Titre: Title:	Extraction des propriétés du code source pour l'annotation automatique
Auteur: Author:	Guillaume Thouvenin
Date:	2002
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Thouvenin, G. (2002). Extraction des propriétés du code source pour l'annotation automatique [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/6999/

UNIVERSITÉ DE MONTRÉAL

EXTRACTION DES PROPRIÉTÉS DU CODE SOURCE POUR L'ANNOTATION AUTOMATIQUE

GUILLAUME THOUVENIN
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
DÉCEMBRE 2002

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

EXTRACTION DES PROPRIÉTÉS DU CODE SOURCE POUR L'ANNOTATION AUTOMATIQUE

présenté par: THOUVENIN Guillaume

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de:

M. MERLO Ettore, Ph.D., président
M. DAGENAIS Michel, Ph.D., membre et directeur de recherche
M. GUIBAULT François, Ph.D., membre

à Marion et à Julie.

REMERCIEMENTS

Je tiens tout d'abord à remercier les membres du jury:

- Ettore Merlo, professeur à l'École Polytechnique de Montréal, pour m'avoir fait l'honneur de présider ce jury, pour sa lecture attentive et ses commentaires avisés concernant mon mémoire.

- Michel Dagenais, professeur à l'École Polytechnique de Montréal, pour m'avoir orienté et conseillé tout au long de cette maîtrise.

Je tiens à remercier Pierre-Luc Brunelle pour m'avoir conseillé sur plusieurs points de mon travail et particulièrement sur la transformation de l'ASG vers le CFG.

Merci aussi à Luc Tétreault et Mathieu Desnoyers pour leur lecture, leurs critiques et toutes les discussions concernant ce mémoire.

RÉSUMÉ

Les analyses statiques désignent des analyses n'ayant pas d'accès aux données réelles d'un programme, qui ne sont disponibles que durant l'exécution, par opposition aux analyses dynamiques. Elles ont pour objectif la détection de certaines propriétés des programmes sans passer par une phase d'exécution de celui-ci.

Ces analyses sont surtout utilisées par les compilateurs dans le but d'optimiser le code à exécuter sur une machine comme par exemple en sortant les invariants d'une boucle. Cependant, elles permettent aussi de prouver de façon formelle certaines propriétés d'un programme. La plupart des analyses statiques sont indécidables. L'un des objectifs va donc être de trouver une approximation correcte d'un programme afin de rendre son analyse possible. Il existe différents outils possédant chacun différentes techniques d'approximations. Parmi ces outils, nous retrouvons *SPLint* qui est développé par David Evans à l'Université de Virginie. Son objectif est double. Tout d'abord il tente de détecter les vulnérabilités d'un programme du point de vue de la sécurité. Ensuite, il tente de détecter les erreurs liées à la programmation. On retrouve notamment parmi ces erreurs celles causées par une mauvaise gestion de la mémoire. L'un des problèmes de SPLint est la génération de faux messages d'erreurs qui nuit à la bonne compréhension des résultats produits par cet outil. Pour contrer ce problème, les développeurs de SPLint ont ajouté la possibilité de mettre dans le code source des annotations

qui peuvent affiner l'analyse. Par exemple, on sait que la fonction *malloc()* peut retourner un pointeur nul. Grâce aux annotations, il est possible de spécifier le comportement de la fonction *malloc()* afin que SPLint tienne compte de cette particularité dans ses analyses. Ces annotations sont actuellement à la charge de l'utilisateur. L'une des améliorations possibles de SPLint serait de permettre l'annotation automatique du code source.

Le but de ce travail est de fournir un moyen permettant l'annotation automatique du code source afin de réduire le nombre de faux messages générés par SPLint. Ceci va permettre d'améliorer la précision du diagnostic tout en facilitant le travail de l'utilisateur. Les étapes nécessaires à l'annotation sont implantées dans l'outil GASTA (Gcc Abstract Syntax Tree Analysis). Dans un premier temps, les annotations concernant les problèmes liés aux allocations mémoires seront couverts. Par la suite, ce travail pourra être étendu à d'autres types d'annotations.

ABSTRACT

The static analysis is performed without access to the real data of a program, in opposition to dynamic analysis. Their goal is to detect some program properties without having to go through the execution phase.

These analysis are mostly used by compilers to optimize the code to execute on a machine. For example, the program's inner loops are good candidates for improvement. Static analysis is used to formally prove some program properties. Most static analysis are impossible to resolve. Finding a correct approximation of a program in order to allow analysis is usally done. There are many different tools with their own static analysis techniques. Among those, there is *SPLint*, developed by David Evans at the University of Virginia. It's objective is twofold. First, it tries to detect code vulnerability. Secondly, it tries to detect program mine errors, including among others memory management problems. One of SPLint's problems is the generation of spurious warning. It's possible to give additional information to *SPLint* using source code annotations. Annotations are stylized user supplied comments that document assumptions about functions, variables, parameters and types. One of *SPLint*'s possible improvement would be extract automatically some of the properties currently specified manually.

This project's goal is to find a way allowing the automatic source code annotation in order to reduce the number of spurious warnings generated by *SPLint*.

This will improve the accuracy of the analysis while facilitating the user's work. The steps which are necessary to accomplish this are implemented in a tool called GASTA (Gcc Abstract Syntax Tree Analysis). First, annotations concerning problems linked to memory allocation is covered. Then, extensions to other kinds of annotations are discussed.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ACRONYMES

ASG	Abstract Syntax Graph
AST	Abstract Syntax Tree
BNF	Backus-Norm Form
CFG	Control flow graph
GASTA	Gcc Abstract Syntax Tree Analysis
GCC	Gnu C Compiler
GLIB	Gnome Library
RAM	Random Access Memory
SAT	Satisfiability Problem
SPLint	SPecifications Lint

CHAPITRE 1

INTRODUCTION

1.1 Motivation et contexte de la recherche

L'analyse statique aide à déterminer, sans exécution et parfois automatiquement, les comportements possibles et donc les propriétés des programmes. Cette analyse se base uniquement sur le code source du logiciel. Ce type d'étude présente au moins deux avantages.

Tout d'abord, l'extraction d'informations concernant le programme peut être utilisée dans le but d'aider, par exemple, à la documentation, à l'annotation du code source, au déverminage ou encore à la validation d'un programme. La détection de code qui ne sera jamais exécuté est un exemple d'analyse statique permettant d'améliorer la qualité du code et donc d'en faciliter sa maintenance. Dans le même registre, il existe aussi des outils permettant la détection de paramètres qui ne sont pas utilisés ou de variables utilisées avant leur définition. Toutes ces propriétés pourraient être détectées par une analyse statique et servir au programmeur dans d'autres tâches comme la documentation du code source. En effet, un intérêt peut être, par exemple, de documenter les signatures des fonctions (variables utilisées à droite et/ou à gauche d'une expression, variables non initialisées) afin de permettre à une personne devant utiliser du code qui lui est étranger d'avoir

une bonne idée de l'évolution des variables dans une procédure.

La seconde utilité de ce genre d'analyse se trouve dans l'optimisation de l'exécution des programmes, et c'est d'ailleurs le champ d'application le plus courant de ces analyses. Les compilateurs implémentent de tels analyses dans le but d'améliorer un programme en supprimant par exemple les bouts de code jamais exécutés, en sortant les invariants d'une boucle, en optimisant l'utilisation des registres, etc... L'analyse de flot de données est aussi utilisée dans les vérifications de type [3].

Pour étudier statiquement les programmes, une simple analyse textuelle du code est bien souvent insuffisante. De plus, la plupart des problèmes d'analyse statique sont indécidables [23] et donc, au cours des années, les informaticiens ont élaboré des modèles pour tenter de rapprocher l'informatique des mathématiques. Une des théories [9] qui est très utilisée aujourd'hui, se place dans un cadre d'interprétation abstraite des programmes. Cette théorie permet de représenter de manière approchée des ensembles d'états par des valeurs abstraites, et propose des techniques pour assurer la convergence des calculs de point fixe sur ces valeurs abstraites.

Sur le plan théorique, il s'agit de prouver formellement la correction de ces algorithmes, d'étudier leur complexité, et le cas échéant de démontrer leur complétude [19]. Une théorie formelle, c'est à dire composée d'un système formel et d'un ensemble d'axiomes[1], est complète si il n'existe pas de formule F telle que ni F ni

[1] un axiome est une formule de base, que l'on considère vraie sans démonstration. C'est en quelque sorte le "point de départ" qui servira à démontrer des théorèmes.

son contraire (non F) ne puissent être démontrés, c'est à dire que si tout ce qui est vrai dans la théorie est démontrable dans la théorie.

Sur le plan expérimental, il s'agit de quantifier l'applicabilité, l'efficacité et la précision de ces nouveaux algorithmes.

Cette introduction constitue le premier chapitre de ce mémoire. Une première étape de ce projet tente de considérer les différents outils qui existent déjà et essaie de voir quels en sont les limitations. Il est à noter que ce travail s'inscrit dans le cadre précis de l'analyse des problèmes liés à la gestion de la mémoire. Ces outils, ainsi qu'une énumération de quelques techniques d'analyse statique font l'objet du second chapitre qui se veut une présentation de l'état actuel des travaux. Le troisième chapitre est une présentation des différents algorithmes développés durant cette maîtrise. Le quatrième chapitre présentera la mise en pratique de la théorie avec la présentation de GASTA, l'outil développé au cours de ce travail. Avant de conclure ce mémoire en tirant un bilan global des travaux accomplis et futurs, des cas pratiques d'analyses ainsi que les résultats obtenus seront présentés dans un cinquième chapitre.

CHAPITRE 2

ANALYSE STATIQUE DES PROBLÈMES DE GESTION DE LA MÉMOIRE

L'une des étapes importantes dans la conception des programmes est la validation du code. Plusieurs techniques sont à la disposition du programmeur. Parmi l'éventail d'outils permettant l'analyse du code, deux catégories se distinguent. La première est constituée des outils d'analyse statique et la seconde des outils d'analyse dynamique. Les analyses statiques reposent sur des analyses approchées qui n'ont pas accès aux données réelles du programme puisque celui-ci n'est jamais exécuté. Les outils d'analyse dynamique, quant à eux, vont tirer profit de l'exécution pour mettre en évidence des erreurs dans la conception d'un logiciel. L'un des grands avantages des outils d'analyse statique est qu'ils peuvent offrir un moyen exhaustif de validation des programmes. De plus, dans certains cas particuliers comme les virus informatique, il peut être intéressant de ne pas avoir à exécuter le programme pour l'analyser.

En règle générale, les analyses statiques sont non décidables [23], les problèmes qu'elles soulèvent ne peuvent être résolus à l'aide d'une machine de Turing, qui est le modèle théorique de tous les ordinateurs actuels. Le problème de la terminaison des programmes en est un exemple. Ces analyses sont alors faites sur des ap-

proximations du programme. De nouveaux problèmes sont alors introduits qui ne sont plus indécidables mais qui sont généralement NP-Complet. Cela signifie qu'il est impossible avec les meilleurs algorithmes existants de résoudre ces problèmes en temps polynomial; au moins une solution existe. Le problème *SAT*, qui est celui de la satisfaisabilité d'un ensemble de clauses, est le premier problème NP-Complet au sens où tous les autres problèmes NP-Complet découlent directement ou indirectement de *SAT*.

Ces analyses sont surtout utilisées par les compilateurs afin d'extraire certaines propriétés du code dans le but de l'optimiser (analyse de flot de données, durée de vie des variables, etc...) mais elles peuvent aussi vérifier et valider, complètement ou partiellement, des programmes par rapport à des spécifications.

Dans le cadre de ce travail une approche permettant d'extraire de façon statique de l'information sur l'utilisation des pointeurs est présentée. Cette information est alors utilisée afin de permettre la détection de propriétés liées à la gestion de la mémoire en C. Cette approche résoud certains des problèmes se rapportant aux mauvaises manipulations des pointeurs. Deux domaines particuliers de l'analyse statique seront couverts:

- les alias et l'analyse de flot de données, qui informent sur l'utilisation des pointeurs;
- le calcul de point fixe et la théorie des treillis, principal outil disponible pour

cette analyse.

L'état de l'art présente une liste non exhaustive des outils d'analyse statique actuellement disponibles même s'ils ne touchent pas directement à l'utilisation des pointeurs. Les outils de vérification formelle ne seront pas abordés puisqu'ils dépassent le cadre ce mémoire. Ce travail se focalisera sur les outils d'aide à la mise au point et plus principalement sur les outils permettant de mettre en évidence les erreurs survenant lors d'une mauvaise gestion de la mémoire.

2.1 État de l'art

2.1.1 Les outils d'analyse statique

Comme cela a été mentionné dans l'introduction de ce chapitre, cette recherche porte sur les outils d'aide au diagnostic. La majorité des programmes appartenant à cette catégorie nécessitent de la part des utilisateurs une spécification du comportement du programme ou du moins de certaines parties du programme. Ces spécifications peuvent prendre la forme d'annotations [14] ou d'assertions [5]. Par exemple, si on considère la fonction *malloc()*, qui peut retourner un pointeur nul, il serait possible de le préciser en ajoutant un commentaire juste avant la fonction. Ce commentaire pourrait être de la forme */*fn-may-ret-nul*/ malloc(....)*. Ces spécifications permettent d'obtenir une information supplémentaire qui n'est pas disponible de façon évidente directement à partir du code.

2.1.1.1 Outils automatiques

Il existe cependant des outils complètement automatiques. Strom et Yellin [32] ont présenté une méthode permettant d'assurer qu'une variable qui est utilisée à droite d'une expression est bien initialisée. Leur technique est une analyse de flot de données qui est effectuée par le compilateur.

Le système *Syntox* permet d'effectuer le déverminage du langage Pascal par la détermination de l'intervalle de variation des variables entières dans un programme. Ces variables ne doivent pas sortir des limites de l'intervalle fixé [5]. Syntox permet par exemple de détecter de façon complètement automatique des erreurs d'index lors d'accès aux tableaux. Il utilise deux algorithmes de calculs de point fixe. Il fonctionne dans le cas ou les bornes des tableaux sont connues à la compilation.

2.1.1.2 Outils nécessitant des spécifications

Parmi les outils nécessitant des spécifications on retrouve *Flavor* [34]. Dans cet outil il est possible de préciser le comportement dynamique de certaines parties d'un programme. Les variables peuvent ainsi avoir différentes "saveurs". Par exemple, si une variable *somme* contient la somme d'un calcul effectué dans une boucle elle pourra prendre trois "saveurs" possibles qui sont : initialisée, somme partielle et somme totale. Flavor se concentre sur les annotations fournies par le programmeur pour déduire si un programme est correct. Ces annotations sont des

formules logiques permettant soit de préciser l'état d'une variable à un point précis du code, soit d'émettre des hypothèses. Le but étant de vérifier les hypothèses par rapport aux assertions proposées. Landi et Ryder ont développé, dans *Flavor*, un algorithme permettant d'évaluer les pointeurs possiblement aliasés. Cet algorithme est très imprécis surtout en présence de pointeurs arbitraires.

ESC (Extended Static Checking) [10] est un outil développé à l'origine pour le langage de programmation Modula-3. Il est maintenant disponible pour Java. Il permet la détection d'erreurs telles que les problèmes d'indexage dans les tableaux, les déréférencement de pointeurs nuls ainsi que les problèmes de verrouillage et de courses dans les programmes multi-fils. ESC part d'un programme annoté avec des spécifications. Ces spécifications vont ensuite permettre la génération de formules mathématiques logiques en utilisant un générateur de conditions de vérification. Ces équations sont alors présentées à un démonstrateur de théorème. Le but ici n'est pas de prouver qu'un programme fait ce qu'il est supposé faire mais seulement qu'il ne comporte pas certaines erreurs. ESC est un outil intéressant puisqu'il réussit à générer des formules mathématiques permettant la vérification de vrais programmes. De plus le démonstrateur de théorème peut travailler de façon complètement automatique mais sa lenteur le rend inutilisable de façon régulière.

Le groupe de recherche de l'Université de Stanford a développé le compilateur *Suif* [33] utilisant des techniques d'analyse statique et dynamique permettant d'améliorer la qualité des logiciels. Il est axé sur la parallélisation des programmes.

On y retrouve diverses analyses comme la détection d'alias en utilisant un algorithme inter-procédural. Un point très intéressant concernant ce compilateur est ça modularité. Suif se compose de deux couches, un noyau et des modules. Le noyau offre des points d'entrées/sorties entre le coeur du compilateur et les modules. Ceci permet le développement simple et rapide de ses propres modules et donc ses propres analyses.

SPLint (que l'on peut interpréter de trois façons selon la FAQ de SPlint par "Secure Programming Lint" ou bien "SPecifications Lint" ou encore "first aid for programmers") [13] est le successeur de LCLint, qui lui-même a succédé à LCL qui était basé sur le langage de spécification Larch C et sur lint, un outil de vérification du langage C. Le premier objectif de LCLint était de rapporter le maximum d'erreurs qui n'étaient pas détectées à la compilation. La plus grosse amélioration apportée à LCLint par rapport à lint a été l'ajout d'annotations. Contrairement aux outils comme ESC ou encore Flavor, qui offrent des spécifications de bas niveau liées au code, les spécifications dans LCLint permettent de décrire des propriétés plus générales du programme. Les analyses faites dans SPLint sont moins évoluées que dans les autres outils présentés. L'inconvénient principal est le nombre de faux messages générés dû à une analyse moins fine; par contre, son principal atout est sa vitesse d'analyse. Cependant, il est possible de diminuer le nombre de fausses erreurs en utilisant les annotations. Le problème dans ce cas est que l'utilisateur doit parcourir l'ensemble du code afin de l'annoter. Depuis SPLint, la démarche

est plus orientée vers la détection de problèmes liés à la sécurité.

2.1.2 L'analyse de flot de données

L'analyse statique est un domaine de l'informatique qui est surtout utilisé par les compilateurs afin de permettre l'optimisation du code. Les propriétés que ces outils mettent en évidence dépendent du langage de programmation utilisé et, bien entendu, elles dépendent aussi de l'architecture de la machine sur laquelle le programme va s'exécuter (par exemple exécution séquentielle ou parallèle, différence dans le jeux d'instructions, etc...). Dans le cadre de cette maîtrise, les problèmes qui découlent d'une mauvaise gestion de la mémoire ont été les premiers à être traités. Afin d'identifier ces problèmes, l'information intéressante qu'il est nécessaire d'extraire du code source concerne les variables adressant la mémoire (les pointeurs). Il s'agit la d'un type précis d'analyse que l'on appelle détection de pointeurs *aliasés*.

Le terme "pointeurs *aliasés*" signifie que deux pointeurs référencent la même zone mémoire. Par exemple, considérons le code suivant :

```
int main() {
  int r, *ptr, *ptr1;
  ptr = &r;
  ptr1 = &r;
}
```

$*ptr$ et $ptr1$ représentent la même zone mémoire et donc ils sont aliasés. La

détection d'alias est une analyse de dépendance entre les différentes variables qui met en évidence les conflits d'accès en mémoire. Ce problème est équivalent à celui posé dans ce mémoire.

Il existe différentes façons de représenter les alias. Il y a le modèle complet [35] dans lequel toutes les paires d'alias sont citées explicitement, le modèle compact [6] dans lequel ne sont conservées que les paires de bases (les autres paires étant déduites de celles de bases) ou encore le modèle basé sur la relation "pointe vers" où la paire s'interprète comme : le premier élément pointe sur le second.

Concernant l'analyse présentée dans ce mémoire, il n'est pas nécessaire de connaître précisément la zone mémoire référencée par un pointeur mais seulement si ce pointeur pointe sur quelque chose de façon certaine. Il est alors intéressant d'utiliser le modèle basée sur la relation "pointe vers" en l'adaptant de la façon suivante:

- $(*p \leftarrow @ - 1)$ indique que p pointe sur une adresse mémoire;
- $(*p \leftarrow @0)$ correspond à un pointeur nul;
- $(*p \leftarrow *q)$ signifie que p et q référencent la même zone mémoire et donc que p dépend de q. C'est à dire que si q est nul alors p est lui aussi nul.

Plus de détails sur cette notation seront donnés dans les chapitres suivants. Les analyses d'alias peuvent être classées selon deux critères. Tout d'abord l'analyse

peut être définitive ou "must-alias" [2] c'est à dire que deux pointeurs partagent la même zone mémoire de façon certaine contrairement à la seconde analyse qui est une analyse possible [11].

La façon de procéder sera de considérer que tous les pointeurs sont initialement nuls. Ensuite, une analyse d'aliasing va permettre de déterminer quelles sont les zones mémoires référencées et par quels pointeurs. Finalement, en comparant les pointeurs retournés par une fonction avec ceux présents dans l'ensemble des pointeurs *aliasés*, il sera possible de savoir si la fonction étudiée retourne ou ne retourne pas de pointeurs possiblement nuls. Cette information va alors permettre l'annotation de la fonction.

Comme nous l'avons suggéré précédemment, la détection d'*alias* tente de déterminer quelles seront les zones mémoire référencées lors de l'exécution. Or, connaître la zone mémoire référencée par un pointeur est un problème NP-Complet [23] [28], c'est a dire qu'il n'existe pas d'algorithme permettant de trouver une solution polynomiale. Ce problème a conduit au développement de différentes méthodes d'analyse et donc à différentes approximations et heuristiques.

Plusieurs critères permettent de comparer les analyses. Il est possible de distinguer les analyses sensibles au flot de contrôle (*flow-sensitive*) et celles qui ne le sont pas (*flow-insensitive*). Les analyses sensibles au flot de contrôle vont tenir compte du graphe de flot de contrôle ou CFG [1] c'est à dire que celui-ci a une incidence sur les résultats de l'analyse. Un CFG est une représentation abstraite

du code que nous analysons. Une définition d'un CFG est donnée dans la section 3.1.2 page 22. Pour bien comprendre la différence entre les analyses sensibles au flot de contrôle et les analyses insensibles au flot de contrôle, considérons l'exemple suivant:

```
    int a,b;
    int *ptr;

    if (a>b)
(1)     ptr = &a;
(2) else
(3)     ptr = &b;
(4)
  ...
```

une analyse sensible au flot de contrôle va conclure que *ptr* va pointer dans la première branche sur a et sur b dans la seconde. Ensuite, cette analyse fusionne les états pour finalement déduire que *ptr* peut à la fois pointer sur a ou sur b en (4). Une analyse insensible au flot de données va considérer que *ptr* pointe sur a ou sur b en tout point du programme ce qui est moins précis que la première analyse.

Une autre distinction qui existe entre les différentes analyses est celle concernant la sensibilité au contexte. Il s'agit des traitements qui seront réalisés lors d'appels à d'autres fonctions. Si on tient compte de ce qui se passe durant ces appels, l'analyse est dite sensible au contexte, autrement l'analyse est insensible au contexte. Un autre aspect à retenir lorsque concernant les analyses de pointeurs

est la propagation de l'information à travers les différents appels de fonctions. Pour traiter ces cas on utilise un graphe d'appel appelé PCG (Program Call Graph). C'est un graphe dont les nœuds sont des fonctions et les arcs dirigés indiquent quelle fonction appelle quelle autre fonction. La figure 2.1 montre un exemple de graphe d'appel.

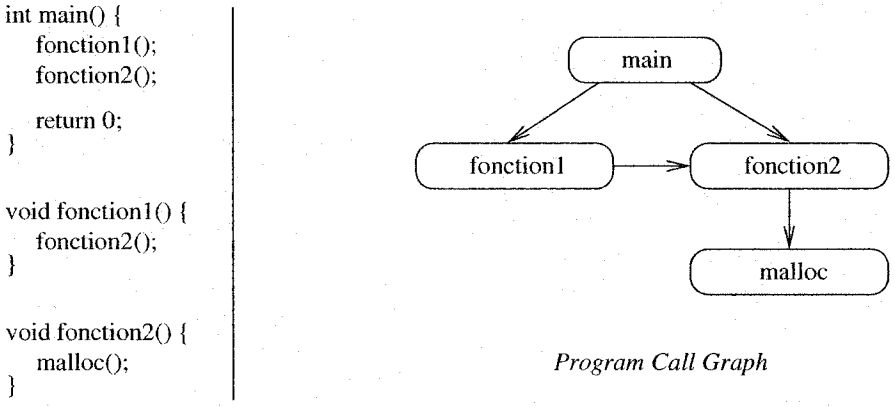


Figure 2.1: Exemple de PCG

Afin de connaître les pointeurs *aliasés*, une analyse triviale serait de considérer que chaque pointeur peut référencer n'importe quelle adresse en mémoire. Cette technique est simple à implémenter mais elle est très imprécise et l'information qu'elle fournit ne serait pas vraiment exploitable. Il existe plusieurs méthodes d'approximation. Michael Hind et Anthony Pioli [18] comparent différentes techniques existantes que l'on retrouve dans la thèse de A. Pioli [27] et qui sont (par ordre croissant de complexité):

- Adress-Taken
- Steensgaard [31]
- Anderson [4]
- Flow-Insensitive [17]
- Flow-Sensitive [7]

L'analyse dite *Address-Taken* est insensible à la fois au contexte et au flot de contrôle. Elle n'utilise qu'un seul ensemble pour stocker tous les alias de tout le programme. Elle s'effectue en traversant toutes les fonctions une seule fois. Le principe est de générer un ensemble contenant toutes les variables utilisées dans le code. Ensuite, on considère que le déréférencement d'un pointeur pointe sur l'ensemble construit et donc sur toutes les variables du code. Cette analyse restreint un peu le domaine par rapport à l'analyse de base qui considère qu'un pointeur peut référencer toute la mémoire mais elle reste cependant très imprécise.

L'analyse de Steensgaard, comme dans l'analyse précédente, calcule un seul ensemble d'alias pour toutes les fonctions. La différence se trouve dans la manière de construire cet ensemble. Son seul avantage par rapport à l'analyse dite *Address-Taken* est donc sa rapidité.

Anderson utilise un algorithme itératif sur les instructions. L'itération s'effectue

jusqu'à atteindre un point fixe. Plus de détails concernant les calculs de point fixe seront donnés dans la section suivante car l'un des algorithmes implémentés durant cette maîtrise utilise cette technique. Cet algorithme, celui d'Anderson, utilise une représentation compacte des alias. L'avantage est qu'un pointeur n'est associé qu'à une seule zone mémoire, d'où une plus grande finesse de l'analyse.

L'analyse Flow-Insensitive est très proche de la précédente. La différence est que cette analyse tient compte du PCG pour le calcul itératif, dans ce cas les aspects inter-procéduraux et intra-procéduraux sont traités.

Enfin, l'analyse la plus fine et la plus complexe est l'analyse Flow-Sensitive. Cette analyse est elle aussi basée sur la théorie de calcul de point fixe. L'itération s'effectue à la fois sur les fonctions (PCG) et sur le CFG. L'ensemble des alias va donc être calculé par rapport aux blocs de base du CFG, c'est à dire que chaque bloc contient de l'information. Là encore, plus de détail à propos de cette analyse seront donnés dans la suite de ce mémoire puisque certains algorithmes inclus dans l'outil élaboré durant cette maîtrise s'en inspirent.

