Licence d'Informatique

Rapport de stage

Laboratoire d'Informatique Fondamentale de Lille
Université Lille 1, Villeneuve d'Ascq

Christophe Bacara
Sous la responsabilité de Laure Gonnord
10 juillet 2013

Implémentation dans LLVM d'analyses de pointeurs

Table des matières

Table des figures

1 Introduction - Contexte

Le calcul scientifique, qui sert notamment à effectuer diverses simulations (météorologiques, physiques), ainsi qu'à analyser de grandes quantités de données, occupe une part importante de la recherche informatique actuellement.

Il est donc crucial d'améliorer le temps d'exécution des calculs, ainsi que de diminuer les ressources nécessaires pour ce calcul. Pour cela, il existe plusieurs types d'optimisations, dont celle à laquelle nous nous intéressons : l'optimisation statique, c'est à dire, effectuée à la compilation, et plus précisément, l'analyse de pointeurs.

L'objet de ce stage est de développer une analyse de pointeurs précise mais à faible coût, qui sera utilisée comme phase préliminaire à d'autres analyses et optimisations. L'implémentation de cet algorithme est fait dans le framework LLVM.

Concernant ce rapport, nous commencerons par justifier notre analyse dans le cadre des optimisations polyédriques (Section 2). Ensuite, nous présenterons la technologie choisie pour la programmation au cours de ce stage (Section 3), ainsi que quelques optimisations courantes réalisées par les compilateurs actuels (Section 4). La section 5 présentera plus en détail les analyses réalisées comme préliminaires à ces optimisations. Enfin, la section 6 présentera nos contributions aux analyses de pointeurs, et la section 7 l'implémentation et les résultats obtenus.

2 Le modèle polyédrique

Généralement, le gros du temps d'exécution d'un programme de calcul est concentré dans une petite partie du code source : les boucles. L'optimisation et la parallélisation de celles-ci constituent un enjeu important dans la recherche informatique, et pour cela, il existe différentes méthodes, dont le modèle polyédrique.

Le modèle polyédrique est un modèle mathématique de représentation de boucles imbriquées. Dans ce modèle, les bornes de boucles *for* sont des fonctions affines des indices des boucles englobantes, et les fonctions d'accès aux tableaux sont aussi des fonctions affines des indices de boucles. Sous cette restriction, on est en mesure de représenter les espaces d'itérations (nids de boucles) par des polyèdres, ainsi que les dépendances de données entre différentes *instructions* du programme. Certaines optimisations (parallélisation automatique, amélioration de placement mémoire) sont alors possibles. [2]

```
for (i = 1; i < N; ++i)
  for (j = 1; j < N; ++j)
    T[i][j] = T[i - 1][j]
                + T[i][j - 1];
```

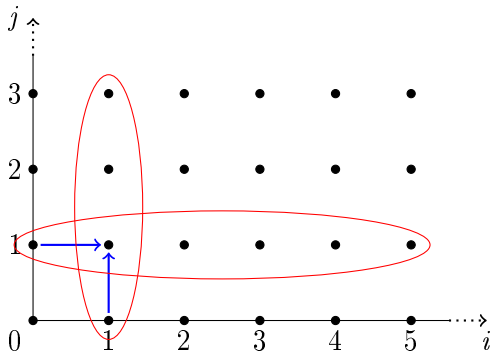


FIGURE 1 – Représentation de boucles imbriquées

Dans la figure 1, nous pouvons constater qu'il est impossible de paralléliser (ellipses rouges) tel quel, car les dépendances de données (représentées par des flèches bleues) l'en empêchent. Le modèle polyédrique permet ici de transformer le code en celui de la figure 2, qui est sémantiquement équivalent, mais qui peut être parallélisé (ellipse verte).

```
for (i = 1; i < N; ++i)
  for (k = 1; k < N - 1; ++k)
    T[i][j] = T[i - 1][k - 1]
            + T[i][k - 1];
```

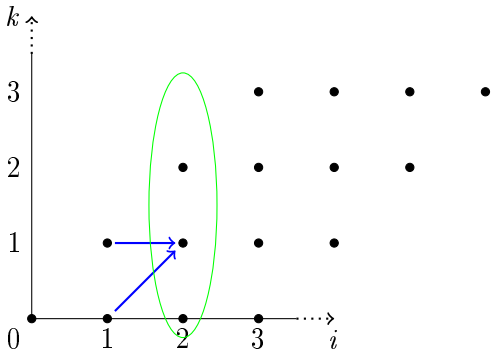


FIGURE 2 – Code parallélisable

3 LLVM

3.1 Un framework de compilation

Le projet LLVM [1] est né en 2000 à l'Université de l'Illinois, sous la direction de Chris Lattner et Vikram Adve. Il s'agit d'une infrastructure de compilateur, fondée sur une représentation intermédiaire du code, de type SSA [2], conçue pour l'optimisation d'un programme à tous les niveaux (compilation, édition de liens, exécution) [4]. A l'origine, l'implémentation concernait les langages C et C++, mais il existe désormais une grande variétés de FRONT-END [3] : Ruby, Python, Java, PHP, et Fortran, parmi d'autres.

LLVM contient nombre d'optimisations, ainsi que des générateurs de code pour de nombreux processeurs. LLVM est diffusé sous licence University Of Illinois Open Source, licence de type BSD. LLVM est codé en C++.

