           <   is read  >: _q_2 p q
//           <   is written>: _p_1 _q_2
```

Prog 6.3 – Les effets résumés pour le programme 6.1

programme 6.3. Cette ressource est calculée par la passe `EFFECTS_WITH_POINTS_TO` qui utilise l'information *points-to* de l'appelé et le mécanisme de traduction de la passe *points-to*. Cette ressource permet la création de l'ensemble des pointeurs qui ont été écrits au niveau de la fonction ; il est appelé Written. Les règles de calcul de cette ressource qui résulte du calcul interprocédural des arcs *points-to* de l'appelée seront détaillées dans la section 6.1.5. Retenons pour l'instant que pour l'exemple 6.1 et après traduction des effets résumés les pointeurs qui ont été écrits sont `pj` et `pi`.

Après l'appel à `swap01` les arcs *points-to* où ces deux pointeurs apparaissent comme sources sont remplacés, via l'ensemble Gen et les propriétés du treillis, par des arcs vers les emplacements `*ANY_MODULE*:*ANYWHERE*` et `undefined`. Ce dernier arc est le résultat d'un éventuel appel à la routine `free` qui libère la zone mémoire mais n'a pas d'effet d'écriture sur son paramètre. Voir le programme 6.4

```
int main()
{
int i = 1, j = 2, z = 3, *pi = &i, *pj = &j, *pz = &z, **ppi = &pi, **ppj = &pj;

// Points To:
// pi -> i , EXACT
// pj -> j , EXACT
// ppi -> pi , EXACT
// ppj -> pj , EXACT
// pz -> z , EXACT
   swap01(ppi, ppj);

// Points To:
// pi -> undefined , MAY
// pj -> undefined , MAY
// pi -> *ANY_MODULE*:*ANYWHERE* , MAY
// pj -> *ANY_MODULE*:*ANYWHERE* , MAY
// ppi -> pi , MAY
// ppj -> pj , MAY
// ppi -> undefined , MAY
// ppj -> undefined , MAY
// pz -> z , EXACT
   return 0;
}
```

Prog 6.4 – Les arcs *points-to* pour l'approche utilisant seulement l'ensemble Kill

L'équation définissant l'ensemble Gen est détaillée à la section 6.5.1. Cette dernière ne permet cependant pas de représenter l'état de la mémoire après un appel à `free` sur un des arguments ou des pointeurs accessibles depuis le site d'appel, d'où l'ajout systématique d'un arc vers `undefined`.

6.1.4 L'approche par résumés

Cette approche est plus précise que les précédentes. Elle consiste à mettre à jour la relation *points-to* en utilisant à la fois les ensembles Kill pour supprimer les pointeurs écrits et Gen pour mettre à jour le programme avec les arcs qui résultent de l'analyse de l'appelée. Son principe consiste 1) à calculer une fonction de traduction entre le contexte effectif du site d'appel et le contexte formel de la fonction appelée (ensemble In), 2) à vérifier que cette fonction de traduction respecte bien les hypothèses de non-aliasing faites lors de l'analyse de l'appelée, 3) à calculer les ensembles Kill et Gen dans le contexte formel en exploitant les ensembles In et Out et l'équation fondamentale :

$$\text{Out} = (\text{In} - \text{Kill}) \cup \text{Gen}$$

pour en déduire des approximations correctes :

$$\text{Kill} = \text{In} - \text{Out}$$

et

$$\text{Gen} = \text{Out} - \text{In}$$

4) à utiliser la fonction de traduction pour obtenir des ensembles Kill et Gen dans le contexte de la fonction appelante et 5) à prendre en comptes les arcs du contexte d'appel qui ne sont pas présents dans la vue que constitue le contexte *points-to* In et qui sont néanmoins modifiés.

En fait, les choses sont un peu plus compliquées pour l'étape 4) parce que tous ces ensembles sont relatifs à l'état mémoire courant, σ, et que ces états ne sont pas les mêmes pour les ensembles In et Out puisqu'il s'agit dans le premier cas de l'état initial et dans le deuxième cas de l'état final. Il faut donc connaître une relation entre ces deux états pour abstraire correctement les équations ci-dessus. Cette relation est abstraite par les effets cumulés de la fonction appelée, appelés Written par la suite.

Nous présentons les résultats de cette approche sur quelques exemples ci-dessous, mais les détails font l'objet de la section 6.3 et des sections suivantes. L'exemple 6.1 est donc analysé à nouveau et les résultats sont illustrés par le programme 6.5. On y voit que les effets de la fonction `swap01` sont décrits puisqu'en sortie `pi` pointe sur `j` et `pj` sur `i`.

```
int main()
{
int i = 1, j = 2, z = 3, *pi = &i, *pj = &j, *pz = &z, **ppi = &pi, **ppj = &pj;

// Points To:
// pi -> i , EXACT
// pj -> j , EXACT
// ppi -> pi , EXACT
// ppj -> pj , EXACT
// pz -> z , EXACT
   swap01(ppi, ppj);

// Points To:
// pi -> j , EXACT
// pj -> i , EXACT
// ppi -> pi , EXACT
// ppj -> pj , EXACT
// pz -> z , EXACT
   return 0;
}
```

Prog 6.5 – Les arcs *points-to* pour le programme 6.1

En termes de graphe *points-to*, l'état de la mémoire est représenté par les figures 6.1 et 6.2 [6].

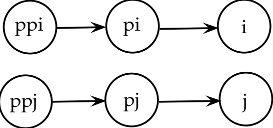


FIGURE 6.1 – Graphe des arcs *points-to* avant l'appel à `swap01`

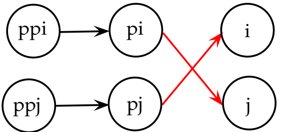


FIGURE 6.2 – Graphe des arcs *points* après l'appel à `swap01`

Les cibles des pointeurs `pi` et `pj` ont été permutées. Une mise à jour précise des arcs *points-to* a été effectuée après l'appel à `swap01`. Ceci a nécessité l'utilisation d'un résumé des arcs *points-to* qui a été calculé au niveau de l'appelée et traduit au niveau de l'appelant. Ce résumé est valide parce que le site d'appel est compatible avec les hypothèses de non aliasing faites lors de l'analyse de `swap01`.

6.1.5 Ordonnancement des phases pour l'analyse *points-to*

Nous avons vu précédemment que l'analyse *points-to* ascendante nécessitait la connaissance des effets des sites d'appel et que le calcul des effets nécessitait la connaissance de l'information *points-to* afin de pouvoir traduire les effets. Il est donc nécessaire de synchroniser ces deux analyses.

6.1.5.1 Ressources et passes impliquées dans l'analyse *points-to*

Les *points-to* L'analyse *points-to* prend en entrée le code source d'un programme pour fournir en résultat les arcs *points-to* calculés pour chaque instruction du programme. Quand c'est l'analyse intraprocédurale qui est effectuée il n'y a pas besoin de tous les modules ; c'est-à-dire que les appelées ne sont pas demandées ni analysées. Les ressources qui sont sauvegardées pour être utilisées par la suite soit par d'autres analyses (voir chapitre 2) soit par l'analyse interprocédurale des pointeurs sont :

- les `points_to_in` qui sont les arcs *points-to* à l'entrée de chaque fonction et qui constituent donc son contexte *points-to* (voir la sous-section 6.3.1.3) ;
- les `points_to_out` qui sont les arcs *points-to* connus à la sortie de chaque fonction et qui sont visibles au niveau du site d'appel (voir la sous-section 6.3.1.3) ;
- les `points_to` qui sont les arcs *points-to* décorant chaque instruction du corps de la fonction.

Notons que dans PIPS les analyses de pointeurs que nous avons introduites précédemment sont appelées :

- `INTRAPROCEDURAL_POINTS_TO_ANALYSIS` pour l'analyse intraprocédurale (sous-section 6.1.1) ;
- `FAST_INTERPROCEDURAL_POINTS_TO_ANALYSIS` pour l'analyse interprocédurale utilisant les effets (sous-section 6.1.3) ;
- `INTERPROCEDURAL_POINTS_TO_ANALYSIS` pour l'analyse interprocédural par transformeur (sous-section 6.1.4).

Les effets propres, calculés ou non avec l'information *points-to* La ressource produite par l'analyse des effets, `proper_effects` (section 2.2.2), a été définie et elle contient la liste des variables qui ont été lues et écrites par une instruction, en utilisant ou non l'information *points-to* pour pouvoir traduire les accès aux variables par déréférencement de pointeurs (comme

6. Le pointeur `pz`, dont la valeur n'est pas modifiée par l'appel à la fonction, n'apparaît pas sur les figures pour plus de lisibilité.

**p par exemple). La nouvelle phase utilisant l’information *points-to* est appelée dans PIPS PROPER_EFFECTS_WITH_POINTS_TO [7] et requiert comme entrée la ressource points_to_in (voir la figure 6.3).

Les effets résumés avec l’information *points-to* La ressource summary_effects est calculée à partir des effets cumulés de la procédure concernée. Les effets cumulés sont dérivés des effets propres en les propageant vers la haut dans l’arbre syntaxique et en y éliminant les dépendances à l’état mémoire courant. En termes de ressources PIPS, les entrées sont donc :

- proper_effects ;
- cumulated_effects.

Les nouvelles analyses sont les deux passes :

- PROPER_EFFECTS_WITH_POINTS_TO ;
- CUMULATED_EFFECTS_WITH_POINTS_TO.

La ressource SUMMARY_EFFECTS permet de récupérer la liste des variables visibles de l’extérieur, qui ont été modifiés par effet de bord de l’appelée, Cette liste doit encore être traduite en faisant la correspondance entre paramètres formels et effectifs, et plus généralement, entre le contexte formel, qui, rappelons le, contient non seulement les paramètres formels mais aussi les *points-to stubs*, et le contexte effectif. Elle permet la génération des ensembles Kill et Gen d’un site d’appel en prenant en compte les sources des arcs *points-to* qui ont été ou non modifiées entre l’entrée et la sortie de la fonction appelée.

6.1.5.2 Ordonnancement des passes relatives aux effets et relations *points-to*

Effets et analyse intraprocédurale des *points-to* Après le calcul des arcs *points-to* pour une fonction, une phase de calcul des effets demande cette ressource pour produire des effets plus précis que ne le ferait un calcul direct des effets (voir la section 2.2.2). Comme indiqué précédemment, ces trois phases sont appelées :

- PROPER_EFFECTS_WITH_POINTS_TO,
- CUMULATED_EFFECTS_WITH_POINTS_TO,
- SUMMARY_EFFECTS.

Les deux premières ont été introduites spécifiquement pour exploiter l’information *points-to* en complément des passes pré-existantes, PROPER_EFFECTS, CUMULATED_EFFECTS et SUMMARY_EFFECTS.

Effets et analyse interprocédurale des *points-to* Pour l’analyse *points-to* interprocédurale, qui requiert non seulement le code de la fonction en cours de traitement mais aussi celui de tous les appelés ainsi que leur arcs *points-to*, les ressources points_to_in et points_to_out sont nécessaires. Les summary_effects des appelés, qui ont été calculés indirectement en utilisant les *points-to* afin de pouvoir déterminer les variables pointeurs qui sont visibles au niveau du site d’appel et qui ont été modifiées par effet de bord ou retournées via l’instruction return, sont aussi nécessaires.

6.1.5.3 Conclusion

L’ordonnancement des phases permet de comprendre comment une des analyses clientes des *points-to*, le calcul des effets, interagit avec l’analyse de pointeurs et comment cette dernière lors du calcul interprocédural exploite à son tour les résultats des effets. Cet ordonnancement est propre à PIPS et à son gestionnaire de phase *pipsmake* [8].

7. Cette phase a été développée par Béatrice Creusillet.
8. Rappelons que *pipsmake* détecte les cycles récursifs et n’y active pas les analyses interprocédurales.

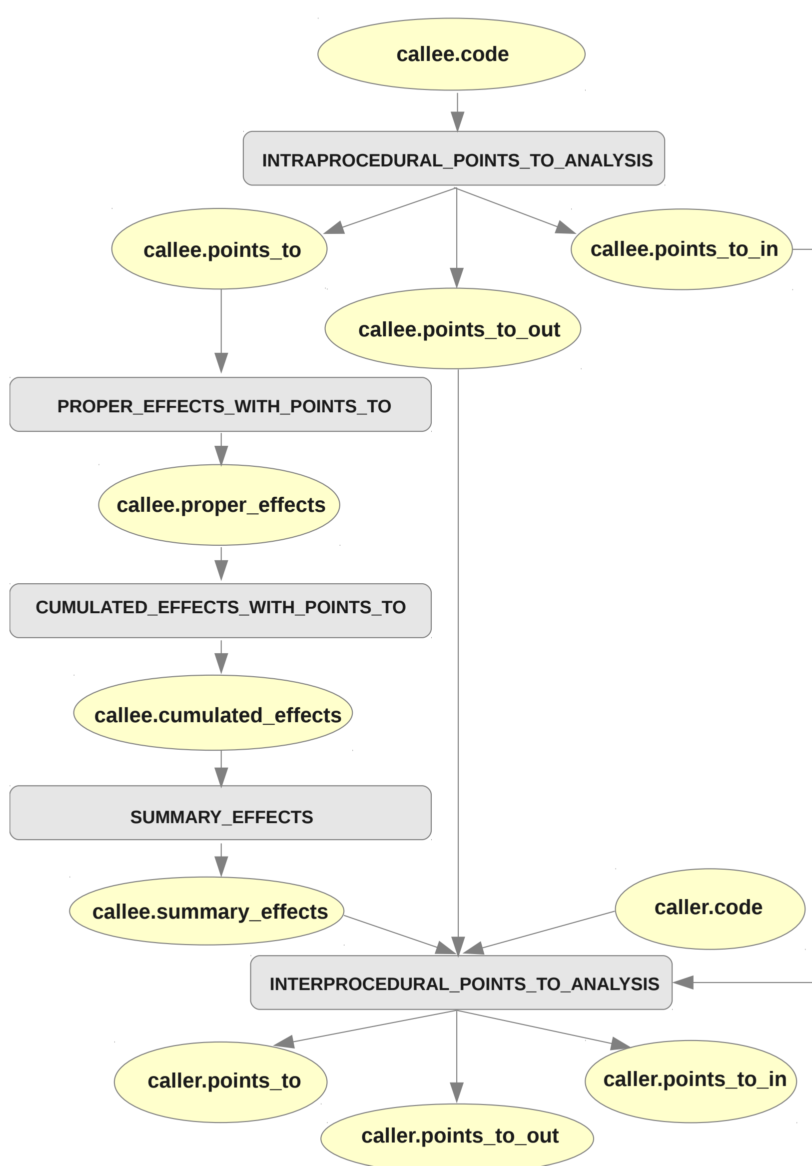


FIGURE 6.3 – Les dépendances entre phases d'analyse

Notons qu'on aurait aussi pu implémenter un calcul des pointeurs modifiés directement dans l'analyse `points-to` mais il vaut mieux réutiliser si possible une analyse existante et ne pas mélanger une analyse de type transformeur, celle des effets, et une analyse de type précondition, celle des *Points-to*.

6.2 Vers un transformeur de *points-to* : les difficultés

Un transformeur de *points-to*, en interprocédural, doit pouvoir exprimer, ou à tout le moins sur-exprimer, toutes les transformations élémentaires de l'information *points-to* qui surviennent dans le cadre intraprocédural. Ces transformations, déjà vues dans les deux chapitres précédents, comportent au minimum :

1. le retour d'une valeur de pointeur, par exemple suite à une allocation effectuée par l'appelée ;
2. l'affectation d'un pointeur, indirectement via des modifications des zones pointées ;
3. l'interruption de l'exécution par « `abort` » ou « `exit` » ;
4. l'impact d'un test ;
5. la libération d'une zone mémoire par « `free` ».

Nous présentons ci-dessous des exemples qui illustrent chacun de ces cinq cas. Les résultats de notre analyse interprocédurale est comparée au résultat de l'analyse intraprocédurale après une transformation du code analysé. La transformation faite par le compilateur consiste à insérer le corps complet de la fonction dans chaque contexte où cette dernière est appelée. Cette transformation est appelée « inlining » ou expansion intégrée. Pour chaque exemple nous montrons les résultats calculés en utilisant les résumés *points-to* des fonctions appelées ainsi que les résultats obtenus en analysant intraprocéduralement le corps expansé de la fonction.

D'autres cas importants sont les détections d'erreurs d'exécution, qu'elles soient dues à la fonction appelée ou bien au site d'appel, comme par exemple le passage d'un pointeur mal initialisé.

Aucun des exemples ne présente d'aliasing entre paramètres effectifs. En effet, nous avons décidé d'analyser finement les procédures en supposant qu'il n'y a pas d'aliasing. Nous nous contentons de détecter l'aliasing au niveau du site d'appel et d'appliquer alors une analyse moins précise.

6.2.1 Affectation à un pointeur de la valeur retournée par une fonction

Comme nous voulons garder le même processus de calcul des arcs *points-to* pour l'intraprocédural ainsi que l'interprocédural, il nous faut définir les ensembles Kill et Gen ou des ensembles équivalents pour pouvoir reproduire l'effet d'une affectation. Le transformeur de *points-to* doit donc « tuer » des arcs *points-to* et en ajouter des nouveaux. Si la fonction retourne de nouveaux emplacements mémoire via l'instruction `return`, comme le montre le programme 6.6, alors la variable qui reçoit le résultat de la fonction doit être mise à jour.

```
typedef int * pointer;
pointer alloc_pointer(int v)
{
  pointer p = malloc(sizeof(int));
  *p = v;
  return p;
}

int main(void)
{
  pointer p1;
  p1 = alloc_pointer(13);
  return;
}
```

Prog 6.6 – Exemple d'allocation dynamique en interprocédural

```
int main(void)
{
   pointer p1;
   //PIPS generated variable
   pointer _return0;
   {
      //PIPS generated variable
      int v0 = 13;
      {
         pointer p;
         p = malloc(sizeof(int));
         *p = v0;
         _return0 = p;
      }
   }
   p1 = _return0;
   return;
}
```

Prog 6.7 – La version « inlinée » du programme 6.6

En effet, au niveau du site d'appel, l'instruction `p1 = alloc_pointer(13);` est équivalente à `p1 = r;` comme les deux opérandes sont de type pointeur, nous devons considérer que c'est une affectation de pointeurs. Le pointeur `p1` pointe vers la même cible que la valeur de retour, c'est à dire dire `alloc_pointer:*HEAP*_l_7`. Au niveau du programme 6.7 nous avons le code de la fonction `alloc_pointer` expansé au niveau du main ; la valeur de retour est une variable désignée par `_return0`.

La comparaison des résultats de l'interprocédural avec l'inlining (voir programme 6.8 et 6.9) permet de se rendre compte que dans ce cas précis d'affectation l'information *points-to* est imprécise. Ceci est dû à la modélisation du tas (voir la section 5.2.1). La version expansée a un arc *points-to* de plus, celui dont la source est la variable de retour. L'autre différence est le numéro de « statement » au niveau duquel l'appel à `malloc` s'effectue ; le site d'appel est différent.

```
int main(void){
   pointer p1;
// Points To:
// p1 -> undefined , EXACT
   p1 = alloc_pointer(13);
// Points To:
// p1 -> alloc_pointer:*HEAP*_l_7 , MAY
   return;
}
```

Prog 6.8 – Les arcs *points-to* pour le programme 6.6

En conclusion, ce premier cas ne présente pas de difficulté pour le calcul de l'ensemble Kill parce qu'il suffit de l'effectuer au niveau de l'appelé et qu'aucune traduction n'est nécessaire. La perte de précision ne pourrait provenir que de l'ensemble Gen qui est ici vide. La différence entre les numéros de sites d'appel de `malloc` est logique et ne constitue pas une perte de précision. L'analyse interprocédurale est aussi précise que l'analyse intraprocédurale du code expansé.

6.2.2 Affectation indirecte d'un pointeur dans une procédure

Le deuxième type d'affectation de pointeurs est produit par un effet de bord d'une fonction. Prenons comme exemple le cas d'échange de valeurs de pointeurs au niveau du pro-

```
 int main(void){
  pointer p1;
  pointer _return0;
   {
      int v0 = 13;
      {
         pointer p;
         p = malloc(sizeof(int));
// Points To:
// _return0 -> undefined , EXACT
// p -> *HEAP*_l_11 , MAY
// p1 -> undefined , EXACT
         *p = v0;

// Points To:
// _return0 -> undefined , EXACT
// p -> *HEAP*_l_11 , MAY
// p1 -> undefined , EXACT
         _return0 = p;
      }
   }
// Points To:
// _return0 -> *HEAP*_l_11 , MAY
// p1 -> undefined , EXACT
   p1 = _return0;

// Points To:
// _return0 -> *HEAP*_l_11 , MAY
// p1 -> *HEAP*_l_11 , MAY
   return;
}
```

Prog 6.9 – La version « inlinée » du programme 6.6 avec les arcs *points-to*

gramme 6.10. Nous commençons par appliquer la phase d'expansion sur le code [9] (voir le programme 6.11). Maintenant que le code de la fonction `alloc_pointer` est disponible au niveau du `main`, nous pouvons appliquer l'analyse des pointeurs aux deux versions et comparer les résultats obtenus.

```
void swap(int **p, int **q)
{
  int *pt = *p;
  *p = *q;
  *q = pt;
  return;
}

int main()
{
  int i = 1, j = 2, *pi = &i, *pj = &j;
  swap(&pi, &pj);

  return 0;
}
```

Prog 6.10 – Affectation d'un pointeur dans une procédure

```
int main()
{
   int i = 1, j = 2, *pi = &i, *pj = &j;
   {
      //PIPS generated variable
      int **P_0, **P_1;
      P_0 = &pi;
      P_1 = &pj;
      {
         int *pt;
         pt = *P_0;
         *P_0 = *P_1;
         *P_1 = pt;
      }
   }

   return 0;
}
```

Prog 6.11 – La version « inlinée » du programme 6.10

La phase d'expansion de procédure de PIPS introduit des variables temporaires pour représenter les paramètres effectifs et par conséquent leur affecte la valeur de ces derniers. Les résultats de l'analyse interprocédurale sur le code sont illustrés par le programme 6.12 ; ceux de la version expansée sont illustrés par le programme 6.14. Au point programme qui suit l'appel à la fonction `swap` et à celui qui suit la fin du corps de la fonction expansée le graphe *points-to* est exactement le même. Notre analyse qui se base sur les résumés fournit dans ce cas le même résultat que si nous avions analysé intraprocéduralement toutes les instructions du corps de la fonction. Par conséquent, le calcul des ensembles Kill et Gen via une traduction peut suffire pour répondre à nos besoins.

Pour ce cas de tests, l'analyse interprocédurale de pointeurs dans PIPS est précise et ses résultats correspondent à ceux attendus.

6.2.3 Interruption de l'exécution

Lors de l'analyse d'un appel de fonction qui se termine inopinément, le graphe *points-to*, traduit au niveau du site d'appel, doit traduire cette terminaison. Prenons comme exemple le programme 6.15, ainsi que sa version expansée (voir le programme 6.16) où le code de la procédure `error_msg` a été intégré aux autres instructions du `main`.

9. Cette phase est disponible dans PIPS. Les résultats présentés dans cette section sont obtenus en l'appliquant. Nous aurions pu aussi appliquer la passe `flatten_code` pour éliminer les accolades inutiles, mais nous pensons qu'elles sont utiles au lecteur pour bien voir qu'un site d'appel a été expansé.