L'analyse de flot de données quant à elle utilise l'analyse de dépendance de données mais dans cette analyse nous ne retiendrons que les opérations apportant la dernière écriture précédent une lecture. Ainsi, lors de deux écritures successives avant une lecture, la première écriture ne sera pas prise en compte. Elle fournit des informations concernant la précédence et la dépendance entre les données. Cette dernière écriture, qui est appelée la *source* est importante car c'est elle qui va

fournir la valeur lue et c'est donc bien cette valeur qui nous informe sur le flot de données. L'analyse de flot de données est utilisable aussi bien pour la mise au point (recherche des variables non initialisées) que pour l'analyse de localité ou la parallélisation automatique.

Il existe deux techniques d'analyses de flot. Une méthode itérative et une méthode géométrique. La plus classique est la première méthode et c'est celle que nous utiliserons. John B. Kam et Jeffrey D. Ullman [21] ont associé l'algorithme de propagation des données proposé par Kildall [22] pour l'optimisation de code avec la technique itérative proposée par Hecht et Ullman qui assure un nombre maximum d'itérations de l'algorithme de propagation, lorsque les données sont représentées par un vecteur de bits. Le travail de Kam et Ullman cherche à associer une propriété particulière à un point donné d'un programme. Par exemple, on peut tenter de prouver qu'une variable i est toujours positive, ou qu'un ensemble de variables vérifient bien telle propriété [20]. La méthode géométrique [15] consiste à lier chacune des valeurs lues à sa source. Les ensembles d'opérations sont représentés par des polyèdres et le calcul de la source se ramène à des opérations d'union, d'intersection et de recherche de maximum sur ces polyèdres.

2.1.3 Calcul de point fixe et treillis

Il a été mentionné à plusieurs reprises que les problèmes d'analyse statique sont indécidables. Lorsque ce genre de problème survient, il est alors nécessaire de

travailler sur une approximation de celui-ci, et il est aussi important d'introduire des hypothèses de finitude afin de s'assurer de la possibilité de les résoudre. C'est dans ce contexte que le treillis se révèle très utile.

Pour introduire les treillis, il est indispensable de commencer par énoncer deux définitions [29] la seconde étant basée sur la première.

Tout d'abord la relation d'ordre: Une relation R dans un ensemble S est appelée une relation de pré-ordre ou d'ordre partiel si elle est réflexive, antisymétrique et transitive. Un ensemble S combiné à une relation de pré-ordre R est appelé un ensemble partiellement ordonné et est noté (S, R).

Il est maintenant possible de définir un treillis comme un ensemble partiellement ordonné dans lequel chaque paire d'éléments a un supremum et un infimum. Les treillis possèdent plusieurs propriétés particulières. Ces propriétés ont été utilisées par Patrick COUSOT [8] afin de présenter en 1978 un doctorat pour obtenir le grade de docteur es sciences mathématiques. Il a proposé une théorie qui permet de résoudre les problèmes de la détermination de propriétés sémantiques d'un programme en utilisant la construction des points fixes extrêmes d'opérateurs monotones sur un treillis.

Cette théorie est particulièrement importante pour l'analyse effectuée dans le cadre de cette maîtrise puisque c'est elle qui assure la convergence de l'algorithme utilisé .

La théorie des treillis n'a pas vraiment évoluée depuis Cousot mais des projets

basés sur le calcul de point fixe se basent sur cette théorie, par exemple *Context Sensitive Flow Analysis to Detect Blocked Statements* de B. Malenfant, G. Antoniol, E. Merlo et M. Dagenais[26].

2.2 Discussion

Deux aspects intéressants se dégagent dans la spécification du code source. Tout d'abord, comme on le voit avec les différents outils d'analyse statique auxquels il a été fait référence, cela permet d'affiner l'analyse et donc cela va permettre de réduire le nombre de faux messages d'erreurs produits; ces faux messages constituent un problème si les informations pertinentes se retrouvent noyées dans ces messages. Il serait sans doute intéressant de disposer d'un outil supplémentaire qui serait capable d'extraire de l'information pertinente concernant les programmes à analyser afin d'annoter leur code pour diminuer le nombre des faux messages d'erreurs. Le second point intéressant est que le fait de rendre explicite les propriétés du code peut simplifier, et la compréhension, et l'entretien des logiciels. Ceci confirme le fait que trouver une méthode permettant d'extraire les propriétés d'un programme à partir de son code source serait intéressant.

CHAPITRE 3

ALGORITHMES

En tirant le bilan de notre revue de littérature, il est apparu qu'une des faiblesses des outils d'analyse statique utilisant des spécifications était l'aspect manuel de ces spécifications. Cette observation constitue le point de départ de ce mémoire. On peut maintenant définir l'objectif de cette recherche comme étant de fournir une méthode permettant d'extraire les propriétés d'un programme à partir de son code source. Par exemple, une de ces propriétés pourrait être le calcul d'un ensemble de fonctions du programme pouvant retourner un pointeur possiblement nul. Les informations ainsi collectées pourraient alors permettre d'annoter le code source afin d'améliorer la détection des erreurs de programmation par des outils d'analyse statique tel que SPLint.

Pour atteindre cet objectif, plusieurs algorithmes sont utilisés. Ils interviennent dans le processus d'annotation qui se décompose en trois parties comme le montre la figure 3.1.

Tout d'abord, dans un premier temps, il est nécessaire d'extraire l'information à partir du code source. Cependant, cette extraction ne se fait pas directement à partir du code source mais elle se fait en utilisant une représentation abstraite du code source comme le graphe de flot de contrôle (CFG). Le CFG est obtenu à

partir d'un graphe de syntaxe abstrait (ASG).

Ensuite, le second algorithme doit extraire, à partir du CFG obtenu précédemment, l'information nécessaire à l'annotation.

Enfin, la dernière étape consiste à annoter ce code source en utilisant l'information recueillie précédemment. Pour cela, à partir du moment où l'information est disponible, par exemple qu'une fonction peut retourner un pointeur possiblement nul, il est facile d'annoter cette fonction comme étant une fonction pouvant retourner un pointeur possiblement nul. Par contre, l'ordre dans lequel les fonctions sont évaluées est important et suit un algorithme précis qui constitue cette dernière étape.

Ce chapitre est donc une présentation de l'ensemble des algorithmes utilisés dans cette étude.

3.1 Algorithme de conversion d'un ASG vers un CFG

3.1.1 Le graphe de syntaxe abstrait (ASG)

Le graphe de syntaxe abstrait est obtenu en utilisant un autre outil, le compilateur *GCC*. Un ASG est un graphe dont chaque nœud est un opérateur et chaque feuille est une opérande, comme le montre la figure 3.2.

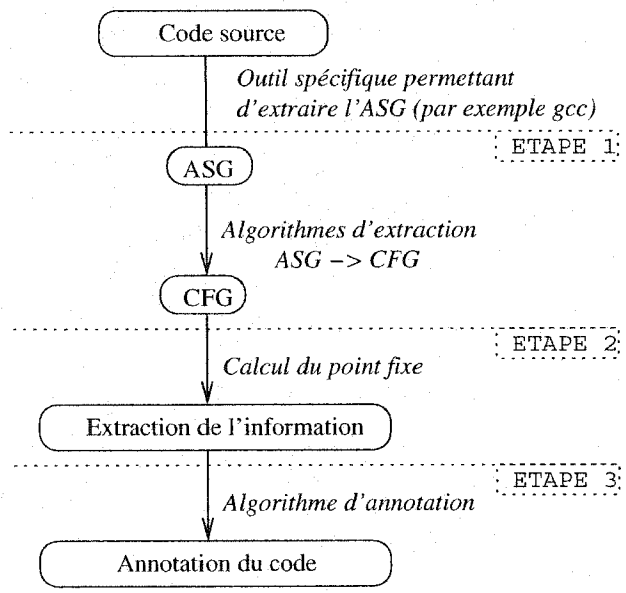


Figure 3.1: Étapes du processus d'annotation

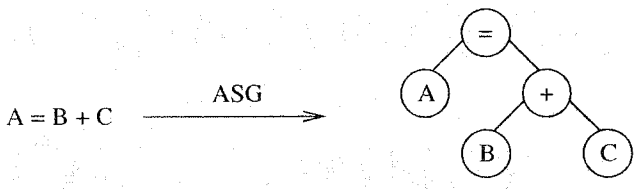


Figure 3.2: Exemple d'ASG

3.1.2 Le graphe de flot de contrôle (CFG)

Les graphes de flot de contrôle sont des graphes orientés $G = (V, E)$, où les nœuds sont les blocs de base d'un programme et les arcs correspondent aux dépendances de contrôle entre les différents blocs. Cela signifie que si deux nœuds sont liés par un arc, alors les blocs correspondants peuvent se suivre immédiatement dans une séquence d'exécution. Un bloc de base est une séquence d'instructions consécutives sans possibilité d'arrêt ou de branchement avant la fin de cette séquence.

Le graphe de flot de contrôle permet donc de représenter les dépendances de contrôle; c'est une représentation de l'ordre d'exécution du programme. La dépendance de contrôle entre deux opérations indique que l'exécution de l'une des opérations dépend du résultat de l'autre.

3.1.3 Le passage de l'ASG au CFG

Avant de présenter les différents algorithmes, il est nécessaire de donner une définition de l'ensemble *open_nodes* utilisé dans tous les algorithmes.

- *open_nodes*: ensemble des nœuds du CFG dont les successeurs n'ont pas encore été tous identifiés. Il s'agit également des prédécesseurs du bloc de base[1] courant.

Un algorithme particulier est utilisé selon les différentes structures de contrôle (*while, if, for, ...*). Seuls deux de ces algorithmes (figures 3.3 et 3.4) seront vus en détail puisque le principe de fonctionnement est sensiblement le même pour tous. Chacune des particularités des autres algorithmes (figure 3.5 à 3.14) seront bien expliquées.

Le premier algorithme abordé dans ce chapitre (Figure 3.3) est aussi le plus simple. Il est utilisé lorsque le bloc de base contient une expression. Dans ce cas, la seule opération consiste à créer un nouveau nœud et mettre à jour les liens

[1]Dans cette étude, un bloc de base correspond à une et une seule instruction.

```
visit_expr_stmt()
Begin
  | a ← new_cfg_node (id + +, "GENERIC");
  | ∀n ∈ open_nodes,
  |   | link (n, a);
  | open_nodes ← {a};
  | visit_next_stmt();
End
```

Figure 3.3: Algorithme pour le "expression"

entre le bloc de base courant et les blocs appartenant à l'ensemble *open_nodes*. La création d'un nouveau nœud est faite par la fonction *new_cfg_node()*. Elle s'occupe de maintenir à jour un numéro de bloc de base et gère l'allocation de la mémoire nécessaire lors la création du nœud. Ensuite, il faut actualiser la liste *open_nodes* de manière à ne contenir que le bloc de base courant. C'est la fonction *link()* qui établit ce lien. Ces étapes faites, il faut visiter le bloc suivant. On remarque que l'ensemble *open_nodes* contient uniquement le bloc qui vient d'être visité et ce bloc est bien le prédécesseur du prochain bloc.

Le second algorithme correspond au "while" et il est décrit à la figure 3.4. Tout comme dans le cas d'une expression, le premier traitement à effectuer consiste à créer un nouveau nœud (qui est aussi un nouveau bloc de base). Ce bloc indique le début de la boucle (la condition). L'étape suivante est là-encore identique au cas d'une expression, puisqu'il s'agit de mettre à jour les liens entre le nouveau nœud et ceux présents dans la liste *open_nodes*. Dans le cas du "while" il est possible d'utiliser, dans le corps, des commandes telles que *break* et *continue*. Il

```
visit_while_stmt()
Begin
  a ← new_cfg_node (id++, "WHILE_COND");
  ∀n ∈ open_nodes,
    | link (n, a);
  open_nodes ← {a};
  visit_while_condition();
  breaksStack.push(∅);
  continuesStack.push(∅);
  visit_while_body();
  ∀n ∈ (open_nodes ∪ continuesStack.pop()),
    | link (n, a);
  open_nodes ← {a} ∪ breaksStack.pop();
  visit_next_stmt();
End
```

Figure 3.4: Algorithme pour le "while"

est alors nécessaire de mettre en place un mécanisme qui va traiter ces cas comme l'utilisation d'une pile. Cette pile est tout d'abord initialisée avec un élément vide et elle sera modifiée lors du traitement des *break* et des *continue* (figures 3.10 et figure 3.11). Les piles étant maintenant à jour, la visite du corps du "while" peut désormais avoir lieu. À la fin de cette visite, les piles contiennent l'ensemble des nœuds correspondant au cas d'un *break* ou d'un *continue*. Il faut alors compléter les liens entre les *continue* et le nœud correspondant à la condition de la structure de contrôle "while" créée au début de l'algorithme. Les nœuds *break* seront, quant à eux, ajoutés à la liste des nœuds ouverts tout comme le nœud initial du "while".

Après avoir expliqué en détail les deux premiers algorithmes de conversion, seules les particularités des suivants seront détaillées.

L'algorithme de la figure 3.5 (cas d'un *if*) utilise une pile supplémentaire qui est locale. Cette pile va permettre de conserver une référence sur le nœud qui

```
visit_if_stmt()
Begin
  a ← new_cfg_node (id++, "IF_BEGIN");
  ∀n ∈ open_nodes,
    link (n, a);
  open_nodes ← {a};
  visit_if_condition();
  visit_if_then();
  thenOpenNodeList ← open_nodes;
  open_nodes ← {a};
  visit_else_expression();
  open_nodes
      ← open_nodes ∪ thenOpenNodeList;
  thenOpenNodeList ← ∅;
  visit_next_stmt();
End
```

Figure 3.5: Algorithme pour le "if"

correspond à la condition vraie du *if*. Il est nécessaire de procéder de cette manière car, lors de la visite de la condition fausse, il va être nécessaire de remettre à jour la liste des nœuds ouverts, afin de conserver une intégrité entre la condition du "if" et les instructions de la branche fausse du *if*. En mettant à jour cette liste, les références avec les instructions de la branche vraie seraient alors perdues sans l'utilisation de cette pile. L'avantage d'une pile par rapport à une autre structure est quelle permet de traiter de façon naturelle les imbrications de *if*.

L'algorithme utilisé dans le traitement d'un *for* est similaire à celui utilisé lors d'un *while*. La seule différence se situe après la visite du corps du *for*. En effet, c'est après la visite du corps du *for* que le nœud correspondant à la condition du *for* doit être créé puisque celle-ci se retrouve à la fin du *for*. Cet aspect apparaît aussi lors de l'extraction du CFG dans une instruction correspondant à un *do* (figure 3.7). Il est à noter qu'une simplification est faite dans le traitement des

```
visit_for_stmt()
Begin
  | visit_for_init();
  | a ← new_cfg_node (id++, "FOR_COND");
  | ∀n ∈ open_nodes,
  |   | link (n, a);
  | open_nodes ← {a};
  | visit_for_condition();
  | breaksStack.push(∅);
  | continuesStack(∅);
  | visit_for_body();
  | b ← new_cfg_node (id++, "FOR_EXPR");
  | ∀n ∈ open_nodes,
  |   | link (n, b);
  | open_nodes ← {b};
  | visit_for_expression();
  | ∀n ∈ (open_nodes ∪ continuesStack.pop()),
  |   | link (n, a);
  | open_nodes ← {a} ∪ breaksStack.pop();
  | visit_next_stmt();
End
```

Figure 3.6: Algorithme pour le "for"

conditions puisque les expressions complexes ne sont pas considérées, contrairement à l'algorithme proposé par Schwartz [30]. Ceci implique que si une affectation est faite soit lors de l'initialisation du *for*, soit lors du calcul des nouvelles valeurs du *for* alors cette affectation ne sera pas prise en compte par notre analyse. Le traitement de conditions complexes s'inscrit dans les futures améliorations de l'outil présenté dans ce mémoire.

La particularité du *switch* réside dans le fait qu'il peut ou non exister un cas par défaut. Pour traiter cela, encore une fois, le recours à une pile s'avère judicieux. Cette pile contient des booléens (figure 3.8). Une valeur fausse indique qu'il n'existe pas de cas par défaut. Donc, au début de l'algorithme, ne sachant pas si une telle condition existe, un booléen faux est ajouté sur la pile. Lorsque le corps du *switch*

```
visit_do_stmt()
Begin
  a ← new_cfg_node (id++, "DO_BEGIN");
  ∀n ∈ open_nodes,
    link (n, a);
  open_nodes ← {a};
  breaksStack.push(∅);
  continueStack.push(∅);
  visit_do_body();
  b ← new_cfg_node (id++, "DO_COND");
  ∀n ∈ open_nodes,
    link (n, b);
  open_nodes ← {b};
  visit_do_condition();
  ∀n ∈ continuesStack.pop(),
    link (n, b);
  link (b, a);
  open_nodes ← {b} ∪ breaksStack.pop();
  visit_next_stmt();
End
```

Figure 3.7: Algorithme pour le "do"

est traité, la valeur présente au sommet de la pile est vérifiée afin de récupérer la valeur du booléen. Cette valeur est mise à jour lors du traitement des *case* (figure 3.9). Si la valeur est toujours fausse (pas de cas par défaut), un lien est ajouté entre le nœud initial du "switch" et les nœuds présents dans la liste *open_nodes*, sinon il n'y a aucun traitement a effectuer.

Traiter une instruction *goto* requiert un algorithme différent (figure 3.13). Le problème est qu'il est impossible de connaître l'endroit où cette instruction va se brancher avant d'avoir parcouru l'ensemble du code. En effet, le branchement peut très bien s'effectuer sur la dernière instruction du programme. Pour résoudre ce problème, une solution consiste à utiliser une table contenant toutes les destinations et à conserver également une référence sur le nœud correspondant au bloc

```
visit_switch_stmt()
  Begin
    defaultcaseStack.push(FALSE);
    visit_switch_condition();
    a ← new_cfg_node (id++, "SWITCH_COND");
    ∀n ∈ open_nodes,
      | link (n, a);
    open_nodes ← {a};
    breaksStack.push(∅);
    labelStack.push(a);
    visit_switch_body();
    defaultCase = defaultcaseStack.pop();
    if (defaultCase == FALSE)
      open_nodes ← {a} ∪ open_nodes;
    open_nodes
        ← open_nodes ∪ breaksStack.pop();
    labelStack.pop();
    visit_next_stmt();
  End
```

Figure 3.8: Algorithme pour le "switch"

```
visit_case_label()
  Begin
    visit_label_value();
    a ← new_cfg_node (id++, "CASE_LABEL");
    ∀n ∈ open_nodes,
      | link (n, a);
    link(labelStack.top(), a);
    open_nodes ← {a};
    if is_inDefaultCase
      defaultcaseStack.top() = TRUE;
    visit_next_stmt();
  End
```

Figure 3.9: Algorithme pour le "case"

```
visit_break_stmt()
  Begin
    a ← new_cfg_node (id++, "BREAK");
    ∀n ∈ open_nodes,
      | link (n, a);
    open_nodes ← ∅;
    breaksStack.top() ← {a} ∪ breaksStack.top() ;
    visit_next_stmt();
  End
```

Figure 3.10: Algorithme pour le "break"

```
visit_continue_stmt()
  Begin
    | a ← new_cfg_node (id + +, "CONTINUE");
    | ∀n ∈ open_nodes,
    |   | link (n, a);
    | open_nodes ← ∅;
    | continuesStack.top() ←
    |     {a} ∪ continuesStack.top() ;
    | visit_next_stmt();
  End
```

Figure 3.11: Algorithme pour le "continue"

```
visit_return_stmt()
  Begin
    | a ← new_cfg_node (id + +, "RETURN");
    | ∀n ∈ open_nodes,
    |   | link (n, a);
    | open_nodes ← ∅;
    | visit_next_stmt();
  End
```

Figure 3.12: Algorithme pour le "return"

```
visit_goto_stmt()
  Begin
    a ← new_cfg_node (id++, "GOTO");
    ∀n ∈ open_nodes,
      | link (n, a);
    open_nodes ← ∅;
    gotoStack.push(Ref, Dest);
    visit_goto_destination();
    visit_next_stmt();
  End
```

Figure 3.13: Algorithme pour le "goto"

de base "goto" dans le CFG. Ainsi, lorsque tout l'ASG aura été visité, il suffira de parcourir la table et, en fonction des informations stockées dans celle-ci, le CFG pourra ainsi être complété en liant chacun des "goto" avec l'étiquette qu'il réfère. Ceci est rendu possible par le fait que les étiquettes sont stockées dans une table de dispersement (figure 3.14) et la clé correspond au numéro du nœud dans l'ASG (la destination d'un "goto" est calculée par rapport à ce numéro). Par exemple, si la destination d'une instruction "goto" est le nœud *@53*, il est possible de trouver le pointeur correspondant à l'étiquette dans le CFG en utilisant la clé *53* et donc, il est possible de relier le "goto" au "label".

3.2 Algorithme itératif pour l'annotation (extraction de l'information)

L'analyse permettant l'extraction de l'information de ce mémoire s'appuie sur un algorithme itératif qui va permettre d'extraire l'information concernant les ensembles de pointeurs possiblement nuls retournés par une fonction.

```
visit_label_stmt()
  Begin
    a ← new_cfg_node (id++, "LABEL");
    ∀n ∈ open_nodes,
      | link (n, a);
    open_nodes ← {a};
    labelHTable.insert(key = nodeNumber, Ref);
    visit_label_value();
    visit_next_stmt();
  End
```

Figure 3.14: Algorithme pour le "label"

Le principe de l'algorithme est de définir des ensembles au niveau de chaque bloc de base du CFG et d'effectuer des opérations sur ces ensembles de manière itérative. Cela signifie que les ensembles définis au niveau de chacun des blocs de base vont évoluer d'un état initial vers un état dans lequel les ensembles contiendront toujours les même éléments. Cette est basée sur le calcul d'un point fixe.

L'atteinte d'un point fixe sous certaines conditions a été mise en évidence par Patrick COUSOT [8]. Il est aussi possible que le calcul de point fixe n'aboutisse jamais à un état dans lequel les ensembles n'évoluent pas. Pour que les calculs se terminent, il faut alors avoir recours à des conditions d'arrêt comme par exemple le nombre d'itérations. Dans ce travail il a été constaté (et non démontré) que le point fixe était toujours atteint. Bien entendu, cette observation ne constitue en rien une preuve mathématique de l'atteinte d'un point fixe. Étant donné que cette démonstration dépasse le cadre de cette maîtrise le point fixe sera considéré comme étant toujours atteint. Si jamais tel n'était pas le cas, il serait alors nécessaire de

modifier cette recherche en ajoutant un critère d'arrêt à l'algorithme (ou bien prouver de façon mathématique que le point fixe est bien atteint). Il serait par exemple possible de fixer le nombre maximum d'itérations à 5. Si lors des tests on se rend compte que dans 70% des cas le nombre maximum d'itérations est atteint alors on augmenterait cette limite et on recommencerait l'expérience jusqu'à l'obtention d'une valeur jugée suffisante.

L'objectif de l'algorithme est de calculer l'ensemble des pointeurs pouvant être nuls. Ceci permettra de comparer les valeurs de retour des fonctions avec cet ensemble et, si le pointeur retourné par la fonction se trouve dans l'ensemble des pointeurs possiblement nul, nous pourrons alors annoter la fonction comme "retournant un pointeur possiblement nul".

L'algorithme itératif de la figure 3.15 repose sur le principe de calcul d'un point fixe. L'idée est de parcourir tous les nœuds du graphe, de comparer l'état de chacun des nœuds par rapport à son état précédent et, si l'un des nœuds a été modifié, recommencer le calcul. La comparaison utilisée est expliquée plus loin dans ce chapitre. L'une des améliorations possibles serait d'utiliser une structure afin de stocker les nœuds modifiés comme dans [25] afin de ne pas avoir à parcourir l'ensemble du graphe à chaque modification d'un bloc de base.

La définition précise des ensembles utilisés et des opérations permises sur ces ensembles par l'algorithme présenté en 3.15 vont maintenant être définis.

Il est utile de préciser dès maintenant que SPLint fournit, entre autres, un

```
algorithme_iteratif()
  Debut
    | Pour chaque bloc B
    | Faire
    |   | init_df();
    | Fait
    | change =VRAI;
    | Tant que change
    | Faire
    |   | change =FAUX;
    |   | Pour chaque bloc B
    |   | Faire
    |   |   | in_null[B] = ⊕_{p∈pred de B} out_null[p];
    |   |   | old_null_out = out_null[B];
    |   |   | out_null[B] = init_null[B] ⊖ in_null[B]
    |   |   | si out_null[B] ≠ old_null_out
    |   |   |   | change = VRAI;
    |   |   | fsi
    |   | Fait
    | Fait
  Fin
```

Figure 3.15: Algorithme itératif

fichier d'en-tête contenant des annotations pour les fonctions de la librairie C. Parmi toutes les annotations, cette analyse s'intéresse en particulier à l'une d'entre elles qui est */*@NULL@*/*. A travers les exemples donnés aux figures 4.10 et 4.11 il a été montré qu'il était possible de supprimer certains faux messages générés par SPLint. L'objectif maintenant est de propager l'information concernant les pointeurs pouvant être nuls à travers le CFG. Maintenant que l'objectif a été précisé, l'algorithme proposé va être détaillé.

Les trois ensembles utilisés dans l'algorithme sont initialisés de la façon suivante:

- L'ensemble *init_null* va contenir les informations concernant les pointeurs

présents dans le bloc de base et particulièrement la manière dont ils sont initialisés. Cette initialisation se fait par l'intermédiaire de la fonction *init_df*() (voir l'algorithme 3.15). Cette fonction, commence par tester si une opération de modification (par exemple $ptr = \&i$) est présente dans le bloc de base courant. Si c'est le cas, il faut alors vérifier que l'opérateur situé à gauche de la modification est bien un pointeur (cette information a été extraite lors de la visite du CFG par le constructeur). S'il s'agit effectivement d'un pointeur, il y a alors trois cas possibles. Soit une valeur retournée par une fonction est assignée au pointeur et cette valeur peut être nulle. Pour le savoir, une base de données contenant les annotations des fonctions de la librairie C (celles fournies par SPLint) est utilisée. Cette librairie est accessible en utilisant une table de dispersement dont la clé est le nom de la fonction. Par exemple, soit le code :

```
ptr = (int *) malloc(sizeof(int));
```

Dans ce cas, le résultat d'un appel de fonction est utilisé par un pointeur. Il faut regarder dans la table contenant la librairie en utilisant la clé "malloc". Les informations retournées par la librairie dans le cas d'un "malloc" sont :

```
extern /*@null@*/ /*@out@*/ /*@only@*/ void *malloc (size_t size);
```

Si la fonction n'est pas dans la librairie, on ne peut rien déduire quant à la valeur du pointeur. Pour *malloc*, on sait que le pointeur retourné par

"malloc()" peut être nul. Dans ce cas il faut ajouter à l'ensemble *init_null* le couple formé par le numéro du pointeur dans l'AST et un numéro @0 qui représente la possibilité d'être null ce qui donne ($@X \leftarrow @0$) indiquant que le bloc de base courant génère l'information : le pointeur ayant pour numéro X dans l'AST peut être nul. Ce type d'annotation est également généré si nous avons directement l'affectation $ptr = NULL$.