3.2 Clang

A l'origine, LLVM devait remplacer le générateur de code existant de GCC, et n'offrait qu'une compilation C, C++, et Obj-C via LLVM-GCC. A présent, LLVM propose Clang, compilateur C, C++, Obj-C et Obj-C++ entièrement développé et compilé à l'aide de l'infrastructure LLVM. De nos jours, le support du C et du C++ est quasi-total (plus de 90% des paquets Debian peuvent être compilés avec Clang). Pour le reste, ainsi que pour l'Ada et le Fortran, *dragonegg* intègre les optimisations et les générations de codes de LLVM avec le parser de GCC.

Clang lui-même est codé en C++, et représente le FRONT-END de LLVM. C'est celui qui est chargé de faire les analyses lexicales et syntaxiques, de construire l'arbre de syntaxe abstrait du programme, et de le convertir en code intermédiaire, compréhensible par LLVM.

Dans ce stage, nous utiliserons uniquement les représentations internes qui sont en amont de la forme SSA, car la librarie Clang nous offre des possibilités d'analyses statiques effectuées directement depuis l'AST (construction de CFG, design pattern Visitor [4], réécriture de code).

3.3 Polly

Polly, qui était au départ un projet de recherche, est un optimiseur polyédrique qui fait désormais partie de la suite de projets officiels de LLVM. Voici un extrait de la page de présentation du projet [5] :

1. `http://llvm.org`
2. *Single-Static Assignment*, représentation intermédiaire de code source dans laquelle chaque variable n'est assignée qu'une fois.
3. Passes de compilation qui effectuent l'analyse lexicale et syntaxique de la source en entrée afin de la transformer en représentation intermédiaire, nécessaire à la suite du processus de compilation (optimisation, édition de liens, ...) [1]
4. Modèle de conception objet dont le but est de visiter des instances d'autres classes afin d'en tirer certaines informations
5. `http://polly.llvm.org`

"Notre but premier est un optimiseur intégré d'accès mémoires et de parallélisme qui prenne avantage des multi-coeurs, de la hiérarchie des caches, et des instructions vectorielles."

Polly a déjà fait ses preuves, mais il n'est efficace pour le moment que sur des syntaxes de boucles très particulières (Boucles *for* et tableaux statiques principalement). Le but de ce stage est de conçevoir une passe de compilation qui tirerait partie d'une analyse des pointeurs C, notamment ceux utilisés comme pseudo-indices de boucles, afin de les transformer en tableaux statiques si possible : une analyse qui se comporterait comme un FRONT-END à Polly, épurant le code source d'un maximum de pointeurs, tout en générant un code sémantiquement équivalent.

4 Optimisations à la compilation

La phase de compilation d'un programme permet d'effectuer de nombreuses optimisations sur son code, dont la majorité visent à réduire le temps d'exécution du programme, ou sa consommation mémoire. Il en existe dont l'objectif est différent, comme par exemple la réduction de la consommation électrique nécessaire à l'exécution du programme, mais nous ne sommes pas concernées dans le cadre de ce stage.

4.1 Quelques types d'optimisations

Les optimisations à la compilation sont très variées, et sont regroupées dans différentes "catégories". Nous expliquerons très rapidement celles qui nous concernent.

Les optimisations locales ne concernent que certaines portions séquentielles du code, les *basic block*, notion qui sera développée dans la section concernant le graphe de contrôle. Ces optimisations nécessitent de très petites analyses, rapides et économes, mais dont la portée est très limitée.

Les optimisations globales, ou intraprocédurales, agissent sur des fonctions complètes. Elles offrent beaucoup plus d'informations, au prix d'analyses beaucoup plus coûteuses que les optimisations locales. Cependant, il faut être très conservateur et considérer le pire cas dès qu'il s'agit de traiter des appels de fonctions ou des accès à des variables globales.

Les optimisations de boucles consistent à transformer les nids de boucles en une version sémantiquement équivalente, mais améliorant la localité des données (accès au cache optimisé), ou facilitant le parallélisme. Parmi celles-ci, on peut citer les optimisations rendues possibles par le modèle polyédrique, implémentées dans Polly.

Les optimisations interprocédurales sont celles qui analysent d'un coup l'ensemble du code source d'un programme. Ces analyses sont extrêmement efficaces, mais elles sont très coûteuses.

Au cours de ce stage, nous nous intéresserons principalement à l'optimisation globale, qui nécessite certaines optimisations locales pour être efficace.

Dans la suite de cette section sont détaillées deux optimisations classiques réalisées sur le graphe de flot, qui permettent une amélioration de la qualité et des performances du code finalement généré.

4.2 Propagation de constantes

La propagation de constantes[6] est une optimisation locale (i.e. qui agit sur un *basic block* à la fois) qui consiste à déterminer à l'entrée et à la sortie de chaque *basic block* si les variables sont constantes, et si oui, leurs valeurs.

Lorsqu'une variable est constante à l'entrée d'un *basic block*, que sa valeur est connue, et que celle-ci n'est aucunement modifiée à l'intérieur de ce bloc, il est alors intéressant de remplacer chaque référence à cette variable, afin d'économiser un accès mémoire pour chacune de ces références. Ce concept est illustré par la figure 3.