```
int main()
{
   int i = 1, j = 2, *pi = &i, *pj = &j;
// Points To:
// pi -> i , EXACT
// pj -> j , EXACT

   swap(&pi, &pj);
// Points To:
// pi -> j , EXACT
// pj -> i , EXACT
   return 0;
}
```

Prog 6.12 – Les arcs *points-to* pour le programme 6.10

```
// Points To OUT:
// _p_1 -> _q_2_2 , EXACT
// _q_2 -> _p_1_1 , EXACT

void swap(int **p, int **q)
{
// Points To:
// _p_1 -> _p_1_1 , EXACT
// p -> _p_1 , EXACT

   int *pt = *p;
// Points To:
// _p_1 -> _p_1_1 , EXACT
// _q_2 -> _q_2_2 , EXACT
// p -> _p_1 , EXACT
// pt -> _p_1_1 , EXACT
// q -> _q_2 , EXACT
   *p = *q;

// Points To:
// _p_1 -> _q_2_2 , EXACT
// _q_2 -> _q_2_2 , EXACT
// p -> _p_1 , EXACT
// pt -> _p_1_1 , EXACT
// q -> _q_2 , EXACT
   *q = pt;

// Points To:
// _p_1 -> _q_2_2 , EXACT
// _q_2 -> _p_1_1 , EXACT
// p -> _p_1 , EXACT
// pt -> _p_1_1 , EXACT
// q -> _q_2 , EXACT
   return;
}
```

Prog 6.13 – Les arcs *points-to* OUT pour le programme 6.10

```
void error_msg(char* msg)
{
  printf("%s", msg);
  exit(1);
}

int main()
{
  char *msg1 = "hello";
  char *msg2 = msg1;
  error_msg(msg2);
  return 0;
}
```

Prog 6.15 – Exemple d'interruption d'exécution en interprocédural

```
int main()
{
   char *msg1 = "hello";
   char *msg2 = msg1;
   printf("%s", msg2);
   exit(1);

   return 0;
}
```

Prog 6.16 – La version « inlinée » du programme 6.15

La fonction `error_msg` affiche à l'écran une chaîne de caractères mais se termine par un ap-

```
int main(){
   int i = 1, j = 2, *pi = &i, *pj = &j;
   {
      //PIPS generated variable
// Points To:
// pi -> i , EXACT
// pj -> j , EXACT
      int **P_0, **P_1;
      P_0 = &pi;
      P_1 = &pj;
      {
// Points To:
// P_0 -> pi , EXACT
// P_1 -> pj , EXACT
// pi -> i , EXACT
// pj -> j , EXACT
         int *pt;
         pt = *P_0;

// Points To:
// P_0 -> pi , EXACT
// P_1 -> pj , EXACT
// pi -> i , EXACT
// pj -> j , EXACT
// pt -> i , EXACT
         *P_0 = *P_1;

// Points To:
// P_0 -> pi , EXACT
// P_1 -> pj , EXACT
// pi -> j , EXACT
// pj -> j , EXACT
// pt -> i , EXACT
         *P_1 = pt;
      }
   }
// Points To:
// pi -> j , EXACT
// pj -> i , EXACT
   return 0;
}
```

Prog 6.14 – La version « inline » du programme 6.10 avec les arcs *points-to*

pel à `exit`. Les résultats de notre analyse interprocédurale sont illustrés par le programme 6.17. Ceux de la version expansée sont illustrés par le programme 6.18. A la fin des deux versions nous retrouvons les mêmes résultats. En effet, après une interruption de programme, le graphe *points-to* reçoit non l'ensemble vide, qui serait ambigu puisque valide en l'absence de pointeurs, mais une valeur particulière, *bottom*, qui est rendue textuellement par l'information `unreachable`. Cette valeur signifie que le code qui vient après une instruction `exit` ou après une violation de la norme C ne peut être exécuté. Il ne faut pas propager l'information *points-to* aux instructions suivantes quand il s'agit de code mort, ce qui permet de préciser l'information *points-to* après un test.

```
int main()
{
   char *msg1 = "hello";
// Points To:
// msg1 -> "hello" , EXACT

   char *msg2 = msg1;
// Points To:
// msg1 -> "hello" , EXACT
// msg2 -> "hello" , EXACT

   error_msg(msg2);
// Points To: unreachable

   return 0;
}
```

Prog 6.17 – Les arcs *points-to* pour le programme 6.15

```
int main()
{
   char *msg1 = "hello";
// Points To:
// msg1 -> "hello" , EXACT

   char *msg2 = msg1;
// Points To:
// msg1 -> "hello" , EXACT
// msg2 -> "hello" , EXACT

   printf("%s", msg2);
// Points To:
// msg1 -> "hello" , EXACT
// msg2 -> "hello" , EXACT

   exit(1);
// Points To: unreachable
   return 0;
}
```

Prog 6.18 – La version « inlinée » du programme 6.15 avec les arcs *points-to*

L'information *points-to* a la valeur `unreachable` pour spécifier à l'utilisateur que lors de l'analyse de l'appelé, ce dernier s'est arrêté inopinément par un appel à `exit` ou `abort`, ou encore par l'exécution d'une opération interdite par la norme du langage `C`, comme le déréférencement d'un pointeur `NULL` ou non-défini.

L'information *points-to* pourrait être utilisée par une nouvelle transformation d'élimination de code non exécuté, similaire à la passe `control_simplification` de PIPS qui utilise, elle, les préconditions sur les valeurs des scalaires.

6.2.4 Impact d'un test

La condition dans un test peut raffiner les cibles d'un pointeur. Si la condition porte sur la valeur d'un pointeur alors elle peut être évaluée en utilisant l'information *points-to*. Prenons comme exemple le programme 6.19 ainsi que sa version expansée ; le corps de la fonction `init` a été intégré au `main`. Ce dernier comporte deux tests :

1. si le premier est évalué à vrai alors le pointeur `q` est alloué correctement au niveau du tas, sinon il vaut `NULL` ;

2. le deuxième comporte deux branches ; si la condition est évaluée à vrai, c'est-à-dire que le pointeur `p` vaut `NULL`, alors le programme se termine à cause de l'instruction `exit` ; le reste du programme est considéré comme du code mort. Si la deuxième condition est évaluée à faux, alors le contenu de l'emplacement mémoire pointé par `p` vaut zéro et, surtout, cela veut dire au niveau du graphe *points-to* que le pointeur `q` ne peut pas valoir `NULL`.

```
void init(int* p)
{
  if(p == NULL)
    exit(1);
  else
    *p = 0;
  return;
}

int main()
{
  int init_p = 1 ;
  int *q = NULL;
  if(init_p)
    q = (int*)malloc(4*sizeof(int));
  init(q);
  return 0;
}
```

Prog 6.19 – Exemple de test en interprocédural

```
int main()
{
   int init_p = 1;
   int *q = (void *) 0;
   if (init_p)
      q = (int *) malloc(4*sizeof(int));
   if (q==(void *) 0)
      exit(1);
   else
      *q = 0;
   return 0;
}
```

Prog 6.20 – La version « inlinée » du programme 6.19

Les résultats de l'analyse interprocédurale sont illustrés par le programme 6.21 et ceux de l'analyse intraprocédurale sur le code expansé par le programme 6.22. Nous remarquons cette fois-ci une imprécision dans notre analyse interprocédurale où l'arc `q -> *NULL_POINTER*, MAY` ne disparaît pas après l'appel à `init`. En effet, notre calcul ne prend pas en compte le fait que, si l'exécution du programme continue après l'appel, cela veut dire que le test sur la nullité du pointeur est évalué à faux. Ce qui devrait avoir comme impact la suppression de l'arc. Par contre la version expansée analysée intraprocéduralement prend bien en compte cette évaluation ; les arcs après le test sur la nullité du pointeur ne contiennent pas la valeur `NULL`. Le résultat de l'analyse interprocédurale n'est pas faux, mais il est moins précis.

Une autre différence apparaît au niveau du numéro de « statement » qui permet d'identifier le site d'appel de `malloc`. Cette différence a été aussi observée dans l'un des exemples précédents (voir le programme 6.6). Malgré cette sur-approximation des résultats, l'analyse interprocédurale reste correcte et permet de garder trace des valeurs de pointeurs après l'appel à la fonction.

6.2.5 Libération d'une zone mémoire

L'effet de la libération de la mémoire via l'appel à la routine « `free` » a été détaillé dans la section 5.2.3. Dans cette dernière nous expliquons que lors de la libération le pointeur lui-même n'est pas modifié, mais sa cible l'est ainsi que la cible de tous les pointeurs qui sont en alias avec le pointeur faisant l'objet du `free`. De même, les arcs partant directement ou indirectement de la zone libéré peuvent disparaître, ce qui peut induire des fuites mémoire indirectes. Nous parcourons donc le graphe pour transformeur en `undefined` les cibles des arcs pointant sur la zone libérée. Les ensembles Gen et Kill associés à une libération sont particulièrement compliqués puisqu'ils font potentiellement intervenir tous les arcs de la relation *points-to*. En interprocédural, nous devons traduire au niveau du site d'appel cette mise à jour du graphe *points-to*. Donc nous devons prendre en compte l'information se trouvant dans la relation *points-to* de sortie de procédure. Ou plutôt, il faut garder trace des cellules mémoires qui sont libérées.

```
int main()
{

   int init_p = 1;
   int *q = (void *) 0;
// Points To:
// q -> *NULL_POINTER* , EXACT

   if (init_p)

// Points To:
// q -> *NULL_POINTER* , EXACT

      q = (int *) malloc(4*sizeof(int));

// Points To:
// q -> *HEAP*_l_17[0] , MAY
// q -> *NULL_POINTER* , MAY

   init(q);

// Points To:
// q -> *HEAP*_l_17[0] , MAY
// q -> *NULL_POINTER* , MAY

   return 0;
}
```

Prog 6.21 – Les arcs *points-to* pour le programme 6.19

```
 int main()
{
   int init_p = 1;
   int *q = (void *) 0;
// Points To:
// q -> *NULL_POINTER* , EXACT

   if (init_p)
// Points To:
// q -> *NULL_POINTER* , EXACT
      q = (int *) malloc(4*sizeof(int));

// Points To:
// q -> *HEAP*_l_6[0] , MAY
// q -> *NULL_POINTER* , MAY
   if (q==(void *) 0)
// Points To:
// q -> *NULL_POINTER* , EXACT
      exit(1);
   else
// Points To:
// q -> *HEAP*_l_6[0] , MAY
      *q = 0;

// Points To:
// q -> *HEAP*_l_6[0] , MAY
   return 0;
}
```

Prog 6.22 – La version « inline » du programme 6.19 avec les arcs *points-to*

6.2.5.1 Exemple de libération simple

Prenons pour commencer deux programmes de libération de mémoire en interprocédural ; le premier est donné par 6.23.

```
typedef int * pointer;
void pointer_free(pointer p){
  free(p);
  p = malloc(sizeof(int));
  return;
}

int main(void){
  pointer p1, p2;
  p1 = malloc(sizeof(int));
  p2 = p1;
  pointer_free(p1);
  return;
}
```

Prog 6.23 – Libération de zone mémoire en interprocédural 1

```
typedef int * pointer;

int main(void)
{
   pointer p1, p2;
   p1 = malloc(sizeof(int));
   p2 = p1;
   free(p1);
   p1 = malloc(sizeof(int));

   return;
}
```

Prog 6.24 – La version « inlinée » du programme 6.23

Le programme 6.23 reçoit en paramètre, `p1`, un pointeur alloué au niveau du tas, qui est en alias avec un deuxième pointeur `p2`. La fonction `pointer_free` libère la zone pointée par son argument et le fait pointer vers une nouvelle zone du tas. Après analyse de la version expansée du code (voir le programme 6.26), nous remarquons que l'effet de la fonction `free` a bien été pris en compte par l'analyse des pointeurs. En effet, le pointeur qui est en alias avec le pointeur `p1` pointe maintenant vers `undefined`. Quant au pointeur `p1` lui même, sa cible a été mise à jour. Après la libération de la zone `*HEAP*_l_4`, il pointe désormais vers l'emplacement `*HEAP*_l_7`. Par contre, notre analyse interprocédurale n'est pas en mesure de reporter ces informations au niveau du site d'appel. En effet, les arcs du programme 6.25 restent inchangés après l'appel à la fonction `pointer_free`. Ni le pointeur `p1` ni le pointeur `p1` ne sont mis à jour.

Pour le deuxième exemple de libération de mémoire illustré par le programme 6.27, la fonction `pointer_free` ne fait que libérer la zone mémoire sans faire pointer le pointeur vers une nouvelle zone.

```
typedef int * pointer;
void pointer_free(pointer p)
{
  free(p);
  return;
}

int main(void)
{
  pointer p1, p2;
  p1 = malloc(sizeof(int));
  p2 = p1;
  pointer_free(p1);
  return;
}
```

Prog 6.27 – Libération de zone mémoire en interprocédural

```
int main(void)
{
   pointer p1, p2;
   p1 = malloc(sizeof(int));
   p2 = p1;
   free(p1);
   return;
}
```

Prog 6.28 – La version « inlinée » du programme 6.27

```
int main(void)
{
   pointer p1, p2;

// Points To:
// p1 -> undefined , EXACT
// p2 -> undefined , EXACT
   p1 = malloc(sizeof(int));

// Points To:
// p1 -> *HEAP*_l_23 , MAY
// p2 -> undefined , EXACT
   p2 = p1;

// Points To:
// p1 -> *HEAP*_l_23 , MAY
// p2 -> *HEAP*_l_23 , MAY

   pointer_free(p1);
// Points To:
// p1 -> *HEAP*_l_23 , MAY
// p2 -> *HEAP*_l_23 , MAY

   return;
}
```

Prog 6.25 – Les arcs *points-to* pour le programme 6.23

```
int main(void)
{
   pointer p1, p2;
// Points To:
// p1 -> undefined , EXACT
// p2 -> undefined , EXACT

   p1 = malloc(sizeof(int));
// Points To:
// p1 -> *HEAP*_l_4 , MAY
// p2 -> undefined , EXACT

   p2 = p1;
// Points To:
// p1 -> *HEAP*_l_4 , MAY
// p2 -> *HEAP*_l_4 , MAY

   free(p1);
// Points To:
// p1 -> undefined , EXACT
// p2 -> undefined , MAY

   p1 = malloc(sizeof(int));
// Points To:
// p1 -> *HEAP*_l_7 , MAY
// p2 -> undefined , MAY

   return;
}
```

Prog 6.26 – Les arcs *points-to* pour la version « inlinée » du programme 6.23

Dans le programme 6.28, le site d'appel a été expansé (le programme 6.28). Maintenant l'appel à la fonction de libération de mémoire `free` se fait dans le corps du `main`. Les deux pointeurs `p1` et `p2` sont en alias et sont alloués dynamiquement dans le tas. L'appel à la fonction `pointer_free` a pour effet de libérer la zone pointée par `p1`. Ceci est traduit au niveau du site d'appel par `p1` pointant vers `undefined` ; ce résultat est visible au niveau du programme 6.29.

```
int main(void)
{
   pointer p1, p2;
   p1 = alloc_pointer(13);

// Points To:
// p1 -> alloc_pointer:*HEAP*_l_7 , MAY
// p2 -> undefined , EXACT
   p2 = p1;

// Points To:
// p1 -> alloc_pointer:*HEAP*_l_7 , MAY
// p2 -> alloc_pointer:*HEAP*_l_7 , MAY
   pointer_free(p1);

// Points To:
// p1 -> undefined, EXACT
// p2 -> alloc_pointer:*HEAP*_l_7, MAY
   return;
}
```

Prog 6.29 – Les arcs *points-to* pour le programme 6.27

```
int main(void)
{
   pointer p1, p2;
   p1 = malloc(sizeof(int));

// Points To:
// p1 -> *HEAP*_l_4 , MAY
// p2 -> undefined, EXACT
   p2 = p1;

// Points To:
// p1 -> *HEAP*_l_4 , MAY
// p2 -> *HEAP*_l_4 , MAY
   free(p1);

// Points To:
// p1 -> undefined, EXACT
// p2 -> undefined, MAY
   return;
}
```

Prog 6.30 – La version « inlinée » du programme 6.27

6.2.5.2 Choix de conception

Les difficultés propres au traitement de la libération de la mémoire en font un bon point de comparaison de différentes conceptions de l'analyse d'un site d'appel. Deux choix ont déjà été envisagés :

1. faut-il ou non traiter explicitement la copie qui est effectuée lors du passage des paramètres scalaires [10] en conservant l'information attachée à chaque paramètre lors de l'entrée dans la fonction analysée ?
2. faut-il ou non conserver explicitement la liste des cellules mémoires libérées ?

Pour évaluer les conséquences de ces choix, deux types de difficultés doivent être pris en compte :

1. l'ordre des opérations : dans le cas où un paramètre formel est modifié, la libération a-t-elle eu lieu avant ou après la modification ?
2. et le fait qu'elles soient exécutées à coup sûr ou non : la modification est-elle gardée par un test ? La libération est-elle conditionnelle ?

Une troisième situation pourrait/devrait aussi être étudiée ; le cas de la libération située dans une boucle.

10. La norme C prévoit que les objets ayant pour type un type de base, struct, union ou enum sont passés par copie, tandis que les tableaux sont passés par référence (cf. section 6.9.1 de la norme C99). Cette différence de traitement nécessite la prise en compte des types dans les algorithmes, ou du moins dans l'implantation.

6.2.5.3 Exemples d'entrelacements entre allocations, libérations et écritures

En ce qui concerne l'ordre des opérations, l'ensemble Written n'en garde pas trace. La connaissance des relations In et Out de la fonction analysée ne permet pas non plus de déterminer si une libération a eu lieu avant ou après une modification du paramètre formel. Il faudrait donc soit imposer un qualificateur `const` à tous les paramètres formels passés par copie [11] et pouvant conduire à des pointeurs, soit prendre en compte explicitement le phénomène de copie en définissant un pseudo-paramètre formel dont la valeur est inchangée durant l'exécution de la procédure puisqu'il n'y apparaît pas.

Les deux ordres possibles sont utilisés dans les deux programmes suivants où la variable `fq` sert de copie au paramètre formel `fp` pour mettre en évidence le gain de précision permis par la gestion d'une copie.

```
int ordered_free01(int *fp) {
  int *fq = fp;
  free(fp);
  fp = (int *) malloc(sizeof(int));
  return 0;
}

int main()
{
  int *p = (int *) malloc(sizeof(int));
  ordered_free01(p);
  return 0;
}
```

Prog 6.31 – Allocation mémoire après un appel à `free`

```
int ordered_free02(int *fp) {
  int *fq = fp;
  fp = (int *) malloc(sizeof(int));
  free(fp);
  return *fq;
}

int main()
{
  int *p = (int *) malloc(sizeof(int));
  ordered_free02(p);
  return 0;
}
```

Prog 6.32 – Libération mémoire après une allocation

Nous commençons par l'expansion du corps des deux fonctions `ordered_free01` et `ordered_free02` (voir les programmes 6.33 et 6.34) pour pouvoir appliquer par la suite l'analyse intraprocédurale des pointeurs.

```
int main()
{
   int *p = (int *) malloc(sizeof(int));
   //PIPS generated variable
   int _return0;
   {
      int *fq;
      fq = p;
      free(p);
      p = (int *) malloc(sizeof(int));
      _return0 = 0;
   }
   _return0;
   return 0;
}
```

Prog 6.33 – Version expansée du programme 6.31

```
int main()
{
   int *p = (int *) malloc(sizeof(int));
   //PIPS generated variable
   int _return0;
   {
      int *fq;
      fq = p;
      p = (int *) malloc(sizeof(int));
      free(p);
      _return0 = *fq;
   }
   _return0;
   return 0;
}
```

Prog 6.34 – La version expansée du programme 6.32

Après expansion du code, l'analyse intraprocédurale est appliquée aux deux exemples pour obtenir les résultats des programmes 6.35 et 6.36.

11. Seuls les tableaux sont passés par référence. Tableaux de pointeurs et tableaux de struct doivent être pris en compte.

```
int main(){
   int *p = (int *) malloc(sizeof(int));
   //PIPS generated variable
// Points To:
// p -> *HEAP*_l_3 , MAY
   int _return0;
   {
// Points To:
// p -> *HEAP*_l_3 , MAY
      int *fq;

// Points To:
// fq -> undefined , EXACT
// p -> *HEAP*_l_3 , MAY
      fq = p;

// Points To:
// fq -> *HEAP*_l_3 , MAY
// p -> *HEAP*_l_3 , MAY
      free(p);

// Points To:
// fq -> undefined , MAY
// p -> undefined , EXACT
      p = (int *) malloc(sizeof(int));

// Points To:
// fq -> undefined , MAY
// p -> *HEAP*_l_10 , MAY
      _return0 = 0;
   }

// Points To:
// p -> *HEAP*_l_10 , MAY
   _return0;

// Points To:
// p -> *HEAP*_l_10 , MAY
   return 0;
}
```

Prog 6.35 – Les arcs *points-to* pour la version expansée du programme 6.31

```
int main(){
   int *p = (int *) malloc(sizeof(int));
   //PIPS generated variable
// Points To:
// p -> *HEAP*_l_3 , MAY
   int _return0;
   {
// Points To:
// p -> *HEAP*_l_3 , MAY
      int *fq;

// Points To:
// fq -> undefined , EXACT
// p -> *HEAP*_l_3 , MAY
      fq = p;

// Points To:
// fq -> *HEAP*_l_3 , MAY
// p -> *HEAP*_l_3 , MAY
      p = (int *) malloc(sizeof(int));

// Points To:
// fq -> *HEAP*_l_3 , MAY
// p -> *HEAP*_l_9 , MAY
      free(p);

// Points To:
// fq -> *HEAP*_l_3 , MAY
// p -> undefined , EXACT
      _return0 = *fq;
   }

// Points To:
// p -> undefined , EXACT
   _return0;

// Points To:
// p -> undefined , EXACT
   return 0;
}
```

Prog 6.36 – Les arcs *points-to* pour la version expansée du programme 6.32

Nous pouvons maintenant comparer ces résultats avec le résultat de l'analyse interprocédurale.