Le second cas arrive si un autre pointeur est assigné au pointeur. Il est alors nécessaire d'ajouter à l'ensemble *init_null* l'information indiquant que le pointeur ainsi assigné va propager l'information du pointeur situé à droite de l'affectation. Ce couple est noté ($@X \leftarrow @Y$). Cela signifie que le pointeur ayant pour numéro X dans l'AST va propager l'information du pointeur ayant pour numéro Y dans l'AST.

Enfin le dernier cas est l'assignation définitivement non nulle. Par exemple, $ptr = \&i$. Dans ce cas, on sait que le pointeur est non nul et le couple ($@X \leftarrow @ - 1$), que nous ajoutons dans l'ensemble *init_null*, permet de savoir que $@X$ est non nul.

- *in_null* est initialement vide. Il va contenir les valeurs propagées par les blocs de base précédents.

- *out_Annot* est initialisé avec les couples contenus dans *init_annot*.

Les phases d'initialisation de l'algorithme viennent d'être présentées. Discutons

maintenant de l'algorithme proprement dit et notamment des deux opérations qu'il est possible d'effectuer sur les ensembles.

La table 5.1 montre un exemple du résultat de l'algorithme sur le CFG de la figure 3.17. Il se termine à la seconde itération car aucun des ensembles *out_null* n'a été modifié. L'information déduite de l'analyse est que le pointeur @91 est possiblement nul (il s'agit de *ptr*) et que le pointeur @103 (c'est à dire *ptr*2) propage les informations de @91. Regardons les opérations qui ont été effectuées.

3.2.1 L'opérateur $\oplus$

in_null correspond aux informations (dans notre cas les pointeurs possiblement nuls) données par les prédécesseurs d'un bloc de base. Lors de la première itération de l'algorithme, *in_null* est modifié en fonction des sorties produites par les prédécesseurs du bloc de base courant. L'opération $\oplus$ consiste à comparer les pointeurs présents dans l'ensemble *out_null* des prédécesseurs avec ceux déjà présents dans l'ensemble *in_null*. S'ils ne sont pas déjà présents dans *in_null* alors ils sont ajoutés à l'ensemble en faisant une union classique. Sinon, on remplace les couples présents dans *in_null* selon l'un trois des principes suivants:

L'exemple suivant montre ce que cela signifie exactement. Soit les hypothèses suivantes :

- Soit un bloc B3 ayant deux prédécesseurs B1 et B2

- (1) $(@X \leftarrow @0) \oplus (@X \leftarrow *) \longrightarrow (@X \leftarrow @0)$
- (2) $(@X \leftarrow @Y) \oplus (@X \leftarrow -1) \longrightarrow (@X \leftarrow @Y)$
- (3) $(@X \leftarrow @Y) \oplus (@X \leftarrow @Z) \longrightarrow (@X \leftarrow @Y), (@X \leftarrow @Z)$
  Si $Y \neq 0$ et $Y \neq -1$ et $Z \neq 0$ et $Z \neq -1$
  sinon nous sommes dans le cas 1 ou 2

Figure 3.16: Règles de l'opérateur $\oplus$

- Soit $out\_null[B1] = \{ (@12 \leftarrow @0),\ (@13 \leftarrow @4),\ (@15 \leftarrow @6) \}$
- Soit $out\_null[B2] = \{ (@12 \leftarrow @8),\ (@14 \leftarrow @ - 1),\ (@15 \leftarrow @9) \}$
- soit $in\_null[B3] = \{ \}$

La première étape va consister à ajouter les éléments présents dans l'ensemble $out\_null[B1]$ (c'est à dire les couples $@X \leftarrow @Y$) dans l'ensemble $in\_null[B3]$. On obtient alors :

$in\_null[B3] = \{ (@12 \leftarrow @0),\ (@13 \leftarrow @4),\ (@15 \leftarrow @6) \}$

Maintenant il faut considérer l'ensemble $out\_null[B2]$. Le premier élément est le couple $(@12 \leftarrow @8)$. Or il existe déjà un couple dans l'ensemble $in\_null[B3]$ dont l'élément de gauche est @12. Si on regarde les règles présentées à la figure 3.16, en prenant le règle numéro 1, on obtient :

$(@12 \leftarrow @0) \oplus (@12 \leftarrow @8) \longrightarrow (@12 \leftarrow @0)$

et donc, le nouvel ensemble

$in\_null[B3] = \{ (@12 \leftarrow @0),\ (@13 \leftarrow @4),\ (@15 \leftarrow @6) \}$

Il se trouve que dans ce cas l'ensemble n'a pas été modifié la suite de l'exemple

montre que ce n'est pas toujours le cas. Il faut maintenant continuer l'opération. Le second élément de $out\_null[B2]$ est le couple $(@14 \leftarrow @-1)$. L'élément gauche du couple ne fait pas partie de l'ensemble $in\_null[B3]$. En appliquant la troisième règle de l'opérateur $\oplus$ qui consiste à ajouter l'élément à l'ensemble, on obtient alors :

$$in\_null[B3] = \{\ (@12 \leftarrow @0),\ (@13 \leftarrow @4),\ (@14 \leftarrow @-1),\ (@15 \leftarrow @6)\ \}$$

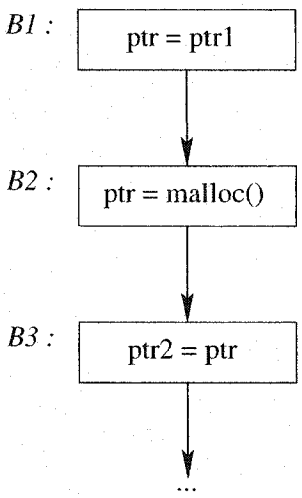
De la même façon, le dernier élément appartenant à l'ensemble $out\_null[B2]$ va être ajouté à $in\_null[B3]$ en utilisant la règle numéro 3. En effet, il existe déjà un couple $(@15 \leftarrow @X)$ dans $in\_null[B3]$ avec $@X \neq @0$ et $@X \neq @-1$. Finalement l'ensemble devient:

$$in\_null[B3] = \{\ (@12 \leftarrow @0),\ (@13 \leftarrow @4),\ (@14 \leftarrow @-1),$$
$$(@15 \leftarrow @6),\ (@15 \leftarrow @9)\ \}$$

Ce résultat est interprété de la façon suivante. Le pointeur ayant le numéro @12 est possiblement nul, le pointeur @13 est possiblement nul si le pointeur @4 est possiblement nul, le pointeur @14 n'est définitivement pas nul et le pointeur @15 est possiblement nul si soit le pointeur @6 est nul, soit le pointeur @9 est nul.

3.2.2 L'opérateur $\ominus$

Avant de mettre à jour l'ensemble *out_null*, il faut en faire une copie. Cette copie est utilisée dans la condition d'arrêt de l'algorithme. Il se termine lorsque les ensembles *out_null* ne subissent plus de modifications. Il aurait été possible de



A titre d'exemple voici les valeurs associés aux pointeurs retournées par notre constructeur :

[Node 91] source: prog.c:6 | name: ptr | type: @11 | used: 1
[Node 102] source prog.c:7 | name: ptr1 | type: @11 | used 1
[Node 103] source prog.c:8 | name: ptr2 | type: @11 | used 1

Nous donnons ces valeurs car nous les utilisons pour montrer le fonctionnement de notre algorithme.

Figure 3.17: Exemple de CFG

choisir l'ensemble *in_null* comme condition d'arrêt.

out_null est le résultat de l'opération $\ominus$ entre *init_null* et *in_null* précédemment calculé. L'opération $\ominus$ agit comme un filtre. Soit l'élément est présent à la fois dans *init_null* et *in_null* auquel cas on prend celui présent dans *init_null* et sinon, si l'élément n'est présent que dans un seul ensemble alors il est ajouté à *out_null*.

Étapes	Ensemble	B1	B2	B3
0	init_null	(@91 ← @102)	(@91 ← @0)	(@103 ← @91)
	in_null			
	out_null	(@91 ← @102)	(@91 ← @0)	(@103 ← @91)
1	init_null	(@91 ← @102)	(@91 ← @0)	(@103 ← @91)
	in_null		(@91 ← @102)	(@91 ← @0)
	out_null	(@91 ← @102)	(@91 ← @0)	(@91 ← @0) (@103 ← @91)
2	init_null	(@91 ← @102)	(@91 ← @0)	(@103 ← @91)
	in_null		(@91 ← @102)	(@91 ← @0)
	out_null	(@91 ← @102)	(@91 ← @0)	(@91 ← @0) (@103 ← @91)

Remarque: $out\_null[Etape2] = out\_null[Etape1] \Rightarrow stop$

Tableau 3.1: Les différentes étapes de l'algorithme

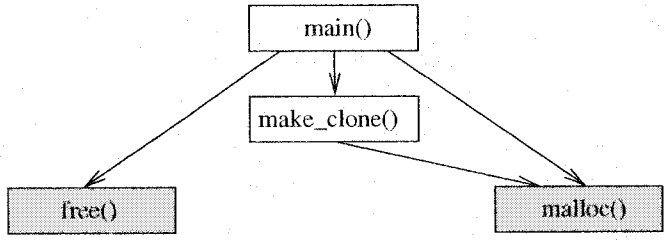


Figure 3.18: Graphe d'appel

3.3 Algorithme d'annotation du graphe d'appel

L'objectif de cette analyse est d'annoter les fonctions non pas dans l'ordre de leur déclaration mais par rapport à leur dépendance envers les fonctions déjà annotées. En regardant les résultats concernant les informations récoltées lors de l'analyse du programme *clone.c* à la figure I.3 de l'annexe I page 126, on remarque que la fonction *main* appelle trois autres fonctions qui sont *make_clone, free* et *malloc*. Ceci conduit au graphe de la figure 3.18. Les parties grisées correspondent aux fonctions annotées fournies par SPLint qui correspondent en général aux fonctions de la librairie C.

Toujours selon ces informations, *free* et *malloc* sont déjà annotées alors que *make_clone* ne l'est pas. Maintenant, en regardant la fonction *make_clone*, elle n'appelle que la fonction *malloc*. Donc, en considérant le graphe d'appel, tous les fils de *make_clone* sont annotés et, donc, cette fonction peut à son tour être annotée. En fait, le problème est équivalent à annoter les feuilles d'abord, ce qui revient à parcourir l'arbre en profondeur. Il est par exemple possible d'utiliser

l'algorithme suivant :

```
/*******************************************************
Notations: un sommet peut avoir trois états qui sont :
BLANC: Non visité et pas en attente de visite
GRIS : Non visité et en attente de visite
NOIR : Visité
*******************************************************/

Annotation_en_profondeur()
  Debut
    Pour tous les nœuds n du graphe
    Faire
      n.couleur ← BLANC;
    Fait
    Visit(Graphe, Racine);
  Fin

Visit(g, n)
  Debut
    n.couleur ← GRIS;
    Pour les voisins v de n
    Faire
      si n.couleur == BLANC
        alors Visit(g, v);
    Fait
    Si v est une feuille ne contenant pas une fonction deja annoter
      alors Annoter(v);
    v.couleur ← NOIR;
  Fin
```

C'est une solution qui est correcte mais qui n'est pas efficace. Il y a un problème si le graphe n'est pas un arbre et contient un cycle. Dans ce cas, une solution facile serait d'annoter sans tenir compte de toutes les dépendances.

3.4 Discussion

Une méthodologie permettant d'annoter les fonctions retournant un pointeur possiblement nul a été mise en place. Cette approche permet de réduire le nombre de faux messages générés par SPLint ce qui constituait un défaut pour cet outil. Le prochain chapitre présente une implémentation, à travers l'outil GASTA (GCC Abstract Syntax Tree Analysis), de cette théorie.

CHAPITRE 4

DÉTAILS DE L'IMPLÉMENTATION

L'outil développé durant cette maîtrise s'appelle GASTA (GCC Abstract Syntax Tree Analysis). Il utilise un graphe de syntaxe abstrait (ASG) qui est une représentation hiérarchique du code. Différents outils permettent de générer un ASG. C'est le compilateur GNU C (GCC) [16] qui a été choisi. Son avantage est qu'il est disponible gratuitement avec ses sources. Il est donc possible de modifier le code source du compilateur si cela devient nécessaire. De plus, ce compilateur est largement utilisé dans le monde UNIX.

GASTA est donc développé en utilisant comme fichier d'entrée la sortie de GCC tel que le montre la figure 4.1. Il se divise en quatre parties. Tout d'abord le programme d'analyse lexicale et syntaxique qui lit le fichier contenant l'ASG et construit le graphe en mémoire. Ensuite, le visiteur permet le parcours du graphe. Durant ce parcours, un graphe de flot de contrôle (CFG) est généré. La troisième partie est le constructeur qui permet la récupération de données durant le parcours de l'ASG. Il est appelé par le visiteur. La boîte en pointillé englobant ces deux "objets" souligne leur interaction. Enfin, la dernière partie est l'analyseur qui va utiliser les valeurs fournies par le constructeur ainsi que le graphe de flot de contrôle (CFG) produit durant la visite afin de générer un ensemble constitué

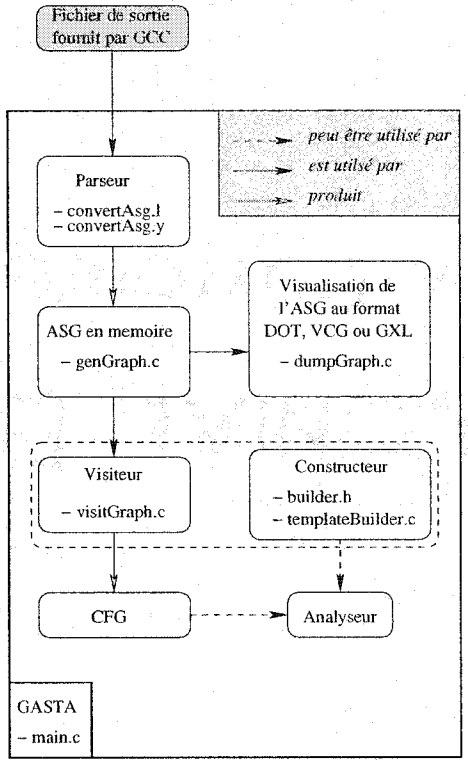


Figure 4.1: L'architecture de GASTA

de couples formés des pointeurs et de la zone mémoire qu'ils référencent. Cette notion de couple est présentée plus loin dans ce chapitre.

La première section présente l'architecture générale de GASTA tandis que la section 4.2 voit plus en détails l'implémentation des algorithmes proposés dans le chapitre précédent. Finalement, la section 4.4 est une discussions des améliorations à apporter.

4.1 L'architecture de GASTA

L'objectif de ce travail est l'extraction des propriétés du code source pour permettre l'annotation automatique de celui-ci. Tout en respectant cet objectif GASTA est construit avec le souci d'offrir un environnement de travail permettant d'exploiter le graphe de syntaxe abstrait fourni par GCC (même si cela ne constitue pas l'objectif premier). Cela signifie que GASTA doit permettre une intégration facile de n'importe quelle analyse basée sur cet ASG. Le squelette d'un constructeur définissant l'interface avec le visiteur est donc inclus dans l'outil.

4.1.1 L'analyseur lexical et syntaxique

Il s'agit d'un programme dont le rôle est de parcourir un fichier afin de lui faire subir un traitement. Il se divise en deux parties : l'analyse lexicale et l'analyse syntaxique. Ces deux parties ont été écrites en utilisant respectivement les outils FLEX et BISON [24] et elle sont décrites dans les sous-sections suivantes.

4.1.1.1 L'analyseur lexical

Le rôle de l'analyse lexicale est de fournir une suite de "tokens" (les éléments terminaux du langage) à partir d'une suite de caractères fournis en entrée. Par exemple, dans la déclaration suivante, x := 2*y + 3, l'analyseur lexical va identifier les caractères par :

- Un identificateur x
- Un symbole d'assignation :=
- Un nombre 2
- Le signe *
- Un identificateur y
- Le signe +
- Un nombre 3

Flex va lire des données afin de convertir les caractères en "tokens" compréhensibles par l'analyseur syntaxique. Pour permettre à Flex d'identifier des patrons de caractères, on utilise les expressions régulières. À chaque patron est associé une action (en général, il s'agit de retourner le "token" correspondant au patron). Par exemple, supposons qu'un identificateur soit de la forme : $lettre(lettre|chiffre)*$ c'est à dire une lettre suivie d'une lettre ou d'un chiffre zéro ou plusieurs fois (x, x1, xx, etc...). Il est alors possible de représenter ce patron par un automate à états finis (FSA : finite state automaton) comme le montre la figure 4.2:

L'état "Fin" représente un état validant notre identificateur. Une manière plus algorithmique, serait par exemple d'écrire :

```
Début  : aller à état 0
état 0 : lire c
         si c est un chiffre
```

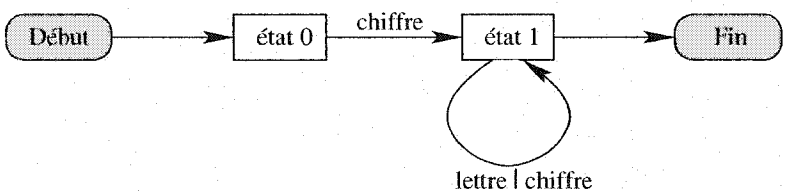


Figure 4.2: automate

```
            alors aller à état 1
            sinon rester à état 0
état 1 : lire c
            si c = lettre
            alors rester à état 1
            sinon si c = chiffre
                  alors rester à état 1
                  sinon aller à Fin
Fin    : accepter l'identificateur
```

C'est la technique utilisée par Flex. Les expressions régulières sont représentées sous forme d'un automate. Dans GASTA, FLEX lit les données contenues dans le fichier de sortie de GCC (figure 4.3). Les patrons reconnus sont les suivants (en prenant comme exemple la sixième ligne de la figure 4.3):

- **@X** est un *NUMBER* (le numéro du nœud)
- **function_decl** est un *IDENT* (identifie le nœud)
- **name: @4** est une étiquette constituée de deux parties, *TAGLABEL : TAG-VALUE*

```
@1  namespace_decl      name: @2
                        srcp: <internal>:0
                        dcls: @3
@2  identifier_node     strg: ::
                        lngt: 2
@3  function_decl       name: @4
                        type: @5
                        srcp: simple4.c:3
                        chan: @6
                        C
                        extern
                        body: @7
@4  ...
```

Figure 4.3: Sortie produite par GCC

En se référant aux précédentes définitions, un nœud du graphe syntaxique abstrait est l'association de trois objets qui sont: son numéro, son identificateur et une suite d'étiquettes.

4.1.1.2 L'analyseur syntaxique

Pour spécifier la syntaxe d'un langage, il est possible d'utiliser ce que l'on appelle les grammaires libres de contexte (context-free) ou BNF (forme de Backus-Norm). Une grammaire libre de contexte possède quatre composantes qui sont :

- Un ensemble de "tokens" ou terminaux du langage,
- Un ensemble de non-terminaux,
- Un ensemble de règles de productions où chacune des règles consiste en un non-terminal à gauche de la production (left-side), une flèche et une séquence de jetons et/ou de non-terminaux à droite de la production (right-side),

- Un non-terminal de départ.

Voici un exemple de grammaire libre de contexte :

```
S -> E

E -> aFa | aEa

F -> b
```

Dans cette grammaire, l'ensemble de jetons est $\{a, b\}$, l'ensemble des non-terminaux est $\{S, E, F\}$, trois règles de production sont utilisées et le non-terminal de départ est S. Le symbole "|" est équivalent à "ou bien". Cette grammaire représente l'ensemble des phrases aba, aabaa, aaabaaa, ...

Les grammaires pour Bison sont décrites en utilisant cette forme. La plupart des langages modernes peuvent être décrits sous cette forme.

La grammaire utilisée dans GASTA est très simple :

```
asg       <- asg node
             |  ;
node      <- NUMBER IDENT tags
             | FUNCTION ;
tags      <- doublet tags
             | IDENT tags
             | IDENT TAGVALUE NUMBER tags
             | ;
doublet   <- TAGLABEL tag_value ;
tag_value <- NUMBER
             | TAGVALUE
             | IDENT
             | ;
```

Soit les règles suivantes:

r1 : S –> E
r2 : E –> aFa | aEa
r3 : F –> b

Soit le mot suivant: aabaa

Parser Top–Down			*Parser Bottom–Up*	
	(1)	S	aabaa	r3
r1	(2)	S –> E	aaFaa	r2
r2	(3)	S –> E –> aEa	aEa	r2
r2	(4)	S –> E –> aEa –> aaEaa	E	r1
r3	(5)	S –> E –> aEa –> aaEaa –> aabaa	S	

Figure 4.4: bottom-up VS top-down

Le rôle de l'analyseur syntaxique est de stocker les nœuds en mémoire. La construction est triviale. Bison est un générateur d'analyseur syntaxique ascendant. C'est à dire que plutôt que de partir d'un non-terminal et générer une expression à partir de la grammaire, il faut partir d'une expression et la réduire à un non-terminal. Cette technique s'appelle un "parsage bottom-up" ou "shift-reduce", et elle utilise des piles pour stocker les termes. Les différences avec les techniques "top-down" sont montrées sur la figure 4.4.

La suite d'étiquettes est construite au fur et à mesure en les ajoutant dans une liste chaînée simple. Lorsqu'on rencontre son identificateur, celui-ci est placé en début de liste (avec les étiquettes), et enfin on utilise le numéro $@X$ comme clé d'entrée dans une table de dispersement. Cette entrée pointe vers la liste associée

au nœud.

Il faut noter qu'il existe des conflits dans la grammaire utilisée. Par exemple, on a choisi de décaler le "lookahead" après l'opérateur '*' alors qu'il y aurait pu avoir une réduction de "E + E" en utilisant la règle r1. Il y a donc un conflit "shift-reduce". Ceci arrive car la grammaire est ambiguë, c'est à dire que pour une expression donnée il existe plusieurs arbres de syntaxe abstraits. Bison résoud ces problèmes en adoptant une action par défaut. Dans le cas d'un conflit "shift-reduce", Bison va choisir le "shift". Dans le cas d'un conflit "reduce-reduce", Bison va choisir la première règle dans la grammaire. En cas de conflits, Bison génère des avertissements sur les règles problématiques. Il est aussi possible de lui spécifier un comportement. Dans le cas de la grammaire utilisée par GASTA pour le parcours du fichier produit par GCC, les actions prises par défaut sont correctes.

4.1.2 Le visiteur

Son rôle est double. Il va parcourir l'ASG en mémoire afin de réaliser deux opérations. La première va consister en construire un CFG en utilisant les algorithmes présentés dans le chapitre précédant et la seconde va être l'appel au constructeur pour chaque nouveau nœud rencontré. Il aurait été possible de construire le CFG directement à partir du fichier généré par GCC en modifiant l'analyseur lexical et syntaxique. Si on a choisi de le faire de cette façon, c'est dans un souci de lisibilité. En effet, le fait de construire l'ASG en mémoire va permettre de mettre

en place des étapes de déverminage.

Le visiteur est spécifique au graphe car le chemin est lié au type du nœud. Ceci implique qu'il n'est pas possible d'utiliser les algorithmes classiques de parcours de graphe (parcours en largeur, parcours en profondeur). Comme cela a été décrit dans la section précédente, un nœud possède une liste d'étiquettes. Par exemple, en considérant le nœud correspondant à la sixième ligne de la figure 4.3, on voit qu'il correspond à une fonction. Les étiquettes associées à ce nœud sont *name:@4, type:@5, srcp:simple4.c:3, chan:@6, C, extern* et *body:@7.* Ce qui est intéressant dans le cas d'une fonction, c'est de récupérer les informations concernant cette fonction avant d'en parcourir le corps pour finalement visiter les autres fonctions du programme. Donc, la dernière étiquette qui sera visitée est *chan:@6.* Cet exemple montre qu'il n'est pas possible d'utiliser un parcours en profondeur ou en largeur du graphe de flot de contrôle puisque l'ordre de visite des fils n'est pas le même selon le type du nœud à visiter.

4.1.3 Le constructeur

Comme il en a déjà été fait mention, le constructeur a été implémenté de façon générique. Cela signifie que si un programmeur souhaite intégrer son constructeur dans GASTA, il lui suffit de copier le fichier *templateBuilder.c* vers *monConstructeur.c* (ainsi que les fichiers d'en-tête *templateBuilder.h* vers *monConstructeur.h*) et de remplacer toutes les occurrences de "templateBuilder" par "monConstruc-

teur" et compléter le corps des fonctions. Ces fichiers contiennent un constructeur, un destructeur ainsi que toutes les fonctions appelées par le visiteur. Donc, à partir de ce moment, il ne reste plus qu'à modifier le programme principal *main()* afin d'ajouter le nouveau constructeur aux paramètres d'appel du visiteur. Ainsi, toutes les fonctions définies dans ce nouveau constructeur seront automatiquement appelées par le visiteur.

Le constructeur va permettre de récupérer les informations nécessaire à l'analyse. Il est appelé à chaque fois que l'on rencontre un nouveau nœud dans le CFG. En fonction du type de nœud, l'information sera conservée ou ignorée. Par exemple, si on est en train de visiter un pointeur, il pourra être intéressant de conserver les informations concernant son type, l'endroit de sa déclaration dans le code, vers quel type d'objet il pointe, etc...

Pour l'analyse faite dans GASTA, le constructeur doit récupérer les informations concernant les pointeurs. Par exemple, l'endroit où est déclaré le pointeur (quelle ligne de quel fichier) et vers quoi il pointe (un entier, un caractère, ...). La figure 4.5 montre le genre d'information récupéré par GASTA.

L'exemple de la figure 4.5 indique que dans la fonction *add_end()* il y a une variable locale à la fonction, que cette variable est un pointeur, que son nom est *next* et elle est déclarée dans le fichier *for.c* à la ligne 6. Il est également connu que ce pointeur n'est ni initialisé ni utilisé. Enfin il y a aussi l'information indiquant que le type référencé par ce pointeur est un *record_type*.

```
Information about type
----------------------
;; Function void add_end(mylist*) (_Z7add_endP6mylist)
[Node 11]       name=pointer_type       ptd=@17
...
[Node 17]       name=record_type
...

Information about variable
--------------------------
;; Function void add_end(mylist*) (_Z7add_endP6mylist)
...
[Node 91]  source: for.c:6 | name: next | type: @11 | used: 0
        -> local variable
        -> not initialized
...
```

Figure 4.5: Informations sur un pointeur

La figure 4.6 montre les détails de l'implémentation. Une partie de la structure est commune à tous les constructeurs alors que certains champs permettent de le personnaliser. Par exemple, le champ *attributes* est un pointeur générique qui est utilisé pour stocker les valeurs particulières d'une analyse (voir par exemple la figure 4.7 qui montre le constructeur asciiBuilder dont le rôle est l'affichage dans le format ASCII des différentes étapes de la visite).

La structure *builder* contient aussi un champ qui est un pointeur de fonction qui permet de lancer un constructeur particulier. L'interaction entre les constructeurs et le programme principal se fait par l'intermédiaire de deux appels de fonctions *template_builder_init()* et *template_builder_destroy()*. C'est dans le premier appel que les champs *attributes* et *start_build* sont initialisés (figure 4.8).