6. *Constant Folding* en anglais.

```
int i, j, k;
i = 1;
j = 10;
k = i + j;
return k;
```

```
int i, j, k;
i = 1;
j = 10;
k = 1 + 10;
return k;
```

```
int i, j, k;
i = 1;
j = 10;
k = 11;
return k;
```

```
int i, j, k;
i = 1;
j = 10;
k = 11;
return 11;
```

FIGURE 3 – Un exemple de propagation de constantes.

Nous pouvons aisément constater que les références à i et j, variables constantes dont les valeurs sont connues, ont été remplacées par leurs valeurs respectives. Ensuite, l'expression *1 + 10* est constante, elle sera remplacée par sa valeur, c'est à dire *11*. Finalement, la référence à k dans la dernière instruction sera remplacée par sa valeur, celle-ci étant désormais connue.

4.3 Élimination du code mort

L'élimination du code mort est une optimisation statique dont le principe est de purger le code source de certaines instructions considérées comme inutiles, i.e. les blocs de codes inatteignables, les déclarations de variables inutilisées, ...

Dans le cas de la figure 3, une fois la propagation de constantes effectuée, on peut remarquer que certaines instructions sont inutilisées. Dans l'exemple suivant, les variables qui ne sont lues à aucun moment peuvent être supprimée, ainsi que, en veillant bien entendu aux éventuels effets de bord, toutes les assignations les concernant. (Figure 4).

```
int i, j, k;
i = 1;
j = 10;
k = 11;
return 11;
```

```
~~int i, j, k;~~
~~i = 1;~~
~~j = 10;~~
~~k = 11;~~
return 11;
```

```
return 11;
```

FIGURE 4 – Un exemple d'élimination du code mort suite à une propagation de constantes.

5 Analyse statique

5.1 Type d'analyse

Les analyses statiques peuvent être définies par certaines propriétés. En voici une brève définition.

Les analyses **sensibles au chemin**[7] considèrent les expressions connues dans le programme en fonction du chemin d'exécution. Par exemple, dans le cas d'un branchement conditionnel, l'analyse va retenir les informations apportées par la condition pendant l'analyse du bloc de code qui suit cette condition (par exemple, que telle variable est inférieure à zéro).

Les analyses **sensibles au contexte**[8] sont des analyses *interprocédurales* qui prennent en compte le contexte d'appel des procédures et fonctions au moment de leur appel. Ainsi, les informations connues au moment de l'appel seront propagées dans l'analyse de la cible appelée, et le résultat de l'analyse de cette cible sera retournée dans l'analyse principale.

Les analyses **sensibles au flot de données**[9] prennent en compte l'ordre d'exécutions des instructions du programme au moment de l'analyse. Une analyse insensible au flot de données pourra par

7. *Path-sensitive*, en anglais.
8. *Context-sensitive*, en anglais.
9. *Flow-sensitive*, en anglais.

exemple déterminer que deux pointeurs sont peut-être des alias [10], alors qu'une analyse sensible au flot de données pourra déterminer précisément à partir et jusqu'à quand les pointeurs sont des alias ou non.

5.2 Graphe de flot de contrôle

Un graphe de flot de contrôle, souvent abrégé CFG [11], est une représentation sous forme de graphe de tous les chemins qui peuvent être empruntés par le programme durant son exécution.

Dans un CFG, le code source est découpé en *basic block*. Un *basic block* représente une suite d'instructions qui seront exécutés séquentiellement, i.e. ne comprenant pas de branchements conditionnels ou de sauts. Un exemple est disponible avec la figure 5 dans laquelle chaque noeud du graphe est un *basic block* provenant du code source adjacent.

```
1  int main(int argc, char *argv[])
2  {
3    int i = 0;
4
5    printf("%i\n", i);
6
7    if (i)
8      i = 42;
9    else
10     i++;
11
12   return i;
13 }
```

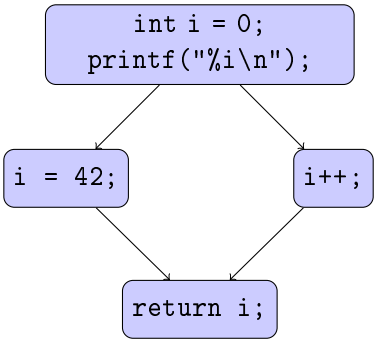


FIGURE 5 – Un code source C, ainsi que le CFG correspondant.

Dans l'exemple de la figure 5, nous pouvons constater qu'il existera, à l'exécution, deux flots d'instructions potentiels : le premier passant à l'intérieur du bloc *if*, le second à l'intérieur du bloc *else*.

Dans la plupart des représentations de CFG, il existe un *bloc d'entrée*, à partir duquel le contrôleur accède au graphe ainsi qu'un *bloc de sortie*, qui est l'unique sortie du graphe et vers lequel convergent tous les chemins qui peuvent potentiellement être exécutés. Il peut exister des chemins n'étant pas liés au bloc d'entrée, de sortie, ou au deux, mais auquel cas, le compilateur considérera généralement qu'il s'agit de code inutile pour l'exécution, et supprimera l'ensemble du code concerné.

Le graphe de flot de contrôle est un outil indispensable pour nombre d'analyses et d'optimisations statiques. Il permet de détecter notamment des blocs de code inatteignable (ceux qui ne sont pas reliés d'une quelconque façon au *bloc d'entrée*), ou certains cas de boucles infinies (par exemple, si le *bloc de sortie* n'est jamais atteint).