```
// Points To OUT: none

int ordered_free01(int *fp)
{
// Points To:
// fp -> _fp_1 , EXACT
   int *fq = fp;

// Points To:
// fp -> _fp_1 , EXACT
// fq -> _fp_1 , EXACT
   free(fp);

// Points To:
// fp -> undefined , EXACT
// fq -> undefined , EXACT
   fp = (int *) malloc(sizeof(int));

// Points To:
// fp -> *HEAP*_l_8 , MAY
// fq -> undefined , EXACT
   return 0;
}
int main() {
   int *p = (int *) malloc(sizeof(int));

// Points To:
// p -> *HEAP*_l_14 , MAY
   ordered_free01(p);

// Points To:
// p -> *HEAP*_l_14 , MAY
   return 0;
}
```

Prog 6.37 – Les arcs *points-to* pour la programme 6.31

```
// Points To OUT:
// fp -> undefined , EXACT
int ordered_free02(int *fp)
{
// Points To:
// fp -> _fp_1 , EXACT
   int *fq = fp;

// Points To:
// fp -> _fp_1 , EXACT
// fq -> _fp_1 , EXACT
   fp = (int *) malloc(sizeof(int));

// Points To:
// fp -> *HEAP*_l_7 , MAY
// fq -> _fp_1 , EXACT
   free(fp);

// Points To:
// fp -> undefined , EXACT
// fq -> _fp_1 , EXACT
   return *fq;
}
int main() {
   int *p = (int *) malloc(sizeof(int));

// Points To:
// p -> *HEAP*_l_14 , MAY
   ordered_free02(p);

// Points To:
// p -> *HEAP*_l_14 , MAY
   return 0;
}
```

Prog 6.38 – Les arcs *points-to* pour le programme 6.32

Contrairement à l'analyse intraprocédurale, l'analyse interprocédurale ne reproduit pas au niveau du site d'appel les effets d'un appel à la fonction `free`. L'argument garde son ancienne cible.

En l'absence d'information sur l'ordre des opérations, l'analyse *points-to* doit signaler au niveau du site d'appel une libération possible ou une conservation de la valeur du paramètre effectif correspondant au paramètre formel `fp`.

La présence d'opérations gardées par des tests augmente considérablement le nombre de scénarios possibles. Il y en a huit, comme le montrent les différents chemins de contrôle possibles pour le code suivant : Si `c1` est vrai, la procédure n'a aucun impact sur son environnement, mis

```
void conditional_free(int *p, bool c1, bool c2, bool c3) {
  if(c1) p = (int *) malloc(sizeof(int));
  if(c2) free(p);
  if(c3) p = (int *) malloc(sizeof(int));
  return;
}
```

Prog 6.39 – Appel conditionnel à `free`

à part une fuite mémoire. Si `c2` est faux, la situation est identique. Enfin, la valeur de `c3` n'a pas d'impact sur le comportement de la procédure.

Malheureusement, les ensembles In et Out ne dépendent pas des conditions `c1`, `c2` et `c3` et l'ensemble Written dépend de `c1` et de `c3`. Ils ne permettent donc pas de déterminer si la cellule mémoire pointée par le paramètre `p` est libérée ou non. Le résultat de l'analyse *points-to* doit donc être une libération possible ou une conservation de la valeur initiale.

6.2.5.4 Conclusion sur le traitement interprocédural des libérations mémoire

L'implémentation actuelle (algorithme 26) ne permet pas la mise à jour du graphe *points-to*. Le transformeur actuel ne permet pas de conserver l'information sur les zones mémoire qui ont été libérées. Nous avons juste une mise à jour de la cible du pointeur. Une idée serait de garder une liste des cellules libérées et de transmettre cette liste au site d'appel pour la traduire et adapter en conséquence le graphe *points-to*. Le but est de pouvoir reproduire les résultats obtenus en appliquant l'analyse intraprocedurale sur le code expansé ; après l'appel à `free` les deux pointeurs pointent vers l'emplacement mémoire `undefined`.

6.2.6 Conclusion sur les difficultés de l'analyse *points-to* interprocédurale

Chaque construction élémentaire du langage `C` peut être cachée derrière un site d'appel. Nous voyons sur les cas étudiés ci-dessus qu'il est impossible de se contenter d'information purement *points-to* pour appliquer la transformation au niveau de l'appelant. Outre les ensembles résumés de la fonction appelée, In et Out, des informations sur les pointeurs qu'elle écrit et sur les cellules qu'elle libère sont aussi utiles pour améliorer le calcul des ensembles Kill et Gen du site d'appel et donc la précision de l'analyse interprocédurale par rapport à l'analyse intraprocédurale.

En effet, les ensembles Kill et Gen ne peuvent être calculés d'une manière ascendante sans prendre en compte l'hypothèse de non-aliasing entre arguments à l'entrée de la fonction. Une vérification de cette hypothèse doit être effectuée avant de commencer l'analyse interprocédurale. Si elle n'est pas vérifiée une autre analyse moins précise est réalisée. Notons qu'il est possible d'approximer Kill et Gen par les ensembles In et Out calculés précédemment par la phase intraprocédurale. Cependant ces deux ensembles ne sont pas suffisants. En effet, comme on l'a montré à la section 6.1.3 nous avons aussi besoin de la liste des pointeurs qui ont été écrits par la fonction. Par le biais des exemples de la sous-section 6.2.5 nous avons montré que les ensembles Written, In et Out ne sont pas suffisants pour traduire au niveau du site d'appel la mise à jour du graphe *points-to* provoqué par un appel à la fonction `free` effectué dans le corps de l'appelée. Ceci est dû au fait que `free` libère la zone allouée par le pointeur sans un effet d'écriture sur ce dernier. Deux solutions peuvent être envisagées :

1. l'ajout d'un nouvel effet associé à `free` ;
2. la création d'une liste des cellules libérées qui sera traduite par la suite au niveau du site d'appel.

Par ailleurs, le graphe *points-to* doit rester borné et vérifier les différentes contraintes posées sur les noeuds et les arcs après la prise en compte d'un site d'appel par le mécanisme de traduction.

6.3 Le schéma interprocédural ascendant général

Comme l'équation d'une analyse de flot de données comporte un ensemble Kill et un ensemble Gen, le transformeur d'une procédure devrait être une paire de relations (Kill, Gen) valables au point d'entrée de la fonction dont on calcule le transformeur de *points-to*. Il faudrait

donc définir une nouvelle analyse alors que nous disposons déjà d'une analyse *points-to* intraprocédurale précise destinée à permettre la parallélisation des procédures feuilles du graphe des appels.

Une relation *points-to* n'est valable que pour une instruction précise. En effet, la sémantique d'un arc $x \to y$ construit sur le produit de l'ensemble des chemins mémoire constants défini section 2.1 est :

$$M(x) = y$$

où M représente l'état mémoire courant. Le calcul ascendant des ensembles Kill et Gen pour une séquence ou pour une autre instruction composite comme les tests et boucles nécessite donc la prise en compte des écritures mémoires et une connaissance symbolique de l'état mémoire courant, symbolique puisque l'analyse est ascendante et statique. De plus, l'aliasing effectif modifiant les résultats, il faudrait générer des listes de paires $(\text{Kill}, \text{Gen})$ dont beaucoup devraient être inutiles.

Peut-on donc éviter cette analyse compliquée en essayant de calculer deux approximations des ensembles Kill_f et Gen_f correctes, avec une sous-approximation de Kill_f et une sur-approximation de Gen_f, à partir des relations *points-to* In et Out synthétisées en entrée et obtenues en sortie d'une procédure ? Comme nous l'avons indiqué en introduction, l'équation de flot de données de base

$$\text{Out} = (\text{In} - \text{Kill}_f) \cup \text{Gen}_f \tag{6.1}$$

peut être satisfaite ou sur-approximée par de nombreux ensembles Kill et Gen. Nous avons déjà utilisé l'ensemble vide comme sous-approximation de Kill pour définir des analyses simplifiées (voir sections 6.1.1 et 6.1.3). Nous pouvons maintenant utiliser Out comme sur-approximation de Gen sans modifier la précision du résultat puisque l'équation précédente est toujours satisfaite, Out constituant une borne supérieur pour Gen qui par définition doit être inclus dedans :

$$\text{Out} = (\text{In} - \text{Kill}_f) \cup \text{Out} \tag{6.2}$$

Le calcul de Kill est plus délicat parce que l'équation

$$\text{Kill}_f = \text{In} - \text{Out} \tag{6.3}$$

n'est pas monotone et parce que les ensemble In et Out ne sont pas calculés pour le même état mémoire. En ce qui concerne la monotonie, l'ensemble In peut être considéré comme exact puisqu'il est synthétisé et définit le contexte formel de la fonction en cours d'analyse, tant que l'hypothèse de non-aliasing est vérifiée.

Il faudrait néanmoins être prudent vis-à-vis des tableaux de pointeurs et des structures de données récursives qui imposent des approximations pour maintenir une représentation finie de la relation *points-to*. L'ensemble Out ne pose pas de problème puisqu'il apparaît derrière le symbole $-$, qu'il constitue une sur-approximation et qu'on souhaite obtenir une sous-approximation de Kill_f.

En ce qui concerne l'évolution de l'état mémoire, il va falloir prendre un sous-ensemble de l'ensemble In ne contenant que des arcs dont l'origine n'aura pas été modifiée au cours de l'exécution de la procédure d'après l'ensemble d'effets Written. Cette restriction est brutale et elle mériterait une comparaison avec une approche ascendante plus normale consistant à synthétiser les ensembles Kill et Gen ou avec une approche consistant à conserver de l'information explicite sur les cellules mémoires libérées, mais elle permet néanmoins de préserver dans les cas réalistes l'information correspondant à une libération d'une cellule du tas ou à une contrainte de la norme C.

On pourrait aussi se demander s'il faut effectuer les traductions de l'information *points-to* du cadre de la fonction appelée vers celui de la fonction appelante avant ou après le calcul des ensembles Gen et Kill. La traduction ajoutant des imprécisions, il nous semble préférable de

calculer les ensembles Gen et kil dans le cadre de la fonction appelée puis de les traduire ensemble dans le cadre de la fonction appelante.

Il faut enfin et surtout prendre en compte le fait que la fonction appelée peut avoir des effets au-delà de son contexte formel In, dans $\text{pt}_{\text{caller}}$, la relation *points-to* connue au niveau du site d'appel. Par exemple, la libération d'une cellule mémoire par un appel peut modifier des arcs pointant vers cette cellule et des arcs sortant de cette cellule, les uns et les autres n'ayant aucune raison d'être générés dans le contexte formel In. Des modifications d'arcs se traduisent par des éléments dans les ensembles effectifs Kill_e et Gen_e qui ne peuvent absolument pas être dérivés des ensembles Kill_f et Gen_f présentés ci-dessus, mais qui nécessitent l'ensemble Written et peut-être aussi l'ensemble des cellules libérées, $Freed$.

L'analyse de Wilson [WHI95] s'inscrit dans le cadre des analyses interprocédurales descendantes sensibles au contexte par résumés et elle présente de nombreuses similarités avec notre algorithme de traitement des sites d'appels. Mais le calcul interprocédural de l'information *points-to* n'est détaillé ni dans la publication de Wilson [WHI95] ni dans le chapitre de sa thèse [Wil97] qui concerne les appels de procédure. Pour autant que nous le sachions, ces publications ne permettent donc pas de reconstruire les algorithmes sous-jacents. C'est pourquoi nous proposons, à la suite de Wilson qui utilise des résumés, un algorithme détaillé original qui calcule de l'information *points-to* après un appel de fonction dans le cadre inhabituel d'une analyse ascendante, respectueuse du code source.

Nous illustrons notre schéma interprocédural par la figure 6.4.

```
void caller()
{
int *e1, *e2, *e3, i = 1, j = 2;

S1 : e1 = &i;

S2 : e2 = &j;

//ptcaller (1)                 //ptbound (2)
S3 : e3 = callee(e1, e2) ;    ------------->  //IN (3)
//ptend (5)                    //binding (3)   int* callee(int *f1, int *f2)
S7:                                            {
}                                              S4: f1= malloc(5*sizeof(int));
                                               S5: ...
                                               S6: return f1;
                                               }
                                               //OUT (4)
```

FIGURE 6.4 – Le schéma interprocédural

Nous y distinguons :

- les informations *points-to* intraprocédurales au niveau de l'appelant *caller*, $\text{pt}_{\text{caller}}$ (In du site d'appel) et pt_{end} (Out du site d'appel) ;
- un résumé au niveau de l'appelée *callee*, valable en l'absence d'aliasing au niveau du site d'appel, et comportant trois ensembles $(\text{In}, \text{Out}, \text{Written})$ où Written est l'ensemble des pointeurs écrits par la fonction appelée ;
- l'information interprocédurale, pt_{bound}, qui permet l'association entre paramètres effectifs et paramètres formels et qui permet de construire la traduction des ensembles *points-to* définis pour la fonction appelée en ensembles *points-to* pour la fonction appelante ; cette association va au-delà des paramètres et prend en compte tout le contexte formel de la procédure appelée, i.e. ses *points-to stubs* (voir section 6.3.4) ; cette association peut nécessiter une modification du contexte formel de la procédure appelante, comme pour n'importe quelle autre instruction ;

- la fonction binding qui associe à chaque élément du contexte formel son correspondant au niveau du site d'appel ; cette fonction est construite avec des constantes ; elle ne dépend donc pas de l'état mémoire courant et peut donc être utilisée au niveau du point de retour de la fonction appelée bien qu'elle soit construite avec des informations disponibles au niveau de son point d'entrée.

Dans la suite du chapitre, après avoir indiqué les notations utilisées (section 6.3.1), nous allons définir au fur et à mesure les algorithmes correspondant :

1. au cadre interprocédural et à ses différentes étapes de calcul pt_{caller}, pt_{bound}, In, Out et pt_{end} (section 6.3.2) ;
2. à l'étape de construction de pt_{bound} qui prend en compte la relation entre paramètres formels et effectifs (section 6.3.3) ;
3. à l'étape de construction de la fonction de traduction binding (section 6.3.4) ;
4. au test de compatibilité vis-à-vis de l'aliasing entre pt_{bound} et l'information *points-to* In du *callee* construite intraprocéduralement (section 6.4) ;
5. au calcul du résultat d'un site d'appel, pt_{end}, dérivé de pt_{caller}, grâce à la fonction de traduction précédemment construite et aux ensemble Kill et Gen globaux pour la procédure appelée, dérivés des relations In et Out et de l'ensemble Written de la procédure appelée (section 6.5).

6.3.1 Notations

Nous présentons dans cette section les notations utilisées pour désigner les ensembles des adresses utilisés lors du calcul interprocédural.

6.3.1.1 Les ensembles de base

Une partie de ces ensembles, $\mathcal{A}$, $\mathcal{E}$ et $\mathcal{H}$, a déjà été définie au niveau du chapitre de l'analyse intraprocédurale simple 4.

6.3.1.2 Relations *points-to*

Toute relation peut être vue comme un ensemble via son graphe.

6.3.1.3 Les ensembles In et Out intégrés dans le résumé de l'appelée

L'analyse interprocédurale du code requiert des ressources calculées lors de l'analyse intraprocédurale (voir la section 6.1.5 pour les règles de communication entre les deux analyses). Plus particulièrement les ressources `points_to_in` et `points_to_out`.

L'ensemble In Ce graphe correspond à la ressource `points_to_in` pré-calculée par l'analyse intraprocédurale à l'entrée d'une fonction. Cet ensemble contient initialement les arcs *points-to* des paramètres formels de type pointeur. Mais au cours de l'analyse du corps de la fonction cet ensemble peut être augmenté par des initialisations à la demande de variables qui appartiennent aux paramètres formels (par exemple un champ pointeur d'une structure qui est passée en argument) ou de variables globales.

CS	ensemble des sites d'appel (call sites) ; un site d'appel permet d'accéder à l'appelant, l'appelé, les paramètres formels ainsi qu'aux paramètres effectifs. L'ensemble des fonctions du chapitre sont définies par rapport à un site d'appel.
$\mathcal{A}$	ensemble des adresses relatives à une fonction donnée, ici la fonction appelante, *caller*, en intraprocédural, la fonction courante. Une adresse est construite à partir d'une variable prise dans I en y ajoutant des indexations pouvant correspondre soit à un élément de tableau, soit à un champ de structure. Ces adresses sont aussi appelées *chemin d'accès constants* à la mémoire. Tous les ensembles décrits par la suite sont inclus dans $\mathcal{A}$, mais sont quand même cités pour plus de précision.
E	un ensemble de noms pour des éléments de $\mathcal{P}(A)$. La connexion est établie pour chaque site d'appel en fonction des paramètres effectifs. E correspond aussi à l'ensemble des chemins constants construits avec ces identificateurs appelés stubs.
$\mathcal{F}$	ensemble des paramètres formels d'une fonction appelée ; les paramètres formels de la fonction appelante sont considérés comme des éléments de $\mathcal{A}$. L'ensemble représente aussi les chemins constants construits à partir des paramètres formels et utilisant des *stubs points-to*. Il est défini par rapport à la fonction appelée.
$\mathcal{G}$	ensemble des adresses des variables globales de l'application ; la fonction appelante et la fonction appelée peuvent avoir accès à des sous-ensembles de $\mathcal{G}$.
$\mathcal{H}$	ensemble des *buckets* alloués dans la zone de mémoire dynamique, le tas (*heap*). En fait, $\mathcal{H}$ peut être un ensemble de noms pointant vers un ensemble de parties d'un autre ensemble. Il correspond à l'ensemble des *buckets* alloués à une ligne particulière d'une fonction. On peut être éventuellement plus précis en prenant en compte la pile des appels. Les éléments h de $\mathcal{H}$ comme les éléments e de E peuvent représenter une ou plusieurs zones mémoires.

Tableau 6.1 – Les ensembles du chapitre interprocédural

$\text{pt}_{\text{caller}} : \mathcal{A} \rightarrow \mathcal{PT}$	ensemble des arcs *points-to* calculé, intraprocéduralement, au niveau du site d'appel
$\text{pt}_{\text{bound}} : (\mathcal{A} \cup \mathcal{F}) \rightarrow \mathcal{PT}$	ensemble des arcs *points-to* calculé, en établissant la correspondance
Written	ensemble des pointeurs qui ont été écrits au niveau de l'appelé et qui sont visibles au niveau du site d'appel
In : $(\mathcal{F} \cup E) \rightarrow \mathcal{PT}$	ensemble des arcs *points-to* calculé intraprocéduralement, en entrée de l'appelée
Out : $(\mathcal{A} \cup \mathcal{F} \cup E \cup \{r\}) \rightarrow \mathcal{PT}$ [12]	ensemble des arcs *points-to* calculé, intraprocéduralement, en entrée de l'appelée entre les paramètres effectifs et formels
pt_{end}	ensemble des arcs *points-to*, au niveau du site d'appel, après traduction du résultat de l'analyse interprocédurale
binding	ensemble de couples (emplacement_appelé, emplacement_appelant) ; cet ensemble permet la traduction de l'ensemble Gen au niveau du site d'appel

Tableau 6.2 – Les notations du chapitre interprocédural

L'ensemble Out Ce graphe correspond à la ressource `points_to_out` pré-calculée par l'analyse intraprocédurale à la sortie d'une fonction. Il s'agit de l'ensemble obtenu après la dernière instruction de la fonction, auquel on ajoute, si la fonction renvoie un pointeur, un arc *points-to* spéciale qui a comme source une entité spécial dans PIPS qui désigne la valeur de retour et qui a comme destination la valeur pointée par le pointeur retourné. Après la construction de cet ensemble ses éléments sont projetés selon leur portée. En effet, seuls les arcs *points-to* qui

peuvent être visibles au niveau du site d'appel sont conservés. Ceci inclut la valeur de retour, les arcs avec des sources au niveau du tas ou des pointeurs de pointeurs. La fonction de projection des arcs *points-to* est la suivante :

Fonction 25 : points_to_function_projection($\mathcal{PT}$: P)

$source, sink : \mathcal{A}$
$res : \mathcal{PT}$
$res = P$
for $pt \in P$ **do**
  if $out_of_scope_p(points_to_source(pt)$ **then**
    $source = points_to_source(pt)$
    $t =$ type_of$(source)$
    if $\neg t = array(t')$ & $\neg formal_parameter_p(source)$ **then**
      $sink = points_to_sink(pt)$
      if $\neg undefined_p(sink)$ **then**
        $res = res - \{pt\}$
return res

6.3.1.4 Conclusion

Après avoir défini les ensembles qui interviennent dans le calcul interprocédural des *points-to* (In,Out, Written, binding, pt_{end}, $\text{pt}_{\text{caller}}$) ainsi que l'algorithme qui consiste à utiliser les résumés *points-to* de l'appelée et à traduire au niveau du site d'appel les arcs dont les sommets sont visibles au niveau de l'appelant. Nous détaillons, dans la suite du chapitre, les algorithmes impliqués dans le traitement d'un site d'appel, en commençant par l'algorithme de traitement global d'un site d'appel.

6.3.2 Algorithme de traitement d'un site d'appel

L'analyse interprocédurale commence par la définition des ensembles nécessaires à son déroulement. Il faut compléter l'ensemble $\mathcal{A}$ des adresses constantes d'une fonction par l'ensemble $\mathcal{F}$ des paramètres formels, l'ensemble E des adresses pointées par les paramètres formels ainsi qu'un élément spécial, r, la valeur retournée, quand elle est de type pointeur. L'ensemble E est construit à la demande lors de l'analyse de la fonction appelée. [13]

Considérons un site d'appel à la fonction *callee*, localisé dans la fonction appelante, *caller* (voir figure 6.4). Les ensembles In et Out calculés précédemment par l'analyse intraprocédurale sont chargés pour être utilisés par l'analyse interprocédurale. S'en suit ensuite la traduction des paramètres formels qui va permettre la traduction de tout l'ensemble In. Et avant de commencer l'analyse interprocédurale l'hypothèse sur l'*aliasing* entre les paramètres formels est vérifiée via la fonction `aliased_translation_p`. Après la vérification l'analyse interprocédurale complète le contexte formel s'il n'est pas construit en entier et traduit les ensembles Gen et Kill au niveau du site d'appel.

13. Les variables globales, $g \in \mathcal{G}$, sont négligées dans un premier temps. Au cours de l'analyse intraprocédurale de l'appelée les variables globales sont initialisées à la demande c.à.d que chaque fois qu'on déréférence un pointeur global on lui crée à la volée l'arc *points-to* nécessaire, comme c'est le cas pour les paramètres formels.

Fonction 26 : points_to_call_site

points_to_call_site(call c, set pt_{caller} , set Written)
let f = call_function(c)
let In = db_get_memory_resource(DBR_POINTS_TO_IN, f)
let Out = db_get_memory_resource(DBR_POINTS_TO_OUT, f)
let eal = effective_arguments_list(f)
let fpcl = formal_parameters_cells_list(c)
let pt_{bound} = points_to_pt_bound(f, eal, pt_{caller})
let translation = formal_parameter_tranlation(fpcl, eal, pt_{caller}, binding)
let (translation, In) =
filter_formal_context_according_to_actual_context(fpcl,In,pt_{bound},binding)
if *aliased_translation_p(fpcl,pt_{bound})* **then**
  pt_{end}= points_to_call_site_intraprocedural(c,pt_{caller})
else
  let Out= filter_formal_out_context_according_to_formal_in_context(Out,In,Written,f)
  let Kill= points_to_kill(Written, pt_{caller}, translation)
  let Gen= points_to_gen(Out, Written, translation, f)
  let pt_{end} = (pt_{caller} - translation(Kill, binding)) ∪ translation(Gen,binding)
return pt_{end}

6.3.3 Fonction de transfert *points-to* associée à une fonction C

Notre fonction de transfert est un quadruplet associant un contexte *points-to* à l'appelée constitué des résumés Inet Out. Du contexte d'appel de la fonction défini par pt_{caller} pour produire des nouveaux arcs traduits au niveau du site d'appel qui constituent l'ensemble pt_{end}.

Les publications de Wilson ne présentent aucun détail de son algorithme, qu'il s'agisse de la construction de pt_{bound} d'une part, ou de l'utilisation de pt_{bound} pour construire pt_{end} à partir de Out d'autre part. La manière dont la compatibilité de pt_{bound} et In est vérifiée n'est pas détaillée non plus, ni la manière dont pt_{end} est calculé.

L'algorithme de Wilson construit la fonction de transfert à la demande quand le graphe des appels est parcouru de manière descendante. Expérimentalement, d'après Wilson [WHI95], il est rarement nécessaire de construire plus d'un couple $(\mathrm{In}, \mathrm{Out})$. Cette information est très encourageante pour nous, puisque nous proposons de ne calculer qu'un triplet comme transformeur, mais nous ne voyons pas comment l'ensemble Written peut ne pas être utilisé tout en obtenant des résultats précis.

Pour expliquer brièvement l'algorithme de Wilson, il faut suivre les étapes mentionnées sur la figure 6.4. En particulier les étapes 4 et 5. Les résumés In et Out permettent de décrire le comportement et l'effet de la fonction sur les paramètres, les variables globales ou encore les variables allouées au niveau du tas. Au moment de la traduction, la fonction de transfert est utilisée pour représenter cet effet au niveau du site d'appel.

6.3.4 Construction de pt_{bound} et de binding

Deux approches sont possibles pour capturer les correspondances entre paramètres formels et paramètres effectifs. La première est fondée sur l'approche *points-to* et la seconde sur de l'information d'aliasing, appelée ici binding ou association entre paramètres formels et effectifs.

6.3.4.1 pt$_{\text{bound}}$

La construction de pt$_{\text{bound}}$ s'effectue en affectant l'expression correspondant au paramètre effectif ep au paramètre formel fp comme par une instruction $fp = ep$. Comme les deux paramètres sont de type pointeur ou bien contiennent des pointeurs, cette instruction a pour effet de faire pointer fp vers les cibles de ep. La valeur initiale de pt$_{\text{bound}}$ est pt$_{\text{caller}}$.

Si le paramètre formel f est appelé avec une expression x comme paramètre effectif et qu'il n'est pas de type tableau, on pourrait définir une variable cachée de copie f_0 pour garder trace de sa valeur à l'entrée et appliquer les deux affectations avant d'effectuer l'analyse afin de garder trace des modifications de l'état mémoire dans le cas où les paramètres formels sont modifiés par la procédure appelée.

$$f = x;$$
$$f_0 = f;$$

Comme nous l'avons expliqué précédemment, nous avons plutôt choisi d'utiliser l'ensemble Written qui a l'avantage de couvrir aussi les modifications des valeurs associées aux *stubs* du contexte formel.
La construction de pt$_{\text{bound}}$ s'effectue en ajoutant à pt$_{\text{caller}}$ les arcs *points-to* résultats des affectations et ceci pour chaque paramètre formel. L'expression x, qui sert de paramètre effectif, peut être de type pointeur, structure, tableau de pointeurs, tableau de structures, prise d'adresse ou arithmétique sur pointeur, affectation de structure, voire une expression avec effet de bord [14]. Tous ces cas sont pris en compte par le traitement de l'affectation. Pour résumer, l'algorithme qui permet de calculer pt$_{\text{bound}}$ est le suivant :

Fonction 27 : compute_pt_bound

compute_pt_bound (list args, list params, graph pt$_{\text{caller}}$)
foreach $p \in$ *parms* **do**
  `find the argument that corresponds to p`
  let arg = find_ith_argument(p, args)
  let Stm = create_assignment(p, arg)
  let pt$_{\text{bound}}$ = set_union(pt$_{\text{bound}}$, points_to_assignment(stm, pt$_{\text{caller}}$))
return pt$_{\text{bound}}$

6.3.4.2 Calcul de la relation de traduction binding

La construction de la relation binding se fait récursivement après le calcul de pt$_{\text{bound}}$. Elle permet d'associer à chaque sommet apparaissant dans un arc de In, le contexte formel d'entrée, son équivalent au niveau de l'appelant, quand il existe [15].

L'algorithme parcourt simultanément les graphes *points-to* de l'ensemble pt$_{\text{bound}}$ et de In et crée au fur et à mesure une relation « alias » [16] *points-to* entre les éléments [17]. Le champ définissant l'approximation permet de savoir si la traduction est unique ou si plusieurs cibles

14. Si le paramètre formel f est une structure, on applique la même technique, l'affectation devenant une affectation de structure.
15. Certains paramètres effectifs permettent de construire une équivalence avec le paramètre formel. Par exemple, l'appel `foo(p)` à une fonction définie par `void foo(int *q)` permet de construire l'équivalence entre p et q. Par contre, l'appel par `foo(&i)` ne le permet pas.
16. dite aussi synonymes
17. La relation alias est implémentée par la même structure de données que la relation *points-to*

réelles peuvent être concernées par un élément du contexte formel de l'appelée. A partir de pt_{bound}, nous pouvons déduire le premier couple en cherchant la relation *points-to* dont la source est un paramètre formel f ainsi que celle dont la source est le paramètre effectif e, correspondant à f et ayant la même destination. Ces deux sources forment le premier élément de binding. A partir de ces deux sources nous allons parcourir les graphes In et pt_{bound} en associant leurs destinations respectives. Si ces destinations apparaissent aussi comme sources de nouveaux arcs *points-to*, alors leurs destinations sont liées à leur tour et ainsi de suite jusqu'à traduire tous les paramètres formels et tous les stubs apparaissant dans l'ensemble In, le contexte formel de l'appelée. En cas de besoin, le contexte formel de l'appelant est complété pour répondre aux besoins de traduction du contexte formel de l'appelant.

6.3.4.3 Exemple pour le calcul de pt_{bound}

Soit l'appel `foo(pp, qq)` pour une fonction `void foo(int *pi, int *pj)` tel que l'on a déclaré dans le programme 6.40 ci-dessous. Les arcs *points-to* obtenus au niveau du site d'appel