Le rôle du constructeur est la récupération des données dans le but de fournir toutes les informations nécessaires à l'analyse. La partie suivante va expliquer de

```
typedef struct {
   int builder_status;
   int node_number;
   int depth;
   stack_s *rvalue_stack;
   stack_s *lvalue_stack;
   const char*node_name;
   GSList *current_node_tag;
} builder_s;

typedef struct {
   GString *name;
   void *attributes;
} build_param_s;

typedef struct {
   void (*start_build) (build_param_s *, const char *);
   GHashTable *match_string_to_builder;
   build_param_s param;
} builder;

/* Default builder (define in defaultBuilder.c)*/
void default_builder_init(builder * b);
```

Figure 4.6: builder.h

```
typedef struct {
   FILE *file;
   char *filename;
} ascii_builder_params;

void ascii_builder_init (builder *b, char *filemame);
void ascii_builder_destroy (builder *b);
```

Figure 4.7: asciiBuilder.h

```
template_builder_init (builder *b) {

  templateb_param_s *tbp = NULL;

  /* We initialize the function start_build */
  b → start_build = (void *) start_template_build;

  /* We initialize the convertToBuilderTable */
  b → match_string_t_builder = g_hash_table_new (g_str_hash, g_str_equal);
  if (!b → match_string_t_builder)
  {
    fprintf (stderr, "Error: init convertToBuilderTable");
    exit (0);
  }

  /* We initialize the methods */
  default_builder_init(b);

  /* Declarations, expressions, types
    storages , statements, others */
  g_hash_table_insert (b → match_string_t_builder, "function_decl", (void *) &template_function_decl);
  ...


  /* Now, we initialize the attributes */
  if (!(b → param.name = g_string_new("Template Builder"))) {
    exit(EXIT_FAILURE);
}

  if (!(b → param.attributes = calloc (1, sizeof (templateb_param_s)))) {
    fprintf(stderr, "Can't allocate memory to b → parm.attributes");
    exit(EXIT_FAILURE);
}

  tbp = (templateb_param_s *) b→parm.attributes;

  /*
   * Initialize your attributes here using :
   * tbp → ... =...
   */

 }
```

Figure 4.8: templateBuilderInit.c

quelle façon l'analyse, pour l'annotation automatique du code source, est faite.

4.2 L'analyse

L'analyseur peut utiliser l'information collectée par le constructeur et/ou le visiteur afin de produire ses résultats. C'est pourquoi les flèches indiquant des liens avec l'analyseur sur la figure 4.1 sont en pointillés.

Pour annoter automatiquement le code source, on utilise à la fois l'information produite par le visiteur (c'est à dire le CFG) et l'information fournie par le constructeur *annotBuilder.c*. Ces informations sont généralement stockées dans une table de dispersement. Ces tables sont accédées en utilisant le numéro du nœud duquel l'information doit être obtenue. Pour l'annotation automatique du code source, quatre tables de dispersement par fonction sont utilisées. Ces tables contiennent des données concernant les fonctions appelées, les paramètres de la fonction, les variables locales et globales ainsi que les types. Il est intéressant d'avoir ce genre de renseignements sous la forme de tables de dispersement car les nœuds de l'ASG fournis par GCC sont de la forme : "@32 var_decl name: @34 type: @6".

Lorsque l'on dispose des tables, la complexité pour connaître le nom et le type de la variable est de l'ordre de un (c'est à dire que la complexité est indépendante de la taille du graphe). La figure I.3 en annexe est un affichage de ce que contiennent les tables de dispersement.

Le code de la figure 4.10 contient des erreurs qui vont être détectées par SPLint.

```
sample.c:8:10: Possibly null storage ptrTemp returned as non-null: ptrTemp
Function returns a possibly null pointer, but is not declared using /*@null@*/
annotation of result.  If function may return NULL, add /*@null@*/ annotation
to the return value declaration. (-nullret will suppress message)
sample.c:7:13: Storage ptrTemp may become null.
```

Figure 4.9: Exemple d'avertissement généré par SPLint

Il va produire certains avertissements comme celui de la figure 4.9. Ce type de message peut-être supprimé en utilisant les annotations (figure 4.11).

Ces annotations doivent être placées dans le code par le programmeur. L'objectif de cette maîtrise est de fournir une méthode et de proposer un outil permettant, dès que c'est possible, d'annoter de façon automatique le code. Un tel outil apporterait deux avantages. Le premier serait de réduire le nombre de fausses alarmes générées par SPLint facilitant ainsi le déverminage du code. Le second serait de fournir une aide à la compréhension du code. Par exemple, sans avoir à lire le code de *mymalloc()*, le programmeur, en regardant les annotations, pourrait savoir que le pointeur retourné par cette fonction peut être nul et incomplet[1].

Comme cela a été présenté dans la partie présentant les algorithmes, le travail est divisé en trois parties. D'un point de vue pratique, les trois étapes sont les suivantes: Premièrement, on doit extraire le CFG en utilisant le visiteur. Dans GASTA, un graphe est produit pour chaque fonction. Deuxièmement, il faut mettre en place l'algorithme itératif (figure 3.15) permettant la propagation des annotations à travers le CFG. L'algorithme utilisé est un algorithme itératif pour

[1]Cela signifie que la référence mémoire est possiblement non initialisée

l'annotation inspiré par celui que l'on trouve dans *Compilers Principles, Techniques, and Tools* [1] et connu sous le nom de "iterative algoritm for reaching definitions". Il a été décrit dans la section 3.2. Enfin, on étendra l'analyse d'une fonction à un ensemble de fonctions en tenant compte du fait qu'une fonction peut être annotée si elle est une feuille du graphe d'appel de fonctions ou, si tous ces fils ont déjà été annotés. Un mécanisme devra être mis en place dans le cas des appels récursifs.

```
#include <stdlib.h>

int *
mymalloc (int size)
{
  int *ptrTemp;

  ptrTemp = (int *) malloc ((size_t) size);
  return ptrTemp;
}
```

Figure 4.10: sample.c sans annotations

```
#include <stdlib.h>

/*@out@*//*@null@*/ int *
mymalloc (int size)
{
  int *ptrTemp;

  ptrTemp = (int *) malloc ((size_t) size);
  return ptrTemp;
}
```

Figure 4.11: sample.c avec annotations

4.2.1 Passage de l'ASG au CFG

Deux structures sont utilisées dans ces algorithmes. Tout d'abord les listes comme *open_nodes* ou *ThenOpenNodeList* sont des listes simplement chaînées. Ce sont les implémentations fournies par la librairie *GLib* de *GNOME* [12] qui sont utilisées. Pour la gestion des chaînes de caractères, c'est aussi l'implémentation de la *GLib* qui est utilisée. La liste *open_nodes* contient des éléments possédant plusieurs champs décrits dans le fichier *visitGraph.h*. Le code suivant montre les différents champs.

```
typedef struct cfg_node_s {

    int id;                     /* The current number of the block for the CFG */
    GString *name;
    int node_number;            /* The number of the node which is currently visited (= the key for hash table) */

    GSList *succ;
    GSList *pred;
    int visited;                /*
                                 * this information is passed to the builder to indicate that the node was
                                 * ever visited
                                 */

    /*  Used when we have two paths TRUE or FALSE
     *  For example, a IF node can have this pointer set to
     *  the node matches with the true path */
    struct cfg_node_s *true_path;
    struct cfg_node_s *false_path;
```

```
    /* Used with modify expression */
    GString *op0;
    GString *op1;

    /* Used with function call */
    GString *func;
    GSList *args;

    /*
     *  This parameter is used when you want to add a special parameter during the analysis. In our
     * case, we use it to add genAnnot, inAnnot and outAnnot. This parameter disable the possibility
     * of parallel analysis.  If you use it and make allocation inside the
     * generic parameter, you must release them.
     */
    void *generic;

} cfg_node_s;
```

Actuellement toutes ces informations ne sont pas utilisées. Certaines ont été placées pour de futurs développements. Par exemple, les pointeurs sur les chemins vrais ou faux qui apparaissent dans des structures de contrôle telles que *if, while* ou autres.

Les champs $op0$ et $op1$ sont utilisés comme référence sur les opérateurs gauche et droite que l'on rencontre dans les expressions.

Une particularité de l'implémentation est dans le traitement des appels de fonction au niveau du CFG.

4.3 Modification de GCC

Une modification au code source de GCC a été nécessaire pour faire fonctionner GASTA. Le problème rencontré est le suivant. Soit le code suivant:

```
fprintf(stderr, "error strg: BUG FOUND");
```

L'ASG produit par gcc va contenir une ligne de la forme:

```
@247    string_cst      type: @268    strg: error strg: BUG FOUND  lngt: 22
```

Le problème ici est qu'il est impossible de savoir que la chaîne de caractère est "error strg: BUG FOUND". En effet il pourrait y avoir deux chaînes "error" et "BUG FOUND". Cela complique singulièrement la parcours de l'ASG car on pourrait imaginer des cas plus problématique. Pour résoudre ce problème sans modifier l'analyseur syntaxique de GASTA, il suffit d'appliquer une rustine à GCC qui permet de générer une ligne de la forme:

```
@247    string_cst      type: @268    strg: "error strg: BUG FOUND"  lngt: 22
```

Cette rustine est distribuée avec GASTA et se trouve à l'adresse http://gasta.sf.net

4.4 Discussion

L'objectif de ce travail était de proposer un outil permettant d'annoter automatiquement le code source dans le but de réduire le nombre de fausses alarmes produites par SPLint. L'analyseur lexical et syntaxique, ainsi que le visiteur, fonctionnent correctement. La génération du CFG durant la visite, et les informations recueillies par le constructeur, sont corrects. Il est à noter que GASTA ne traite pas les expressions complexes dans les conditions des structures de contrôle. Ceci constitue un ajout futur.

La partie analyse proprement dite comporte quelques limitations liées à l'état actuel des travaux. Par exemple, une contrainte est causée par le graphe syntaxique abstrait généré par GCC qui représente les structures utilisées en C de façon incomplète. Ce problème devrait être résolu dans les futures versions du compilateur.

L'algorithme utilisé pour la génération des annotations fonctionne. Il doit être amélioré afin de prendre en compte un plus grand nombre de cas car, cela a été mentionné dans ce mémoire, son rôle est pour le moment limité à la propagation des pointeurs pouvant être nuls. Une autre étape de ce travail sera l'intégration de l'algorithme itératif dans un graphe d'appel. Il serait aussi intéressant que GASTA indique dans un fichier de "logs" toutes les modifications apportées au code source ainsi que la raison de cette modification.

Pour conclure, il est encore trop tôt pour juger de l'amélioration apportée par GASTA dans l'analyse produite par SPLint. Par contre, il existe maintenant une base intéressante de travail et d'analyse de l'ASG fournie par GCC. Cette base est réutilisable pour d'autres analyses que celle présentée dans ce mémoire.

CHAPITRE 5

RÉSULTATS

Les tests effectués sur GASTA ont pour objectif la validation de l'algorithme itératif. Les opérations $\oplus$ et $\ominus$ décrites respectivement dans les sections 3.2.1 et 3.2.2 vont donc être testées. Les tests vérifieront également la bonne propagation de l'information tout au long du CFG. Étant donné que le nombre de cas possible se limite à une dizaine, il est possible de tous les tester.

Afin de pouvoir tester les différents codes avec GASTA, il est nécessaire de générer un ASG en utilisant l'option *-fdump-tree-original*[1] de GCC. C'est la version 3.1.1 de GCC qui est utilisée pour ces tests sur laquelle la rustine fournie avec GASTA a été appliquée. Cette rustine corrige une erreur lors de l'écriture des chaînes de caractères.

Toutes les sorties générées par GASTA ont été placées en annexe II page 136. Elles contiennent des métriques concernant la durée de l'analyse et le nombre de passes effectuées par l'algorithme. Elles contiennent aussi les informations récupérées au niveau des instructions *return* d'une fonction (si il y en a).

Cette section traitant des résultats se découpe en six parties. La première partie est la vérification de la propagation de l'information, la seconde ainsi que

[1] Cette option est équivalente à l'option *-fdump-ast-original* pour les versions inférieures à GCC-3.1

```
#include <stdlib.h>

int
main()
{
    int *ptr1;
    int *ptr2;
    int i = 3;

    ptr1 = (int *) malloc(sizeof (int));
    ptr1 = &i;

    if (i > 10) {
        ptr2 = (int *) malloc(sizeof (int));
        ptr2 = &i;
    } else {
        *ptr1 = 10;
    }

    *ptr1 = *ptr2 + 3;
    return i;
}
```

Figure 5.1: Code source de propagation.c

la troisième partie valident respectivement les opérateurs $\oplus$ et $\ominus$ qu'on retrouve dans l'algorithme itératif pour l'annotation. La quatrième partie est l'étude d'une fonction extraite du code d'une application réelle. Dans la cinquième on retrouve un tableau comparatif des différents résultats obtenus et finalement, la dernière partie tire un bilan des ces analyses.

5.1 Vérification de la propagation de l'information

Le premier test permet de montrer la bonne propagation de l'information à travers le CFG. Le test comporte quatre étapes. Le code de ce test est donné à la figure 5.1

La première étape va consister à allouer un espace mémoire pour un pointeur ce qui va avoir pour effet de générer le couple $(@X \leftarrow @0)$ avec $@X$ le numéro du

pointeur. Lors de ce test GASTA génère un fichier *annot_builder.txt* qui contient les informations pertinentes dans le cadre de l'analyse présentée dans ce mémoire. Le contenu de ce fichier se trouve sur la figure 5.3. En regardant ce fichier on remarque que le couple généré sera (@28 ← @0). En effet on retrouve bien la section suivante:

```
[Node 28]  source: propagation.c:6 | name: ptr1 | type: @31 | used: 1
        -> local variable
        -> not initialized
```

On peut remarquer d'autres informations comme l'endroit où la variable $ptr1$ est déclarée ce qui sera intéressant pour l'annotation. Si on regarde la figure 5.4, on voit bien que l'ensemble *init_null* du bloc $N°2$ contient effectivement l'élément (@28 ← @0). Le fait qu'on retrouve cet élément dans l'ensemble *out_null* du même bloc de base montre que l'information est bien transmise à l'intérieur d'un bloc.

La seconde étape va assigner une adresse mémoire au pointeur. On ne s'intéresse pas au bon fonctionnement du programme car dans ce cas il est bien évident qu'on vient de perdre la référence sur l'espace mémoire précédemment alloué. L'objectif ici est de générer une autre information qui est (@28 ← @ − 1). Dans cette étape il y a deux choses intéressantes. Si on regarde le contenu des ensembles du bloc de base $N°3$ on remarque deux choses. Tout d'abord l'ensemble *in_null* contient le couple (@28 ← @0), ce qui permet de dire que l'information provenant du bloc de base précédant a bien été transmise. Ensuite, *out_null* contient le couple (@28 ← @ − 1) ce qui montre que l'information générée par le bloc de base $N°3$ sera bien transmise au bloc suivant. L'opération effectuée dans ce cas est

```
;; Function int main() (main)
;; enabled by -dump-tree-original

@1      function_decl    name: @2       type: @3       srcp: propagation.c:5
                         C              extern         body: @4
@2      identifier_node  strg: main     lngt: 4
@3      function_type    size: @5       algn: 64       retn: @6
                         prms: @7
@4      compound_stmt    line: 22       body: @8       next: @9
@5      integer_cst      type: @10      low : 64
@6      integer_type     name: @11      size: @12      algn: 32
                         prec: 32       min : @13      max : @14
@7      tree_list        valu: @15
@8      scope_stmt       line: 5        begn           clnp
                         next: @16
@9      return_stmt      line: 22       expr: @17
@10     integer_type     name: @18      size: @5       algn: 64
                         prec: 36       unsigned       min : @19
                         max : @20
@11     type_decl        name: @21      type: @6       srcp: <internal>:0
@12     integer_cst      type: @10      low : 32
@13     integer_cst      type: @6       high: -1       low : -2147483648
@14     integer_cst      type: @6       low : 2147483647
@15     void_type        name: @22      algn: 8
@16     compound_stmt    line: 5        body: @23      next: @24
@17     init_expr        type: @6       op 0: @25      op 1: @26
@18     identifier_node  strg: bit_size_type           lngt: 13
@19     integer_cst      type: @10      low : 0
@20     integer_cst      type: @10      high: 15       low : -1
@21     identifier_node  strg: int      lngt: 3
@22     type_decl        name: @27      type: @15      srcp: <internal>:0
@23     decl_stmt        line: 6        decl: @28      next: @29
@24     scope_stmt       line: 22       end            clnp
@25     result_decl      type: @6       scpe: @1       srcp: propagation.c:5
                         size: @12      algn: 32
@26     integer_cst      type: @6       low : 0
@27     identifier_node  strg: void     lngt: 4
@28     var_decl         name: @30      type: @31      scpe: @1
                         srcp: propagation.c:6         size: @32
                         algn: 32       used: 1
@29     decl_stmt        line: 7        decl: @33      next: @34
@30     identifier_node  strg: ptr1     lngt: 4
@31     pointer_type     size: @32      algn: 32       ptd : @6
@32     integer_cst      type: @10      low : 32
@33     var_decl         name: @35      type: @31      scpe: @1
                         srcp: propagation.c:7         size: @32
                         algn: 32       used: 1
@34     decl_stmt        line: 8        decl: @36      next: @37
@35     identifier_node  strg: ptr2     lngt: 4
@36     var_decl         name: @38      type: @6       scpe: @1
                         srcp: propagation.c:8         init: @39
                         size: @12      algn: 32       used: 1
@37     expr_stmt        line: 10       expr: @40      next: @41
@38     identifier_node  strg: i        lngt: 1
@39     integer_cst      type: @6       low : 3
@40     modify_expr      type: @31      op 0: @28      op 1: @42
@41     expr_stmt        line: 11       expr: @43      next: @44
...
```

Figure 5.2: Extrait de l'ASG obtenu sur le fichier propagation.c de la figure 5.1

$(@28 \leftarrow @0) \ominus (@28 \leftarrow @-1) \longrightarrow (@28 \leftarrow @-1)$. Cette opération sera validée dans la section 5.3.2. Dans cette seconde étape il est important de noter que c'est bien l'information $(@28 \leftarrow @-1)$ qui est transmise.

La troisième étape permet de diviser le CFG en deux branches par l'intermédiaire d'un test. Le but est de prouver que l'information se propage bien dans les deux branches. Si on regarde les blocs de base $N°5$ et $N°8$, on remarque l'élément $(@28 \leftarrow @-1)$ appartient bien à l'ensemble *in_null* de chacun d'eux. L'information est donc bien diffusée sur les deux branches du CFG.

La quatrième et dernière étape permet de voir le bon comportement de l'algorithme lorsqu'un bloc de base possède deux antécédents. Il est facile de vérifier en observant les éléments contenus dans les ensembles du bloc de base $N°9$ qu'ils ont été obtenus en utilisant l'opération $\oplus$ et que l'information ainsi générée est bien valide.

La figure 5.5 permet de voir à quoi ressemble un ASG généré par GCC. Cet ASG a été obtenu en utilisant l'option *-t dot* de GASTA. Cette figure n'est là qu'à titre d'exemple et, afin de ne pas surcharger ce mémoire, les ASG des autres programmes de test ne seront pas fournis sachant qu'ils ne sont pas primordiaux pour la compréhension de la validation de opérations de l'algorithme itératif.

Après avoir mis en évidence la bonne propagation de l'information à travers le CFG, les deux prochaines sections ont pour objectif la validation des deux opérateurs $\oplus$ et $\ominus$.

```
Information about type
----------------------

;; Function int main() (main)
[Node 55]       name=pointer_type       ptd=@63
[Node 3]        name=function_type
[Node 58]       name=boolean_type
[Node 15]       name=void_type
[Node 48]       name=pointer_type       ptd=@15
[Node 6]        name=integer_type
[Node 72]       name=function_type
[Node 63]       name=function_type
[Node 31]       name=pointer_type       ptd=@6
[Node 10]       name=integer_type
[Node 65]       name=integer_type

Information about variable
--------------------------

;; Function int main() (main)
[Node 33]  source: propagation.c:7 | name: ptr2 | type: @31 | used: 1
        -> local variable
        -> not initialized

[Node 36]  source: propagation.c:8 | name: i | type: @6 | used: 1
        -> local variable
        -> not initialized

[Node 28]  source: propagation.c:6 | name: ptr1 | type: @31 | used: 1
        -> local variable
        -> not initialized


Information about parameters
----------------------------

;; Function int main() (main)

Information about functions
---------------------------

;; Function int main() (main)
[Node 1]        name=main       source=propagation.c:5  annotated=0
[Node 56]       name=malloc     source=stdlib.h:527     annotated=1
```

Figure 5.3: Contenu des tables pour le programme propagation.c

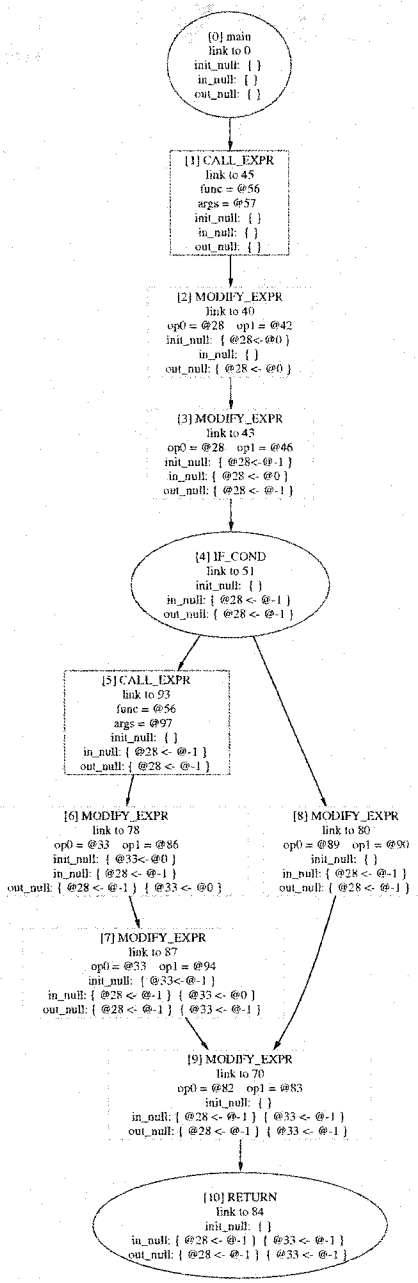


Figure 5.4: CFG après analyse de propagation.c

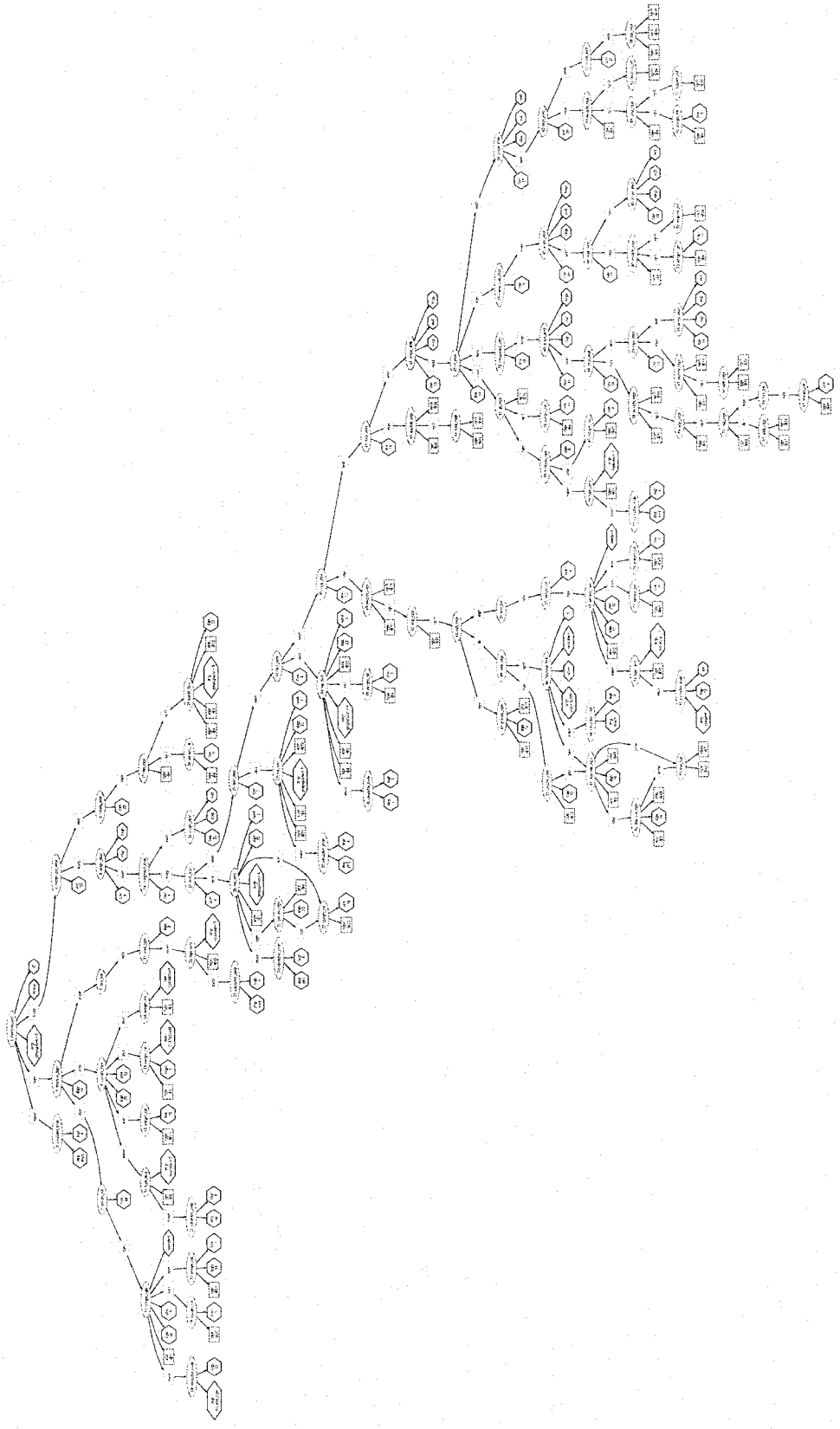


Figure 5.5: ASG de propagation.c

5.2 validation de l'opérateur $\oplus$

L'opérateur $\oplus$ permet de calculer l'ensemble *in_null* d'un bloc de base. Cette opération s'effectue en utilisant les ensembles *out_null* de tous les blocs de base précédent celui dont on calcule l'ensemble *in_null*. Cette opération se rapproche d'une union entre les ensembles avec quelques particularités qui feront l'objet de cette étude. La figure 3.16 montre qu'il existe trois règles :

- Règle $N°1 \quad (@X \leftarrow @0) \oplus (@X \leftarrow *) \longrightarrow (@X \leftarrow @0)$
- Règle $N°2 \quad (@X \leftarrow @Y) \oplus (@X \leftarrow -1) \longrightarrow (@X \leftarrow @Y)$
- Règle $N°3 \quad (@X \leftarrow @Y) \oplus (@X \leftarrow @Z) \longrightarrow (@X \leftarrow @Y), (@X \leftarrow @Z)$

  Si $Y \neq 0$ et $Y \neq -1$ et $Z \neq 0$ et $Z \neq -1$

  sinon nous sommes dans le cas 1 ou 2

Le test de ces trois règles constitue les trois prochaines étapes de la validation de l'algorithme itératif qui va permettre la récupération de l'information nécessaire à l'annotation.