De plus, grâce à des optimisations comme la propagation de constantes, ou l'élimination de code mort, il est possible de faire varier le flot de contrôle, ce qui peut entraîner la fusion de blocs adjacents, et détecter à nouveau d'éventuels codes inatteignables ou des boucles infinies. Par exemple, dans la figure 6, nous pouvons constater qu'après avoir effectué une propagation de constantes, le CFG dispose d'un bloc désormais inatteignable, qui peut donc être supprimé.

10. Voir section 5.3.
11. *Control-Flow Graph* en anglais

```
int i = 0;
if (i < 0)
i = 0;
else
i = 1;
return i;
```

```
int i = 0;
if (0 < 0)
  i = 0;
else
  i = 1;
return i;
```

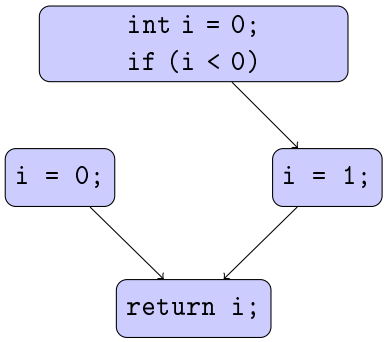


FIGURE 6 – Un code source C, ainsi que le CFG correspondant.

Pour conclure, le graphe de flot de contrôle est une structure de données très facile à parcourir grâce à la présence des structures de contrôles originales, et en conséquence la facilité d'expression des analyses dites *flow-sensitive* [1]. C'est d'ailleurs pour cette raison que nous utiliserons principalement cette représentation pendant ce stage, car l'analyse des alias dont nous avons besoin se doit d'être *flow-sensitive* étant donné que nous avons besoin de savoir précisément à quel moment un pointeur pointe sur tel espace mémoire.

5.3 Aliasing

En théorie des langages, un *objet* est un espace mémoire contenant des données, tandis qu'une *l-value* est une expression désignant un *objet*. Ainsi, lorsque que deux *l-value* désignent le même objet, elles sont considérées comme des alias (Figure 7).

```
int i;
int tab[10];
int *p1, *p2;

p1 = &i;
p2 = &tab[4];
```

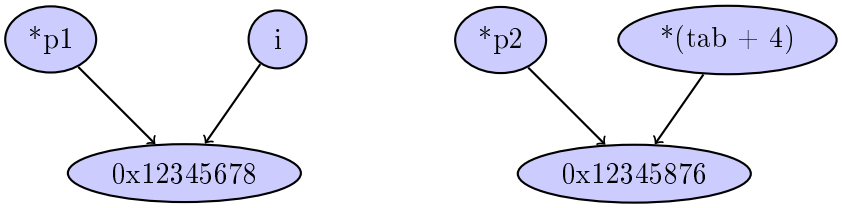


FIGURE 7 – Un code source C et deux exemples d'aliasing pour ce code.

Les informations d'aliasing dans un programme C sont très importantes, à cause de la présence des pointeurs, car elles permettent d'améliorer significativement la précision d'autres analyses (analyse des dépendances, analyse des variables vivantes, vérification des bornes de tableaux), notamment interprocédurale [12], ainsi que d'optimisations (propagation de constantes notamment) [3]. L'analyse des pointeurs est aussi cruciale pour l'exploitation du parallélisme en C.

Dans le cas de la figure 8, aucune propagation de constantes n'est possible sans informations d'aliasing, ou si ces informations ne sont pas suffisamment précise. De plus, le résultat de la propagation de constantes est différent en fonction de l'état d'aliasing entre p et q ("peuvent être des alias", "sont des alias", "ne sont pas des alias").

Il faut donc s'attarder sur les algorithmes qui proposent des analyses d'aliasing, tel que l'algorithme de Steensgaard qui, malgré son aspect *flow-insensitive*, possède une propriété intéressante.

12. Analyse qui prend en compte le lien entre les différentes procédures et fonctions du programme : qui est appelée et par qui, avec quels paramètres, ...)

```
int n, *p, *q;

[...]

*p = 4;
*q = 6;

n = *p + *q;
```

(a) Sans informations d'aliasing

```
int n, *p, *q;

[...]

*p = 4;
*q = 6;

n = 10;
```

(b) p et q ne sont pas des alias

```
int n, *p, *q;

[...]

~~*p = 4;~~
*q = 6;

n = 12;
```

(c) p et q sont des alias

FIGURE 8 – Exemple de sortie après une propagation de constantes en fonction des informations d'alias connues.

6 Contributions aux analyses de pointeurs

6.1 Algorithme de Steensgaard

L'algorithme de Steensgaard retourne un graphe qui modélisent les relations *points to* entre les différentes variables du code. Dans ce graphe, un noeud représente une variable et un arc une relation *points-to* entre deux variables. [5]

```
int x, y;
int *p, *px, *py;

px = &x;
p = px;

py = &y;
p = py;
```

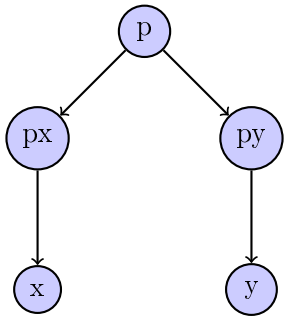


FIGURE 9 – Construction de graphe avec Steensgaard

Le graphe de la figure 9 est construit après l'analyse des instructions dans le code correspondant. Une fois cette analyse effectuée, l'algorithme de Steensgaard va "fusionner" certains noeuds (figure 10), car il ne permet la présence que d'un seul arc de sortie. On constate donc une perte de précision, qui compense la complexité linéaire de l'algorithme. [5]