```
void foo(int ** pi, int ** pj) {
  int *r, *s;
  r = *pi;
  s = *pj;
  **pi=1, **pj=2;
  return;
}

int main() {
 int **qq, **pp, *q, *p, i = 0, j = 1;
  p = &i;
  q = &j;
  pp = &p;
  qq = &q;
  foo(pp, qq);
  return 0;
}
```

Prog 6.40 – Exemple pour le calcul de pt_{bound}

à `foo` sont :

```
// Points To:
// p -> i , EXACT
// pp -> p , EXACT
// q -> j , EXACT
// qq -> q , EXACT
```

Prog 6.41 – Les arcs *points-to* pour la fonction `foo`

L'ensemble $\text{pt}_{\text{caller}}$ qui a été calculé intraprocéduralement au point programme qui précède l'appel à la fonction `foo` vaut :

$$\text{pt}_{\text{caller}} = \{(p,i),(pp,p),(q,j),(qq,q)\}$$

A partir de $\text{pt}_{\text{caller}}$ et en appliquant les deux affectations `pi=pp` et `qq=pj`, l'ensemble pt_{bound} obtenu est :

$$\text{pt}_{\text{bound}} = \{(pi,p),(p,i),(pj,q),(pp,p),(q,j),(qq,q)\}$$

Il est plus pratique de garder dans pt_{bound} l'information contenue dans $\text{pt}_{\text{caller}}$ pour faciliter l'opération inverse qui est de traduire pt_{end} au niveau du site d'appel après l'analyse de l'appelée.

6.3.4.4 Exemple de calcul de la relation binding

Après le calcul de pt_{bound} et en utilisant la relation In crée par l'analyse de la fonction appelée :

$$\text{In} = \{(_pi_1, _pi_1_1)(_pj_2, _pj_2_2), (pi, _pi_1), (pj, _pj_2)\}$$

nous pouvons construire l'ensemble binding en suivant les arcs des deux graphes en partant des paramètres formels. Pour cela nous allons suivre les arcs des graphes In et pt_{bound} décomposés au niveau des figures 6.5 et 6.6.

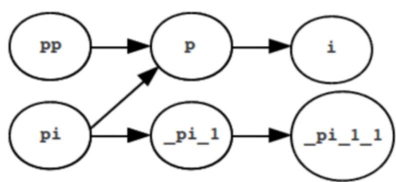


FIGURE 6.5 – L'ensemble In et pt_{bound} pour la variable `pi`

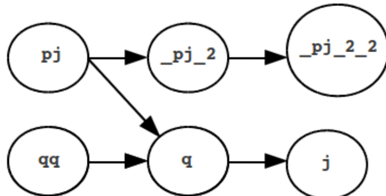


FIGURE 6.6 – L'ensemble In et pt_{bound} pour la variable `pj`

En associant chaque emplacement de In à son correspondant dans pt_{bound} l'ensemble binding obtenu est le suivant :

$$\text{binding} = \{(pi, pp), (p, _pi_1), (i, _pi_1_1), (pj, qq), (_pj_2, q), (_pj_2_2, j)\}$$

Cet ensemble permet la traduction de Kill et de Gen pour effectuer le calcul de pt_{end} en remplaçant chaque élément de $\mathcal{F}$ ou de E par son image dans $\mathcal{A}$.

6.3.4.5 Utilisation de la fonction binding

La fonction binding est utilisée par la suite pour traduire des cellules et des arcs de la fonction appelée dans le repère de l'appelante. La traduction proprement dite est appelée $translation$ pour les deux types d'arguments ou ensembles de tels arguments afin de simplifier la présentation, la surcharge étant facile à lever.

La fonction $translation$ n'est pas égale à :

$$translation(C, \text{binding}) = \{a \in A | \exists c \in C\ t.q.(a, c) \in \text{binding}\}$$

En effet, l'ensemble des cellules est plus grand que $F \cup E$ à cause des expressions d'indices. Par exemple, si c est un champ de structure, le champ doit être reporté dans a.

Notons que $translation(c, \text{binding})$ n'est pas forcément un singleton. Quand c'est le cas, nous disons que la traduction est exacte. La fonction $exact_translation$ appliquée à un ensemble ne renvoie que les éléments exactement traduits :

$$exact_translation(C, \text{binding}) = \{a \in A | \exists! c \in C\ t.q.(a, c) \in \text{binding}\}$$

Son résultat peut être vide, alors que la fonction $translation$ doit toujours renvoyer un ensemble non vide, sauf en cas d'erreur de programmation comme le passage d'un pointeur `NULL`ou undefined.

La fonction $translation$ peut aussi prendre en compte les variables qu'il ne faut pas traduire et qu'on ne trouve pas dans binding comme la valeur retournée par la fonction, les cellules du tas et les variables globales.

6.3.4.6 Conclusion

La première phase de l'analyse interprocédurale consiste à calculer la relation binding qui relie les paramètres formels et tous les éléments du contexte formel de l'appelée aux destinations des paramètres effectifs, y compris ceux qui appartiennent au contexte formel de l'appelant. Cette relation sert à traduire au niveau du site d'appel les arcs *points-to* des ensembles Kill et Gen Dans la section suivante, cet ensemble est aussi utilisé pour vérifier la condition de compatibilité nécessaire à la suite du calcul interprocédural, à savoir l'absence, au niveau du site d'appel, d'aliasing visible par la fonction appelée.

6.4 Compatibilité entre binding et In : correction de la fonction de traduction

A l'entrée de la fonction appelée les hypothèses d'aliasing faites sur les paramètres formels doivent être au préalable vérifiées afin de respecter la sémantique de la fonction. Il est possible de traiter le cas général que traite l'algorithme de Wilson, à savoir la compatibilité entre une fonction et un site d'appel, ou bien de traiter le cas particulier qui permet de vérifier que la relation binding n'implique pas d'« aliasing » [18] entre paramètres formels ou éléments du contexte formel. C'est cette dernière solution qui a été retenue.

On calcule pour chaque paramètre formel (et pour les variables globales) la liste des adresses concrètes ou abstraites pouvant être atteintes par la relation binding et on vérifie que les ensembles atteints ont tous deux à deux une intersection vide. En considérant la relation binding comme une fonction retournant une liste d'adresses mémoires, on demande donc :

$$\forall f \forall f' f != f' \Rightarrow \text{binding}(f) \cap \text{binding}(f') = \emptyset \tag{6.4}$$

6.4.1 Exemple pour le test de compatibilité de binding

Prenons comme exemple une fonction `foo(int *p, int *q)`, appelée au niveau de la fonction `main` par `foo(&i,&i)`. La construction de l'ensemble E par l'analyse intraprocédurale de `foo` donnera deux éléments `_p_1` et `_q_2` pointés respectivement par `p` et `q`. Comme il n'y a pas d'« aliasing » statique en `C`, les deux pseudo-variables `_p_1` et `_q_2` sont considérées comme deux entiers différents. En particulier, si l'analyse sémantique utilise l'information *points-to*, la séquence :

```
*p = 1;
*q = 2;
```

Prog 6.42 – Une séquence `C`

conduira à `_p_1` == 1 et `_q_2` == 2 et jamais à `_p_1` == `_q_2` == 2.

6.4.2 Deuxième exemple

En appliquant la formalisation au cas de la fonction `foo` (figure 6.40) et en définissant les relations par leurs graphes, nous obtenons :

$$\text{binding} \quad = \quad \{(pi, pp), (pj, qq), (_pi_1, p), (_pj_2, q), (_pi_1_1, i), (_pj_2_2, j)\}$$

18. On rappelle la définition d'« aliasing », dit aussi synonymie : elle consiste à avoir deux pointeurs ou plus qui pointent vers la même zone mémoire.

avec $A = \{i, j, pp, qq\}$, $F = \{pi, pj\}$ et $E = \{_pi_1, _pj_2, _pi_1_1, _pj_2_2\}$. On constate trivialement qu'aucun conflit n'existe. On constate aussi que si un conflit existait au niveau des entiers, il ne perturberait pas l'analyse *points-to*. La contrainte de non-aliasing utilisée est donc un peu trop forte, mais elle est utile pour protéger les analyses utilisatrices des effets.

Il y aurait de l'aliasing si l'appel avait été `foo(pp,pp)` ou bien si `pp` et `qq` pointaient vers le même élément, ou bien si c'était le cas de `p` et `q`.

Le cas de la fonction `foo` est trop simple et n'illustre pas à sa juste mesure la formalisation complexe définie précédemment. Pour mieux l'illustrer il faut utiliser un exemple avec pointeur de pointeur, par exemple un `int ** p`, pour mettre en évidence le mécanisme de traduction des éléments de E.

6.4.2.1 Conclusion

La formalisation du problème de compatibilité entre le site d'appel et le contexte formel de la fonction appelée est très simple puisqu'elle ne fait intervenir que la relation binding. Nous présentons dans la prochaine section la dernière étape du calcul interprocédural qui consiste à déterminer l'ensemble des arcs *points-to* qui peuvent être supprimés du site d'appel parce qu'ils ne sont certainement plus valides. Cet ensemble correspond à l'ensemble Kill. Le deuxième ensemble à déterminer est l'ensemble Gen qui comporte les nouveaux arcs à traduire au niveau du site d'appel. Et finalement l'ensemble pt_{end} qui est le résultat final traduit au niveau du site d'appel.

6.5 Calcul de Kill, Gen et pt_{end}

L'objectif est de proposer un algorithme qui maximise de manière sûre l'ensemble Kill et un autre algorithme qui minimise, toujours de manière sûre, l'ensemble Gen.

6.5.1 Calcul de Kill

Une approche intuitive consisterait à construire l'ensemble Kill à partir de $\text{pt}_{\text{caller}}$ en supprimant tous les arcs *points-to* dont la source a été écrite au niveau de l'appelé. Pour se faire, il faudrait utiliser l'ensemble des pointeurs écrits, exprimé dans le repère de l'appelant, à savoir $translation(\text{Written}, \text{binding})$ (voir la section 6.1.3). Cet ensemble serait ensuite soustrait à $\text{pt}_{\text{caller}}$ ce qui supprimerait tous les arcs *points-to* de $\text{pt}_{\text{caller}}$ dont la source a été écrite. Mais en tenant compte de l'analyse intraprocédurale, il est plus simple de traiter le problème comme une simple affectation et de considérer plusieurs ensembles Kill. Les destructions d'arcs peuvent provenir :

1. soit de sources qui n'ont pas été écrites par la fonction appelée mais dont l'ensemble des destinations a été modifié soit par des désallocations mémoire, soit par des contraintes d'exécution du langage C ;
2. soit de sources, autres que les paramètres formels, qui ont été certainement modifiées par la fonction appelée ;
3. soit de sources, autres que les paramètres formels, qui ont été certainement ou peut-être modifiées par la fonction appelée et qui correspondent à des arcs exacts ; ces arcs doivent être substitués par des arcs approchés qui, eux, doivent apparaître dans un ensemble Gen correspondant.

6.5.1.1 Calcul de Kill_1

L'ensemble Kill_1 est obtenu en projetant le contexte formel de la procédure appelée selon les cellules écrites pour ne garder que les sources inchangées et en soustrayant au résultat la relation *points-to* de sortie de l'appelée. Les arcs obtenus doivent ensuite être traduits dans le repère de l'appelée.

$$\text{Kill}_1 = translation(induced_subgraph(\text{In}, \overline{\text{Written}}) - \text{Out}, \text{binding})$$

Comme dans d'autres cas, on pourrait envisager d'effectuer la traduction plus tôt, avant la projection, sur In et Written, ou bien plus tard après l'union des trois différentes composantes de Kill. La deuxième alternative n'est pas possible à cause de la manière dont Kill_2 et Kill_3 sont calculés.

Rappelons que l'ensemble Out de l'exemple 6.27 a la valeur `p, undefined , EXACT` par conséquent l'arc `(p1, alloc_pointer:*HEAP*_l_7)` de $\text{pt}_{\text{caller}}$ doit être enlevé du résultat et remplacé par l'arc `(p1, undefined)` qui provient de la traduction de Out par binding (voir la sous-sous-section 6.5.2.1 sur le calcul de Gen_1).

6.5.1.2 Calcul de Kill_2

L'ensemble Kill_2 est dérivé de $\text{pt}_{\text{caller}}$ qui doit comporter plus d'arcs que le contexte formel In effectif pour le site d'appel [19]. Il faut obtenir l'ensemble des cellules mémoire qui sont certainement écrites, $\text{Written}_{\text{EXACT}}$, et le traduire dans le repère de l'appelant tout en maintenant l'exigence d'écriture exacte. Le processus de traduction peut donc réduire l'ensemble des cellules mémoire écrites. Il reste ensuite à projeter $\text{pt}_{\text{caller}}$ sur l'ensemble des cellules écrites.

$$\text{Kill}_2 = induced_subgraph(\text{pt}_{\text{caller}}, exact_translation(\text{Written}_{\text{EXACT}}, \text{binding}))$$

Comme les sources ont été écrites, ces arcs seront remplacés par des arcs de Gen_1.

Par exemple, pour `swap01`, les arcs (pi, i) et (pj, j) appartiennent à Kill_2, tandis que les nouveaux arcs (pi, j) et (pj, i) sont obtenus par traduction des arcs $(_p_1, _q_2_2)$ et $(_q_2, _p_1_1)$ de Out (voir programme 6.2).

6.5.1.3 Calcul de Kill_3

Dans le cas où l'on ne dispose pas d'information exacte sur l'effet de la fonction appelée, soit parce que la traduction est imprécise, soit parce que la fonction appelée génère un arc *points-to* imprécis, il faut penser à modifier l'approximation des arcs exacts se trouvant dans $\text{pt}_{\text{caller}}$ et dont la source a peut-être été modifiée. La modification de l'approximation passe formellement par l'ajout d'un arc dans Kill via Kill_3 et dans Gen via Gen_3.

$$\text{Kill}_3 = exact_filter(induced_subgraph(\text{pt}_{\text{caller}}, translation(\text{Written}, \text{binding})) - \text{Kill}_2)$$

Voici deux exemples montrant l'un l'impact d'une traduction imprécise, programme 6.43, et l'autre, programme 6.44, l'impact d'une analyse imprécise de la fonction appelée.

19. $\text{pt}_{\text{caller}}$ qui est défini au niveau de l'appelant est a priori plus riche que In qui a été synthétisé à partir de la fonction appelée. S'il existe un arc dans In qui n'exite pas dans $\text{pt}_{\text{caller}}$, on peut soit le retirer de In soit l'ajouter dans $\text{pt}_{\text{caller}}$. Par exemple, l'analyse intraprocédurale de la fonction appelée peut avoir prévu qu'un paramètre formel puisse pointer vers `NULL`. Il doit être supprimé s'il n'existe pas d'arc équivalent dans $\text{pt}_{\text{caller}}$. Par contre, la construction à la demande du contexte formel de la fonction appelante peut ne pas avoir encore abouti. S'il existe par exemple un arc $(fp, _fp_1)$ dans In et que fp est associé à un paramètre formel de l'appelante ap qui est lui-même un paramètre formel, il se peut que l'arc $(ap, _ap_1)$ n'existe pas encore et qu'il faille le créer.

```
void my_malloc(int **p)
{
  *p = (int *) malloc(sizeof(int));
  return;
}

int main()
{
  int i = 1, j = 2, *pi = &i, *pj = &j, **p

  pp = (i>j) ? &pi : &pj;
  my_malloc(pp);

  return 0;
}
```

Prog 6.43 – Exemple de traduction imprécise

```
void my_malloc(int c, int **fpp)
{
  if(c)
    *fpp = (int *) malloc(sizeof(int));
  return;
}

int main()
{
  int i = 1, *pi = &i, **pp= &pi;

  my_malloc(i, pp);

  return 0;
}
```

Prog 6.44 – Exemple d’analyse imprécise

Après analyse interprocédurale des deux programmes, nous obtenons les résultats ci-dessous.

```
int main() {
int i = 1, j = 2, *pi = &i, *pj = &j, **pp;
// Points To:
// pi -> i , EXACT
// pj -> j , EXACT
// pp -> undefined , EXACT
   pp = i>j?&pi:&pj;

// Points To:
// pi -> i , EXACT
// pj -> j , EXACT
// pp -> pi , MAY
// pp -> pj , MAY
   my_malloc(pp);

// Points To:
// pi -> *ANY_MODULE*:*HEAP**ANYWHERE*,MAY
// pj -> *ANY_MODULE*:*HEAP**ANYWHERE*,MAY
// pp -> pi , MAY
// pp -> pj , MAY
   return 0;
}
```

Prog 6.45 – Les arcs *points-to* pour le programme 6.43

```
int main()
{

// Points To: none

   int i = 1, *pi = &i, **pp = &pi;
// Points To:
// pi -> i , EXACT
// pp -> pi , EXACT


   my_malloc(i, pp);