5.2.1 Validation de la règle $N°1$

Cette première règle indique que s'il existe un pointeur pouvant être nul appartenant à l'ensemble *out_null* d'un des successeur du bloc de base courant, alors le pointeur est possiblement nul dans le bloc courant. Afin de valider la première

règle, trois tests sont nécessaires. Ces trois tests correspondent aux trois cas possibles suivant:

- Cas A: $(@X \leftarrow @0) \oplus (@X \leftarrow @ - 1) \longrightarrow (@X \leftarrow @0)$
- Cas B: $(@X \leftarrow @0) \oplus (@X \leftarrow @0) \longrightarrow (@X \leftarrow @0)$
- Cas C: $(@X \leftarrow @0) \oplus (@X \leftarrow @Y) \longrightarrow (@X \leftarrow @0)$

5.2.1.1 Cas A

Le code utilisé pour ce test est celui de la figure 5.6. C'est l'instruction *switch* qui a été utilisée car elle permet de créer autant de branches dans le CFG qu'il est nécessaire et elle offre un moyen simple de contrôler les ensembles au sein de chacun des blocs.

Comme à chaque utilisation de GASTA, un fichier contenant les informations sur les variables, les types, les paramètres ainsi que les functions est généré. Ce fichier permet de voir si toutes les informations relatives à l'annotation du code source sont présentes. En effet, étant donné que GASTA ne réalise pas encore l'annotation automatique, ce fichier permet de constater que toute l'information est bien récupérée. Le contenu de ce fichier se trouve sur la figure 5.7.

Le résultat de l'analyse faite par GASTA apparaît sur le CFG donné à la figure 5.8.

```
#include <stdlib.h>

int
main()
{

    char s = 'a';
    int i = 3;
    int *ptr;


    switch (s) {
    case 'a':
        ptr = (int *) malloc(sizeof (int));  /* (@X <- @0) */
        break;
    case 'b':
        ptr = &i;                            /* (@X <- @-1) */
        break;

    }

    i = *ptr + 1;

    return s;
}
```

Figure 5.6: Code de test du premier cas (cas_a.c)

Le point important de ce test se trouve au bloc de base N^o9. Dans chacune des branches de l'instruction *switch*, une information a été générée. Dans un cas il s'agit d'un pointeur possiblement nul et dans l'autre cas du même pointeur ne pouvant pas être nul. Ce qui est important pour pouvoir annoter une fonction avec */*@null@*/* c'est de connaître s'il existe un chemin dans le CFG par lequel un pointeur peut être nul. Si un tel chemin existe, alors la fonction peut retourner un pointeur nul et c'est bien ce que recherche l'algorithme mis en œuvre dans ce mémoire. Donc, lors du calcul de l'ensemble *in_null* du bloc de base N^o9, on devrait voir apparaître l'information indiquant qu'il est possible qu'un pointeur soit nul. Or, c'est exactement ce que montre l'analyse puisque (@41 $\leftarrow$ @0). C'est

```
Information about type
----------------------

;; Function int main() (main)
[Node 45]       name=pointer_type       ptd=@6
[Node 3]        name=function_type
[Node 80]       name=function_type
[Node 15]       name=void_type
[Node 70]       name=pointer_type       ptd=@15
[Node 82]       name=integer_type
[Node 6]        name=integer_type
[Node 83]       name=function_type
[Node 31]       name=integer_type
[Node 75]       name=pointer_type       ptd=@80
[Node 10]       name=integer_type

Information about variable
--------------------------

;; Function int main() (main)
[Node 34]  source: cas_a.c:8 | name: i | type: @6 | used: 1
        -> local variable
        -> not initialized

[Node 28]  source: cas_a.c:7 | name: s | type: @31 | used: 1
        -> local variable
        -> not initialized

[Node 41]  source: cas_a.c:9 | name: ptr | type: @45 | used: 1
        -> local variable
        -> not initialized


Information about parameters
----------------------------

;; Function int main() (main)

Information about functions
---------------------------

;; Function int main() (main)
[Node 1]        name=main       source=cas_a.c:5        annotated=0
[Node 76]       name=malloc     source=stdlib.h:527     annotated=1
```

Figure 5.7: Contenu des tables pour cas_a.c

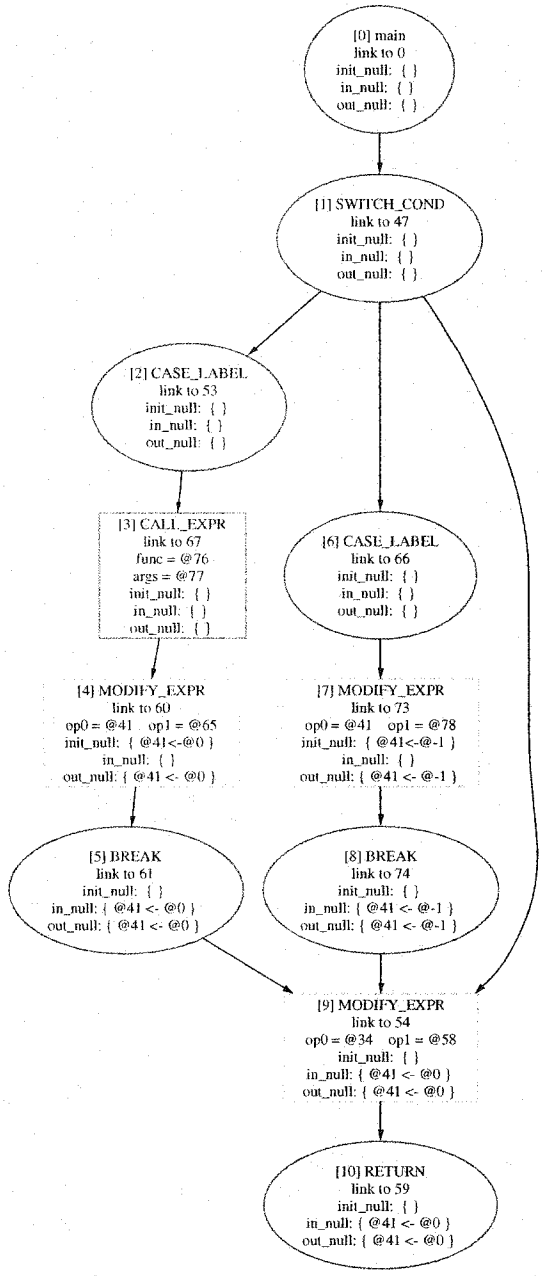


Figure 5.8: CFG après analyse de cas_a.c

```
#include <stdlib.h>

int
main()
{

    int i;
    int *ptr1;

    i = rand();

    if (i < 10)
        ptr1 = (int *) malloc(sizeof (int));  /* (@X <- @0) */
    else
        ptr1 = (int *) malloc(10 * sizeof (int));  /* (@X <- @0) */

    i++;
    return i;
}
```

Figure 5.9: Code de test du second cas (cas_b.c)

à dire en se référant au contenu du fichier de la figure 5.7, la variable *ptr* est possiblement nulle.

On peut également voir que le bloc de base N^o9 du CFG contient trois prédécesseurs dont l'un est la condition de l'instruction *switch*. Cela signifie qu'il n'y a pas de cas *default*.

5.2.1.2 Cas B

Ce cas peut arriver lorsqu'une zone mémoire est allouée en suivant deux chemins différents. Par exemple, il est possible d'allouer deux zones de tailles différentes par rapport à une condition se trouvant dans un *if*, comme le montre la figure 5.9

Le fichier de la figure 5.10 est généré par GASTA lors de son exécution.

Dans ce cas, il est évident que s'il existe deux chemins possibles menant à un pointeur possiblement nul, lorsque ces deux chemins se rejoignent le pointeur est

```
Information about type
----------------------

;; Function int main() (main)
[Node 34]       name=pointer_type       ptd=@6
[Node 45]       name=pointer_type       ptd=@52
[Node 3]        name=function_type
[Node 47]       name=boolean_type
[Node 80]       name=function_type
[Node 15]       name=void_type
[Node 71]       name=pointer_type       ptd=@15
[Node 82]       name=integer_type
[Node 6]        name=integer_type
[Node 83]       name=function_type
[Node 52]       name=function_type
[Node 10]       name=integer_type
[Node 76]       name=pointer_type       ptd=@80


Information about variable
--------------------------

;; Function int main() (main)
[Node 28]  source: cas_b.c:7 | name: i | type: @6 | used: 1
        -> local variable
        -> not initialized

[Node 31]  source: cas_b.c:8 | name: ptr1 | type: @34 | used: 1
        -> local variable
        -> not initialized


Information about parameters
----------------------------

;; Function int main() (main)

Information about functions
---------------------------

;; Function int main() (main)
[Node 77]       name=malloc     source=stdlib.h:527     annotated=1
[Node 1]        name=main       source=cas_b.c:5        annotated=0
[Node 46]       name=rand       source=stdlib.h:446     annotated=0
```

Figure 5.10: Contenu des tables pour cas_b.c

possiblement nul. C'est bien ce qu'on observe sur la figure 5.11.

5.2.1.3 Cas C

Comme cela a déjà été mentionné, l'objectif de cette maîtrise et donc de l'analyse présentée ici est de mettre en évidence les pointeurs possiblement nuls afin de pouvoir évaluer un ensemble de pointeurs possiblement nuls retournés par une fonction. Dans ce troisième cas, s'il existe un chemin par lequel un pointeur est possiblement nul, il faut propager cette information. Ce peu importe les valeurs de ce pointeur sur les autres chemins de données, y compris si il est *aliasé* à un autre pointeur. En effet, cet *alias* peut conduire à deux cas. Soit le pointeur *aliasé* est nul soit il ne l'est pas. Dans les deux cas, le résultat correspond au cas A et au cas B traités précédemment, c'est à dire que le pointeur possiblement nul est propagé. Ce test permet de voir si cette propriété est bien respectée.

Les informations récupérées par GASTA durant l'exécution sont données sur à figure 5.13.

Tout comme dans le premier test (règle1: cas A), le point important se trouve au bloc de base $N°9$. Dans chacune des branches de l'instruction *switch*, une information a été générée. Dans un cas, il s'agit d'un pointeur possiblement nul, et dans l'autre cas, du même pointeur *aliasé* à une autre pointeur. En regardant les tables de la figure 5.13, on peut noter que le couple (@41 $\leftarrow$ @47) correspond respectivement aux pointeurs $ptr1$ et $ptr2$. Pour en revenir au test, on voit que

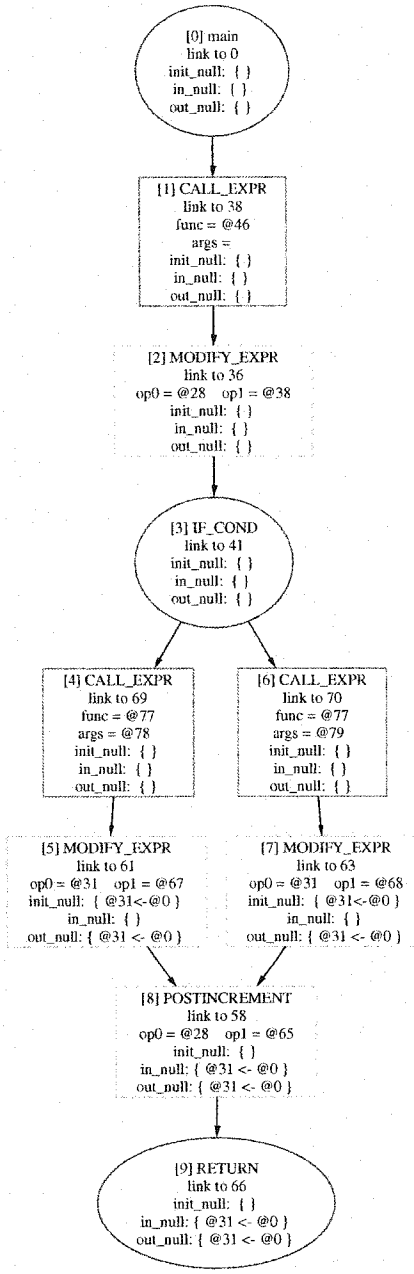


Figure 5.11: CFG après analyse de cas_b.c

```
#include <stdlib.h>

int
main()
{

    char s = 'a';
    int i = 3;
    int *ptr1;
    int *ptr2;


    switch (s) {
    case 'a':
        ptr1 = (int *) malloc(sizeof (int));
        break;
    case 'b':
        ptr1 = ptr2;
        break;

    }

    i = *ptr1 + 1;

    return s;
}
```

Figure 5.12: Code de test du troisième cas (cas_c.c)

c'est bien (@41 $\leftarrow$ @0) qui est propagé, et que cela correspond bien au résultat attendu.

Les trois cas précédemment traités (A, B et C) ont permis de valider la règle N^o1. Il reste maintenant deux règles à vérifier pour permettre de montrer que l'opérateur $\oplus$ fonctionne de la manière attendue.

```
Information about type
----------------------

;; Function int main() (main)
[Node 45]       name=pointer_type       ptd=@6
[Node 78]       name=pointer_type       ptd=@82
[Node 3]        name=function_type
[Node 15]       name=void_type
[Node 82]       name=function_type
[Node 6]        name=integer_type
[Node 73]       name=pointer_type       ptd=@15
[Node 84]       name=integer_type
[Node 85]       name=function_type
[Node 31]       name=integer_type
[Node 10]       name=integer_type

Information about variable
--------------------------

;; Function int main() (main)
[Node 34]  source: cas_c.c:8 | name: i | type: @6 | used: 1
        -> local variable
        -> not initialized

[Node 47]  source: cas_c.c:10 | name: ptr2 | type: @45 | used: 1
        -> local variable
        -> not initialized

[Node 28]  source: cas_c.c:7 | name: s | type: @31 | used: 1
        -> local variable
        -> not initialized

[Node 41]  source: cas_c.c:9 | name: ptr1 | type: @45 | used: 1
        -> local variable
        -> not initialized


Information about parameters
----------------------------

;; Function int main() (main)

Information about functions
---------------------------

;; Function int main() (main)
[Node 1]        name=main       source=cas_c.c:5        annotated=0
[Node 79]       name=malloc     source=stdlib.h:527     annotated=1
```

Figure 5.13: Contenu des tables pour cas_c.c

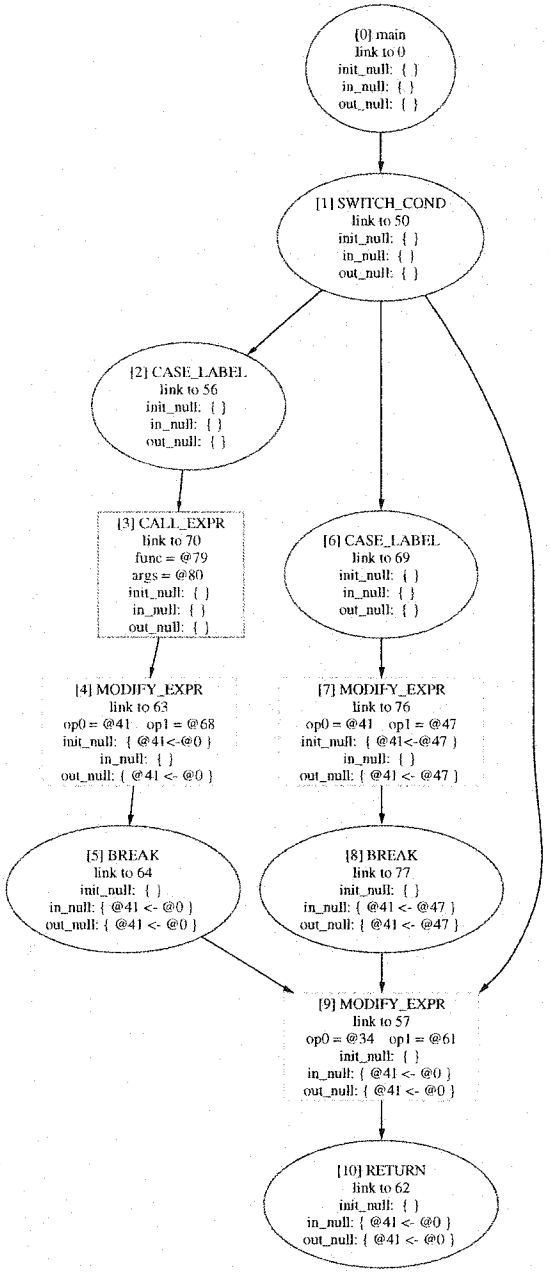


Figure 5.14: CFG après analyse de cas_c.c

5.2.2 Validation de la règle N^o2

Cette règle indique que si un pointeur est *aliasé* à un autre pointeur dans une branche du CFG, et non nul dans une autre branche, alors l'union de ces deux états est le couple $(@X \leftarrow @Y)$. Le code de la figure 5.15 est utilisé pour ce test.

```
#include <stdlib.h>

int
main()
{

    char s = 'a';
    int i = 3;
    int *ptr1;
    int *ptr2;

    switch (s) {
    case 'a':
        ptr1 = &i;                          /* (@X <- @-1) */
        break;
    case 'b':
        ptr1 = ptr2;                        /* (@X <- @Y) */
        break;

    }

    i = *ptr1 + 1;

    return i;
}
```

Figure 5.15: Code de test pour la règle N^o2 (regle2.c)

Les informations récupérées par GASTA lors de l'analyse sont données à la figure 5.16. Les deux pointeurs $ptr1$ et $ptr2$ sont associés respectivement aux adresses @41 et @47 de l'ASG.

Le CFG obtenu après l'analyse est donné à la figure 5.17. On remarque sur ce graphe que les ensembles *out_null* des blocs de base N^o4 et N^o7 contiennent respectivement les couples $(@41 \leftarrow @-1)$ et $(@41 \leftarrow @47)$, ce qui correspond

```
Information about type
----------------------

;; Function int main() (main)
[Node 45]       name=pointer_type       ptd=@6
[Node 3]        name=function_type
[Node 15]       name=void_type
[Node 6]        name=integer_type
[Node 31]       name=integer_type
[Node 10]       name=integer_type

Information about variable
--------------------------

;; Function int main() (main)
[Node 34]  source: regle2.c:8 | name: i | type: @6 | used: 1
        -> local variable
        -> not initialized

[Node 47]  source: regle2.c:10 | name: ptr2 | type: @45 | used: 1
        -> local variable
        -> not initialized

[Node 28]  source: regle2.c:7 | name: s | type: @31 | used: 1
        -> local variable
        -> not initialized

[Node 41]  source: regle2.c:9 | name: ptr1 | type: @45 | used: 1
        -> local variable
        -> not initialized


Information about parameters
----------------------------

;; Function int main() (main)

Information about functions
---------------------------

;; Function int main() (main)
[Node 1]        name=main       source=regle2.c:5       annotated=0
```

Figure 5.16: Contenu des tables pour regle2.c

bien à la règle qu'on souhaite tester. Le résultat de l'opération se retrouve dans l'ensemble *in_null* du bloc de base $N^o 8$ et il s'agit bien du couple attendu.

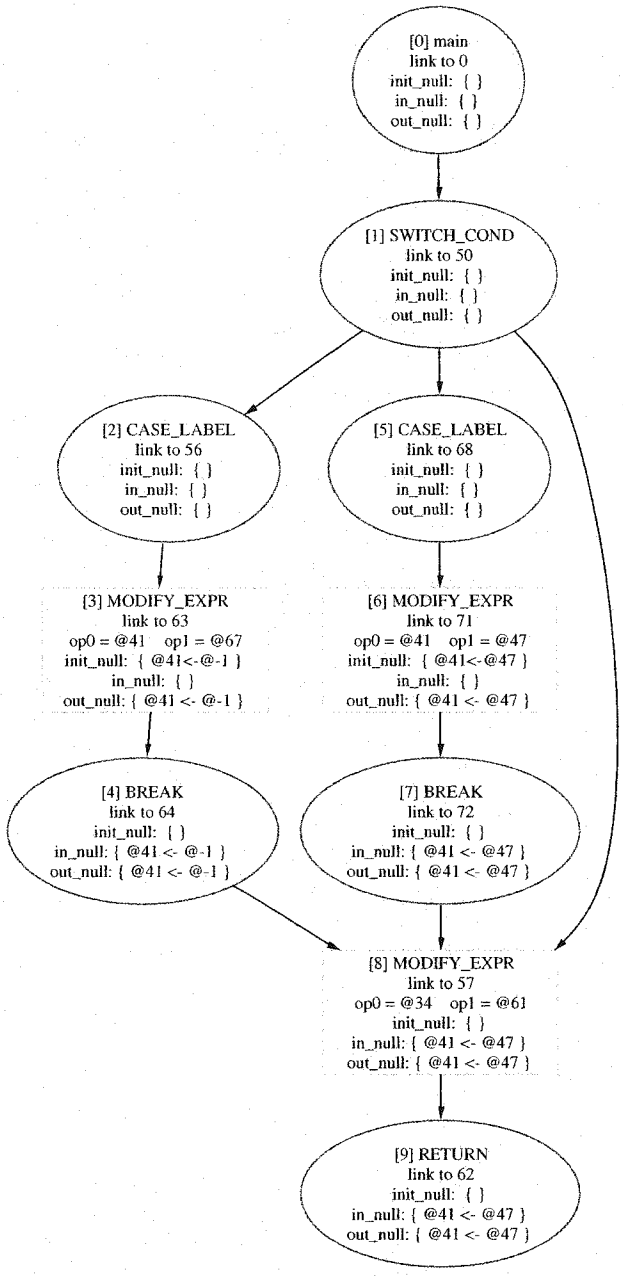


Figure 5.17: CFG après analyse de regle2.c

```
#include <stdlib.h>

int
main()
{

    char s = 'a';
    int i = 3;
    int *ptr1;
    int *ptr2;
    int *ptr3;

    switch (s) {
    case 'a':
        ptr1 = ptr2;        /* (@X <- @Y) */
        break;
    case 'b':
        ptr1 = ptr3;        /* (@X <- @Z) */
        break;

    }

    i = *ptr1 + 1;
    return i;
}
```

Figure 5.18: Code de test pour la règle N^o3 (regle3.c)

5.2.3 Validation de la règle N^o3

Afin de valider le comportement de l'opérateur $\oplus$ il reste à vérifier la troisième règle $(@X \leftarrow @Y) \oplus (@X \leftarrow @Z) \longrightarrow (@X \leftarrow @Y), (@X \leftarrow @Z)$. Cette règle s'applique dans le cas ou un pointeur est *aliasé* à deux pointeurs différents en suivant deux branches différentes du CFG. Dans ce cas, étant donné qu'il est impossible de prédire la valeur du pointeur (possiblement nul ou non-nul), il est nécessaire de conserver les deux couples afin de s'assurer que toutes les possibilités sont bien propagées. La figure 5.18 montre le code utilisé pour le test de la règle N^o3.

Lors de l'analyse, GASTA a récupéré les informations présentées sur la figure

5.19. On peut remarquer que les trois pointeurs $ptr1, ptr2$ et $ptr3$ sont bien du même type et qu'ils ont les nœuds de l'ASG de valeurs respectives @41, @47 et @50.

Le CFG obtenu après l'analyse est celui donné à la figure 5.20. On remarque sur ce graphe que les ensembles *out_null* des blocs de base $N°4$ et $N°7$ contiennent respectivement les couples (@41 ← @47) et (@41 ← @50) ce qui correspond bien à la règle qu'on souhaite tester. Le résultat de l'opération se retrouve dans l'ensemble *in_null* du bloc de base $N°8$ et les deux couples représentant les deux *alias* ont bien été ajoutés. Le comportement est donc bien celui attendu.

5.2.3.1 discussion

Les trois règles ont donc été testées. Étant donné qu'elles couvrent l'ensemble des cas possibles pouvant survenir lors d'une analyse, il est possible de conclure que le comportement de l'opérateur correspond effectivement à la définition de celui-ci.

A ce point du mémoire, il reste à valider l'opérateur $\ominus$. C'est l'objectif de la prochaine section.

```
Information about type
----------------------

;; Function int main() (main)
[Node 45]       name=pointer_type       ptd=@6
[Node 3]        name=function_type
[Node 15]       name=void_type
[Node 6]        name=integer_type
[Node 31]       name=integer_type
[Node 10]       name=integer_type

Information about variable
--------------------------

;; Function int main() (main)
[Node 34]  source: regle3.c:8 | name: i | type: @6 | used: 1
        -> local variable
        -> not initialized

[Node 47]  source: regle3.c:10 | name: ptr2 | type: @45 | used: 1
        -> local variable
        -> not initialized

[Node 28]  source: regle3.c:7 | name: s | type: @31 | used: 1
        -> local variable
        -> not initialized

[Node 50]  source: regle3.c:11 | name: ptr3 | type: @45 | used: 1
        -> local variable
        -> not initialized

[Node 41]  source: regle3.c:9 | name: ptr1 | type: @45 | used: 1
        -> local variable
        -> not initialized


Information about parameters
----------------------------

;; Function int main() (main)

Information about functions
---------------------------

;; Function int main() (main)
[Node 1]        name=main       source=regle3.c:5       annotated=0
```

Figure 5.19: Contenu des tables pour regle3.c

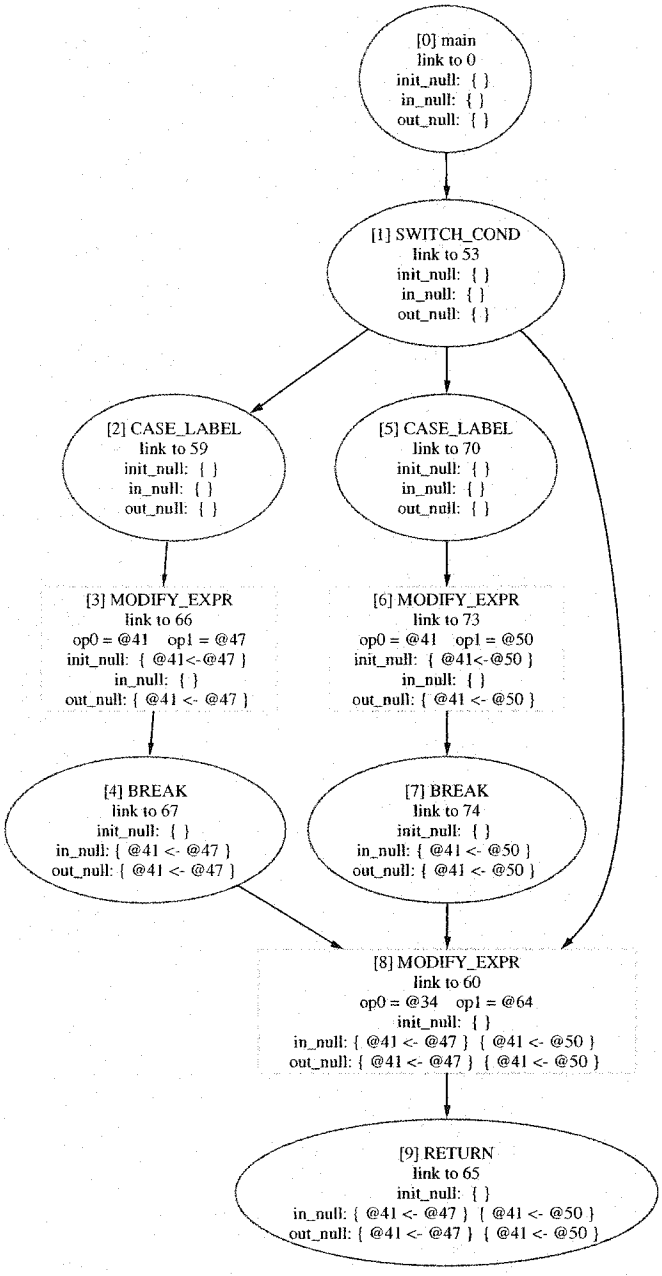


Figure 5.20: CFG après analyse de regle3.c

5.3 validation de l'opérateur $\ominus$

L'opérateur $\ominus$ permet de calculer l'ensemble *out_null* d'un bloc de base. Cette opération s'effectue en utilisant les ensembles *init_null* et *in_null* du bloc de base. Si l'ensemble *init_null* est vide, on se retrouve dans le cas d'une propagation de l'information et *out_null* = *in_null*. Dans la suite de cette étude, l'ensemble *init_null* sera donc considéré non vide. Étant donné que les blocs de base du CFG construit par GASTA ne contiennent qu'une seule instruction, l'ensemble *init_null* ne peut contenir qu'un seul couple, ce qui conduit à trois cas possibles qui sont :

- cas A $(@X \leftarrow @0) \in init\_null$
- cas B $(@X \leftarrow @-1) \in init\_null$
- cas C $(@X \leftarrow @Y) \in init\_null$

Ce couple agit comme un filtre. Cela signifie que si on le retrouve dans l'ensemble *in_null*, cela aura comme effet d'effacer le valeur précédemment évaluée (celle donnée par le couple présent dans *in_null*). Les diverses possibilités concernant les couples qu'on peut retrouver dans l'ensemble *in_null* méritent d'être examinées. Il peut ne pas contenir le couple $(@X \leftarrow @*)$. Dans ce cas, on se retrouve encore dans le cas d'une propagation de l'information. Si il contient un tel couple, $@*$ peut avoir trois valeurs possibles qui sont $@0, @-1$ ou encore

$@Y$. Il existe donc neuf possibilités qui vont faire l'objet des trois prochaines sections. Pour les tests, l'instruction *switch* va être utilisée. Contrairement aux tests précédents elle possédera trois cas possibles qui sont les trois valeurs possible du couple $(@X \leftarrow @*)$ de l'ensemble *in_null*.