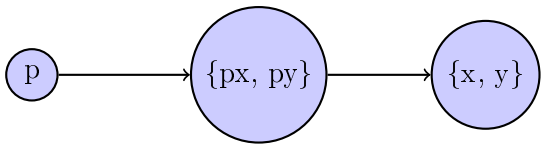


FIGURE 10 – Graphe final avec Steensgaard

Étant donné que le but de ce stage est de concevoir une analyse visant à purger un code C d'un maximum de pointeurs, le manque de précision de l'algorithme de Steensgaard risque de nous handicaper. En effet, nous avons besoin de connaître à tout moment la dernière cible connue d'un pointeur (dans le cas où c'est possible), ce que l'algorithme de Steensgaard ne peux gérer. En effet, dans l'exemple de la figure 9, nous pouvons constater qu'à la lecture du graphe de la figure 10, il n'est pas possible de déterminer précisément si *px* pointe vers x, alors qu'à la lecture du code source, il ne fait aucun doute que c'est le cas. Cependant, la modélisation de la relation *points to* à l'aide d'un graphe semble être une solution adaptée à nos besoins.

6.2 Notre proposition

Nous allons donc parcourir le CFG de chaque fonction du code source, afin de tenter de la débarrasser d'un maximum de pointeurs, et de retourner un graphe d'aliasing pour les pointeurs restants, afin de pouvoir transmettre les informations rassemblées à une future analyse.

6.3 Choix de la structure de donnée interne

Pour gérer l'aliasing, la solution retenue est donc de gérer une matrice représentant le graphe d'aliasing. Chaque noeud représente un pointeur, et chaque arc une relation de type *points to*. Ainsi, après avoir effectué une fermeture transitive du graphe, il suffit d'établir l'existence d'un arc entre deux noeuds pour établir que les pointeurs correspondants à ces deux noeuds sont des alias (figure 11).

```
int x, *p1, *p2;
p1 = &x;
p2 = p1;
```

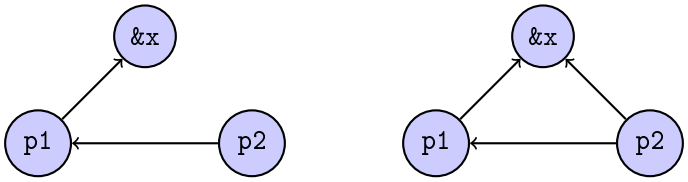
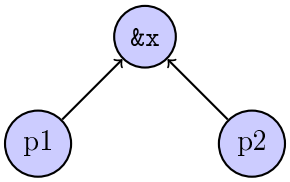


FIGURE 11 – Un extrait de code C, le graphe d'aliasing correspondant, et la fermeture transitive dudit graphe.

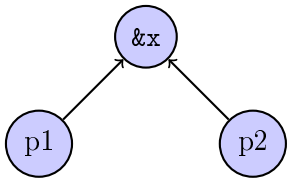
Outre le coût de la fermeture transitive, $O(n^3)$, il est nécessaire de stocker jusqu'à trois matrices en mémoire afin de pouvoir conserver les informations. La première est la matrice qui stocke le graphe d'aliasing, la seconde est la matrice qui stocke la fermeture transitive du graphe d'aliasing (pour éviter de multiples calculs si elle ne varie pas). Pour la troisième, il s'agit de la transformation du graphe d'aliasing (orienté) en graphe non-orienté. Cela se fait en symétrisant la matrice qui représente le graphe d'aliasing. Cette opération est nécessaire, car sinon, il existe un cas où la fermeture transitive ne permet pas de récupérer l'information d'aliasing (figure 12).

```
int x, *p1, *p2;

p1 = &x;
p2 = &x;
```



(a) Graphe d'aliasing après 3.



(b) Fermeture transitive du même graphe : aucun arc n'a été ajouté.

FIGURE 12 – Exemple du problème lié à la fermeture transitive d'un graphe orienté.

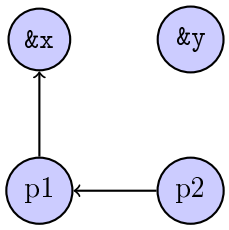
Comme nous pouvons le constater, la fermeture transitive du graphe ne nous permettra pas d'obtenir l'information *"p1 et p2 sont des alias"*, alors qu'ils le sont. La matrice intermédiaire est donc nécessaire pour la fermeture transitive, et la stocker permet d'éviter les pertes d'informations liées à l'orientation du graphe : *X pointe sur Y, Z pointe sur ...*

6.4 Construction du graphe d'aliasing à partir d'un programme donné

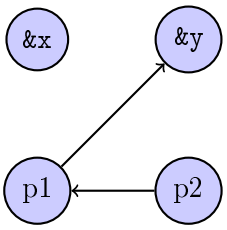
Lors de la construction du graphe d'aliasing, un problème se pose lorsque la cible d'un pointeur *p1* varie alors qu'il existe un ou plusieurs arcs entre *p1* et d'autres pointeurs. Dans ce cas, l'information de cible peut-être perdue, comme nous pouvons le constater dans la figure 13 dans laquelle le graphe d'aliasing final est erroné : *p2* devrait être aliasé avec *x*.