// Points To:
// pi -> *ANY_MODULE*:*HEAP**ANYWHERE*,MAY
// pi -> i , MAY
// pp -> pi , EXACT
   return 0;
}
```

Prog 6.46 – Les arcs *points-to* pour le programme 6.44

Pour 6.43, l’instruction au niveau de l’appelée déréference l’argument `p` ; ceci implique une écriture sur la liste `{pi, pj}`. Par conséquent, on ne connaît pas exactement le pointeur qui va être mis à jour. Au niveau du site d’appel nous devrions conserver les anciens arcs des pointeurs `{pi, pj}` avec une approximation `MAY`.

6.5.2 Calcul de Gen

L’ensemble Gen se compose de plusieurs parties, Gen_i qui correspond aux arcs *points-to* créés ou modifiés par effet de bord de la fonction appelée, en établissant un mécanisme de traduction pour récupérer les arcs *points-to* qui sont visibles au niveau du site d’appel. Ceci

requiert l'utilisation de l'ensemble Written des pointeurs qui ont été écrits par la fonction ainsi que la liste des cellules libérées.

6.5.2.1 Calcul de Gen_1

L'ensemble Gen_1 comprend les nouveaux arcs *points-to* qui ont été créés au niveau de l'appelée et qui sont visibles au niveau de l'appelant. L'ensemble des arcs au point de sortie de l'appelée est Out. L'objectif est de minimiser la taille de cet ensemble pour maximiser la précision de l'analyse.

Il faut en supprimer les arcs où des paramètres formels modifiées par l'appelée apparaissent comme source. Pour cela une projection de cet ensemble est effectuée selon la composante $F' \cap$ Written qui comporte tous les paramètres formels passés par copie à la fonction, F', et modifiés par elle. Après cette projection il suffit de traduire les arcs restants dans le repère de l'appelant. Pour traduire ces arcs nous devons utiliser l'ensemble binding qui associe à chaque élément de F son correspondant au niveau du site d'appel. Nous réutilisons $translation$, la fonction qui permet de chercher dans binding la traduction des nœuds de Out.

$$\text{Gen}_1 = translation(induced_graph(\text{Out}, \overline{F' \cap \text{Written}}), \text{binding}) \quad (6.5)$$

Dans l'exemple 6.27, la projection préserve, avant traduction, l'arc $(fp, undefined)$, qui devient $(ap, undefined)$ dans Gen_1. Cet ensemble est complémentaire de Kill_1.

6.5.2.2 Calcul de Gen_2

Il ne faut pas perdre l'information portée par la pseudo-variable, `return_value`, qui porte l'information retournée par la fonction. Elle ne doit pas être projetée mais ses cibles doivent être traduites. Cet ensemble est le résultat de l'algorithme d'affectation de pointeurs où la partie gauche correspond à la variable qui reçoit le résultat de la fonction, au niveau de l'appelant, et la partie droite correspond à la valeur de retour après traduction.

6.5.2.3 Calcul de Gen_3

L'ensemble Gen_3 est calculé en fonction de Kill_3 qui est un ensemble d'arc exacts qu'il faut transformeur en arcs MAY.

$$\text{Gen}_3 = downgrade_approximation(\text{Kill}_3)$$

où $downgrade_approximation$ vérifie que chaque arc de Kill_3 est et en fait une copie MAY.

6.5.2.4 Conclusion

Il ne reste plus qu'à faire l'union des différentes composantes de Kill qui ont toutes été calculées dans le repère de la fonction appelante. Ainsi nous obtenons :

$$\text{Kill} = \text{Kill}_1 \cup \text{Kill}_2 \cup \text{Kill}_3 \quad (6.6)$$

De même que pour Gen :

$$\text{Gen} = \text{Gen}_1 \cup \text{Gen}_2 \cup \text{Gen}_3 \quad (6.7)$$

6.5.3 Calcul de pt_{end}

Il est maintenant possible de traiter le site d'appel très simplement en utilisant l'équation habituelle que nous redonnons ci-dessous :

$$\text{pt}_{\text{end}} = (\text{pt}_{\text{caller}} - \text{Kill}) \cup \text{Gen} \quad (6.8)$$

6.6 Preuve de correction

La correction globale provient du respect de la monotonie à chaque étape de calcul, ce qui garantit d'obtenir une sur-approximation comme résultat.

La correction de chaque étape a été montrée localement, dans chacune des sections concernées.

6.7 Conclusion

Nous avons présenté plusieurs algorithmes permettant de traiter les sites d'appel lors d'une analyse de *points-to* ascendante par rapport au graphe des appels.

Notre approche pour effectuer un calcul interprocédural ascendant précis des relations *points-to* présente des similarités avec l'algorithme d'analyse descendante de Wilson dont nous n'avons malheureusement trouvé aucune description suffisamment détaillée pour justifier le mot d'algorithme. Les similarités s'arrêtent au fait que nous avons comme lui construit un transformeur ou résumé permettant d'abstraire les effets des fonctions et de ne pas les réanalyser systématiquement à chaque site d'appel.

Nous avons soigneusement étudié les problèmes que devait permettre de résoudre un transformeur de points-to. Nous avons détaillé et justifié tous les algorithmes utilisés. Nous avons mis en œuvre notre propre mécanisme de traduction des arcs ainsi que le test d'aliasing au niveau des sites d'appel.

Notre analyse est contrôlée par une propriété `ALIASING_ACROSS_FORMAL_PARAMETERS` qui stipule que l'analyse est effectuée sous l'hypothèse d'absence d'aliasing direct ou indirect entre les paramètres formels, une hypothèse plus forte que le simple qualificateur *restrict*.

Dans le cas contraire une approche conservatrice, assurée par l'analyse `FAST_INTERPROCEDURAL_POINTS_TO_ANALYSIS` est adoptée (section 6.1.3). Par conséquent nous ne pouvons plus tirer profit des résumés et l'analyse perd en précision. Heureusement ce cas de figure reste rare.

Cependant un autre problème lié aux boucles doit aussi être pris en compte ; celui des appels aux fonctions dans le corps de boucles. En effet, lors de l'analyse d'un corps de boucle le calcul interprocédural peut vite augmenter la taille du graphe *points-to*. Comme lors de la définition d'une propriété pour définir le nombre d'itérations effectuées sur le corps de boucle pour atteindre le point fixe, nous devons aussi définir une nouvelle propriété qui contrôle le nombre d'itérations que nous pouvons effectuer sur la fonction.

CHAPITRE 7

Implémentation et expériences

Dans les trois chapitres précédents, nous avons introduit les composants de notre analyse en utilisant quatre sous-ensembles de C, les langages $\mathcal{L}_0$, $\mathcal{L}_1$, $\mathcal{L}_2$ et $\mathcal{L}_3$. Ces langages supposent que les déclarations sont effectuées hors du corps des fonctions. Ce n'est pas le cas en C en général et encore moins pour C99. Il nous faut donc ajouter le traitement des portées pour prétendre traiter C. Cela consiste à projeter les arcs relatifs aux variables qui ne sont plus vivantes à la fermeture des blocs et à détecter d'éventuelles erreurs. Cette fonction est très proche de celle qui est utilisée pour les corps de fonction (voir section 6.3.1.3).

Dans ce chapitre nous présentons tout d'abord le paralléliseur interprocédural dans lequel nous avons implémenté notre analyse *points-to*. Il s'agit de PIPS (parallélisation interprocédurale de programmes scientifiques) dont on présente la vue générale, l'architecture logicielle ainsi que son outil de génération de données `Newgen` (section 7.1). Dans la deuxième partie du chapitre, les définitions données dans le chapitre interprocédural sont illustrées sur des exemples tirés d'applications scientifiques. Cinq exemples avec passage d'une information exacte au niveau du site d'appel sont introduits : un exemple de chaînage de listes (section 7.2), un exemple d'échange indirect via des pointeurs (section 7.3), un exemple de tableau dynamique dérivé des techniques utilisées dans les applications scientifiques (section 7.4), un exemple d'affectation de structures de données (section 7.5) et l'exemple de Wilson [Wil97](section 7.6).

7.1 Implémentation de l'analyse *points-to*

Dans cette section, nous présentons le cadre général dans lequel nous avons conçu notre analyse. Le cadre est le compilateur PIPS dont nous donnons une vue générale ainsi que l'architecture logicielle et les outils qu'il comporte et qui permettent la manipulation des structures de données et l'ordonnancement des phases de compilation.

7.1.1 Le compilateur PIPS

L'architecture logicielle du compilateur PIPS est illustrée par la figure 7.1. Les différentes fonctionnalités qu'il offre ainsi que ses domaines d'applications sont détaillés dans les sous-sections ci-dessous.

7.1.1.1 Vue générale

Le but du projet PIPS[IJT91][Iri93] est de développer un logiciel libre, ouvert et extensible pour analyser et transformer automatiquement les applications, en particulier scientifiques, mais aussi les applications de traitement de signal. Il permet la compilation, le « reverse-engineering », la vérification, l'optimisation et la parallélisation source à source des programmes. Ses analyses interprocédurales permettent d'améliorer la compréhension du code et la vérification des transformations automatiques dans le but de réduire le coût et le temps d'exécution. Les optimisations développées dans PIPS peuvent être appliquées à des applications écrites en Fortran ou en C.

PIPS effectue des analyses statiques qui produisent les graphes d'appel, les effets mémoires, les chaînes « use-def », les graphes de dépendance, les transformeurs, les préconditions, les

conditions de continuation, l'estimation de complexité, la détection des réductions et les régions (read, write, In et Out, MAY ou EXACT). Les résultats des analyses peuvent être affichés avec le code source, avec le graphe d'appel ou avec un graphe de flot de contrôle, ces graphes pouvant être affichés sous forme de texte ou de graphe. Plusieurs algorithmes de parallélisation sont disponibles, comme par exemple celui d'Allen&Kennedy ainsi que la distribution automatique de code et un prototype de compilateur HPF. Quant aux transformations de programmes, elles incluent la distribution des boucles, la privatisation des scalaires et des tableaux, le déroulement des nids de boucles partiel et complet, la permutation ou la normalisation de boucles, l'évaluation partielle, l'élimination du code mort ainsi que l'élimination de code inutile à l'aide des « use-def ».

7.1.1.2 Architecture logicielle

PIPS offre une architecture logicielle modulaire et évolutive (voir la figure 7.1). Le projet capitalise de nombreuses contributions faites au cours des années de la part de chercheurs, de doctorants et d'étudiants. Ces exigences ont requis une structuration en phases, où chaque phase manipule une représentation intermédiaire commune des programmes. Le mot phase a été choisi de préférence à passe dans la mesure où les passes d'un compilateur sont associées à des transformations de la représentation interne plutôt qu'à de simples enrichissements. Un problème majeur de cette architecture est qu'il faut déterminer l'ordre d'application des différentes phases du processus de parallélisation tout en maintenant la cohérence. PIPS adopte une approche innovante pour traiter ce problème : l'ordre est dirigé par la demande dynamique et automatiquement déduit des spécifications des dépendances interprocédurales qui existent entre les différentes phases. Cet ordre est calculé par un programme appelé *pipsmake*. Les utilisateurs peuvent envoyer des requêtes à *pipsmake* et obtenir des structures de données particulières ou bien appliquer des passes ou phases spécifiques. Chaque structure de données utilisée, produite ou transformée par une phase dans PIPS, comme l'arbre syntaxique abstrait, le graphe de flot de contrôle ou les chaînes *use-def*, est considérée comme une ressource sous le contrôle d'un gestionnaire de base de données appelé *pipsdbm*. L'emplacement où les données résident (mémoire ou fichier) est géré par *pipsdbm* qui assure la persistance en se basant sur les fonctions de sérialisation de `NewGen`.

Les structures de données effacées de la mémoire par effet de bord peuvent être récupérées à partir de leur copie sur disque. C'est aussi le cas des structures de données sauvegardées par des exécutions précédentes de PIPS, si elles sont encore cohérentes. Appelée par *pipsmake*, chaque phase commence par récupérer les ressources dont elle a besoin via la fonction `db_get` fournie par *pipsdbm*, effectue des traitements et déclare la disponibilité de son résultat via `db_put`. Chaque structure de données est attachée à son unité de programme. Pour le moment, les unités connues par *pipsmake* sont le programme en entier, un module donné (une routine Fortran ou une fonction `C`), les appelés (la liste des modules invoqués par un module donné) et les appelants (la liste des modules qui invoquent un module donné). *pipsmake* gère l'interprocéduralité de PIPS *via* la notion d'appelés et d'appelants.

PIPS offre plusieurs interfaces utilisateurs. La plus utilisée reste l'interface `tpips` qui permet d'écrire des scripts ordonnançant des phases. Ces interfaces permettent à l'utilisateur de choisir les programmes qu'il veut transformer ainsi que les options à appliquer. Bien sûr, toutes les transformations et les analyses restent sous le contrôle de *pipsmake* qui tient aussi compte des dépendances indirectes qui sont dues à l'interprocéduralité. Pour cela, *pipsmake* a toujours recours au graphe des appels du programme en cours d'analyse. Chaque fois que l'utilisateur spécifie les fichiers sources de son programme, PIPS calcule le graphe des appels.

Quant à *pipsdbm*, il gère les ressources qui doivent soit résider dans la mémoire, soit résider sous forme de fichiers, soit migrer entre les deux formes : les ressources pouvant migrer sont dites actives lorsqu'elles sont en mémoire , sinon elles sont dites inactives. L'utilisation d'une ressource nécessite que celle-ci soit active. Chaque phase dans PIPS, comme l'analyse du code

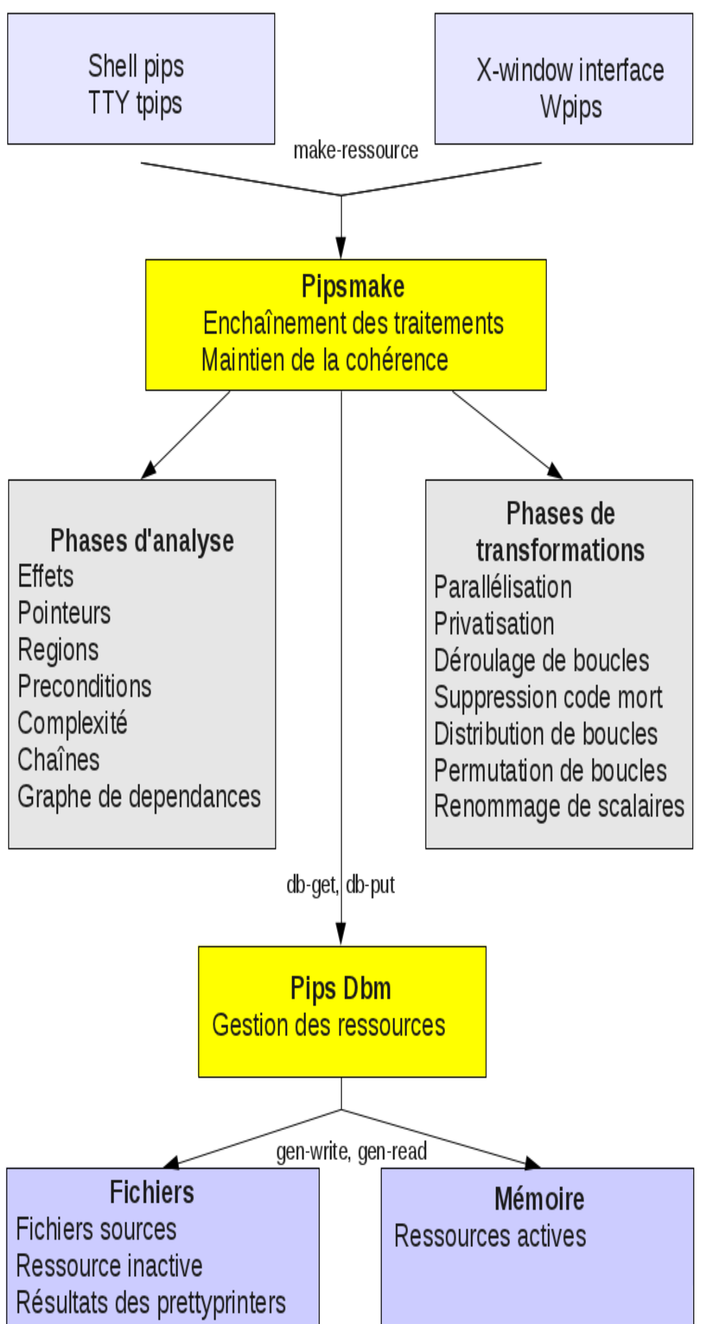


FIGURE 7.1 – Architecture logicielle de PIPS

source d'un programme, peut utiliser et/ou produire des ressources. Chaque phase est déclarée via des règles de production stockées dans un fichier de configuration qui est utilisé par pipsmake pour ordonnancer l'exécution de chaque fonction, selon les besoins de l'utilisateur et de la phase.

7.1.1.3 Représentation intermédiaire et outil `NewGen`

Au niveau de PIPS, pour la représentation intermédiaire des programmes, le métalangage de description des structures des données utilisé est le langage de définition `NewGen` [JT89]. L'outil `NewGen` permet à partir d'un ensemble de spécifications de types de données complexes défini par l'utilisateur la génération d'une bibliothèque de fonctions et de macros pour créer, initialiser, mettre à jour, supprimer, sérialiser et désérialiser les objets de ces types de données, perçus comme des types abstraits. `NewGen` permet l'utilisation des domaines basiques, à partir desquels des types plus élaborés peuvent être définis. Par exemple `NewGen` fournit les domaines basiques `int` et `float`. À partir de ces domaines prédéfinis, `NewGen` permet la définition des domaines produits, listes, fonctions ou tableaux multidimensionnels. Les principales caractéristiques de cet outil sont :

- le support de l'abstraction fonctionnelle ;
- la garantie d'un transfert de données qui facilite l'évolution du logiciel ;
- un environnement de programmation homogène.

Cet outil a été utilisé pour définir les structures de données choisies pour représenter les relations *points-to* dans notre analyse. Cette définition s'est faite en se fondant sur la représentation intermédiaire de PIPS [RI].

7.1.1.4 Conclusion

Notre analyse a été intégrée dans le compilateur PIPS, un framework de compilation et d'analyse de code. La particularité de ce dernier est l'aspect interprocédural de ses analyses qui permet de recueillir des informations sur le programme en entier. Dans la suite nous présentons des exemples avec les résultats *points-to* associés.

7.2 Chaînage d'une structure de données

Le premier exemple, programme 7.1, consiste à chaîner deux éléments dans une fonction appelée, `chain`, et à remonter l'information de chaînage dans la fonction appelante, `main`.

```
typedef struct foo {int ip; struct foo *next;} il_t, *ilp_t;

void chain(ilp_t c1, ilp_t c2) {
  c1->next = c2;
}

int main() {
  ilp_t x1 = (ilp_t) malloc(sizeof(il_t));
  ilp_t x2 = (ilp_t) malloc(sizeof(il_t));
  x1->next = NULL, x2->next = NULL;
  chain(x1,x2);
  return 0;
}
```

Prog 7.1 – Chaînage de deux éléments

7.2.1 Analyse intraprocédurale de la fonction `chain`

Le résultat calculé par la phase intraprocédurale pour le transformeur (In, Out) est, pour l'entrée, c'est-à-dire l'initialisation des paramètres formels, donné par le programme 7.2.

```
// Points To IN:
// c1 -> _c1_1 , EXACT
// c2 -> _c2_2 , EXACT
```

Prog 7.2 – In pour l'exemple 7.1

```
// Points To OUT:
// _c1_1.next -> _c2_2 , EXACT
```

Prog 7.3 – Out pour l'exemple 7.1

Pour la sortie, après projection des variables, c'est-à-dire après la suppression des variables locales qui ne sont pas visibles depuis le site d'appel, les arcs sont donnés par le programme 7.3. soit en termes de graphes de fonctions [1] :

$$\begin{aligned} \text{In} &= \{(c1, _c1_1), (c2, _c2_2)\} \\ \text{Out} &= \{(_c1_1.next, _c2_2)\} \end{aligned} \tag{7.1}$$

L'ensemble E des variables cibles, aussi appelées *stubs*, créées par l'analyse contient donc deux éléments :

$$E = \{_c1_1, _c2_2\}$$

7.2.2 Analyse intraprocédurale de la fonction `main`

L'information *points-to*, obtenue au point programme précédant l'appel à `chain` au niveau de la fonction `main` et notée $\text{pt}_{\text{caller}}$, est :

```
// Points To:
// *HEAP*_l_12.next -> *NULL_POINTER* , EXACT
// *HEAP*_l_13.next -> *NULL_POINTER* , EXACT
// x1 -> *HEAP*_l_12 , MAY
// x2 -> *HEAP*_l_13 , MAY
```

Prog 7.4 – $\text{pt}_{\text{caller}}$ pour l'exemple 7.1

qui peut se réécrire plus synthétiquement pour les éléments du tas [2] sous la forme :

$$\text{pt}_{\text{caller}} = \{(x1, h12), (x2, h13), (h12.next, \texttt{NULL}), (h13.next, \texttt{NULL})\}$$

Pour faciliter la lecture des arcs *points-to*, ils sont reportés au niveau du graphe figure 7.2. La présence du pointeur `NULL` ne modifie pas l'algorithme proposé parce qu'il est intégré comme une simple adresse particulière qui ne posera de problème que dans les fonctions cherchant à le déréférencer.

7.2.3 Analyse interprocédurale

Construction de pt_{bound} Pour obtenir pt_{bound}, les deux affectations `c1=x1;` et `c2=x2;` sont calculées en utilisant l'information contenue dans $\text{pt}_{\text{caller}}$:

$$\text{pt}_{\text{bound}} = \{(x1, h12), (x2, h13), (h12.next, \texttt{NULL}), (h13.next, \texttt{NULL}), (c1, h12), (c2, h13)\}$$

1. Les approximations ne sont pas mentionnées pour une meilleure représentation en graphe.
2. `*HEAP*_l_12` est abrégé en $h12$.

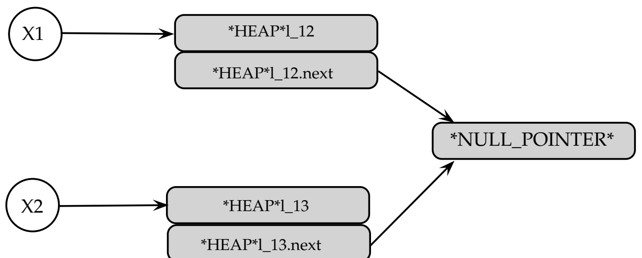


FIGURE 7.2 – Graphe *points-to* avant l'appel à `chain`

Construction de binding A partir de pt_{bound} et de In qui est de la forme suivante :

$$\text{In} = \{(c1, _c1_1), (c2, _c2_2)\}$$

Nous pouvons construire binding, qui associe à chaque élément de E et F le ou les éléments équivalents dans A :

$$\text{binding} = \{(c1, x1), (c2, x2), (_c1_1, h12), (_c2_2, h13)\}$$

Compatibilité entre pt_{bound} et In Pour vérifier que pt_{bound} ne contient pas d'« aliasing » il faut reprendre l'équation (6.4) et déterminer les correspondances entre les éléments de Out et ceux du site d'appel en utilisant binding. A partir de l'information contenue dans binding, nous définissons les chemins concrets suivants :

$$C_c(_c1_1) = \{h12\}$$

$$C_c(_c2_2) = \{h13\}$$

$$C_c(_c1_1) \cap C_c(_c2_2) = \emptyset$$

Comme l'intersection des deux chemins concrets est vide, nous concluons qu'il n'y a pas d'« aliasing » au niveau de pt_{bound}.

Calcul de Kill Pour calculer l'ensemble Kill, il faut d'abord appliquer $\text{pt}_{\text{caller}}$ à l'équation 4.11 :

$$\text{pt}_{\text{caller}} = \{(x1, h12), (x2, h13), (h12.next, \texttt{NULL}), (h13.next, \texttt{NULL})\}$$

puis récupérer les effets résumés de la fonction appelée :

```
//          <    is read   >: c1 c2
//          <    is written>: _c1_1.next
```

Prog 7.5 – Effets résumés pour l'exemple 7.1

À partir des effets résumés, seul l'effet d'écriture sur un pointeur est préservé pour obtenir l'ensemble Written suivant :

$$\text{Written} = \{_c1_1.next\}$$

Nous traduisons cet ensemble en appelant la fonction `Translation` et en utilisant l'information binding :

$$\text{Translation}(\{_c1_1.next\}, \text{binding}) = \{h12.next\}$$

Finalement, la case mémoire qui a été écrite au niveau du site d'appel est $h12.next$. D'où l'ensemble Kill suivant :

$$\text{Kill} = \{(h12.next, \texttt{NULL})\}$$

Calcul de Gen Pour calculer Gen, il faut reprendre l'ensemble Out ci-dessous :

$$\text{Out} = \{(_c1_1.next, _c2_2), (c1, _c1_1), (c2, _c2_2)\}$$

L'équation 6.5 est appliquée à l'ensemble Out pour traduire au niveau du site d'appel les nouveaux arcs *points-to* :

$$\text{Gen} = \{(h12.next, h13)\}$$

Calcul de pt_{end} Le calcul de pt_{end} se fait en combinant la nouvelle information *points-to* générée par la fonction appelée et l'information disponible au niveau du site d'appel et qui a été préservée. L'information finale, obtenue après la suppression des arcs « tués » par la fonction `chain` et l'ajout des nouveaux arcs, est la suivante :

$$\begin{aligned}\text{pt}_{\text{end}} &= \{(x1, h12), (x2, h13), (h12.next, \texttt{NULL}), (h13.next, \texttt{NULL})\} - \{(h12.next, \texttt{NULL})\} \\ &\quad \cup\{(h12.next, h13)\} \\ &= \{(h13.next, \texttt{NULL}), (h12.next, h13), (x1, h12), (x2, h13)\}\end{aligned}$$

qui donne bien l'état des deux structures de données allouées dans le programme principal et mises à jour par la procédure appelée. En termes de graphe *points-to*, ce résultat est traduit par la figure 7.3.