5.3.1 cas A

Ce premier test correspond au cas ou un pointeur possiblement nul appartient à l'ensemble *init_null*, par exemple lors de l'allocation d'une zone mémoire. Comme cela a été indiqué, l'instruction *switch* permet de créer trois branches, trois chemins possibles dans le CFG, comme le montre le code en C de la figure 5.21.

Les différentes informations récupérées dans les tables de GASTA durant l'analyse sont données à la figure 5.22.

Le CFG obtenu après l'analyse se trouve à la figure 5.23. On remarque par rapport aux tests précédents qu'il possède plus de blocs de bases. Ceci s'explique par le fait que l'instruction *switch* possède une condition supplémentaire. Les trois possibilités pour ce premier cas se retrouvent dans les blocs de base $N°6$, $N°12$ et $N°18$. Il est facile de constater que dans les trois cas l'information propagée est bien celle générée par l'ensemble *init_null*, ce qui est logique puisque cet ensemble annule les définitions précédentes. Le résultat est donc bien conforme aux attentes.

```
#include <stdlib.h>

int
main()
{
    char s = 'a';
    int i = 3;
    int *ptr;
    int *ptr1;

    switch (s) {
    case 'a':
        ptr = (int *) malloc(sizeof (int));  /* (@X <- @0) */
        ptr = (int *) malloc(sizeof (int));  /* (@X <- @0) */
        *ptr = i;
        break;
    case 'b':
        ptr = ptr1;                          /* (@X <- @Y) */
        ptr = (int *) malloc(sizeof (int));  /* (@X <- @0) */
        *ptr = i;
        break;
    case 'c':
        ptr = &i;                            /* (@X <- @-1) */
        ptr = (int *) malloc(sizeof (int));  /* (@X <- @0) */
        *ptr = i;
        break;
    }

    i = *ptr + 1;
    return s;
}
```

Figure 5.21: Code de test pour le cas A (cas_a.c)

```
Information about type
----------------------

;; Function int main() (main)
[Node 88]       name=function_type
[Node 45]       name=pointer_type       ptd=@6
[Node 90]       name=integer_type
[Node 3]        name=function_type
[Node 15]       name=void_type
[Node 81]       name=pointer_type       ptd=@88
[Node 6]        name=integer_type
[Node 94]       name=function_type
[Node 31]       name=integer_type
[Node 75]       name=pointer_type       ptd=@15
[Node 10]       name=integer_type

Information about variable
--------------------------

;; Function int main() (main)
[Node 34]  source: cas_a.c:7 | name: i | type: @6 | used: 1
        -> local variable
        -> not initialized

[Node 47]  source: cas_a.c:9 | name: ptr1 | type: @45 | used: 1
        -> local variable
        -> not initialized

[Node 28]  source: cas_a.c:6 | name: s | type: @31 | used: 1
        -> local variable
        -> not initialized

[Node 41]  source: cas_a.c:8 | name: ptr | type: @45 | used: 1
        -> local variable
        -> not initialized


Information about parameters
----------------------------

;; Function int main() (main)

Information about functions
---------------------------

;; Function int main() (main)
[Node 1]        name=main       source=cas_a.c:5        annotated=0
[Node 82]       name=malloc     source=stdlib.h:527     annotated=1
```

Figure 5.22: Contenu des tables pour cas_a.c

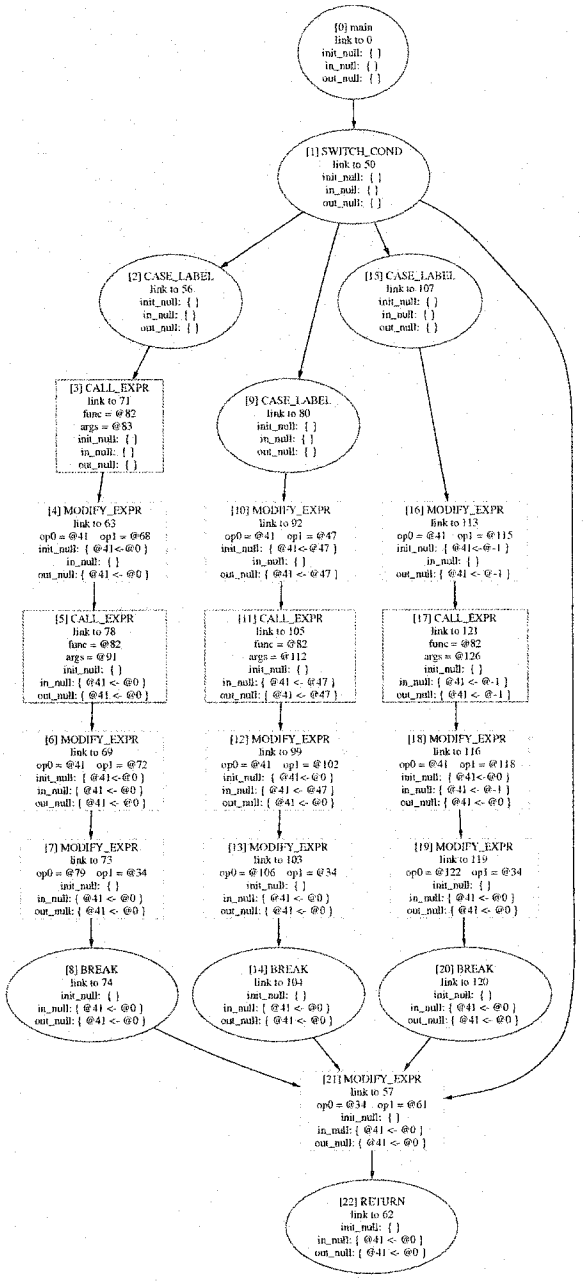


Figure 5.23: CFG après analyse de cas_a.c

```
#include <stdlib.h>

int
main()
{
    char s = 'a';
    int i = 3;
    int j = 4;
    int *ptr;
    int *ptr1;

    switch (s) {
    case 'a':
        ptr = (int *) malloc(sizeof (int));  /* (@X <- @0) */
        ptr = &i;                            /* (@X <- @-1) */
        *ptr++;
        break;
    case 'b':
        ptr = ptr1;                          /* (@X <- @Y) */
        ptr = &i;                            /* (@X <- @-1) */
        *ptr++;
        break;
    case 'c':
        ptr = &i;                            /* (@X <- @-1) */
        ptr = &j;                            /* (@X <- @-1) */
        *ptr++;
        break;
    }

    i = *ptr + 1;
    return s;
}
```

Figure 5.24: Code de test pour le cas B (cas_b.c)

5.3.2 cas B

Ce second test correspond au cas ou un pointeur non nul appartient à l'ensemble *init_null*. C'est la cas lorsqu'on assigne une adresse au pointeur. Tout comme dans le cas précédent, l'instruction *switch* va permettre de créer trois chemins possibles dans le CFG permettant de traiter les trois possibilités. On retrouve le code utilisé pour le test à la figure 5.24.

Les différentes informations récupérées pendant l'analyse par GASTA sont données à la figure 5.25.

```
Information about type
----------------------

;; Function int main() (main)
[Node 89]       name=function_type
[Node 79]       name=pointer_type       ptd=@15

[Node 3]        name=function_type
[Node 91]       name=integer_type
[Node 15]       name=void_type
[Node 49]       name=pointer_type       ptd=@6

[Node 6]        name=integer_type
[Node 94]       name=function_type
[Node 84]       name=pointer_type       ptd=@89

[Node 31]       name=integer_type
[Node 10]       name=integer_type

Information about variable
--------------------------

;; Function int main() (main)
[Node 34]  source: cas_b.c:7 | name: i | type: @6 | used: 1
        -> local variable
        -> not initialized

[Node 46]  source: cas_b.c:9 | name: ptr | type: @49 | used: 1
        -> local variable
        -> not initialized

[Node 28]  source: cas_b.c:6 | name: s | type: @31 | used: 1
        -> local variable
        -> not initialized

[Node 51]  source: cas_b.c:10 | name: ptr1 | type: @49 | used: 1
        -> local variable
        -> not initialized

[Node 41]  source: cas_b.c:8 | name: j | type: @6 | used: 1
        -> local variable
        -> not initialized


Information about parameters
----------------------------

;; Function int main() (main)

Information about functions
---------------------------

;; Function int main() (main)
[Node 1]        name=main       source=cas_b.c:5        annotated=0
[Node 85]       name=malloc     source=stdlib.h:527     annotated=1
```

Figure 5.25: Contenu des tables pour cas_b.c

Le CFG obtenu après l'analyse se trouve à la figure 5.26. Les trois possibilités pour ce second cas se retrouvent dans les blocs de base N^o5, N^o10 et N^o15. Comme dans le cas A, on constate que dans les trois cas l'information propagée est bien celle générée par l'ensemble *init_null*. Il s'agit bien du résultat attendu.

5.3.3 cas C

Ce dernier cas survient lorsqu'un pointeur *aliasé* appartient à l'ensemble *init_null*. On retrouve ce cas lors d'assignation entre pointeurs. Là encore, l'instruction *switch* va permettre de créer trois chemins possibles dans le CFG, permettant d'étudier les trois possibilités. On retrouve le code utilisé pour le test à la figure 5.27.

Comme à chaque analyse, GASTA récupèrent différentes informations concernant les variables utilisées dans le programme testés. Ces informations sont celles données à la figure 5.28.

Le CFG obtenu après l'analyse se trouve à la figure 5.29. Les trois possibilités pour ce second cas se retrouvent dans les blocs de base N^o6, N^o11 et N^o16. Contrairement aux deux autres cas, on observe que cette fois les ensembles *in_null* possèdent non pas un élément mais deux éléments. Pour valider le cas C, on ne doit considérer que les couples intervenant lors du calcul. Par exemple, le couple (@41 $\leftarrow$ @47) de l'ensemble *init_null* du bloc de base N^o6 va être un opérateur de $\ominus$ avec le couple (@41 $\leftarrow$ @0) de l'ensemble *in_null*. Le résultat de l'opération

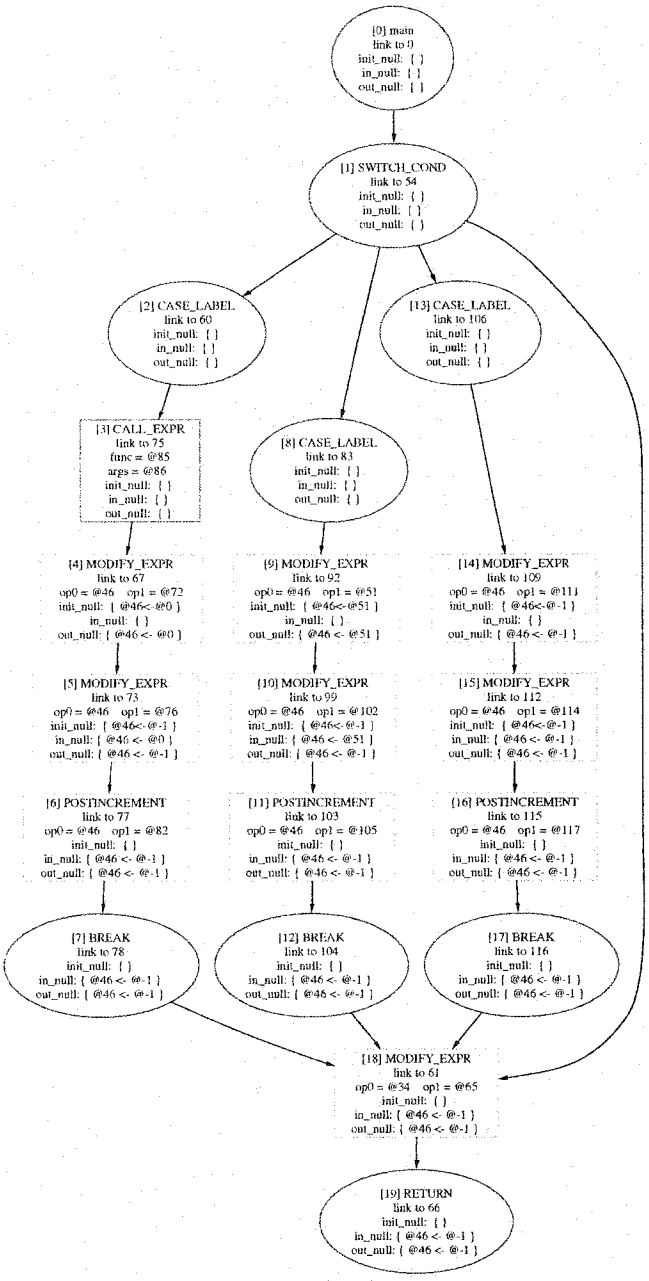


Figure 5.26: CFG après analyse de cas_b.c

```
#include <stdlib.h>

int
main()
{
    char s = 'a';
    int i = 3;
    int *ptr;
    int *ptr1;
    int *ptr2;

    ptr1 = &i;

    switch (s) {
    case 'a':
        ptr = (int *) malloc(sizeof (int));  /* (@X <- @0) */
        ptr = ptr1;                          /* (@X <- @Y) */
        *ptr = i;
        break;
    case 'b':
        ptr = ptr2;                          /* (@X <- @Z) */
        ptr = ptr1;                          /* (@X <- @Y) */
        *ptr = i;
        break;
    case 'c':
        ptr = &i;                            /* (@X <- @-1) */
        ptr = ptr1;                          /* (@X <- @Y) */
        *ptr = i;
        break;
    }

    i = *ptr + 1;
    return s;
}
```

Figure 5.27: Code de test pour le cas C (cas_c.c)

```
Information about type
----------------------

;; Function int main() (main)
[Node 45]       name=pointer_type       ptd=@6

[Node 90]       name=function_type
[Node 3]        name=function_type
[Node 80]       name=pointer_type       ptd=@15

[Node 15]       name=void_type
[Node 92]       name=integer_type
[Node 6]        name=integer_type
[Node 95]       name=function_type
[Node 85]       name=pointer_type       ptd=@90

[Node 31]       name=integer_type
[Node 10]       name=integer_type

Information about variable
--------------------------

;; Function int main() (main)
[Node 34]  source: cas_c.c:7 | name: i | type: @6 | used: 1
        -> local variable
        -> not initialized

[Node 47]  source: cas_c.c:9 | name: ptr1 | type: @45 | used: 1
        -> local variable
        -> not initialized

[Node 28]  source: cas_c.c:6 | name: s | type: @31 | used: 1
        -> local variable
        -> not initialized

[Node 50]  source: cas_c.c:10 | name: ptr2 | type: @45 | used: 1
        -> local variable
        -> not initialized

[Node 41]  source: cas_c.c:8 | name: ptr | type: @45 | used: 1
        -> local variable
        -> not initialized


Information about parameters
----------------------------

;; Function int main() (main)

Information about functions
---------------------------

;; Function int main() (main)
[Node 1]        name=main       source=cas_c.c:5        annotated=0
[Node 86]       name=malloc     source=stdlib.h:527     annotated=1
```

Figure 5.28: Contenu des tables pour cas_c.c

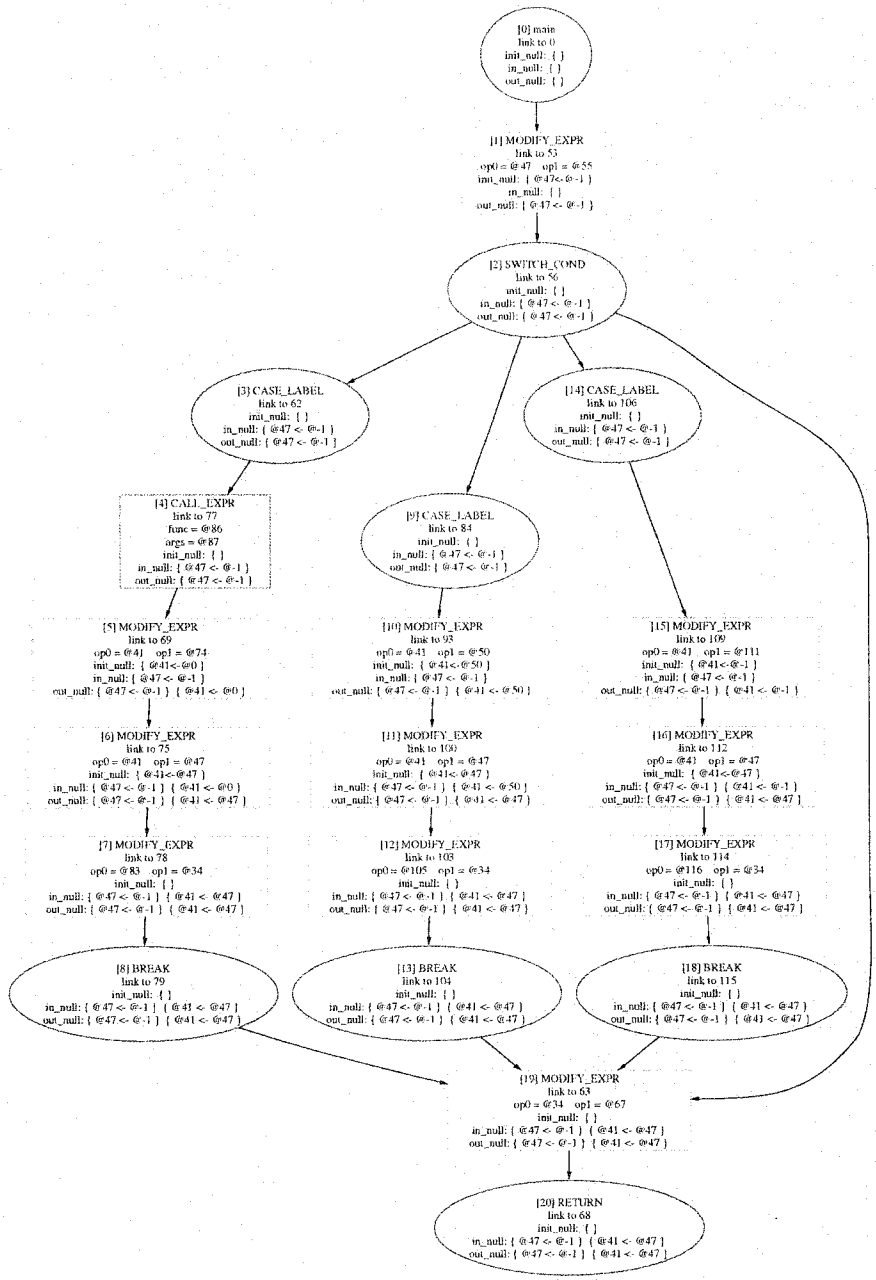


Figure 5.29: CFG après analyse de cas_c.c

entraîne l'apparition du couple (@41 ← @47) dans l'ensemble *out_null*. On remarque que l'autre couple est simplement propagé puisque la partie gauche de ce couple n'apparaît pas dans la partie gauche d'un couple qui appartiendrait à l'ensemble *init_null*. Afin de valider ce cas, le même raisonnement est appliqué aux deux autres blocs de base (le N^o11 et le N^o16). Les ensembles obtenus sont bien conformes aux prédictions.

La section suivante est une récapitulation des résultats précédents synthétisés dans un tableau.

5.3.4 Tableau des résultats

Programmes testés	Temps d'exécution	Nombre de blocs de base	Nombre d'itérations	Nombre de ligne de code	Résultats conformes
Propagation					
propagation.c	0.0229s	10	2	18	oui
Opérateur *oplus*					
Règle1: cas_a.c	0.0217s	10	2	18	oui
Règle1: cas_b.c	0.0216s	9	2	14	oui
Regle1: cas_c.c	0.0001s	10	2	19	oui
Règle2: regle2.c	0.0215s	9	3	19	oui
Règle3: regle3.c	0.0220s	9	3	20	oui
Opérateur *ominus*					
cas_a.c	0.0196s	22	2	28	oui
cas_b.c	0.0215s	19	2	29	oui
cas_c.c	0.0218s	20	2	30	oui

Tableau 5.1: Comparaison des différents tests

Le tableau de résultats 5.1 permets de voir que le temps d'exécution d'une analyse pour une fonction est très court. Il est donc intéressant de voir que pour un coût en temps d'analyse très faible il est possible de réduire le nombre de faux

messages. Un autre point positif est que GASTA ne tient compte actuellement que d'un seul type d'analyse. Il est très probable qu'en ajoutant d'autres annotations on parvienne à diminuer d'autant le nombre de faux messages d'avertissements.

Les trois cas ont donc été validés. Ceci conclut une série de test permettant de valider la véracité des équations posées dans l'algorithme itératif. Ces tests permettent de confirmer que les opérations se comportent de la façon souhaitée. Ils ont aussi permis de tester l'outil GASTA dans des conditions se rapprochant de situation qu'on retrouve dans des programmes réels. La prochaine section est l'étude du comportement de GASTA dans une contexte d'analyse d'un programme réel.

5.4 Cas réel: La librairie GLib

Traiter un cas réel permets de montrer l'intérêt de l'outil développé durant cette maîtrise. Les tests ont été réalisés sur la librairie *glib* qui est utilisée, entre autre, dans les programmes développés au sein du projet GNOME. La prochaine section est une explication des différentes étapes ainsi qu'un présentation des résultats obtenus lors de cette analyse.

5.4.1 L'analyse par GASTA

Cette analyse se déroule en deux étapes. Une première étape va consister à exécuter GASTA sur le fichier C contenant l'implantation de *g_malloc()*. La seconde étape va être l'interprétation de l'analyse de GASTA. Cette seconde étape devrait être automatique mais, compte tenu de quelques problèmes pour récupérer certaines informations comme par exemple l'endroit exact d'une déclaration si celle-ci fait partie d'une structure, on doit faire cette étape manuellement.

La première étape est la génération de l'AST par GCC avec le commande :

```
maranello[GCC]:~/projetMaitrise/tests/glib-1.2.10 $ g++ -fdump-tree-original -c gmem.c
gmem.c: In function 'GMemChunk* g_mem_chunk_new(gchar*, int, long unsigned int,
   int)':
gmem.c:495: 'SIZEOF_LONG' undeclared (first use this function)
gmem.c:495: (Each undeclared identifier is reported only once for each function
   it appears in.)
gmem.c: In function 'void* g_mem_chunk_alloc(GMemChunk*)':
gmem.c:580: invalid conversion from 'void*' to 'GMemArea*'
gmem.c: In function 'void g_mem_chunk_free(GMemChunk*, void*)':
```

```
gmem.c:730: invalid conversion from 'void*' to 'GMemArea*'
gmem.c: In function 'void g_mem_chunk_clean(GMemChunk*)':
gmem.c:769: invalid conversion from 'void*' to 'GMemArea*'
```

Il en résulte quelques messages d'erreurs mais le fichier *gmem.c.original* est tout de même généré[2]. Maintenant il est possible d'utiliser GASTA pour analyser les fonctions se trouvant dans le fichier *gmem.c*.

```
maranello[GCC]:~/projetMaitrise/tests/glib-1.2.10 $ gasta -t dot gmem.c.original
<BEGIN gasta-2.1.1>
Recovering options ...                  ok
Open gmem.c.original ...                ok
Start parse input file ...              ok
Generate graph ...                      ok
Initialisation of ascii builder ...     ok
Initialisation of annot builder ...     ok
Initialisation of the visitor ...       ok
Start visiting graph ...                ok
Start iterative algorithm :


 There were 2 passe(s)
 - Analysis of g_malloc :
        Return blocks are annotated as follow :
        [BB2]
                return->expr->op1 = @57
        [BB7]    { @35 <- @0 }
                return->expr->op1 = @77
 There were 2 passe(s)
 - Analysis of g_malloc0 :
        Return blocks are annotated as follow :
        [BB2]
```

[2]C'est une version 3.1.1 patchée avec la rustine fournie sur le site http://gasta.sf.net

```
                return->expr->op1 = @57
        [BB7]    { @35 <- @0 }
                return->expr->op1 = @79
 There were 2 passe(s)
...
----8<-----
 Cette partie est coupée mais elle présente les résultats de l'analyse
 pour chacune des fonctions de la même façon que pour g_malloc ou g_malloc0
----8<-----
...
iterative algorithm done
******
DURATION = 0.164506 us
******
Start automatic annotations ...          ok
Dump control flow graph ...              ok
Destruction of visitor ...               ok
Destruction of ascii builder ...         ok
Destruction of annot builder ...         ok
Close input file ...                     ok
Release the memory ...                   ok
<END>
```

Durant cette analyse GASTA génère de nombreux fichiers puisque l'option "-t dot" a pour but de créer un fichier au format dot pour chacune des fonctions (soit 22 pour le fichier *gmem.c*). Ces fichiers contiennent une représentation graphique du CFG comme le montre la figure III.1 de l'annexe III à la page 146. Sur la sortie standard GASTA écrit un certain nombre de messages. Comme cela a été dit en introduction de cette analyse d'un cas réel, on s'intéresse à la fonction *g_malloc()*. On

constate qu'il y a deux points de retour. La première instruction *return* retourne une expression numérotée @57. En regardant les tables contenant les informations se rapportant à la fonction *g_malloc()* on remarque que rien ne correspond à la valeur @57. GASTA doit alors accéder directement à l'AST. Ceci est possible car l'implantation est telle que les numéros de type @X correspondent au nœud numéro X de l'AST généré par GCC. GASTA peut alors déduire de cette information que la valeur associée à @57 correspond à l'entier 0. Or cette fonction retourne un pointeur et l'entier 0 peut donc être interprété comme un pointeur nul. La fonction *g_malloc()* peut donc retourner un pointeur nul. Il n'est alors pas nécessaire d'étudier le second *return* car il suffit d'un seul pouvant retourner un pointeur possiblement nul pour pouvoir annoter la fonction. On remarque également que la fonction *g_malloc0()* peut aussi retourner un pointeur possiblement nul et elle peut donc être annotée. Cette étape est faite automatiquement. Sans GASTA, l'utilisateur aurait dû exécuter SPLint, regarder les messages générés, annoter la fonction *g_malloc()* comme pouvant retourner un pointeur nul (indication générée par GASTA), exécuter SPLint à nouveau. Cette nouvelle annotation provoque de nouveaux messages puisque maintenant SPLint considère que *g_malloc()* peut retourner *NULL*. Si on pousse un peu plus l'analyse, on se rend compte qu'en fait, la fonction *g_malloc()* peut retourner un pointeur nul seulement si l'argument qui lui est passé est nul. En effet, si *malloc()* retourne un pointeur nul, alors cette erreur est interceptée par *g_malloc()* et la fonction génère une exception qui est

prise en charge. Cette constatation a alors conduit a se poser la question suivante:

"GASTA a été construit dans le souci de faciliter l'intégration d'autres analyses. Ne serait-il pas possible de créer un module permettant de gérer ce cas spécifique?"