Sachant que l'une des informations les plus importantes dans le cadre de notre sujet est la dernière cible connue d'un pointeur, cet algorithme ne semble pas entièrement adapté. La gestion de matrices étant indispensable pour stocker les informations d'aliasing, le changement se fera à l'établissement

```
1.  p1 = &x;
2.  p2 = p1;
3.  p1 = &y;
```



(a) Aliasing après 2.



(b) Aliasing après 3.

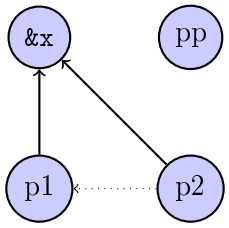
FIGURE 13 – Exemple du problème de construction de graphe.

d'une relation *points to* entre deux pointeurs. Ainsi, dans le cas d'une instruction comme *p2 = p1*, si la cible de p1 est connue, alors nous considérerons que l'instruction équivaut à *p2 = pts_to(p1)*, afin d'être capable de déterminer, sans doutes possibles, la cible de p2, et sans avoir à effectuer une fermeture transitive du graphe d'aliasing. Réciproquement, si la cible d'un pointeur n'est pas connue, c'est qu'il pointe sur un espace mémoire externe à la fonction (par exemple, un pointeur passé en paramètre). Dans ce cas, l'analyse se doit d'être conservative et de considérer le pire cas : le pointeur peut pointer sur n'importe quoi.

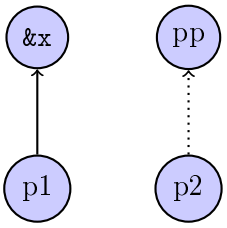
```
void
f(int *pp)
{
(1) int x, *p1, *p2;
(2) p1 = &x;
(3) p2 = p1;

  // [...]

(4) p2 = pp;
}
```



(a) Aliasing après (3)



(b) Aliasing après (4)

FIGURE 14 – Gestion des relations *points to* dans le cas d'une cible extérieure à la fonction.

Dans la figure 14, les arcs en gras représentent la dernière cible connue du pointeur, tandis que les arcs en pointillés représentent la dernière valeur qui a été affectée au pointeur.

Maintenant que l'algorithme que nous allons utiliser pour gérer l'aliasing est construit, il faut regarder comment l'implémenter grâce à LLVM, et plus précisément Clang.

7 Implémentation

7.1 Représentation des instructions dans LLVM/Clang

Clang utilise une hiérarchie de classes pour représenter les différentes instructions rencontrées dans un programme. *Stmt* est la super-classe de cette hiérarchie, et représente de manière abstraite n'importe quelle instruction. Une partie de cette architecture est présentée dans la figure 15

De plus, *Stmt*, ainsi que toutes les classes qui en héritent, se comporte comme un conteneur. Elle peut donc stocker des pointeurs vers des objets enfants de type *Stmt* (et donc, par polymorphisme [13], de n'importe quelle classe héritant de *Stmt*). Ainsi, en itérant sur un objet de type *Stmt*, il est possible de récupérer l'ensemble des sous-instructions contenues dans cet objet. Un exemple de représentation d'une instruction est disponible avec la figure 16.

13. Principe de programmation orientée objet, qui considère qu'un objet d'une classe B, héritant d'une classe A, peut-être considéré comme une extension de la classe A, et peut donc être manipulé comme un objet de classe A.