```
// Points To:
// *HEAP*_l_12.next -> *HEAP*_l_13 , MAY
// *HEAP*_l_12.next -> *NULL_POINTER* , MAY
// *HEAP*_l_13.next -> *NULL_POINTER* , MAY
// x1 -> *HEAP*_l_12 , MAY
// x2 -> *HEAP*_l_13 , MAY
```

Prog 7.6 – Les arcs *points-to* pour le programme 7.1

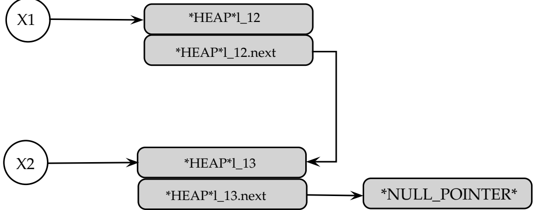


FIGURE 7.3 – Graphe *points-to* après l'appel à `chain`

7.3 Échange indirect

Cet exemple montre que nous pouvons traiter interprocéduralement des chemins d'accès à des champs de structures. Pour illustrer les modifications possibles de cases mémoires visibles depuis l'appelée, nous introduisons des doubles pointeurs passés en arguments de la fonction `swap`, voir le programme 7.7.

7.3.1 Analyse intraprocédurale de la fonction `swap`

Le résultat de l'analyse intraprocédurale en entrée de la fonction `swap` et qui correspond à l'initialisation des paramètres formels est illustré par le programme 7.8 :
L'ensemble In est réécrit comme suit :

$$\text{In} = \{(_p_1, _p_1_1), (_q_2, _q_2_2), (p, _p_1), (q, _q_2)\}$$

Le résultat de l'analyse intraprocédurale pour l'appel à *return* et après suppression des variables locales et des paramètres est :

```
void swap(int **p, int **q) {
  int *pt = *p;
  *p = *q;
  *q = pt;
  return;
}
int main() {
  int i = 1, j = 2, *pi = &i, *pj = &j;
  swap(&pi,&pj);
  return 0;
}
```

Prog 7.7 – Échange indirect

```
// Points To IN:
// _p_1 -> _p_1_1 , EXACT
// _q_2 -> _q_2_2 , EXACT
// p -> _p_1 , EXACT
// q -> _q_2 , EXACT
```

Prog 7.8 – In pour l'exemple d'échange indirect

```
// Points To OUT:
// _p_1 -> _q_2_2 , EXACT
// _q_2 -> _p_1_1 , EXACT
```

Prog 7.9 – Out pour l'exemple d'échange indirect

L'information peut être réécrite comme suit :

$$\text{Out} = \{(_p_1, _q_2_2), (_q_2, _p_1_1)\}$$

En termes de graphe, la mémoire est modélisée par la figure suivante.

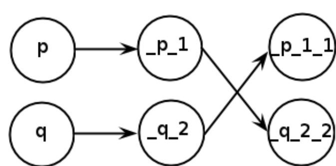


FIGURE 7.4 – Etat de la mémoire après l'analyse intraprocédurale de `swap`

A partir des élément de Out et In, nous pouvons déduire les éléments de E :

$$E = \{_p_1, _q_2_2, _q_2, _p_1_1\}$$

7.3.2 Analyse intraprocédurale de la fonction `main`

Le résultat de l'analyse intraprocédurale, au point programme qui précède l'appel à `swap` et qui correspond à l'ensemble $\text{pt}_{\text{caller}}$, est le suivant :

L'ensemble $\text{pt}_{\text{caller}}$ est réécrit en :

$$\text{pt}_{\text{caller}} = \{(pi, i), (pj, j)\}$$

```
// Points To:
// pi -> i , EXACT
// pj -> j , EXACT
```

Prog 7.10 – $\text{pt}_{\text{caller}}$ pour l'exemple d'échange indirect

7.3.3 Analyse interprocédurale

Construction de pt_{bound} Les deux affectations correspondant au site d'appel, `p=&pi` et `q=&pj`, sont appliquées à $\text{pt}_{\text{caller}}$ pour obtenir :

$$\text{pt}_{\text{bound}} = \{(pi, i), (pj, j), (p, pi), (q, pj)\}$$

Les éléments de E peuvent être remplacés par les variables équivalentes du programme appelant, en construisant binding à partir de In et pt_{bound} :

$$\begin{aligned} \text{In} &= \{(_p_1, _p_1_1), (_q_2, _q_2_2), (p, _p_1), (q, _q_2)\} \\ \text{binding} &= \{(_p_1, pi), (_q_2, pj), (_p_1_1, i), (_q_2_2, j)\} \end{aligned}$$

Compatibilité entre pt_{bound} et In En rappelant les éléments de pt_{bound} :

$$\text{pt}_{\text{bound}} = \{(p, pi), (pi, i), (q, pj), (pj, j)\}$$

et ceux de In :

$$\text{In} = \{(p, _p_1), (_p_1, _p_1_1), (q, _q_2), (_q_2, _q_2_2)\}$$

les chemins concrets correspondants aux éléments de In et renvoyés par binding sont :

$$\begin{aligned} C_c(_p_1) &= \{pi\} \\ C_c(_p_1_1) &= \{i\} \\ C_c(_q_2) &= \{pj\} \\ C_c(_q_2_2) &= \{j\} \end{aligned}$$

La conversion est à nouveau extrêmement simple parce que tous les ensembles intermédiaires sont des singletons. Il y a une correspondance directe entre les cibles formelles de l'appelé et les adresses de l'appelant.

Calcul de Kill Pour calculer l'ensemble Kill, il faut reprendre l'ensemble $\text{pt}_{\text{caller}}$ et lui appliquer l'équation 4.11 :

$$\text{pt}_{\text{caller}} = \{(pi, i), (pj, j)\}$$

puis récupérer les effets résumés de la fonction `swap` :

```
//          <    is read   >: _p_1 _q_2 p q
//          <    is written>: _p_1 _q_2
```

Prog 7.11 – Les effets résumés pour l'exemple d'échange indirect

A partir des effets résumés, l'ensemble Written est déduit en ne gardant que les effets d'écriture sur pointeurs :

$$\begin{aligned} \text{Written} &= \{_p_1, _q_2\} \\ \text{Translation}(\text{Written}, \text{binding}) &= \{pi, pj\} \end{aligned}$$

pour obtenir :

$$\text{Kill} = \{(pi, i), (pj, j)\}$$

Calcul de Gen Pour calculer Gen, il faut reprendre l'ensemble Out :

$$\text{Out} = \{(_p_1, _q_2_2), (_q_2, _p_1_1), (p, _p_1), (q, _q_2)\}$$

et lui appliquer l'équation 6.5 pour obtenir :

$$\text{Gen} = \{(pi, j), (pj, i)\}$$

Résultat L'information finale obtenue est :

$$\text{pt}_{\text{end}} = \{(pi, j), (pj, i)\}$$

Soit, en termes de graphe :

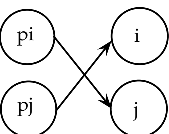


FIGURE 7.5 – Etat de la mémorie après l'appel à `swap`

Les deux pointeurs `pi` et `pj` ont échangé leurs cibles et l'information *points-to* après le site d'appel a été mise à jour en conséquence.

7.4 Tableau dynamique

Du fait de l'absence de tableau ajustable, c'est à dire de type dépendant, dans les normes `C` antérieures à `C99`, les tableaux ajustables sont souvent implémentés par des structures de données contenant les dimensions et une zone allouée dynamiquement. Ils sont très fréquents dans les applications scientifiques (voir le programme 7.12).

```
typedef struct {int dim; float *data;} darray_t, *parray_t;

void allocate_array(parray_t pa) {
  pa->data = (float *) malloc(pa->dim*sizeof(float));
  return;
}

int main() {
  parray_t ma = (parray_t)malloc(sizeof(darray_t));
  allocate_array(ma);

  return 0;
}
```

Prog 7.12 – Tableau dynamique

7.4.1 Analyse intraprocédurale de la fonction `allocate_array`

Le résultat calculé par la phase intraprocédurale pour le transformer (In, Out) pour la fonction `allocate_array` est :

```
// Points To IN:
// pa -> _pa_1 , EXACT
```

Prog 7.13 – In pour l'exemple de tableau dynamique

```
// Points To OUT:
// _pa_1.data -> *HEAP*_l_7[0] , EXACT
```

Prog 7.14 – Out pour l'exemple de tableau dynamique

soit, en termes de graphes de fonctions :

$$\begin{aligned} \text{In} &= \{(pa, _pa_1)\} \\ \text{Out} &= \{(_pa_1, h_7[0])\} \end{aligned}$$

L'ensemble E contient donc l'élément :

$$E = \{_pa_1\}$$

7.4.2 Analyse interprocédurale

Construction de pt_{bound} Pour le site d'appel de la fonction `allocate_array`, la correspondance entre paramètres formels et effectifs donne :

$$\begin{aligned} \text{pt}_{\text{bound}} &= \{(ma, h20), (pa, h20)\} \\ \text{binding} &= \{(ma, pa)\} \end{aligned}$$

Compatibilité entre pt_{bound} et In À partir des ensembles pt_{bound} et In, ainsi que l'information binding, nous pouvons déduire le chemin concret qui correspond à `_pa_1` :

$$\begin{aligned} \text{pt}_{\text{bound}} &= \{(ma, h20), (pa, h20)\} \\ \text{In} &= \{(pa, _pa_1)\} \\ C_c(_pa_1) &= \{h20\} \end{aligned}$$

Il n'y a qu'un seul argument ; l'intersection du chemin concret est vide.

Calcul de Kill pour `allocate_array` Pour calculer l'ensemble Kill, l'ensemble $\text{pt}_{\text{caller}}$ est repris et l'équation 4.11 lui est appliquée :

$$\text{pt}_{\text{caller}} = \{(ma, h20)\}$$

Les effets résumés obtenus pour `allocate_array` sont donnés par le programme 7.15.

```
//          <   is read  >: _pa_1.dim pa _MALLOC_EFFECTS:_MALLOC_
//          <   is written>: _pa_1.data _MALLOC_EFFECTS:_MALLOC_
```

Prog 7.15 – les effets résumés pour la fonction `allocate_array`

A partir desquels nous obtenons l'ensemble :

$$\text{Written} = \{_pa_1.data\}$$

Ainsi que Translation :

$$\text{Translation}(\{_pa_1.data\}, \text{binding}) = \{h20.data\}$$

pour obtenir :

$$\text{Kill} = \emptyset$$

Calcul de Gen Pour le calcul de Gen, il faut rappeler Out :

$$\text{Out} = \{(_pa_1.data, h_7[0]\}$$

Puis lui appliquer l'équation 6.5 pour obtenir :

$$\text{Gen} = \{(h_20.data, h_7[0])\}$$

Résultat L'information finale obtenue est :

$$\text{pt}_{\text{end}} = \{(ma, h20), (h_20.data, allocate_array : h_7[0])\}$$

Où $allocate_array : h_7[0]$ désigne le site d'allocation dynamique qui apparaît à la ligne 7 de la fonction $allocate_array$.

7.5 Affectation de structures

Abordons maintenant un exemple qui illustre le passage des structures de données en argument. La difficulté de cet exercice vient de la traduction des chemins d'accès complexes dus au besoin de modéliser les champs des structures de type pointeur :

```
struct foo {int *ip1; int *ip2;};
void assignment(struct foo **t1, struct foo **t2)
{
  (**t1).ip1 =(**t2).ip2;
}

int main() {
 struct foo s1, s2, **ppt, **pps, *pt = &s1, *ps = &s2;
 int i11 = 1, i12 = 2, i21 = 3, i22 = 4;
 s1.ip1 = &i11;
 s1.ip2 = &i12;
 s2.ip1 = &i21;
 s2.ip2 = &i22;
 *pt = s2;
 ppt = &pt;
 pps = &ps;
 assignment(pps, ppt);
 return 0;
}
```

Prog 7.16 – Affectation de structures

7.5.1 Analyse intraprocédurale de la fonction `main`

Le résultat calculé par la phase intraprocédurale avant l'appel à `assignment`, comprend l'initialisation des structures ainsi que de leur champs pointeurs est illustré par le programme 7.17. L'ensemble $\text{pt}_{\text{caller}}$ peut être réécrit comme suit :

$$\text{pt}_{\text{caller}} = \{(pps, ps), (ppt, pt), (ps, s2), (pt, s1), (s1.ip1, i21), (s1.ip2, i22), (s2.ip1, i21), (s2.ip2, i22)\}$$

```
// Points To:
// pps -> ps , EXACT
// ppt -> pt , EXACT
// ps -> s2 , EXACT
// pt -> s1 , EXACT
// s1.ip1 -> i21 , EXACT
// s1.ip2 -> i22 , EXACT
// s2.ip1 -> i21 , EXACT
// s2.ip2 -> i22 , EXACT
```

Prog 7.17 – pt_{caller} pour l'exemple d'affectation de structures

```
// Points To IN:
// _t1_1 -> _t1_1_1 , EXACT
// _t2_2 -> _t2_2_2 , EXACT
// _t2_2_2.ip2 -> _t2_2_2_2 , EXACT
// t1 -> _t1_1 , EXACT
// t2 -> _t2_2 , EXACT
```

Prog 7.18 – In pour l'exemple d'affectation de structures

7.5.2 Analyse intraprocédurale de la fonction `assignment`

L'initialisation des paramètres formels fournit l'information *points-to* du programme 7.18. Après l'analyse du corps de la fonction `assignment` et la suppression des variables locales et de celles qui sont non visibles depuis le site d'appel, l'information obtenue est la suivante :

```
// Points To OUT:
// _t1_1 -> _t1_1_1 , EXACT
// _t1_1_1.ip1 -> _t2_2_2_2 , EXACT
// _t2_2 -> _t2_2_2 , EXACT
// _t2_2_2.ip2 -> _t2_2_2_2 , EXACT
```

Prog 7.19 – Out pour l'exemple d'affectation de structures

7.5.3 Analyse interprocédurale

Construction de pt_{bound} Pour la construction de pt_{bound}, il faut commencer par rappeler l'ensemble pt_{caller} [3] ainsi que l'ensemble Out. Les deux ensembles sont mis sous la forme de graphes, figure 7.6 et 7.7.

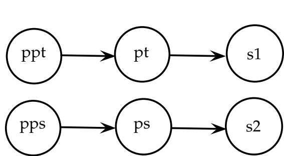


FIGURE 7.6 – L'ensemble pt_{caller} avant l'appel à `assignment` et sans les champs des structures

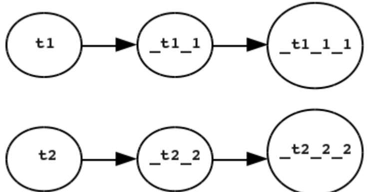


FIGURE 7.7 – L'ensemble Out à la sortie de la fonction à `assignment` et sans les champs des structures

Pour obtenir pt_{bound}, un arc est créé de `t1` vers la cible de `ppt` et un autre de `t2` vers `pps` (figure 7.8).

3. Sans les champs pointeurs contenus dans les structures `s1` et `s2` pour une meilleure lisibilité.

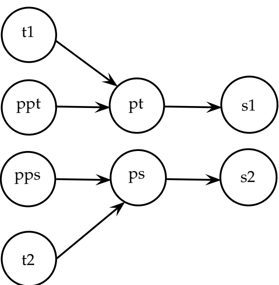


FIGURE 7.8 – Construction de l'ensemble pt_{bound} à l'entrée de la fonction `assignment`

Finalement, après l'ajout des champs pointeurs contenus dans les structures, pt_{bound} est défini comme suit :

$$\begin{aligned}\text{pt}_{\text{bound}} &= \{(t1, pt), (t2, ps), (pps, ps), (ppt, pt), (ps, s2), (pt, s1),\\ &\quad (s1.ip1, i21), (s1.ip2, i22), (s2.ip1, i21), (s2.ip2, i22)\}\end{aligned}$$

En utilisant In, nous pouvons aussi définir binding :

$$\begin{aligned}\text{binding} = \{&(_t2_2_2, s1),\\ &(_t2_2, pt),\\ &(t2, ppt),\\ &(_t1_1_1, s2),\\ &(_t1_1, ps),\\ &(_t2_2_2_2, i22),\\ &(t1, pps),\\ &(_t2_2_2.ip2, s1.ip2)\}\end{aligned}$$

Compatibilité entre pt_{bound} et In Pour les éléments de E qui sont :

$$E = \{_t1_1, _t2_2, _t1_1_1, _t2_2_2, _t2_2_2_2, _t2_2_2.ip2\}$$

les chemins concrets obtenus sont :

$$\begin{gathered} C_c(_t1_1) = \{ps\}\\ C_c(_t1_1_1) = \{s2\}\\ C_c(t_t2_2) = \{pt\}\\ C_c(_t2_2_2) = \{s1\}\\ C_c(_t2_2_2.ip2) = \{s1.ip2\}\\ C_c(_t2_2_2_2) = \{i22\}\end{gathered}$$

Comme les intersections des chemins concrets deux à deux est vide, nous concluons qu'il n'y a pas d'aliasing au niveau de pt_{bound}.

Calcul de Kill Comme les effets résumés de l'instruction d'écriture `(**t1).ip1 = (**t2).ip2;` ne peuvent donner d'information précise sur les pointeurs qui ont été écrits, à cause du double déréférencements, nous ne pouvons pas calculer l'ensemble Kill et le posons donc égal à l'ensemble vide. Nous aurons une information redondante au moment de traduire le résultat final au niveau du site d'appel mais correcte.

Calcul de Gen Pour calculer Gen, l'ensemble Out est rappelé :

$$\text{Out} = \{(_t1_1, _t1_1_1), (_t2_2, _t2_2_2), (_t2_2_2.ip2, _t2_2_2_2), (_t1_1_1.ip1, _t2_2_2_2)\}$$

L'ensemble Out est traduit en utilisant l'information contenue dans binding, ce qui permet d'obtenir l'ensemble Gen suivant :

$$\text{Gen} = \{(pt, s1), (s1.ip2, i22), (ps, s2), (s2.ip1, i22)\}$$

Calcul de pt_{end} L'information finale obtenue est la suivante :

$$\begin{aligned} \text{pt}_{\text{end}} &= \{(pps, ps), (ppt, pt), (ps, s2), (pt, s1), \\ &\quad (s1.ip1, i21), (s1.ip2, i22), (s2.ip1, i22), (s2.ip2, i22)\} \end{aligned}$$

soit, en termes de graphe, et en omettant les champs pointeurs contenus dans les structures :

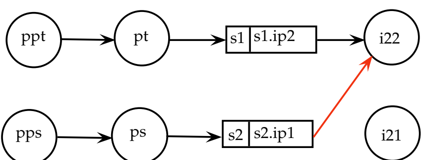


FIGURE 7.9 – Graphe des arcs *points-to* après le traitement de l'appelée

Cet exemple montre que le mécanisme de traduction permet de mettre à jour le site d'appel malgré des chemins d'accès à des champs de structure.

7.6 Exemple de Wilson

Dans son article, Wilson [Wil97] propose un exemple, programme 7.20, pour montrer l'impact de l'aliasing au niveau du site d'appel sur les résultats.

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

int main() {
  int x, y, z;
  int *x0 = &x, *y0 = &y, *z0 = &z;
  bool test1, test2;
  if(test1)
    f(&x0, &y0, &z0);
  else if(test2)
    f(&z0,&x0,&y0);
  else
    f(&x0,&y0,&x0);

  return 0;
}
```

Prog 7.20 – Exemple de Wilson

Voici les conclusions tirées par Wilson à propos de la fonction `f`, sans évaluation des conditions `test1` et `test2` :

- dans le premier cas, si `r` et `p` ne sont pas aliasés, alors `*q` pointe vers la cible de `*r` ;
- dans le deuxième cas, si `r` et `p` pointent exactement vers la même cible, alors `*q` garde son ancienne cible ;
- enfin, dans le troisième cas, si `r` et `p` peuvent pointer vers la même cible mais que ces cibles ne sont pas exactement les mêmes, alors `*q` peut garder son ancienne cible ou peut pointer vers la cible de `*r`.

Cet exemple montre que le comportement d'une fonction est dépendant des arcs d'alias entre ses paramètres.

Pour faciliter la compréhension de cet exemple, nous allons le décomposer en trois sous-exemples qui contiennent chacun un site d'appel. La décomposition donne les programmes 7.21, 7.22 et 7.23.

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

int main() {
  int x, y, z;
  int *x0 = &x, *y0 = &y, *z0 = &z;
  bool test1, test2;

  if(test1)
    f(&x0, &y0, &z0);

  return 0;
}
```

Prog 7.21 – Premier site d'appel de l'exemple Wilson

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

int main() {
  int x, y, z;
  int *x0 = &x, *y0 = &y, *z0 = &z;
  bool test1, test2;

 if(test2)
    f(&z0,&x0,&y0);

  return 0;
}
```

Prog 7.22 – Deuxième site d'appel de l'exemple Wilson

7.6.1 Analyse intraprocédurale de la fonction `f`

La fonction In calculée à l'entrée de la fonction `f` est donnée par le programme 7.24.

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

int main() {
  int x, y, z;
  int * x0=&x, *y0=&y, *z0=&z;
  bool test1, test2;

  if(test1)
    f(&x0,&y0,&x0);

  return 0;
}
```

Prog 7.23 – Troisième site d'appel de l'exemple Wilson

```
// Points To IN:
// _p_1 -> _p_1_1 , EXACT
// _q_2 -> _q_2_2 , EXACT
// _r_3 -> _r_3_3 , EXACT
// p -> _p_1 , EXACT
// q -> _q_2 , EXACT
// r -> _r_3 , EXACT
```

Prog 7.24 – In pour l'exemple de Wilson

Cette information peut être réécrite comme suit :

$$\text{In} = \{(p, _p_1), (_p_1, _p_1_1), (q, _q_2), (_q_2, _q_2_2), (r, _r_3), (_r_3, _r_3_3)\}$$

De même, la fonction Out, calculée après l'analyse de la fonction `f`, est donnée par le programme 7.25.