En effet, il serait très intéressant si, avec un minimum d'efforts, l'utilisateur pouvait utiliser GASTA pour l'adapter, grâce à un module, à un problème particulier. La section suivante montre comment cela est possible avec GASTA.

5.4.2 GASTA, un outil modulaire

GASTA permet de créer des crochets au niveau de l'ASG généré par GCC. Cela signifie qu'en tout point de l'ASG il est possible d'insérer des appels à ses propres fonctions et ce sans modification du code de GASTA. La création d'un module est réalisée en trois étapes. Tout d'abord on doit se poser la question sur ce que doit faire le module. Ensuite, on crée un fichier de base et finalement, on complète les fonctions de façon à réaliser les opérations mises en œuvre dans la première étape. Concrètement voici ce que cela signifie pour le cas où on souhaite connaître la valeur d'un argument passé à la fonction *g_malloc()*.

Deux informations sont nécessaires. Premièrement, on doit identifier la fonction *g_malloc()*; ensuite, il faut identifier l'argument de cette fonction. Si on regarde la figure IV.1 de l'annexe IV à la page 153, on remarque deux choses. Tout d'abord, le nom de la fonction est disponible au niveau du nœud *identifier_node* de l'ASG. Ensuite, La valeur de l'argument passé à la fonction est disponible juste après le

nœud *tree_list*. Donc, pour récupérer l'information il faut aller chercher le nom de l'*identifier_node* et la valeur qu'on retrouve après *tree_list*. Ici, il faut tout de même faire attention à ne pas regarder tous les nœuds du type *identifier_node* car tous ne correspondent pas à un appel de fonction. Pour résoudre ce problème, il suffit d'ajouter un booléen au niveau du nœud *call_expr* qui indiquera si oui ou non nous sommes dans l'appel d'une fonction. Ceci est possible car dans GASTA les fonctions associées à un nœud sont appelées deux fois. Une première fois lorsque on commence la visite du nœud et une seconde fois lorsque on termine cette visite.

Maintenant, la seconde étape consiste à générer le squelette du module. GASTA fournit deux fichiers, *templateBuilder.c* et *templateBuilder.h* qui définissent la structure d'un module. Il suffit donc de copier ces fichiers en les nommant par exemple, *mon_module.c* et *mon_module.h*. Ensuite, il suffit de remplacer dans ces fichiers toutes les occurrences du mot *template* par *mon_module*. Une fois cette étape faite, l'utilisateur dispose du squelette d'un module. Il reste donc à "remplir" les fonctions avec le code approprié.

De base, il existe trois fonctions dans le module. Une fonction d'initialisation, une fonction de destruction et une fonction appelée au tout début de la visite. Les fonctions à définir dans le module correspondent à ce qu'on souhaite faire selon le type de nœud de l'ASG. Pour reprendre le cas où on souhaite détecter la fonction *g_malloc()*, il suffit d'ajouter un crochet au nœud *identifier_node* et, dans la fonction associée à ce nœud, on peut tester si le champ *str:* correspond

bien à *g_malloc*. Si c'est le cas alors on met un drapeau à 1 pour indiquer qu'on se trouve dans *g_malloc*, sinon le drapeau reste à 0. GASTA sait quelles sont les fonctions à appeler pour les différents types de nœuds car chacun des modules doit enregistrer ses fonctions en fonction du nom du nœud. L'ensemble du code du module permettant de détecter si le paramètre passé à la fonction *g_malloc()* est nul est donné en annexe IV à la page 147.

5.5 Analyse de la performance

L'objectif de cette analyse est l'évaluation de la complexité de l'algorithme itératif pour l'annotaion développé dans GASTA. La complexité qu'on souhaite calculer est celle concernant le temps d'analyse en fonction de la taille de la fonction et de la taille du fichier. La machine de test est un pentium III 1GHz avec 1Go de mémoire RAM et 3Go de swap. Les fichiers utilisés pour le test sont ceux de la glib version 1.2.10. Tous les fichiers ont été compilés avec la commande suivante : g++ -fdump-tree-original -O0 -c <fichier.c>

Le tableau 5.2 montre que le temps d'analyse moyen est de $0.01s$ par ligne de code. Ce n'est pas le temps qui est important ici mais la complexité. Les résultats montrent que la complexité de l'analyse semble indiquer que le temps de traitement croît de manière linéaire avec le volume de code à analyser, ce qui est conforme aux attentes puisque l'algorithme utilisé s'effectue sur des blocs de

Fichiers testés	Nombre de fonctions	Nombre de ligne de code	Durée de l'analyse
garray.c	26	335	0.24s
gcache.c	8	144	0.11s
gcompletion.c	7	221	0.09s
gdataset.c	18	455	0.28s
gdate.c	46	948	0.60s
gerror.c	4	227	0.05s
ghash.c	15	286	0.17s
ghook.c	23	488	0.31s
giochannel.c	9	77	0.08s
giounix.c	12	235	0.10s
giowin32.c	43	830	0.36s
glist.c	28	532	0.22s
gmain.c	38	1022	0.40s
gmem.c	22	722	0.22s
gmessages.c	15	601	0.16s
gmutex.c	6	124	0.07s
gnode.c	37	773	0.38s
gprimes.c	1	48	0.01s
grel.c	20	323	0.25s
gslist.c	26	470	0.20s
gstrfuncs.c	24	1280	0.21s
gstring.c	25	349	0.25s
gtimer.c	6	132	0.06s
gtree.c	27	592	0.25s
gutils.c	24	696	0.19s
Total (25 fichiers)	510	11910	5.23s

Tableau 5.2: Évaluation de la performance de l'algorithme

base et que le nombre de blocs de base croît avec le nombre de lignes de code. À l'intérieur de chaque bloc, l'algorithme effectue un nombre d'opérations sur des ensembles dépendant de la quantité de pointeurs dans une fonction. Cette quantité est négligeable par rapport au nombre de ligne de code et donc aux nombres de blocs de base. Ainsi, il semble normal que la complexité de l'analyse soit linéaire comme on le voit sur les figures 5.30 et 5.31. Les droites présentées sur les figures sont celles obtenues en utilisant les différents totaux.

5.6 Discussion

L'analyse d'un cas réel a permis de mettre en évidence deux aspects très intéressants de GASTA. En effet, dans un premier temps GASTA a permis d'automatiser l'annotation de la fonction *g_malloc()*. Ensuite, cette nouvelle annotation a introduit de nouveaux messages d'erreurs et ces nouveaux messages ont alors pu être traités en créant un module répondant précisément aux besoins de la nouvelle analyse. Ces deux points montrent l'intérêt d'un tel outil. La prochaine et dernière partie est un récapitulatif des objectifs ainsi qu'un bilan du travail effectué durant cette maîtrise recherche.

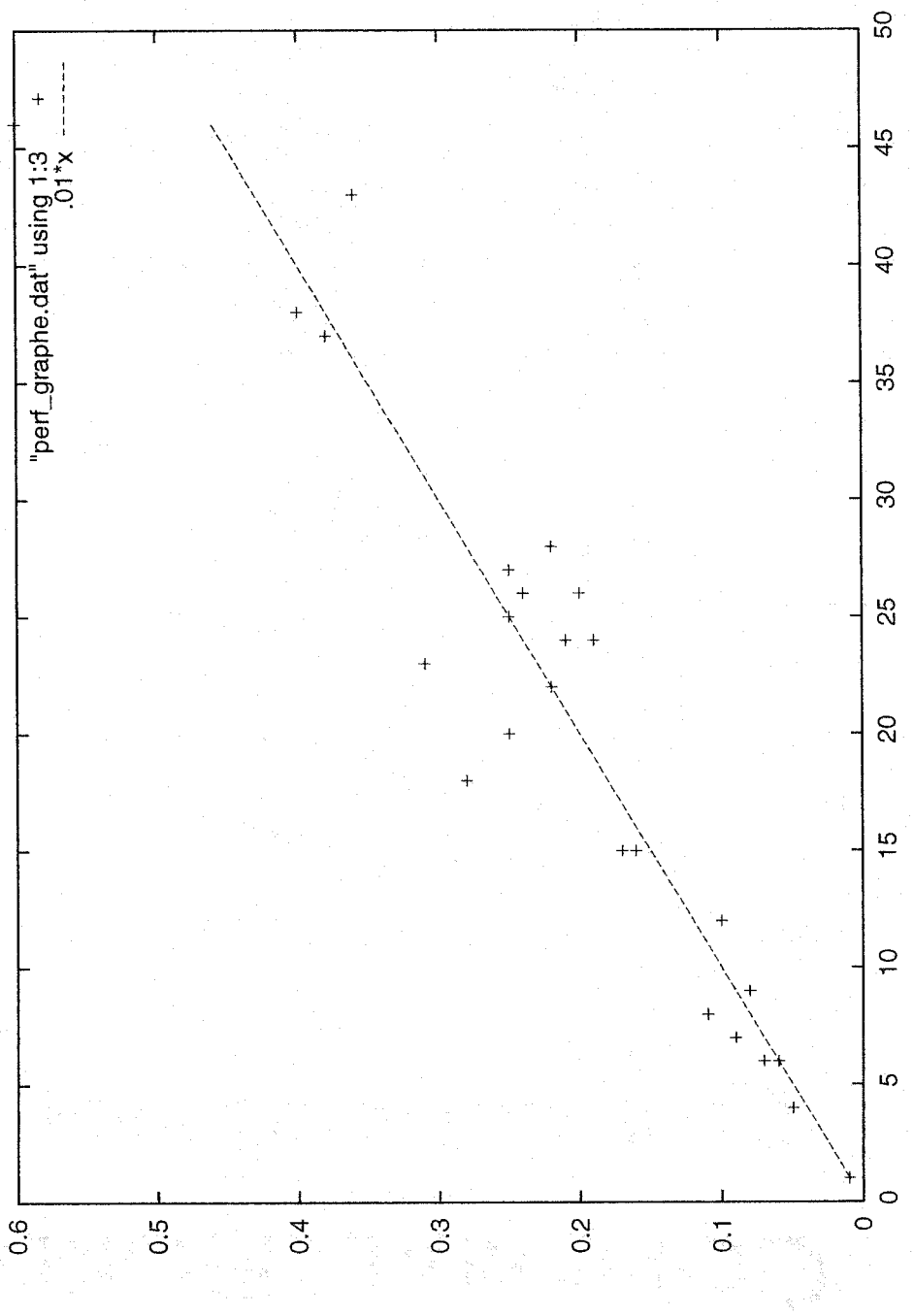


Figure 5.30: Temps d'exécution en fonction du nombre de fonction

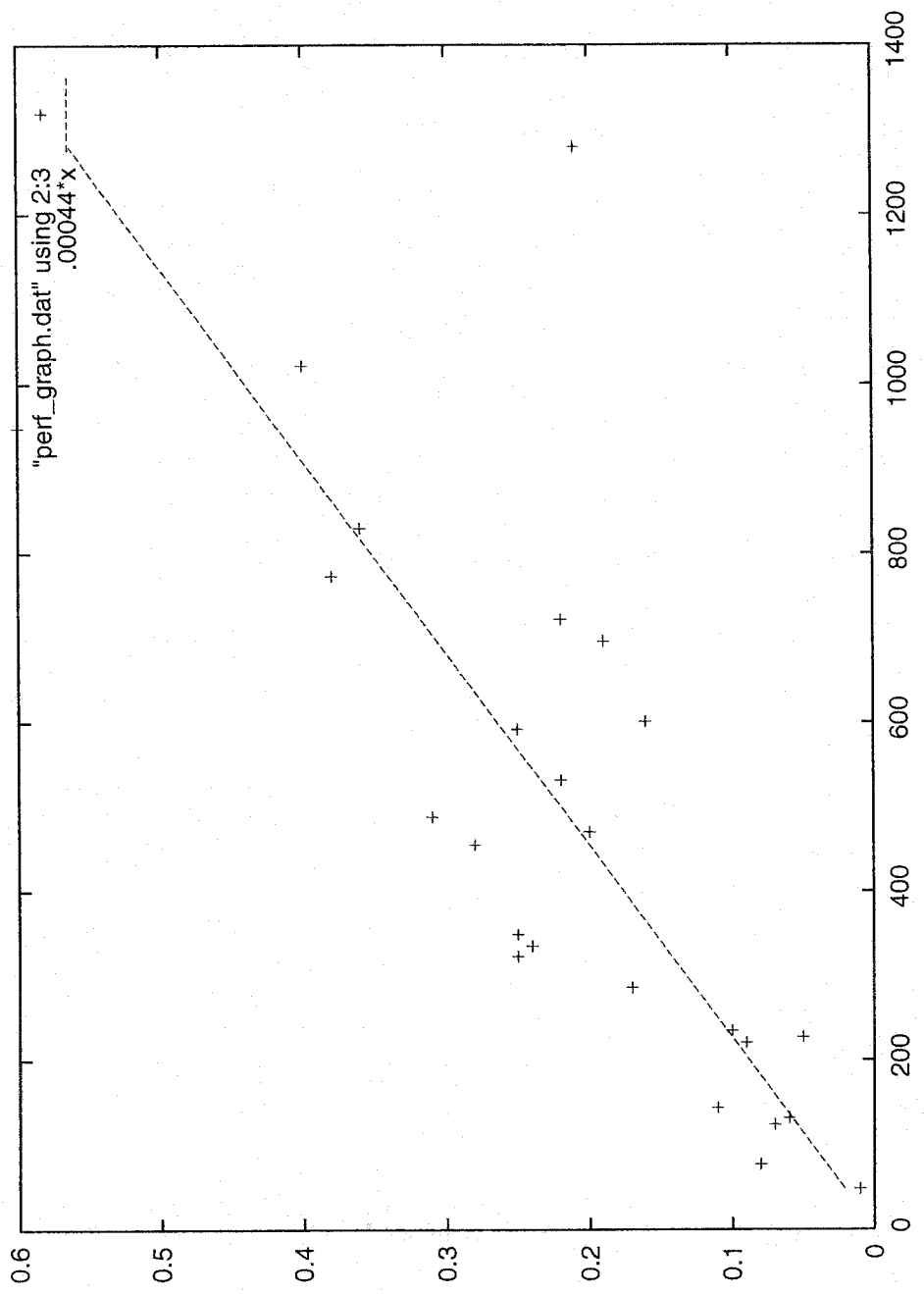


Figure 5.31: Temps d'exécution en fonction du nombre de ligne de code

CHAPITRE 6

CONCLUSION

6.1 Rappel des objectifs

En observant les outils disponibles dans le domaine de l'analyse statique et en comparant ces outils aux besoins actuels du marché, il apparaît qu'il est nécessaire de détecter les erreurs de programmation le plus rapidement possible dans le processus de développement. Malheureusement, les outils d'analyse statique s'avèrent trop imprécis pour être utilisables. Diverses méthodes ont été élaborées afin de remédier à ce problème. Parmi les différentes méthodes existantes, l'outil *SPLint* propose la possibilité d'annoter le code source afin d'accroître la précision de l'analyse. Son inconvénient majeur était la difficulté de trouver l'information pertinente parmi le nombre important de faux messages générés.

L'objectif était donc fixé, fournir un outil permettant d'automatiser dans la mesure du possible ces annotations qui autrement étaient à la charge de l'utilisateur.

6.2 Discussion

Le bilan est difficile à tirer puisque le travail était ambitieux et malheureusement l'implémentation de GASTA n'est pas complète. Cependant un certain nombre de concepts ont été développés et testés. GASTA regroupe en effet un ensemble

de fonctions qui permettent à partir de l'ASG fourni par GCC de bâtir un CFG en mémoire et d'extraire des informations pertinentes de cet ASG. Ces fonctions font partie de l'analyse lexicale, de l'analyse syntaxique, du visiteur et enfin du constructeur.

Jusqu'à la construction du CFG, GASTA est parfaitement fonctionnel et a été testé sur plusieurs programmes disponibles comme *sendmail-8.12.2, textutils-2.0.21* ou encore *bind-9.2.0.* Les CFG extraits à partir de ces différents programmes n'a pas été présentés dans ce mémoire car les graphes sont trop volumineux pour être visualisés sur une feuille de papier.

Concernant l'analyse proprement dite, un premier bilan peut être fait. L'algorithme itératif, qui évalue si une fonction retourne un pointeur possiblement nul (dans le cas où une fonction retourne un pointeur), a bien été implémenté et les algorithmes ont été validés. L'étude de l'analyse sur la librairie développée par GNOME a aussi permis de montrer les avantages que peut apporter GASTA, l'annotation automatique et la modularité. Concernant l'annotation automatique du code source, qui était notre objectif, elle n'est pas encore effective. Il reste des problèmes pour récupérer certaines informations (par exemple sur les structures utilisées dans un programme). Les tests ont tout de même prouvé la capacité à extraire l'information concernant les pointeurs nuls. En ajoutant à GASTA d'autres analyses permettant d'ajouter de nouvelles annotations on pourrait obtenir des résultats très intéressants.

Finalement, l'outil développé dans ce mémoire est un premier pas vers l'annotation automatique du code source et il offre une conception modulaire qui permet à l'utilisateur d'ajouter ses propres analyses.

6.3 Améliorations futures

Trois améliorations sont possibles. Ce sont des points qui ont été abordés dans ce mémoire mais qui n'ont pas été implémentés durant ce travail de recherche.

Tout d'abord, il serait intéressant de ne pas annoter les fonctions dans l'ordre de leur déclaration comme c'est actuellement le cas mais il serait plus judicieux de les annoter en tenant compte des dépendances du graphe d'appel. Une solution à ce problème a été présentée dans la section 3.3. Même si cette solution n'est certainement pas optimale elle peut servir de base à d'autres réflexions. Ensuite, il faudrait annoter de façon complètement automatiquement le code source et enfin l'ajout de nouvelles annotations permettraient d'accroître l'efficacité de GASTA.

BIBLIOGRAPHIE

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, tools.* Addison-Wesley, 1986.

[2] R. Z. Altucher and W. Landi. An extended form of must alias analysis for dynamic allocation. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 74–84. ACM Press, 1995.

[3] B. A.M. and S. H.J. Does apl really need run time checking?, 1974.

[4] L. Anderson. *Program Analysis and Specialization for the C programming Language.* PhD thesis, University of Copenhagen, 1994.

[5] F. Bourdoncle. Assertion-based debugging of imperative programs by abstract interpretation. In I. Sommerville and M. Paul, editors, *Proceedings of the Fourth European Software Engineering Conference*, pages 501–516. Springer-Verlag, 1993.

[6] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, Charleston, South Carolina, 1993.

[7] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. *20th ACM Transactions on Programming Languages and Systems*, 1993.

[8] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[10] D. L. Detlefs. An overview of the extended static checking system. In *Proceedings of The First Workshop on Formal Methods in Software Practice*, pages 1–9, 1996.

[11] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. *ACM SIGPLAN Notices*, 29(6):230–241, 1994.

[12] G. Developers. Gnome 2.0 api reference, http://developer.gnome.org/doc/api/.

[13] D. Evans. Splint manual, version 3.0.6, February 2001.

[14] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 87–96, 1994.

[15] P. Feautrier. Dataflow analysis of scalar and array references. In *Ini. Journal of Parallel Programming*, pages 23–53, 1991.

[16] F. S. Foundation. Gcc 3.0.2 manual, http://gcc.gnu.org/onlinedocs/gcc-3.0.2/gcc.html, 2001.

[17] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21, 1999.

[18] M. Hind and A. Pioli. Which pointer analysis should i use? In *International Symposium on Software Testing and Analysis*, pages 113–123, 2000.

[19] D. Hofstadter. *Gödel, Escher, Bach.* InterEditions, 1960.

[20] M. Jones. *Program Flow analysis: theory and applications.* Prentice-Hall, 1981.

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

[22] G. A. Kildall. A unified approach to global program optimization. *ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[23] W. Landi. Undecidability of static analysis, 1992.

[24] M. E. Lesk and E. Schmidt. The lex and yacc page, http://www.combo.org/lex_yacc_page/.

[25] B. Malenfant, G. Antoniol, E. Merlo, and M. Dagenais. Context sensitive flow analysis to detect blocked statements. *ICSM 2001*, 2001.

[26] E. Merlo, G. Antoniol, P. Brunelle, and M. Dagenais. Fast flow analysis to compute fuzzy estimates of risk levels. Technical Report TR2002-xxx, 2002.

[27] A. Pioli. Conditional pointer aliasing and constant propagation. Master's thesis, SUNY at New Paltz, 1998.

[28] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, September 1994.

[29] K. H. Rosen. *Mathématiques discrètes.* Chenelière/McGraw-Hill, 1998.

[30] N. Schwartz. Steering clear of triples: Deriving the control flow graph directly from the abstract syntax tree in C programs. Technical Report TR1998-766, 16, 1998.

[31] B. Steensgaard. Points-to analysis in almost linear time. *23rd ACM SIGPLAN Notices*, pages 32–41, 1996.

[32] R. Strom and D. Yellin. Extending typestate checking using conditional liveness analysis. *Software Engineering, IEEE Transactions*, 19:478–485, 1993.

[33] C. S. L. S. University. An overview of the suif2 compiler infrastructure, http://suif.stanford.edu/suif/suif2/doc-2.2.0-4/.

[34] H. W.E. Comments analysis and programming errors. *Software Engineering, IEEE Transactions*, 16:72–81, 1990.

[35] W.Landi and B.Ryder. A safe approximate algorithm for inter-procedural pointer aliasing. *ACM SIGPLAN Notices*, 1992.