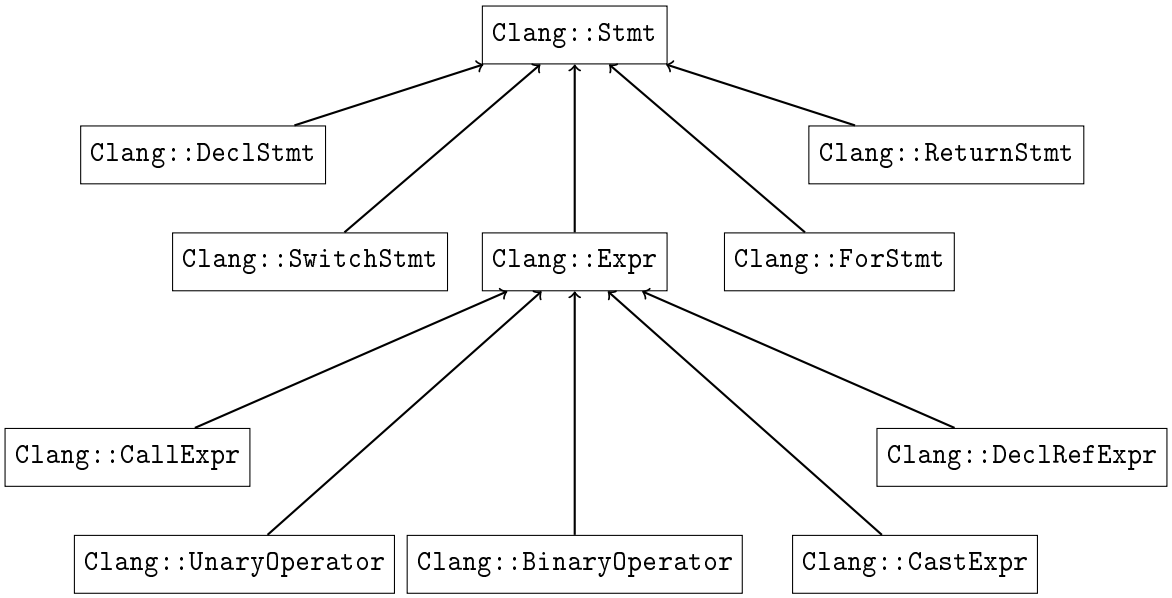


FIGURE 15 – Extrait de la hiérarchie de classes Clang pour la représentation des instructions.

```
for (int i = 0;
     i < SZ;
     ++i)
  tab[i]= 0;
```

```
ForStmt
|-DeclStmt
| `-VarDecl i 'int'
|   `-IntegerLiteral 0 'int'
|-BinaryOperator '<'
| |-ImplicitCastExpr 'int'
| | `-DeclRefExpr Var 'i' 'int'
| `-IntegerLiteral 0 'int'
|-BinaryOperator '='
| |-ArraySubscriptExpr
| | |-ImplicitCastExpr 'int*'
| | | `-DeclRefExpr ParmVar 'tab' 'int*'
| | `-ImplicitCastExpr 'int'
| |   `-DeclRefExpr Var 'i' 'int'
| `-IntegerLiteral 0 'int'
`-UnaryOperator prefix '++'
  `-DeclRefExpr Var 'i' 'int'
```

FIGURE 16 – Exemple d'instruction C avec sa représentation Clang.

7.2 Modèle de conception : *Visitor*

La bibliothèque Clang offre des modèles de conception de type *Visitor* pour faciliter le travail de développement au sein du FRONT-END de LLVM. Nous utiliserons ces modèles pour parcourir l'arbre de syntaxe abstrait du programme, à la recherche des définitions de fonctions, ainsi que pour parcourir les CFGs de chacune de ces fonctions et y effectuer notre analyse.

Le modèle de conception *Visitor* présent dans Clang dispose d'un comportement similaire, qu'il s'agisse d'un visiteur d'AST ou de CFG. Dans cette section, nous considérerons uniquement l'exemple du CFG. Comme le montrent les figures 15 et 16, chaque type d'instruction possède une représentation propre au sein de la hiérarchie de classes. Le modèle de visite implémenté utilise, grâce à un système de génération de code à la compilation, une gestion pseudo-dynamique des différentes méthodes de visite. Dans le cas d'un objet représentant une instruction de la classe FooStmt, le visiteur cherchera à appeler sa propre méthode définie comme *RetTy VisitFooStmt(FooStmt *F)*, où *RetTy* est un type

paramétrable. Dans le cadre du modèle *Visitor* récursif, si la méthode n'est pas définie, le visiteur itérera sur les enfants de l'objet courant (F dans le cas de *VisitFooStmt(..)*) pour effectuer le même processus avec chacun d'eux.

```
#define DISPATCH(NAME, CLASS) \
return static_cast<ImplClass*>(this)->Visit ## NAME(static_cast<PTR(CLASS)>(S))

RetTy Visit(Stmt *S)
{

  // [...]

  // Top switch stmt: dispatch to VisitFooStmt for each FooStmt.
  switch (S->getStmtClass())
  {
    default: llvm_unreachable("Unknown stmt kind!");

#define ABSTRACT_STMT(STMT)
#define STMT(CLASS, PARENT) \
    case Stmt::CLASS ## Class: DISPATCH(CLASS, CLASS);

    #include "clang/AST/StmtNodes.inc"
  }
}

// If the implementation chooses not to implement a certain visit method, fall
// back on VisitExpr or whatever else is the superclass.
#define STMT(CLASS, PARENT) \
RetTy Visit ## CLASS(PTR(CLASS) S) { DISPATCH(PARENT, PARENT); }
#include "clang/AST/StmtNodes.inc"

 // If the implementation doesn't implement binary operator methods, fall back
 // on VisitBinaryOperator.
 // [...]

 // If the implementation doesn't implement compound assignment operator
 // methods, fall back on VisitCompoundAssignOperator.
 // [...]

 // If the implementation doesn't implement unary operator methods, fall back
 // on VisitUnaryOperator.
 // [...]
```

FIGURE 17 – Extrait du code source de *Clang : :StmtVisitor*, classe abstraite qui définit l'implémentation du *Visitor* au sein de Clang.

Dans cet extrait, le code est généré à la volée à la compilation. La compilation va générer un fichier *StmtNodes.inc*, qui contiendra des appels aux macros définies précédemment (*DISPATCH* par exemple) pour chaque classe trouvée dans l'arborescence de Clang. Ainsi, pour chaque classe *Foo* existante, un *case* sera généré, contenant un appel à *VisitFoo*. Concrètement, pour implémenter un visiteur en suivant de modèle, il suffit de créer une classe qui héritera de ce modèle, à l'intérieur de laquelle on définira uniquement les méthodes *VisitXXX* pour les types d'instructions qui nous intéressent.