```
// Points To OUT:
// _p_1 -> _q_2_2 , EXACT
// _q_2 -> _r_3_3 , EXACT
// _r_3 -> _r_3_3 , EXACT
```

Prog 7.25 – Out pour l'exemple de Wilson

L'ensemble Out peut être réécrit comme suit :

$$\text{Out} = \{(_p_1, _q_2_2), (_q_2, _r_3_3), (_r_3, _r_3_3)\}$$

A partir de In et Out, nous pouvons déduire l'ensemble E qui contient six éléments :

$$E = \{_p_1, _p_1_1, _q_2, _q_2_2, _r_3, _r_3_3\}$$

7.6.2 Analyse intraprocédurale de la fonction `main`

La valeur de $\text{pt}_{\text{caller}}$, qui correspond à l'analyse intraprocédurale de la fonction `main`, est la même pour les trois sites d'appel :
ce qui donne :

$$\text{pt}_{\text{caller}} = \{(x0, x), (y0, y), (z0, z)\}$$

```
// Points To:
// x0 -> x , EXACT
// y0 -> y , EXACT
// z0 -> z , EXACT
```

Prog 7.26 – $\text{pt}_{\text{caller}}$ pour l'exemple de Wilson

soit, en termes de graphe :

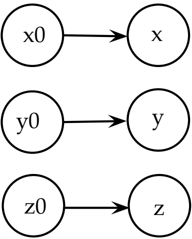


FIGURE 7.10 – Graphe des arcs *points* avant le site d'appel

7.6.3 Analyse interprocédurale pour le premier site d'appel

Construction de pt_{bound} pour le premier site d'appel Pour le premier site d'appel, la construction de pt_{bound} consiste à effectuer les affectations suivantes :

```
p=&x0;
q=&y0;
r=&z0;
```

Prog 7.27 – Les opérations pour le premier site d'appel

A partir de ces affectations, nous obtenons l'ensemble pt_{bound} suivant :

$$\text{pt}_{\text{bound}} = \{(x0, x), (y0, y), (z0, z), (p, x0), (q, y0), (r, z0)\}$$

La fonction binding est plus facile à construire en suivant les arcs *points-to* de la figure 7.11. En effet, les associations entre paramètres formels et effectifs aboutissent à l'information binding suivante :

$$\text{binding} = \{(_p_1, x0), (_p_1_1, x), (_q_2, y0), (_q_2_2, y), (_r_3, z0), (_r_3_3, z)\}$$

Compatibilité entre pt_{bound} et In pour le premier site d'appel Comme le premier site d'appel ne présente aucun aliasing, il y a compatibilité entre pt_{bound} et In. Il n'y a qu'un chemin d'appel concret pour chaque élément de E :

$$\begin{aligned} C_c(_p_1) &= \{x0\} \\ C_c(_p_1_1) &= \{x\} \\ C_c(_q_2) &= \{y0\} \\ C_c(_q_2_2) &= \{y\} \\ C_c(_r_3) &= \{z0\} \\ C_c(_r_3_3) &= \{z\} \end{aligned}$$

Il n'y a pas d'intersection entre les résultats des chemins concrets des différents éléments de E, d'où la conclusion : l'absence d'aliasing entre les paramètres.

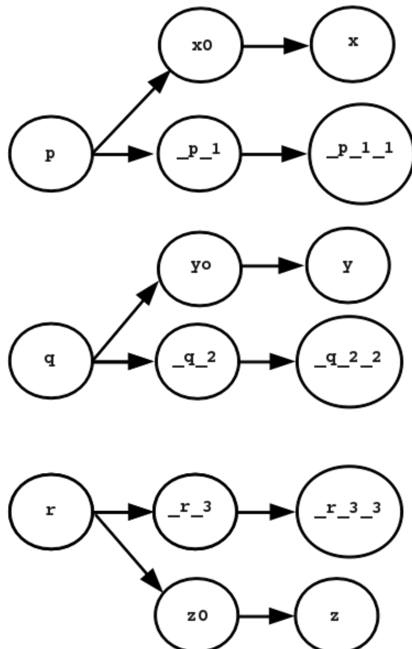


FIGURE 7.11 – Graphe des arcs *points* pour la construction de binding pour le premier site

Calcul de Kill pour le premier site d'appel Pour calculer l'ensemble Kill, il faut reprendre $\text{pt}_{\text{caller}}$ et lui appliquer l'équation 4.11 :

$$\text{pt}_{\text{caller}} = \{(x0, x), (y0, y), (z0, z)\}$$

Les effets résumés obtenus pour `f` sont :

```
//          <   is read  >: _q_2 _r_3 p q r
//          <   is written>: _p_1 _q_2
```

Prog 7.28 – Les effets résumés pour la fonction `f`

A partir d'eux, est obtenu l'ensemble suivant :

$$\text{Written} = \{_p_1, _q_2\}$$

La traduction de Written en utilisant l'information binding est effectuée par la fonction Translation et fournit le résultat suivant :

$$\text{Translation}(\text{Written}, \text{binding}) = \{x0, y0\}$$

Enfin, à partir de cette traduction, l'ensemble Kill est obtenu :

$$\text{Kill} = \{(x0, x), (y0, y)\}$$

Calcul de Gen Pour calculer Gen, il faut repartir de Out :

$$\text{Out} = \{(_p_1, _q_2_2), (_q_2, _r_3_3), (_r_3, _r_3_3)\}$$

et lui appliquer l'équation 6.5 pour obtenir :

$$\text{Gen} = \{(x0, y), (y0, z), (z0, z)\}$$

Calcul de pt_{end} pour le premier site d'appel L'information finale obtenue est :

$$\text{pt}_{\text{end}} = \{(x0, y), (y0, z), (z0, z)\}$$

Compte tenu du fait que l'appel se fait dans le cadre d'un test, les arcs *points-to* pt_{end} sont fusionnés avec les arcs qui existaient avant l'appel. Ceci abouti au résultat suivant :

```
// Points To:
// x0 -> x , MAY
// x0 -> y , MAY
// y0 -> y , MAY
// y0 -> z , MAY
// z0 -> z , EXACT
```

Prog 7.29 – les arcs *points-to* pour le premier site d'appel

Soit, en termes de graphe :

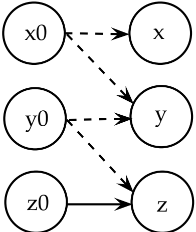


FIGURE 7.12 – Graphe des arcs *points-to* pour le premier site d'appel

7.6.4 Analyse interprocédurale pour le deuxième site d'appel

Construction de pt_{bound} pour le deuxième site d'appel Le deuxième site d'appel correspond aux affectations suivantes :

```
p=&z0;
q=&y0;
r=&x0;
```

Prog 7.30 – Les opérations pour le deuxième site d'appel

qui conduisent à :

$$\text{pt}_{\text{bound}} = \{(x0, x), (y0, y), (z0, z), (p, z0), (q, y0), (r, x0)\}$$

La fonction binding est plus facile à construire en suivant les arcs *points-to* de la figure 7.13.

$$\text{binding} = \{(_p_1, z0), (_p_1_1, z), (_q_2, y0), (_q_2_2, y), (_r_3, x0), (_r_3_3, x)\}$$

Compatibilité entre pt_{bound} et In pour le deuxième site d'appel Comme le deuxième site d'appel ne présente aucun aliasing, il y a compatibilité entre pt_{bound} et In. Il n'y a qu'un chemin d'appel concret pour chaque élément de E :

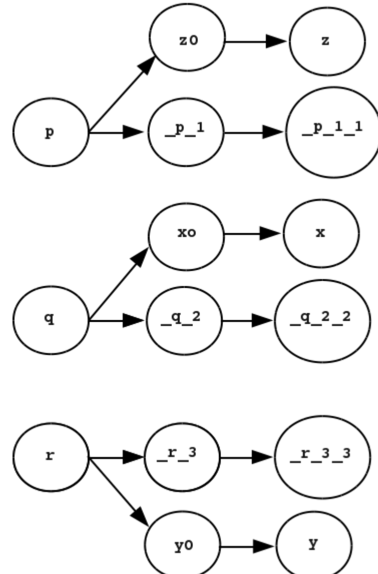


FIGURE 7.13 – Graphe des arcs *points* pour la construction de binding pour le deuxième site

$$
\begin{aligned}
C_c(_p_1) &= \{z0\} \\
C_c(_p_1_1) &= \{z\} \\
C_c(_q_2) &= \{x0\} \\
C_c(_q_2_2) &= \{x\} \\
C_c(_r_3) &= \{y0\} \\
C_c(_r_3_3) &= \{y\}
\end{aligned}
$$

Il n'y a pas d'intersection entre les résultats des chemins concrets des différents éléments de E. En conclusion, il n'y a pas d'aliasing entre les paramètres.

Calcul de Kill pour le deuxième site d'appel Pour calculer l'ensemble Kill, il faut reprendre $\text{pt}_{\text{caller}}$ et lui appliquer l'équation 4.11 :

$$\text{pt}_{\text{caller}} = \{(x0, x), (y0, y), (z0, z)\}$$

Les effets résumés obtenus pour `f` sont toujours :

```
//           <   is read  >: _q_2 _r_3 p q r
//           <   is written>: _p_1 _q_2
```

Prog 7.31 – Les effets résumés pour la fonction `f`

ce qui conduit à l'ensemble :

$$\text{Written} = \{_p_1, _q_2\}$$

et à Translation :

$$\text{Translation}(\text{Written}, \text{binding}) = \{z0, x0\}$$

pour obtenir finalement :

$$\text{Kill} = \{(z0, z), (x0, x)\}$$

Calcul de Gen Pour calculer Gen, l'ensemble Out est repris :

$$\text{Out} = \{(_p_1, _q_2_2), (_q_2, _r_3_3), (_r_3, _r_3_3)\}$$

L'équation 6.5 est utilisée pour obtenir :

$$\text{Gen} = \{(x0, y), (y0, y), (z0, x)\}$$

Calcul de pt_{end} pour le deuxième site d'appel On obtient finalement l'information attendue :

$$\text{pt}_{\text{end}} = \{(x0, z), (y0, y), (z0, x)\}$$

Compte tenu du fait que l'appel se fait dans le cadre d'un test, les arcs *points-to* pt_{end} sont fusionnés avec les arcss qui existaient avant l'appel. Ceci aboutit au résultat suivant :

```
// Points To:
// x0 -> x , MAY
// x0 -> y , MAY
// y0 -> y , EXACT
// z0 -> x , MAY
// z0 -> z , MAY
```

Prog 7.32 – Les arcs *points-to* pour le deuxième site d'appel

Soit en terme de graphe :

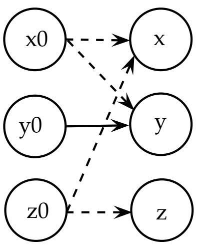


FIGURE 7.14 – Graphe des arcs *points-to* pour le deuxième site d'appel

7.6.5 Analyse interprocédurale pour le troisième site d'appel

Construction de pt_{bound} Le troisième site d'appel consiste à effectuer les affectations suivantes : ce qui conduit à :

```
p=&x0;
q=&y0;
r=&x0;
```

Prog 7.33 – Les opérations pour le troisième site d'appel

$$\text{pt}_{\text{bound}} = \{(x0, x), (y0, y), (z0, z), (p, x0), (q, y0), (r, x0)\}$$

La fonction binding est plus facile à construire en suivant les arcs *points-to* de la figure 7.15.

$$\text{binding} = \{(_p_1, x0), (_p_1_1, x), (_q_2, y0), (_q_2_2, y), (_r_3, x0), (_r_3_3, x)\}$$

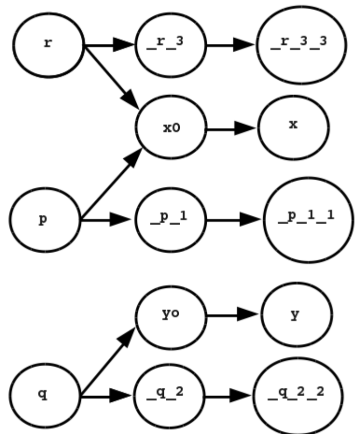


FIGURE 7.15 – Graphe des arcs *points* pour la construction de binding pour le troisième site

Compatibilité entre pt_{bound} et In pour le troisième site d'appel Les chemins concrets pour chaque élément de E sont les suivants :

$$\begin{array}{rcl} C_c(_p_1) & = & \{x0\} \\ C_c(_p_1_1) & = & \{x\} \\ C_c(_q_2) & = & \{y0\} \\ C_c(_q_2_2) & = & \{y\} \\ C_c(_r_3) & = & \{x0\} \\ C_c(_r_3_3) & = & \{x\} \end{array}$$

Il y a donc une intersection non vide avec les couples $(_p_1, _r_3)$ et $(_p_1_1, _r_3_3)$ et par conséquent présence d'aliasing entre les paramètres effectifs. Comme notre analyse se fait sous l'hypothèse qu'il n'existe pas d'aliasing entre paramètres, nous choisissons de ne pas continuer l'analyse interprocédurale complète mais plutôt l'analyse rapide, qui permet juste de supprimer les arcs *points-to* dont les pointeurs ont été écrits au niveau de l'appelé (revoir les différentes approches de la section 6.1).
En faisant appel à l'analyse `FAST_INTERPROCEDURAL_POINTS_TO_ANALYSIS`, nous obtenons au niveau du troisième site d'appel les arcs *points-to* suivants :

```
// Points To:
// x0 -> undefined , MAY
// x0 -> x , MAY
// y0 -> undefined , MAY
// y0 -> y , MAY
// z0 -> z , EXACT
```

Prog 7.34 – Les arcs *points-to to* pour le troisième site d'appel

Soit, en termes de graphe la figure 7.16.
Ce résultat est dû au fait que l'analyse crée un nouvel ensemble Gen avec les pointeurs écrits pointant vers la zone `undefined`.

7.6.6 Conclusion sur l'exemple de Wilson

L'exemple de Wilson a été découpé en trois cas :

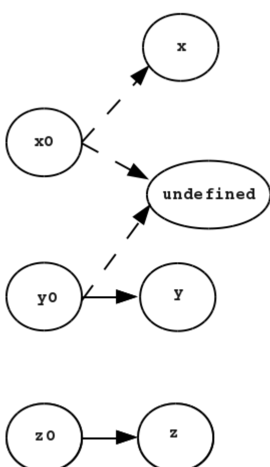


FIGURE 7.16 – Graphe des arcs *points* pour le troisième site d'appel

- en ce qui concerne le premier cas, celui du premier site d'appel, nous nous rendons compte que l'explication de Wilson est incomplète puisqu'elle ne spécifie pas ce vers quoi pointe `p` , les mêmes conclusions s'appliquent au deuxième cas ; de plus, une fois de retour dans l'espace de l'appelant, il faudrait exprimer les effets en terme de `x0`, `y0` et `z0` plutôt que des cibles de `p`, `q` et `r`.
- en ce qui concerne le troisième cas, qui correspond au troisième site d'appel, il est dit que la cible de `q`, `y0`, conserve sa cible initiale, `y`. On constate que ce résultat est obtenu de manière très imprécise quand il y a absence d'hypothèse sur la concrétisation des éléments de `E`, mais qu'un résultat faux est obtenu quand cette hypothèse est faite ; ce n'est pas un problème pour l'algorithme proposé, puisqu'il suppose que les paramètres réels ne sont pas aliasés, condition qui n'est pas vérifiée pour ce site d'appel ; quand cette condition n'est pas vérifiée, nous appliquons l'analyse `FAST_INTERPROCEDURAL_POINTS_TO_ANALYSIS` qui permet de continuer l'analyse interprocédurale de façon conservatrice.

7.7 Conclusion

Nous avons présenté dans ce chapitre expérimental des exemples illustrant les différentes difficultés liées à l'analyse interprocédurale. Parmi elles se pose la difficulté de traduction de chemins d'accès complexes comme les champs de structure, la traduction des nouveaux emplacements alloués au niveau de l'appelée ainsi que celle des arguments de type tableau dynamique. Le dernier exemple analysé est celui de Wilson qui montre bien le problème lié à la condition d'aliasing en entrée de la fonction. Il permet aussi de comparer nos résultats à ceux de Wilson.

CHAPITRE 8

Contributions et perspectives

L'analyse des pointeurs est encore un domaine de recherche très actif à cause des applications qui ont besoin de plus en plus de manipuler des objets de tailles inconnues à la compilation. De nombreux travaux, avant le notre, ont été menés et plusieurs analyses efficaces ont vu le jour. Notre analyse se différencie des autres analyses par la combinaison d'une analyse intraprocédurale précise et d'une analyse interprocédurale modulaire ascendante. Les contributions originales de notre travail sont les suivantes :

- une représentation unifiée des accès à la mémoire, compatible avec les autres analyses (use-def chains et test de dépendance) ;
- la modélisation des champs de structures de données (appelé dans la littérature *field-sensitive*) ;
- l'utilisation des chemins d'accès constants qui ne nécessitent pas de passage en code 3 adresses ;
- le respect intégral du code source ;
- la sensibilité au flot de contrôle pour le calcul intraprocédural des arcs *points-to* ;
- une analyse intraprocédurale précise grâce au calcul à la demande du contexte formel d'appel ;
- une étude expérimentale de la notion de transformeur de *points-to* ;
- l'analyse modulaire du code via l'utilisation des résumés pour le calcul interprocédural ascendant des arcs *points-to* ;
- une modélisation du tas ;
- la gestion des erreurs détectées par l'analyse.

Nous récapitulons ci-dessous les nouveautés apportées ainsi que les perspectives. L'impact de notre analyse sur plusieurs phases du compilateur PIPS est montré sur de nombreux exemples au chapitre 2 ainsi qu'au chapitre expériences 7.

8.1 Contexte de la thèse

Cette thèse s'inscrit dans le cadre de l'optimisation des applications de modélisation scientifiques et de traitement de signal. Elle est intégrée dans le compilateur PIPS [AAC^{+}] développé au sein du centre de recherche en informatique de l'école MINES ParisTech.

8.1.1 Profil des applications scientifiques

La première étape de cette thèse a consisté à étudier les besoins des applications scientifiques qui sont analysées par PIPS.

On retrouve essentiellement des applications où domine les opérations sur les tableaux comme la convolution ou la transposée. Ces applications sont écrites en `fortran` et en `C`. Lorsqu'elles sont écrites en `C` les tableaux qui y sont traités sont des tableaux dynamiques. Ces tableaux sont utilisés via des pointeurs, leurs dimensions sont généralement des champs de structures (le type `struct`), ce qui augmente la difficulté de l'analyse du code. Avant d'orienter notre choix sur l'analyse de pointeurs à concevoir, il était important de déterminer quelles applications nous allions cibler pour répondre aux mieux à leur besoins. Rétrospectivement le choix d'une analyse

de pointeurs est une question empirique qui se base sur les analyses clientes ainsi que le profil du code à analyser.

8.1.2 Besoins des analyses clientes

La deuxième étape a permis de cibler les analyses clientes qui pourraient tirer profit de l'information *points-to*. Une information imprécise quant aux les cases mémoire lues ou écrites par une instruction inhibe plusieurs analyses et optimisations de code. Tous les outils d'optimisation, comme gcc, llvm ou encore rose, contiennent aujourd'hui une analyse qui fournit une information sur l'état de la mémoire. Mais après l'intégration de l'analyse, il faut pouvoir déterminer son impact sur l'outil. Plusieurs expériences ont été mené afin de prouver l'importance de cet impact [Rak01] [WHI95] [GH98] [CH00].

Dans ce manuscrit, nous avons consacré le chapitre 2 à étudier cette question sur les analyses de PIPS. La première analyse qui utilise cette information est l'analyse des effets dans PIPS [Cre96], où une passe a été développée par Beátrice Creusillet et François Irigoin afin que ces derniers soient calculés en exploitant l'information *points-to*. Une fois cette passe appliquée, toutes les autres analyses comme le calcul des « use-def », le calcul du graphe de dépendances, la suppression du code inutile peuvent profiter directement ou indirectement de l'information *points-to via* les effets mémoires.

8.1.3 Intégration des résultats dans le compilateur PIPS

Notre analyse a été intégrée au compilateur PIPS, dans une phase qui intervient avant le calcul des effets. Les effets contiennent les variables écrites ou lues par une instruction. Lors d'un déréférencement de pointeur, par exemple `*p`, si l'information `p` pointe vers la variable `i` n'est pas fournie, l'analyse des effets renvoie un résultat sur-approximé, l'emplacement abstrait `*ANYWHERE*` qui est lu ou écrit. Ce résultat signifie que n'importe quelle case mémoire du programme peut être lue ou écrite en déréférençant le pointeur `p`. Ce qui est certes vrai mais qui par contre inhibe toute optimisation de l'environnement où le déréférencement apparaît. Le calcul des effets dans PIPS peut se faire désormais en évaluant chaque déréférencement avec l'information *points-to* pré-calculés. Ce n'est que lorsque l'information n'est plus calculable que l'analyse *points-to* opte pour la solution `*ANYWHERE*`.

8.2 Contributions

Nous détaillons ci-dessous les principales contributions de notre travail de thèse.

8.2.1 Étude des besoins

Avant de commencer à concevoir une analyse de pointeurs il fallait étudier les besoins des analyses clientes qui allaient en tirer profit. Il fallait aussi déterminer le profil des applications scientifiques, qui sont essentiellement des applications de traitement de signal,ont nous voulons améliorer les performances [Ami08]. L'étude des besoins des analyses clientes est le sujet du chapitre 2.

8.2.2 État de l'art

Après l'analyse des besoins, nous avons effectué une étude bibliographique des analyses de pointeurs, d'alias ou encore de *shape analysis*. Cette question étant très importante pour les compilateurs `C`, elle a été longtemps étudiée et a été le sujet de nombreux travaux. Lors de notre

étude de l'état de l'art, nous nous sommes intéressé aux premiers algorithmes étant donné qu'ils restent d'actualité et qu'ils ont été repris et améliorés dans plusieurs outils. La première analyse est celle d'Emami [Ema93] qui domine la catégorie de analyses sensibles au flot de contrôle et au contexte. La deuxième est celle d'Andersen [And94] qui domine la catégorie des analyses insensibles au flot de contrôle et au contexte. Notre étude bibliographique nous a permis de comprendre l'importance d'une analyse précise pour l'intraprocédural, ainsi que les exigences en terme de performances (temps d'exécution ou encore l'espace mémoire utilisé) des outils. Ceci nous a amené à combiner une analyse intraprocédurale sensible au flot de contrôle et une analyse interprocédurale ascendante, insensible au contexte.

8.2.3 Conception de l'analyse intraprocédurale

Notre analyse intraprocédurale est une généralisation de l'analyse d'Emami [EGH94] [Ema93] ; une mise à jour des arcs *points-to* est effectuée à chaque point du programme. C'est-à-dire que l'analyse suit le flot de contrôle et prend en considération les instructions de type test et boucles. L'analyse vise en premier lieu à lever l'ambiguïté sur les accès tableau de type `p[i]` où `p` est un pointeur. Cette notation est équivalente au déréférencement suivant (`*(p+i)`). D'où l'importance de connaître les cibles du pointeur dans le but final de pouvoir paralléliser les accès aux tableaux.

Notre analyse traite toutes les instructions du langage `C`, même celles qui contiennent des expressions complexes. Elle prend en compte tous les opérateurs comme la virgule, le conditionnel ainsi que l'arithmétique sur pointeurs. Concernant ce dernier point, nous nous sommes tournés vers la norme pour développer une solution qui la respecte. Le traitement des autres instructions du langage a été détaillé dans le chapitre 4.

8.2.4 Conception de l'analyse interprocédurale

Après comparaison des analyses interprocédurales existantes, nous avons conclu que pour satisfaire les besoins des applications que nous traitions, il fallait concevoir une analyse performante qui ne dégrade pas le temps d'exécution. L'analyse interprocédurale que nous avons conçu permet d'analyser un programme en entier et de propager l'information *points-to* à travers toutes les fonctions de manière ascendante, en l'absence de récursion et d'aliasing. L'analyse peut être sensible au contexte, c'est à dire lorsqu'un appel de fonction est rencontré cette dernière est analysée, à nouveau, et les résultats sont traduits au niveau du site d'appel. Ceci signifie que pour chaque appel, il peut y avoir un résultat différent. Comme elle peut être insensible au contexte c'est-à-dire que l'analyse ne prend pas en compte le contexte d'appel et applique le même résumé à tous les sites.

Ceci dit l'analyse peut être insensible au contexte sans perdre d'information sur les pointeurs ; pour cela nous avons choisi d'utiliser des transformeurs de fonctions. Après l'analyse intraprocédurale du corps de la fonction, nous créons un résumé qui contient les arcs *points-to* pertinents. Par pertinent, nous voulons dire les arcs qui peuvent être visibles et avoir un effet sur le site d'appel. Et au moment de l'appel, ces résumés sont traduits au niveau du site d'appel, après la vérification de la condition de non aliasing entre paramètres effectifs (voir le chapitre 6).

8.3 Nouveaux concepts

Nous présentons maintenant les originalités de notre travail ainsi que nos apports par rapport aux travaux déjà réalisés dans le domaine de l'analyse de pointeurs.

8.3.1 Formulation de l'analyse de pointeurs

L'analyse de pointeurs a fait l'objet de plusieurs publications. La plus part des travaux appliquaient la correspondance de modèles (*pattern matching*) qui consiste à appliquer une règle spécifique à chaque instruction du langage. Cette méthode est très laborieuse ; de plus, vu la complexité des expressions en `C`, les analyses sont obligées d'ignorer certaines instructions lorsqu'elles n'arrivent pas à déterminer le modèle qu'il faut appliquer. Notre approche est plus généraliste ; elle itère sur toutes les sous expressions composant l'expression complexe et calcule à chaque fois les arcs *points-to* associées à la sous-expression, tout en respectant l'ordre de l'analyse imposé par les points de séquence.

Nous avons formulé notre analyse sous la forme d'équations sémantiques qui associent à chaque *statement* un ensemble d'arcs *points-to*. Nous sommes partis d'un sous-langage de `C` pour déterminer au début les équations pour les instructions les plus importantes. Ces dernières sont l'assignation, qui est la principale instruction génératrice d'arcs *points-to*, ainsi que la structure de contrôle test (`if ... else`). Et finalement la boucle `while` qui nécessite le calcul de points fixes pour converger en cas de présence de structures de données récursives 5.

8.3.2 Le treillis des emplacements mémoire

La modélisation des emplacements mémoire sous la forme d'un treillis typé à plusieurs niveaux est une originalité de notre analyse. Nous avons pu établir une relation d'ordre entre les emplacements et abstraire des zones mémoire. Ces zones incluent la pile, le tas, la zone allouée aux paramètres formels ou encore les variables statiques du programme. Le sommet du treillis, noté `*ANYWHERE*`, correspond à une cible de pointeur bien définie, mais qui peut se trouver n'importe où dans la mémoire. Pour le test de dépendances, cela signifie que le pointeur peut être aliasé avec tous les autres pointeurs du programme qui ont le même type.

Le treillis des emplacements contient implicitement plusieurs dimensions. Parmi ces dimensions figurent le nom de la variable, le nom du module dans lequel la variable est définie, la liste des indices et finalement une dimension très importante qui est le type. L'utilisation du treillis nous permet de créer des arcs *points-to* correctement typés où la cible doit avoir le même type de base que le pointeur (par exemple un pointeur vers un entier doit pointer vers un entier). La comparaison de toutes les dimensions nous permet de déterminer si un des emplacements contient l'autre. Ou si deux pointeurs ont les mêmes cibles ce qui impliquent qu'ils sont en alias, une information très utile à l'analyse de dépendances.

8.3.3 Implémentation dans un compilateur source-à-source

Tandis qu'un compilateur traditionnel prend en entrée un programme écrit dans un langage haut niveau pour générer en sortie un programme écrit en langage machine prêt à être exécuté, un compilateur source-à-source prend en entrée un programme écrit en un langage haut niveau pour générer un programme écrit dans le même langage ou un autre langage du même niveau d'abstraction [Def]. L'avantage d'un tel compilateur est de pouvoir traduire du code, mais aussi de pouvoir fournir en sortie le même codé annoté de propriétés ou optimisé. Le compilateur paralléliseur PIPS fait partie de cette catégorie de compilateurs, aussi appelé compilateur de haut niveau. L'analyse de pointeurs est effectué sur un code traduit dans la représentation interne de PIPS qui est très proche du langage `C` ce qui facilite encore plus le traitement des différentes instructions du programme.

8.3.4 Utilisation des chemins d'accès constants

La plus part des analyses utilisaient la décomposition en instructions plus simples pour pouvoir évaluer un expression d'adresse, par exemple :