Annexe I

Exemple d'une analyse

I.1 Code source

```
#include <stdlib.h>

static int *make_clone (int *ptr);

int
main ()
{

  int *ptr;
  int *clone = NULL;

  ptr = (int *) malloc (sizeof (int));
  if (ptr == NULL)
    return 1;
  else
    {
      *ptr = 5;
      clone = make_clone (ptr);

      if (clone == NULL)
        {
          free (ptr);
          return 2;
```

```
        }
      else
        {
          free (ptr);
          free (clone);
          return 0;
        }
    }

}

static int *
make_clone (int *ptr)
{

  int *ptrTmp;

  if (ptr == NULL)
    return NULL;
  else
    {
      ptrTmp = (int *) malloc (sizeof (int));
      if (ptrTmp == NULL)
        return NULL;
      else
        {
          *ptrTmp = *ptr;
          return ptrTmp;
        }
    }
```

```
}
```

I.2 Résumé de l'analyse

```
(guill@maranello:~/projetMaitrise) $ gasta -t dot clone.c.original
<BEGIN gasta-2.1.1>
Recovering options ...                  ok
Open clone.c.original ...               ok
Start parse input file ...              ok
Generate graph ...                      ok
Initialisation of ascii builder ...     ok
Initialisation of annot builder ...     ok
Initialisation of the visitor ...       ok
Start visiting graph ...                ok
Start iterative algorithm :


 There were 2 passe(s)
 - Analysis of main :
        Return blocks are annotated as follow :
        [BB4]    { @28 <- @0 }
                 return->expr->op1 = @73
        [BB10]   { @28 <- @0 }
                 return->expr->op1 = @114
        [BB13]   { @28 <- @0 }
                 return->expr->op1 = @26
 There were 2 passe(s)
 - Analysis of make_clone :
        Return blocks are annotated as follow :
        [BB2]
```

```
                return->expr->op1 = @52
        [BB6]    { @29 <- @0 }
                return->expr->op1 = @84
        [BB8]    { @29 <- @0 }
                return->expr->op1 = @29

iterative algorithm done
******
DURATION = 0.057635 us
******
Start automatic annotations ...          ok
Dump control flow graph ...              ok
Destruction of visitor ...               ok
Destruction of ascii builder ...         ok
Destruction of annot builder ...         ok
Close input file ...                     ok
Release the memory ...                   ok
<END>
```

I.3 Le fichier *annot_builder*

```
Information about type
----------------------

;; Function int main() (main)
[Node 111]      name=pointer_type       ptd=@120
[Node 46]       name=pointer_type       ptd=@15
[Node 68]       name=function_type
[Node 123]      name=function_type
[Node 3]        name=function_type
[Node 15]       name=void_type
[Node 60]       name=function_type
```

```
[Node 49]       name=boolean_type

[Node 93]       name=function_type

[Node 6]        name=integer_type

[Node 62]       name=integer_type

[Node 31]       name=pointer_type       ptd=@6

[Node 53]       name=pointer_type       ptd=@60

[Node 10]       name=integer_type

[Node 87]       name=pointer_type       ptd=@93

[Node 120]      name=function_type


;; Function int* make_clone(int*) (_Z10make_clonePi)

[Node 67]       name=pointer_type       ptd=@72

[Node 13]       name=integer_type

[Node 24]       name=void_type

[Node 79]       name=function_type

[Node 14]       name=integer_type

[Node 4]        name=function_type

[Node 38]       name=boolean_type

[Node 60]       name=pointer_type       ptd=@24

[Node 72]       name=function_type

[Node 8]        name=pointer_type       ptd=@14

[Node 74]       name=integer_type


Information about variable

--------------------------


;; Function int main() (main)

[Node 33]  source: clone.c:10 | name: clone | type: @31 | used: 1

        -> local variable

        -> not initialized
```

```
[Node 28]  source: clone.c:9 | name: ptr | type: @31 | used: 1
        -> local variable
        -> not initialized


;; Function int* make_clone(int*) (_Z10make_clonePi)
[Node 29]  source: clone.c:39 | name: ptrTmp | type: @8 | used: 1
        -> local variable
        -> not initialized


Information about parameters
----------------------------

;; Function int main() (main)

;; Function int* make_clone(int*) (_Z10make_clonePi)
[Node 5]  source: clone.c:37 | name: ptr | type: @8 | lused: 1 | rused: 1

Information about functions
---------------------------

;; Function int main() (main)
[Node 88]       name=make_clone         source=clone.c:3        annotated=0
[Node 1]        name=main       source=clone.c:7        annotated=0
[Node 112]      name=free       source=stdlib.h:538     annotated=1
[Node 54]       name=malloc     source=stdlib.h:527     annotated=1

;; Function int* make_clone(int*) (_Z10make_clonePi)
[Node 1]        name=make_clone         source=clone.c:37       annotated=0
[Node 68]       name=malloc     source=stdlib.h:527     annotated=1
```

I.4 L'ASG

```
;; Function int main() (main)

;; enabled by -dump-tree-original


@1      function_decl    name: @2       type: @3       srcp: clone.c:7
                         C              extern         body: @4
@2      identifier_node  strg: main     lngt: 4
@3      function_type    size: @5       algn: 64       retn: @6
                         prms: @7
@4      compound_stmt    line: 33       body: @8       next: @9
@5      integer_cst      type: @10      low : 64
@6      integer_type     name: @11      size: @12      algn: 32
                         prec: 32       min : @13      max : @14
@7      tree_list        valu: @15
@8      scope_stmt       line: 7        begn           clnp
                         next: @16
@9      return_stmt      line: 33       expr: @17
@10     integer_type     name: @18      size: @5       algn: 64
                         prec: 36       unsigned       min : @19
                         max : @20
@11     type_decl        name: @21      type: @6       srcp: <internal>:0
@12     integer_cst      type: @10      low : 32
@13     integer_cst      type: @6       high: -1       low : -2147483648
@14     integer_cst      type: @6       low : 2147483647
@15     void_type        name: @22      algn: 8
@16     compound_stmt    line: 7        body: @23      next: @24
@17     init_expr        type: @6       op 0: @25      op 1: @26
@18     identifier_node  strg: bit_size_type           lngt: 13
@19     integer_cst      type: @10      low : 0
@20     integer_cst      type: @10      high: 15       low : -1
@21     identifier_node  strg: int      lngt: 3
```

```
@22     type_decl       name: @27      type: @15      srcp: <internal>:0
@23     decl_stmt       line: 9        decl: @28      next: @29
@24     scope_stmt      line: 33       end            clnp
@25     result_decl     type: @6       scpe: @1       srcp: clone.c:7
                        size: @12      algn: 32
...

;; Function int main() (main)

(B)  FUNCTION_DECL  [node = 1,  depth = 0, source = clone.c:7]
    (B)  IDENTIFIER_NODE  [node = 2,  depth = 1, name = main]
    (E)  IDENTIFIER_NODE  [node = 2,  depth = 1
    (B)  FUNCTION_TYPE  [node = 3,  depth = 1]
        (B)  INTEGER_TYPE  [node = 6,  depth = 2]
            (B)  TYPE_DECL  [node = 11,  depth = 3, source = <internal>:0]
                (B)  IDENTIFIER_NODE  [node = 21,  depth = 4, name = int]
                (E)  IDENTIFIER_NODE  [node = 21,  depth = 4
                (=)  INTEGER_TYPE  [node = 6, depth = 4]
            (E)  TYPE_DECL  [node = 6,  depth = 3
            (B)  INTEGER_CST  [node = 12,  depth = 3, low = 32]
                (B)  INTEGER_TYPE  [node = 10,  depth = 4]
                    (B)  IDENTIFIER_NODE  [node = 18,  depth = 5, name = bit_size_type]
                    (E)  IDENTIFIER_NODE  [node = 18,  depth = 5
                    (B)  INTEGER_CST  [node = 5,  depth = 5, low = 64]
                        (=)  INTEGER_TYPE  [node = 10, depth = 6]
                    (E)  INTEGER_CST  [node = 10,  depth = 5
                    (B)  INTEGER_CST  [node = 19,  depth = 5, low = 0]
                        (=)  INTEGER_TYPE  [node = 10, depth = 6]
                    (E)  INTEGER_CST  [node = 10,  depth = 5
                    (B)  INTEGER_CST  [node = 20,  depth = 5, low = -1]
```

```
                    (=)  INTEGER_TYPE  [node = 10, depth = 6]
                (E)  INTEGER_CST  [node = 10,  depth = 5
            (E)  INTEGER_TYPE  [node = 10,  depth = 4
        (E)  INTEGER_CST  [node = 10,  depth = 3
        (B)  INTEGER_CST  [node = 13,  depth = 3, low = -2147483648]
            (=)  INTEGER_TYPE  [node = 6, depth = 4]
        (E)  INTEGER_CST  [node = 6,  depth = 3
        (B)  INTEGER_CST  [node = 14,  depth = 3, low = 2147483647]
            (=)  INTEGER_TYPE  [node = 6, depth = 4]
        (E)  INTEGER_CST  [node = 6,  depth = 3
    (E)  INTEGER_TYPE  [node = 6,  depth = 2
    (B)  TREE_LIST  [node = 7,  depth = 2]
        (B)  VOID_TYPE  [node = 15,  depth = 3]
            (B)  TYPE_DECL  [node = 22,  depth = 4, source = <internal>:0]
                (B)  IDENTIFIER_NODE  [node = 27,  depth = 5, name = void]
                (E)  IDENTIFIER_NODE  [node = 27,  depth = 5
                (=)  VOID_TYPE  [node = 15, depth = 5]
            (E)  TYPE_DECL  [node = 15,  depth = 4
        (E)  VOID_TYPE  [node = 15,  depth = 3
    (E)  TREE_LIST  [node = 15,  depth = 2
    (=)  INTEGER_CST  [node = 5, depth = 2]
(E)  FUNCTION_TYPE  [node = 5,  depth = 1
(B)  COMPOUND_STMT  [node = 4,  depth = 1, line = 33]
    (B)  SCOPE_STMT  [node = 8,  depth = 2, line = 7]
        (B)  COMPOUND_STMT  [node = 16,  depth = 3, line = 7]
            (B)  DECL_STMT  [node = 23,  depth = 4, line = 9]
                (B)  VAR_DECL  [node = 28,  depth = 5, source = clone.c:9]
                    (B)  IDENTIFIER_NODE  [node = 30,  depth = 6, name = ptr]
                    (E)  IDENTIFIER_NODE  [node = 30,  depth = 6
                    (B)  POINTER_TYPE  [node = 31,  depth = 6]
                        (B)  INTEGER_CST  [node = 32,  depth = 7, low = 32]
```

```
                (=)  INTEGER_TYPE  [node = 10, depth = 8]
            (E)  INTEGER_CST  [node = 10,  depth = 7
            (=)  INTEGER_TYPE  [node = 6, depth = 7]
        (E)  POINTER_TYPE  [node = 6,  depth = 6
        (=)  FUNCTION_DECL  [node = 1, depth = 6]
        (=)  INTEGER_CST  [node = 32, depth = 6]
    (E)  VAR_DECL  [node = 32,  depth = 5
    (B)  DECL_STMT  [node = 29,  depth = 5, line = 10]
        (B)  VAR_DECL  [node = 33,  depth = 6, source = clone.c:10]
            (B)  IDENTIFIER_NODE  [node = 35,  depth = 7, name = clone]
            (E)  IDENTIFIER_NODE  [node = 35,  depth = 7
            (B)  INTEGER_CST  [node = 36,  depth = 7, low = 0]
                (=)  POINTER_TYPE  [node = 31, depth = 8]
            (E)  INTEGER_CST  [node = 31,  depth = 7
            (=)  POINTER_TYPE  [node = 31, depth = 7]
            (=)  FUNCTION_DECL  [node = 1, depth = 7]
            (=)  INTEGER_CST  [node = 32, depth = 7]
        (E)  VAR_DECL  [node = 32,  depth = 6
        (B)  EXPR_STMT  [node = 34,  depth = 6, line = 12]
...
```

I.5 Le CFG

Annexe II

Messages produits par GASTA

II.1 Propagation de l'information

```
(guill@maranello:~/projetMaitrise/) $ gasta -t dot propagation.c.original
<BEGIN gasta-2.0.1>
Recovering options ...                  ok
Open propagation.c.original ...         ok
Start parse input file ...              ok
Generate graph ...                      ok
Initialisation of ascii builder ...     ok
Initialisation of annot builder ...     ok
Initialisation of the visitor ...       ok
Start visiting graph ...                ok
Start iterative algorithm :


 There were 2 passe(s)
 - Analysis of main :
        Return blocks are annotated as follow :
        [BB10]   { @28 <- @-1 }  { @33 <- @-1 }
                return->expr->op1 = @36


iterative algorithm done
******
DURATION = 0.022868 us
******
Start automatic annotations ...         ok
Dump control flow graph ...             ok
```

```
Destruction of visitor ...              ok
Destruction of ascii builder ...        ok
Destruction of annot builder ...        ok
Close input file ...                    ok
Release the memory ...                  ok
<END>
```

II.2 Opérateur $\oplus$

II.2.1 Règle 1

II.2.1.1 Cas A

```
[guill@maranello:~/projetMaitrise/] $ gasta -t dot cas_a.c.original
<BEGIN gasta-2.0.1>
Recovering options ...                  ok
Open cas_a.c.original ...               ok
Start parse input file ...              ok
Generate graph ...                      ok
Initialisation of ascii builder ...     ok
Initialisation of annot builder ...     ok
Initialisation of the visitor ...       ok
Start visiting graph ...                ok
Start iterative algorithm :

 There were 2 passe(s)
 - Analysis of main :
        Return blocks are annotated as follow :
        [BB10]   { @41 <- @0 }
                 return->expr->op1 = @64

iterative algorithm done
```

```
******
DURATION = 0.021663 us
******
Start automatic annotations ...          ok
Dump control flow graph ...              ok
Destruction of visitor ...               ok
Destruction of ascii builder ...         ok
Destruction of annot builder ...         ok
Close input file ...                     ok
Release the memory ...                   ok
<END>
```

II.2.1.2 Cas B

```
[guill@maranello:~/projetMaitrise/] $ gasta -t dot cas_b.c.original
<BEGIN gasta-2.0.1>
Recovering options ...                   ok
Open cas_b.c.original ...                ok
Start parse input file ...               ok
Generate graph ...                       ok
Initialisation of ascii builder ...      ok
Initialisation of annot builder ...      ok
Initialisation of the visitor ...        ok
Start visiting graph ...                 ok
Start iterative algorithm :

 There were 2 passe(s)
 - Analysis of main :
        Return blocks are annotated as follow :
        [BB9]    { @31 <- @0 }
                 return->expr->op1 = @28
```

```
iterative algorithm done

******

DURATION = 0.021623 us

******

Start automatic annotations ...          ok

Dump control flow graph ...              ok

Destruction of visitor ...               ok

Destruction of ascii builder ...         ok

Destruction of annot builder ...         ok

Close input file ...                     ok

Release the memory ...                   ok

<END>
```

II.2.1.3 Cas C

```
[guill@maranello:~/projetMaitrise/] $ gasta -t dot cas_c.c.original

<BEGIN gasta-2.0.1>

Recovering options ...                   ok

Open cas_c.c.original ...                ok

Start parse input file ...               ok

Generate graph ...                       ok

Initialisation of ascii builder ...      ok

Initialisation of annot builder ...      ok

Initialisation of the visitor ...        ok

Start visiting graph ...                 ok

Start iterative algorithm :


 There were 2 passe(s)

 - Analysis of main :

        Return blocks are annotated as follow :

        [BB10]    { @41 <- @0 }

                  return->expr->op1 = @67
```

```
iterative algorithm done

******

DURATION = 0.000166 us

******

Start automatic annotations ...          ok

Dump control flow graph ...              ok

Destruction of visitor ...               ok

Destruction of ascii builder ...         ok

Destruction of annot builder ...         ok

Close input file ...                     ok

Release the memory ...                   ok

<END>
```

II.2.2 Règle 2

```
[guill@maranello:~/projetMaitrise/] $ gasta -t dot regle2.c.original

<BEGIN gasta-2.0.1>

Recovering options ...                   ok

Open regle2.c.original ...               ok

Start parse input file ...               ok

Generate graph ...                       ok

Initialisation of ascii builder ...      ok

Initialisation of annot builder ...      ok

Initialisation of the visitor ...        ok

Start visiting graph ...                 ok

Start iterative algorithm :


 There were 3 passe(s)

 - Analysis of main :

        Return blocks are annotated as follow :

        [BB9]    { @41 <- @47 }
```

```
                return->expr->op1 = @34


iterative algorithm done

******

DURATION = 0.021535 us

******

Start automatic annotations ...          ok

Dump control flow graph ...              ok

Destruction of visitor ...               ok

Destruction of ascii builder ...         ok

Destruction of annot builder ...         ok

Close input file ...                     ok

Release the memory ...                   ok

<END>
```

II.2.3 Règle 3

```
[guill@maranello:~/projetMaitrise/] $ gasta -t dot regle3.c.original

<BEGIN gasta-2.0.1>

Recovering options ...                   ok

Open regle3.c.original ...               ok

Start parse input file ...               ok

Generate graph ...                       ok

Initialisation of ascii builder ...      ok

Initialisation of annot builder ...      ok

Initialisation of the visitor ...        ok

Start visiting graph ...                 ok

Start iterative algorithm :


 There were 3 passe(s)

 - Analysis of main :

        Return blocks are annotated as follow :
```

```
        [BB9]    { @41 <- @47 }  { @41 <- @50 }
                 return->expr->op1 = @34

iterative algorithm done
******
DURATION = 0.022054 us
******
Start automatic annotations ...          ok
Dump control flow graph ...              ok
Destruction of visitor ...               ok
Destruction of ascii builder ...         ok
Destruction of annot builder ...         ok
Close input file ...                     ok
Release the memory ...                   ok
<END>
```

II.3 Opérateur $\ominus$

II.3.1 $(@X \leftarrow @0) \in Init\_null$

```
[guill@maranello:~/projetMaitrise/] $ gasta -t dot cas_a.c.original
<BEGIN gasta-2.0.1>
Recovering options ...                   ok
Open cas_a.c.original ...                ok
Start parse input file ...               ok
Generate graph ...                       ok
Initialisation of ascii builder ...      ok
Initialisation of annot builder ...      ok
Initialisation of the visitor ...        ok
Start visiting graph ...                 ok
Start iterative algorithm :
```

```
There were 2 passe(s)
- Analysis of main :
        Return blocks are annotated as follow :
        [BB22]   { @41 <- @0 }
                return->expr->op1 = @67


iterative algorithm done
******
DURATION = 0.019576 us
******
Start automatic annotations ...          ok
Dump control flow graph ...              ok
Destruction of visitor ...               ok
Destruction of ascii builder ...         ok
Destruction of annot builder ...         ok
Close input file ...                     ok
Release the memory ...                   ok
<END>
```

II.3.2 $(@X \leftarrow @ - 1) \in Init\_null$

```
[guill@maranello:~/projetMaitrise/] $ gasta -t dot cas_b.c.original
<BEGIN gasta-2.0.1>
Recovering options ...                   ok
Open cas_b.c.original ...                ok
Start parse input file ...               ok
Generate graph ...                       ok
Initialisation of ascii builder ...      ok
Initialisation of annot builder ...      ok
Initialisation of the visitor ...        ok
Start visiting graph ...                 ok
Start iterative algorithm :
```

```
There were 2 passe(s)
- Analysis of main :
        Return blocks are annotated as follow :
        [BB19]   { @46 <- @-1 }
                 return->expr->op1 = @71

iterative algorithm done
******
DURATION = 0.021461 us
******
Start automatic annotations ...          ok
Dump control flow graph ...              ok
Destruction of visitor ...               ok
Destruction of ascii builder ...         ok
Destruction of annot builder ...         ok
Close input file ...                     ok
Release the memory ...                   ok
<END>
```

II.3.3 $(@X \leftarrow @Y) \in Init\_null$

```
[guill@maranello:~/projetMaitrise/] $ gasta -t dot cas_c.c.original
<BEGIN gasta-2.0.1>
Recovering options ...                   ok
Open cas_c.c.original ...                ok
Start parse input file ...               ok
Generate graph ...                       ok
Initialisation of ascii builder ...      ok
Initialisation of annot builder ...      ok
Initialisation of the visitor ...        ok
Start visiting graph ...                 ok
```

```
Start iterative algorithm :


 There were 2 passe(s)

 - Analysis of main :

        Return blocks are annotated as follow :

        [BB20]   { @47 <- @-1 }  { @41 <- @47 }

                return->expr->op1 = @73


iterative algorithm done

******

DURATION = 0.021763 us

******

Start automatic annotations ...          ok

Dump control flow graph ...              ok

Destruction of visitor ...               ok

Destruction of ascii builder ...         ok

Destruction of annot builder ...         ok

Close input file ...                     ok

Release the memory ...                   ok

<END>
```

Annexe III

Analyse de la librairie glib

III.1 Le CFG

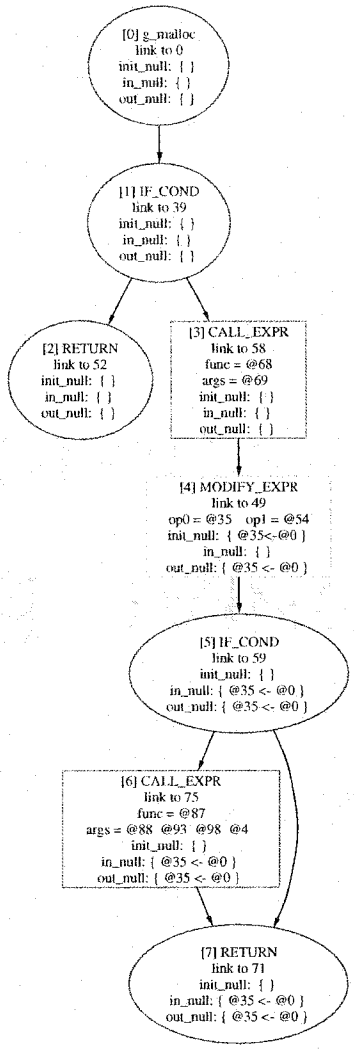


Figure III.1: CFG de g_malloc()

Annexe IV

Les modules dans GASTA

IV.1 ASG d'un appel de fonction

IV.2 Le module *g_malloc*

Le fichier gmallocPlugin.c :

```
#include <stdlib.h>
#include <assert.h>

#include "gastaCommon.h"
#include "builder.h"
#include "genGraph.h"
#include "gmallocPlugin.h"

static void start_gmp_build(build_parm_s * bp, const char *name);

static void gmp_call_expr(build_parm_s * bp, builder_s * bt);
static void gmp_identifier_node(build_parm_s * bp, builder_s * bt);
static void gmp_addr_expr(build_parm_s * bp, builder_s * bt);
static void gmp_tree_list(build_parm_s * bp, builder_s * bt);

void
gmp_builder_init(builder * b, char *filename)
{
    gmpb_parm_s *gmpbp = NULL;

    /* We initialize the function start_build */
```

```
    b->start_build = (void (*)(build_parm_s *, const char *)) start_gmp_build;
    /* We initialize the match_string_to_builder */
    b->match_string_to_builder = g_hash_table_new(g_str_hash, g_str_equal);
    if (!b->match_string_to_builder) {
        fprintf(stderr, "Unable to initialize match_string_to_builder\n");
        exit(EXIT_FAILURE);
    }
    /* We initialize the methods */
    default_builder_init(b);
    /* Declarations */
    g_hash_table_insert(b->match_string_to_builder, "call_expr", (void *) &gmp_call_expr);
    g_hash_table_insert(b->match_string_to_builder, "identifier_node", (void *) &gmp_identifier_node);
    g_hash_table_insert(b->match_string_to_builder, "addr_expr", (void *) &gmp_addr_expr);
    g_hash_table_insert(b->match_string_to_builder, "tree_list", (void *) &gmp_tree_list);
    /* Now, we initialize the attributes */
    if (!(b->parm.name = g_string_new("gmalloc plugin"))) {
        exit(EXIT_FAILURE);
    }
    if (!(b->parm.attributes = calloc(1, sizeof (gmpb_parm_s)))) {
        fprintf(stderr, "Can't allocate memory to b->parm.attributes\n");
        exit(EXIT_FAILURE);
    }
    gmpbp = (gmpb_parm_s *) b->parm.attributes;
    gmpbp->filename = g_string_new(filename);
    gmpbp->file = fopen(gmpbp->filename->str, "w");
    gmpbp->in_call = FALSE;
    gmpbp->in_gmalloc = FALSE;
    gmpbp->propagate = FALSE;
}

void
```

```
gmp_builder_destroy(builder * b)
{
    gmpb_parm_s *gmpbp = (gmpb_parm_s *) b->parm.attributes;

    g_hash_table_destroy(b->match_string_to_builder);
    g_string_free(b->parm.name, 1);
    g_string_free(gmpbp->filename, 1);
    fclose(gmpbp->file);
    free(gmpbp);
}

static void
start_gmp_build(build_parm_s * bp, const char *name)
{
    (void) bp; (void) name;
}

static void
gmp_call_expr(build_parm_s * bp, builder_s * bt)
{
    gmpb_parm_s *gmp_b = NULL;

    assert((bt) && (bp) && (bp->attributes));
    gmp_b = (gmpb_parm_s *) bp->attributes;
    switch (bt->builder_status) {
    case BUILDER_BEGIN:
        gmp_b->in_call = TRUE;
        break;
    case BUILDER_END:
        gmp_b->in_call = FALSE;
        break;
```

```
    }
}

static void
gmp_identifier_node(build_parm_s * bp, builder_s * bt)
{
    gmpb_parm_s *gmp_b = NULL;

    switch (bt->builder_status) {
    case BUILDER_BEGIN:
        assert((bt) && (bp) && (bp->attributes));
        gmp_b = (gmpb_parm_s *) bp->attributes;
        if (gmp_b->in_call) {
            tag_s *node = NULL;
            node = find_string_inside_list("strg:", bt->current_node_tags);
            if ((node && node->value))
                if (STREQ(node->value->str, "g_malloc"))
                    gmp_b->in_gmalloc = TRUE;
        }
        break;
    }
}

static void
gmp_addr_expr(build_parm_s * bp, builder_s * bt)
{
    gmpb_parm_s *gmp_b = NULL;

    switch (bt->builder_status) {
    case BUILDER_END:
        assert((bt) && (bp) && (bp->attributes));
```

```
        gmp_b = (gmpb_parm_s *) bp->attributes;
        if (gmp_b->in_gmalloc) {
            gmp_b->propagate = TRUE;
            gmp_b->in_gmalloc = FALSE;
        }
        break;
    }
}

static void
gmp_tree_list(build_parm_s * bp, builder_s * bt)
{
    gmpb_parm_s *gmp_b = NULL;

    switch (bt->builder_status) {
    case BUILDER_BEGIN:
        assert((bt) && (bp) && (bp->attributes));
        gmp_b = (gmpb_parm_s *) bp->attributes;
        if (gmp_b->propagate) {
            tag_s *node = NULL;

            node = find_string_inside_list("valu:", bt->current_node_tags);
            if ((node) && (node->value)) {
                fprintf(stderr, "g_malloc parameter is %s\n", node->value->str);
            }
            gmp_b->propagate = FALSE;
        }
        break;
    }
}
```

Le fichier gmallocPlugin.h :

```
#ifndef GMPBUILDER_H
#define GMPBUILDER_H

#include <glib.h>

typedef struct {

    FILE *file;
    GString *filename;

    int in_call;
    int in_gmalloc;
    int propagate;

} gmpb_parm_s;

void gmp_builder_init(builder * b, char *filename);
void gmp_builder_destroy(builder * b);

#endif
```

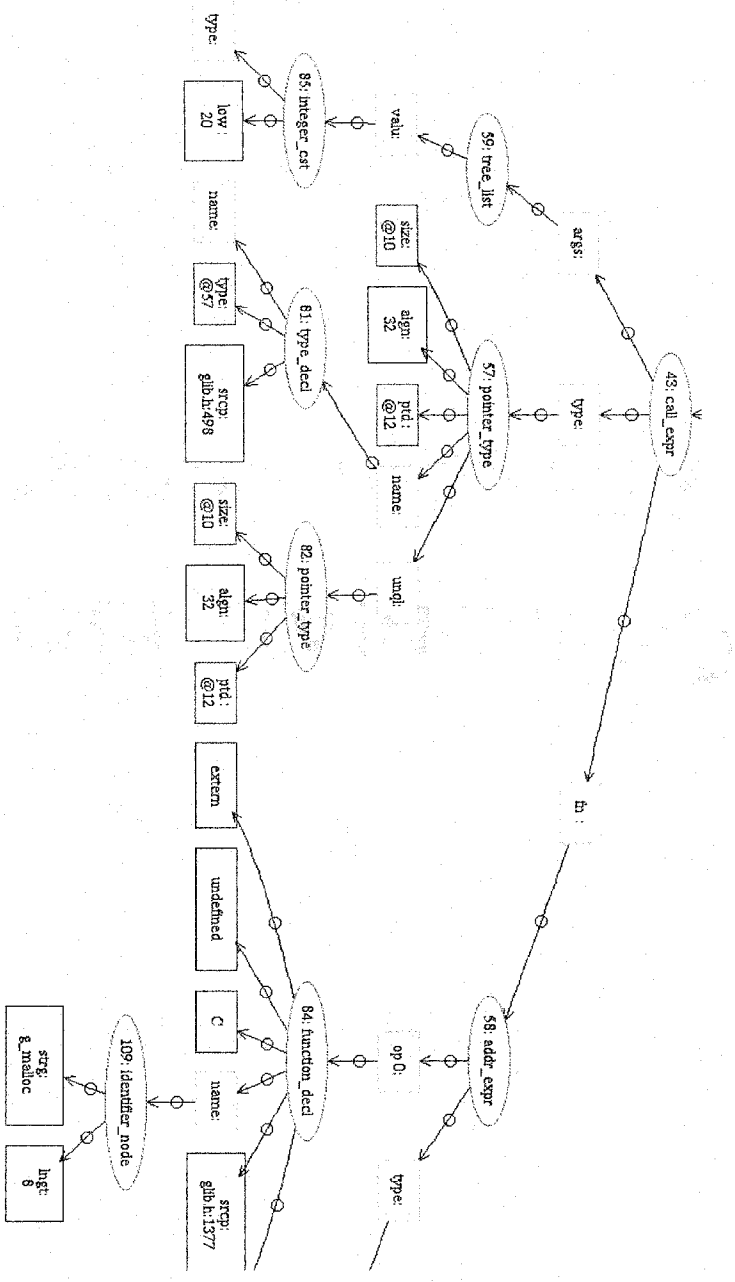


Figure IV.1: Patron d'un appel de fonction dans l'ASG