7.3 *Visitor* : Utilisation

```
// Notre propre implémentation doit dériver du pattern paramétré par
// la classe qui en hérite (nécessaire pour une gestion précise des appels
// internes de méthodes).
//
// Il est possible de paramétrer le type de retour des méthodes de visites
// à la suite du paramétre de classe. Par exemple :
// class ClassName : public Visitor<ClassName, bool>
// Le type de retour par défaut est void.
class CustomCFGRecursiveVisitor
    : public CFGRecStmtVisitor<CustomCFGRecursiveVisitor> {

public:
  // Constructeur
  CustomCFGRecursiveVisitor(FunctionDecl *F, CFG &cfg)
    : m_cfg(cfg)
  {
    typedef FunctionDecl::param_iterator it;

    // Parcours des paramètres de la fonction à la recherche d'éventuelles
    // déclarations de pointeurs.
    for (it b = f->param_begin(), e = f->param_end(); b != e; ++b)
      if (isa<PointerType>(getTypePtrFromVarDecl(*b)))
        (*b)->dumpColor();
  }

  // Méthodes de visite
  void VisitBinaryOperator(BinaryOperator *BOP);
  void VisitDeclStmt(DeclStmt *DS);
};

void CustomCFGRecursiveVisitor::VisitBinaryOperator(BinaryOperator *BOP)
{
  if (BOP->getOpcode() == BO_Assign)
    BOP->dumpColor();
}

void CustomCFGRecursiveVisitor::VisitDeclStmt(DeclStmt *DS)
{
  VarDecl *VD;

  // Si la variable déclarée est de type "Pointeur" ...
  if ((VD = dyn_cast<VarDecl>(DS->getDecl()))
      && isa<PointerType>(getTypePtrFromVarDecl(VD)))
    DS->dumpColor();
}
```

FIGURE 18 – Exemple d'utilisation du modèle de conception *Visitor* : le code ici présent définit une classe de visiteur qui recherche les déclarations de pointeurs ainsi que les opérations d'assignations simples, et affiche simplement les informations contenues dans leurs représentations respectives (dump).

8 Conclusion

Pour conclure, il est important de préciser que l'implémentation de cette analyse n'est pas finie. Elle est capable à ce jour de détecter certains usages de pointeurs, de construire le graphe d'aliasing, mais n'effectue pas toutes les détections voulues, ni une quelconque réécriture de code. Actuellement,

le projet comporte environ 1500 lignes de codes, et à terme, j'estimerais sa teneur à 2000-2500 lignes, le code manquant étant le plus difficile à mettre en place.

Lors de ce stage, j'ai pu découvrir le milieu de la recherche informatique, et j'ai beaucoup appris en terme de compilation avancée, d'algorithmique, et de notions de bas niveau (ASM, parallélisation, pipeline processeur, ...). Je suis très content d'avoir complété mon enseignement en compilation que j'ai eu au S5, et d'être maintenant capable de comprendre l'ensemble du processus qui compose le FRONT-END d'un compilateur. De plus, j'ai acquis une bonne expérience en C++, ainsi que des connaissances du framework LLVM, connaissances qui me serviront très surement à nouveau.

9 Benchmarks

```
int main(void)
{
  int sz = 10;
  int *p;

  // Propagation de constantes, taille connue a la
  // compilation. Si la taille est multiple de la taille
  // du type pointe, et si les acces sont multiples de la
  // taille du type pointe, nous pourrons reecrire un tableau
  // statique.
  *p = (int*)malloc(sizeof(int) * sz);

  // Execution symbolique peut-etre necessaire.
  while (sz--) {
    *(p + sz) = 0; // Pointeur est incremente, puis dereference.
  }

  // A ce stade, on connait la taille du tableau, on sait que
  // tous les acces sont alignes sur le type pointe, et que le
  // pointeur qui a recu l'adresse de l'allocation dynamique est
  // constant ..
  // ======> Reecriture du code :
  //
  // int sz = 10;
  // int t[10];
  //
  // while (sz--) {
  // t[sz] = 0;
  // }
  //
  // Une analyse plus fine pourra par la suite transformer la
  // boucle "while" en "for", et il sera alors possible de
  // paralléliser le traitement.
}
```

```
int f(int *p)
{
        int x = 10;
        int *p1 = p, *p2 = &x;

        // Propagation de constantes : bloc "if" inatteignable.
        if (x + 1 < 0)
                *p1 += 1;
        else
                *p2 = *p2 * *p1;

        // A ce niveau, p1 est toujours constant et pointe sur p,
        // on renvoit dont la valeur du parametre. La prochaine
        // analyse interprocedurale aura les informations d'aliasing
        // et effectuera des optimisations si possible.
        return *p1;
}
```

```c
/*Source Code From Laure Gonnord*/

#include <stdio.h>

void find_1d(double *q)
{
  int i;
  for (i = 0;i < 2; ++i) {
    *(q + i) = 12 - i;
  }
}

// q est un tableau 2D linéarisé.
void find_2d(double *q)
{
  int i,j;
  int ro;

  int qdim1 = 12;

  for (i = 0 ; i < 2; ++i) {
    for (j = 0;j < 3; ++j) {
      q[i+ 2*j] = 1515+i*j;
    }
  }
}

int main()
{
  double t[2] = {23.9,1515.42};
  find_1d(t);

  printf("%lf\n",t[1]);

  double t2[6]={1,2,3,4,5,6};
  find_2d(t2);

  return 0;
}
```

Références

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Pierre Boulet. *Contributions aux environnements de programmation pour le calcul intensif*. Hdr, Université des Sciences et Technologie de Lille - Lille I, 2002. H355 H355.

[3] Michael Hind. Pointer analysis : Haven't we solved this problem yet ? In *PASTE'01*, pages 54–61. ACM Press, 2001.

[4] Chris Lattner and Vikram Adve. Llvm : A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization : feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[5] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.