```
**p = 1;
```

Prog 8.1 – Double déréférencement

```
t = *p;
*t = 1;
```

Prog 8.2 – Introduction d'une variable temporaire

Ceci nécessitait une passe en plus sur le programme pour transformer tous les multiples pointeurs.

Mais, comme en langage C les deux notations `**p` et `p[0][0]` sont synonymes, le fait que `p` soit un pointeur est contenue dans son type et non dans la syntaxe. Pour ne pas développer une passe de transformation de code comme les autres analyses, nous avons choisi de traduire les déréférencements en accès constants à des éléments de tableau. Une expression comme `**p` est traduite en chemin d'accès non constant, ne faisant intervenir que des indices, puis évaluée de gauche à droite, en remplaçant chaque dimension de « tableau » par sa cible quand elle correspond à un déréférencement. Cette traduction facilite l'évaluation des cases mémoire et surtout économise une passe de transformation du code. Cette représentation nous permet aussi de distinguer les champs pointeurs des structures de données. Reprenons l'exemple tiré de l'article de Ghiya [Rak01] :

```c
struct foo {int *p; int *q;} s1, s2;
int x,y;
s1.p = &x;
s2.q = &y;
```

Prog 8.3 – Modélisation des champs pointeurs

Nous faisons la distinction entre les champs `p` et `q` ainsi que entre le même champ de deux instances distinctes. Ainsi le champ `s1.p`, traduit comme chemin constant en `s1[p]`, est distinct du champ `s2.p`, traduit comme chemin constant en `s2[p]`. Cette distinction est un grand avantage pour notre analyse qui, contrairement à ce qui est fait traditionnellement, ne fusionne pas structure et champs en un seul objet. Cette dimension de l'analyse est appelée « field-sensitivity ». Elle augmente considérablement la précision de l'analyse et faisait partie des exigences initiales, puisqu'elle est nécessaire pour traiter et paralléliser les anciens codes C qui représentent les nombres complexes par des structures.

8.3.5 Analyse modulaire

PIPS adopte une approche modulaire pour l'analyse des programmes. L'analyse de pointeurs profite de cette approche où les sous-programmes sont analysés indépendamment et à la demande. Donc seulement une partie du programme réside dans la mémoire. Cet avantage permet à l'utilisateur de sélectionner les modules qu'il désire analyser et de ne pas analyser le programme en entier, surtout si seule une portion du code est candidate à l'optimisation.

Pendant l'analyse intraprocédurale, chaque fonction est analysée comme une unité indépendante, avec l'initialisation de ses paramètres et des emplacements qu'ils peuvent pointer, à des emplacements abstraits créés par l'analyse, à la demande. À la fin de l'analyse, un résumé décrivant l'effet de la fonction est créé. Il comporte :

1. les arcs *points-to* en entrée : ce sont les paramètres formels et les stubs ;
2. les arcs *points-to* en sortie : ce sont la valeur de retour, si c'est un pointeur, les multiples pointeurs, les pointeurs vers le tas ou encore les variables globales.

Lorsque l'analyse interprocédurale est activée, au moment où un site d'appel est rencontré, le résumé de la fonction en question est chargé dans la mémoire. Ce résumé est traduit et les arcs *points-to* sont mis à jour dans l'appelant.

8.3.6 Analyse paramétrable

Un grand avantage offert par notre analyse est constitué par les propriétés. Ces dernières permettent d'activer ou désactiver certaines fonctionnalités. Parmi ces propriétés citons :

- `ABSTRACT_HEAP_LOCATIONS`, la modélisation du tas : le tas peut être assimilé à un unique emplacement mémoire abstrait pour tout le programme, ou à un ensemble d'emplacements, un par site d'appel ;
- `ALIASING_ACROSS_FORMAL_PARAMETERS`, l'*aliasing* entre paramètres formels : l'analyse interprocédurale peut se faire sous l'hypothèse qu'il n'y a pas d'alias entre les paramètres .
- `ALIASING_ACROSS_TYPES`, la prise en compte du type des variables pour décider de l'*aliasing*.

Ces propriétés contrôlent le comportement de l'analyse. Si l'utilisateur sait que l'application ne contient que très peu d'allocation dynamique et que la plus part des arcs *points-to* sont au niveau de la pile, il peut choisir un niveau de précision faible pour la modélisation du tas. La description détaillée des autres propriétés se trouve dans la section 5.7.

Une autre liberté offerte à l'utilisateur est le choix de l'analyse interprocédurale. En effet comme cette phase est la plus délicate et qu'elle peut affecter l'efficacité d'une analyse de pointeurs, nous avons choisi de laisser le choix à l'utilisateur d'activer une des trois analyses interprocédurales suivantes :

1. une analyse de base qui fait pointer les pointeurs, qui ont pu être modifiés ou libérés par l'appelée, vers les emplacements abstraits `*ANYWHERE*` et `undefined` ;
2. une analyse, encore rapide mais plus précise que la précédente, qui récupère la liste des pointeurs qui ont été effectivement modifiés par l'appelée et qui supprime les arcs *points-to* où ces pointeurs apparaissent comme source ou les transforme en réduisant leur précision ;
3. une analyse précise qui utilise un transformeur pour traduire au niveau du site d'appel les effets de la fonction appelée.

Selon le besoin de l'utilisateur et le profil du code analysé, l'analyse appropriée est choisie.

Notre analyse utilise aussi l'initialisation à la demande des paramètres formels et des variables qu'ils peuvent pointer pour constituer un contexte formel minimal. Ceci permet de ne pas propager une grande liste d'arcs *points-to* dans la procédure appelée et de ne pas consommer un grand espace mémoire, ce qui augmente l'efficacité de l'analyse en temps et en espace.

8.3.7 Détection des erreurs

Un effet de bord de notre analyse est la détection des erreurs impliquant des pointeurs. En effet, en calculant les arcs *points-to*, nous pouvons nous rendre compte qu'un pointeur est utilisé alors qu'il n'a pas été initialisé, connu dans la littérature sous le nom de « wild pointer ». D'autres erreurs sont aussi détectées :

- les pointeurs pendants (« danglig pointer ») : qui pointent vers des objets non consistants, par exemple une zone libérée ou une partie inactive de la pile ;
- les fuites mémoire (« memory leak ») : c'est la perte de référence vers des zones mémoire ; ces zones deviennent inaccessibles pour l'utilisation ainsi que pour la libération ; la taille de l'espace mémoire peut augmenter considérablement et aboutir à l'arrêt brutal de l'application.

8.4 Perspectives

Dans cette dernière section nous introduisons les perspectives de notre travail de recherche et les axes qui peuvent être améliorés.

8.4.1 Modélisation du tas

Notre représentation actuelle du tas ne permet pas de distinguer entre les différentes instances d'une allocation dynamique dans une boucle :

```
int *p[10], i;
for(i = 0; i<10; i++)
 p[i] = malloc(4*sizeof(int));
```

Prog 8.4 – Appel à `malloc` dans une boucle

Pour le programme ci-dessus, notre analyse conclura que `p` pointe vers le tas alloué à la ligne 3. Il faut rajouter une autre information où il serait possible d'ajouter une indication sur l'itération en cours de la boucle. Ainsi nous aurons `p` qui pointe vers différents emplacements alloués au niveau du tas.

8.4.2 Les instructions du langage `C`

Notre analyse traite toutes les instructions du langage. Ceci dit certaines instructions présentent plus de difficulté que d'autres et leur effet sur l'analyse de pointeurs est important. L'implémentation actuelle offre des résultats très satisfaisants, mais une amélioration peut encore être apportée. Parmi les constructions concernées figurent :

- le cast ;
- les unions ;
- l'opérateur XOR ;
- le traitement de certains intrinsèques qui ne sont pas encore pris en compte.

Nous ciblons essentiellement les intrinsèques qui manipulent la mémoire comme par exemple `memcpy`.

8.4.3 Conception d'une analyse des structures de données récursives

Appelée aussi *shape analysis*, cette analyse permettrait de calculer les arcs *points-to* pour les types comme `struct` ou `union`. C'est une analyse coûteuse mais ses résultats pourraient améliorer considérablement la précision de notre analyse de pointeurs. Le développement d'une telle analyse dépasse le cadre de ce travail.

Bibliographie

[AAC+] Mehdi AMINI, Corinne ANCOURT, Fabien COELHO, Béatrice CREUSILLET, Serge GUELTON, François IRIGOIN, Pierre JOUVELOT, Ronan KERYELL, and Pierre VILLALON. Pips is not (just) polyhedral software. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT 2011). Chamonix, France, avril 2011*.

[AALwT93] Saman P. Amarasinghe, Jennifer M. Anderson, Monica S. Lam, and Chau wen Tseng. An overview of the suif compiler for scalable parallel machines. In *In Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 662–667, 1993.

[ACSGK11] Corinne Ancourt, Frédérique Chaussumier-Silber, Serge Guelton, and Ronan Keryell. PIPS : An interprodedural, extensible, source-to-source compiler infrastructure for code transformations and instrumentations. tutorial at International Symposium on Code Generation and Optimization, April 2011. Chamonix, France.

[ADLL05] Dzintars Avots, Michael Dalton, V. Benjamin Livshits, and Monica S. Lam. Improving software security with a c pointer analysis. In *Proceedings of the 27th international conference on Software engineering*, pages 332–341, New York, NY, USA, 2005. ACM.

[Ami08] Amira.Mensi. Pointeurs et aliasing en c. Master's thesis, Mines ParisTech,Centre de Recherche en Informatique (CRI), 2008. http ://cri.ensmp.fr/classement/doc/E-302.pdf.

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[BCCH95] Michael G. Burke, Paul R. Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *LCPC '94 : Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, pages 234–250, London, UK, 1995. Springer-Verlag.

[BEF+94] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, William Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris : Improving the effectiveness of parallelizing compilers. In *PROCEEDINGS OF THE SEVENTH WORKSHOP ON LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING*, pages 141–154. Springer-Verlag, 1994.

[BHR08] Uday Bondhugula, Albert Hartono, and J. Ramanujam. A practical automatic polyhedral parallelizer and locality optimizer. In *In PLDI '08 : Proceedings of the ACM SIGPLAN 2008 conference on Programming language design and implementation*, 2008.

[CH00] Ben-Chung Cheng and Wen-Mei W. Hwu. Modular interprocedural pointer analysis using access paths : design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 57–69, New York, NY, USA, 2000. ACM.

[Cou00] P. Cousot. Interprétation abstraite. *Technique et science informatique*, 19(1-2-3) :155–164, January 2000.

[Cre96] Béatrice Creusillet. *Array Region Analyses and Applications*. PhD thesis, Ecole des Mines de Paris, centre de recherche en informatique, 1996.

[Def] Définition d'un compilateur source à source. http ://en.wikipedia.org/wiki/Source-to-source_compiler.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers : beyond k-limiting. In *PLDI '94 : Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 230–241, New York, NY, USA, 1994. ACM.

[DM98] Leonardo Dagum and Ramesh Menon. Openmp : An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, pages 46–55, 1998.

[DMM98] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 106–117, New York, NY, USA, 1998. ACM.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94 : Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 242–256, New York, NY, USA, 1994. ACM.

[Ema93] Maryam Emami. A pratical interprocedural alias analysis for an optimizing/parallelizing c compiler. Master's thesis, School of Computer Science, McGill University, Montreal, 1993.

[GH96] Rakesh Ghiya and Laurie J. Hendren. Connection analysis : A practical interprocedural heap analysis for c. In *LCPC '95 : Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, pages 515–533, London, UK, 1996. Springer-Verlag.

[GH98] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 121–133, New York, NY, USA, 1998. ACM.

[Ghi96] Rakesh Ghiya. *Pratical Techniques for Interprocedural Heap Analysis*. PhD thesis, Shool of Computer Science McGill University, Montreal, 1996.

[gnu91] Gnu scientific library, 1991. http ://en.wikipedia.org/wiki/GNU_Multi-Precision_Library.

[gnu96] Gnu scientific library, 1996. http ://en.wikipedia.org/wiki/GNU_Scientific_Library.

[HBCC99] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. volume 21, pages 848–894, New York, NY, USA, 1999. ACM.

[HDE+92] Laurie J. Hendren, Chris Donawa, Maryam Emami, Guang R. Gao, Justiani, and Bhama Sridharan. Designing the mccat compiler based on a family of structured intermediate representations. In *In Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing, number 757 in LNCS*, pages 406–420. Springer-Verlag, 1992.

[Hei99] Nevin Heintze. Analysis of large code bases :the compile-link-analyse model, 1999.

[Hei01] Nevin Heintze. Ultra-fast aliasing analysis using cla :a million lines of c code in a second. pages 254–263. ACM, 2001.

[HHN92] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. Abstract description of pointer data structures : an approach for improving the analysis and optimization of imperative programs. *ACM Lett. Program. Lang. Syst.*, 1(3) :243–260, 1992.

[Hin01] Michael Hind. Pointer analysis :haven't we solved this problem yet ? In *PASTE'01*, pages 54–61. ACM Press, 2001.

[HP00] Michael Hind and Anthony Pioli. Which pointer analysis should i use ? In *In Proc. of the 2000 Int. Symp. on Soft. Testing and Analysis*, pages 113–123, 2000.

[HT01] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, 2001.

[IJT91] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization : an overview of the pips project. In *ICS '91 : Proceedings of the 5th international conference on Supercomputing*, pages 244–251, New York, NY, USA, 1991. ACM.

[Int] Définition de l'analyse interprocédurale. http ://en.wikipedia.org/wiki/Interprocedural_optimization.

[Iri93] François. Irigoin. Interprocedural analyses for programming environments. pages 333–350, Amsterdam, The Netherlands, The Netherlands, 1993. Elsevier Science Publishers B. V.

[JT89] Pierre Jouvelot and Rémi Triolet. Newgen : A language-independent program generator. Technical report, Mines ParisTech, 1989.

[JVIW05] Thomas H. Dunigan Jr., Jeffrey S. Vetter, James B. White III, and Patrick H. Worley. Performance evaluation of the cray x1 distributed shared-memory architecture. *IEEE Micro*, 25 :30–40, 2005.

[KSS00] Kazuhiro Kusano, Shigehisa Satoh, and Mitsuhisa Sato. Performance evaluation of the omni openmp compiler. In *In Proceedings of International Workshop on OpenMP : Experiences and Implementations (WOMPEI), volume 1940 of LNCS*, pages 403–414, 2000.

[LLV] Analyse d'alias dans le compilateur llvm. http ://llvm.org/docs/AliasAnalysis.html.

[mat70] Matlab, 1970. http ://en.wikipedia.org/wiki/MATLAB.

[mat88] Mathematica, 1988. http ://en.wikipedia.org/wiki/Mathematica.

[Men08] Amira. Mensi. Analyse des flots de données, 2008.

[Min07] Antoine Miné. Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. *CoRR*, 2007.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6 :40–53, 2008.

[ope] Site officiel de openmp. http ://openmp.org.

[PBD]

[pip] le site officiel de pips. http ://pips4u.org.

[PKH04] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis for c. In *In ACM workshop on Program Analysis for Software Tools and Engineering (PASTE*, pages 37–42. ACM Press, 2004.

[Qia] Wu Qiang. Survey of alias analysis. http ://www.cs.princeton.edu/ jqwu/Memory/-survey.html.

[Rak01] David.Sehr Rakesh.Ghiya, Daniel. Lvery. On the importance of points-to analysis and other memory disambiguation methods for c programs. In *In Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 47–58. ACM Press, 2001.

[Ran01] and Ken.Kennedy Randy.Allen. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers Inc, 2001.

[Ray05] Derek Rayside. Points-to analysis. Technical report, MIT CSAIL, 2005.

[RI] La représentation intermediaire de pips. http ://cri.ensmp.fr/pips/newgen/ri.htdoc/.

[Rug05a] Radu. Rugina. Pointer analysis overview and flow-sensitive analysis. Technical report, Cornell University, 2005.

[Rug05b] Radu Rugina. Pointer analysis overview and flow-sensitive analysis. Technical report, Cornell University, 2005.

[SA] shape analysis. http ://en.wikipedia.org/wiki/Shape_analysis_(program_analysis).

[SJR10] Pascal Sotin, Bertrand Jeannet, and Xavier Rival. Concrete memory models for shape analysis. *Electron. Notes Theor. Comput. Sci.*, pages 139–150, 2010.

[SOHL+98] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1 : The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, pages 217–298, 2002.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[SUI] Le compilateur suif. http :http ://suif.stanford.edu/.

[TDD+02] The BlueGene/L Team, T Domany, Mb Dombrowa, W Donath, M Eleftheriou, C Erway, J Esch, J Gagliano, A Gara, R Garg, R Germain, Me Giampapa, B Gopalsamy, J Gunnels, B Rubin, A Ruehli, S Rus, Rk Sahoo, A Sanomiya, E Schenfeld, M Sharma, S Singh, P Song, V Srinivasan, Bd Steinmacher-burow, K Strauss, C Surovic, Tjc Ward, J Marcella, A Muff, A Okomo, M Rouse, A Schram, M Tubbs, G Ulsh, C Wait, J Wittrup, M Bae (ibm Server Group, K Dockser (ibm Microelectronics, and L Kissel. An overview of the bluegene/l supercomputer, 2002.

[WFPS02] Peng Wu, Paul Feautrier, David Padua, and Zehra Sura. Instance-wise points-to analysis for loop-based dependence testing. In *Proceedings of the 16th international conference on Supercomputing*, pages 262–273, New York, NY, USA, 2002. ACM.

[WHI95] Robert Paul Wilson, John L. Hennessy, and John T. Gill Iii. Efficient context-sensitive pointer analysis for c programs. pages 1–12. ACM, 1995.

[Wil97] Robert Paul Wilson. *EFFICIENT, CONTEXT-SENSITIVE POINTER ANALYSIS FOR C PROGRAMS*. PhD thesis, Standford University, 1997.

[WSR00] Reinhard Wilhelm, Mooly Sagiv, and Thomas Reps. Shape analysis. In *International Conference on Compiler Construction*, number 1781 in LNCS, pages 1–16. Springer Verlag, 2000.

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

Index

$apply_pt$, 116
eval_local, 67
$killable_{MAYM}$, 103
$killable_{MAYN}$, 106
$killable_{MAYT}$, 108
$killable_{killMAYV}$, 109
$killable_{EXACTM}$, 103
$killable_{EXACTN}$, 106
$killable_{EXACTT}$, 108
$killable_{killEXACTV}$, 109
max_N, 105
$inter_M$, 103
$inter_N$, 105
$inter_T$, 107
$inter_{Vref}$, 109
max_M, 102
max_T, 107
max_{cp}, 101
$T_{\mathcal{PT}}$, 76
eval, 69
eval_prog, 66
eta, 71
etv, 73
points_to_free, 97
sti, 72
sources, 70
stub_cp, 70
type_of, 63, 64

any_loop_to_points_to, 130
atomic_location_p, 110

compute_pt_binded, 170

Gen, 74

Kill, 75

points_to_call_site, 169
points_to_cyclic_graph, 133
points_to_function_projection, 168
points_to_graph, 134
points_to_pointer_assignment, 87

transformeur_pts_to_exp, 84

u_ntp, 134

Analyse des pointeurs pour le langage C

RESUME : Les analyses statiques ont pour but de déterminer les propriétés des programmes au moment de la compilation. Contrairement aux analyses dynamiques, le comportement exact du programme ne peut être connu. Par conséquent, on a recours à des approximations pour remédier à ce manque d'information. Malgré ces approximations, les analyses statiques permettent des optimisations et des transformations efficaces pour améliorer les performances des programmes. Parmi les premières analyses du processus d'optimisation figure l'analyse des pointeurs. Son but est d'analyser statiquement un programme en entrée et de fournir en résultat une approximation des emplacements mémoire vers lesquels pointent ses variables pointeurs. Cette analyse est considérée comme l'une des analyses de programmes les plus délicates et l'information qu'elle apporte est très précieuse pour un grand nombre d'autres analyses clientes. En effet, son résultat est nécessaire à d'autres optimisations, comme la propagation de constante, l'élimination du code inutile, le renommage des scalaires ainsi que la parallélisation automatique des programmes. Le langage C présente beaucoup de difficultés lors de son analyse par la liberté qu'il offre aux utilisateurs pour gérer et manipuler la mémoire par le biais des pointeurs. Ces difficultés apparaissent par exemple lors de l'accès aux tableaux par pointeurs, l'allocation dynamique «malloc» ainsi que les structures de données récursives. L'un des objectifs principaux de cette thèse est de déterminer les emplacements mémoire vers lesquels les pointeurs pointent. Ceci se fait en assurant plusieurs dimensions comme :

- la sensibilité au flot de contrôle, c'est-à-dire la mise à jour des informations d'un point programme à un autre ;
- la non-sensibilité au contexte, c'est-à-dire l'utilisation de résumés au lieu de l'analyse du corps de la fonction à chaque appel ;
- la modélisation des champs pointeurs des structures de données agrégées, dans laquelle chaque champ représente un emplacement mémoire distinct.

D'autres aspects sont pris en compte lors de l'analyse des programmes écrits en C comme la précision des emplacements mémoire alloués au niveau du tas, l'arithmétique sur pointeurs ou encore les pointeurs vers tableaux. Notre travail, implémenté dans le compilateur parallélliseur PIPS (Parallélisation interprocédurale de programmes scientifiques), permet d'analyser les applications scientifiques de traitement du signal tout en assurant une analyse intraprocédurale précise et une analyse interprocédurale efficace via les résumés.

Mots clés : analyse statique de programmes, analyse de pointeurs, sensibilité au flot de contrôle, sensibilité au contexte.

Points-to Analysis for The C Language

ABSTRACT: Static analysis algorithms 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 to determine. This information is often used to assess if two pointers are potential aliases, i.e. if they can point to the same memory area. An analysis of pointers, also called points-to analysis, may provide more precision to other analyses such as constant propagation, analysis of dependencies or analysis of live variables. The analysis of pointers is very important for the exploitation of parallelism in scientific C programs since the most important structures they manipulate are arrays, which are typically accessed by pointers. It is necessary to analyse the dependencies between arrays in order to exploit the parallelism between loops. Points-to analysis may also attempt to handle recursive data structures and other structures that are accessed by pointers. This work provides a points-to analysis which is:

- flow-sensitive, by taking into account the order of execution of instructions;
- field-sensitive, since structure fields are treated as individual locations;
- context-insensitive, because functions summaries are computed to avoid re-analyzing functions bodies.

Other issues such as heap modeling, pointer arithmetics and pointers to arrays are also taken into account while analyzing C programs. Our intraprocedural analysis provides precise results to client analyses, while our interprocedural one allows to propagate them efficiently. Our work is implemented within the PIPS (Parallélisation interprocédurale de programmes scientifiques) parallelizer, a framework designed to analyze, optimize and parallelize scientific and signal processing applications.

Keywords: static analysis, points-to analysis, flow-sensitive, context-insensitive, field-sensitive.