Université Claude Bernard Lyon 1

Analyses statiques pour les pointeurs, expérimentations autour d'algorithmes.

Tristan Dubois

Janvier-Février 2015

Résumé

Mots-clés Programmes C, analyse statique, algorithme de Steensgaaard, arithmétique de pointeurs, implémentations.

Table des matières

1 Introduction **2**
1.1 L'équipe Compsys . . . 2
1.2 Contexte – analyse statique des *alias* pour la compilation . . . 2
1.3 État de l'art des analyses de pointeurs . . . 3
1.4 Objectif du stage . . . 4

2 Réalisation 1 : Un traducteur C vers Prolog **5**
2.1 Contexte théorique . . . 5
2.1.1 Interprétation abstraite et analyse d'intervalle . . . 5
2.1.2 L'analyse de pointeurs proposée par CompSys . . . 5
2.1.3 Détails du prototype Prolog . . . 6
2.2 Contexte technologique – LLVM . . . 7
2.3 Réalisation et résultats . . . 8

3 Réalisation 2 : Expérimentations autour de l'algorithme de Steensgaaard **9**
3.1 Objectif . . . 9
3.2 Réalisation et premiers résultats expérimentaux . . . 9

4 Conclusion **10**

1 Introduction

Ce TER a été proposé par Laure Gonnord (Compsys, LIP/ENS Lyon, Université Lyon1).

1.1 L'équipe Compsys

COMPSYS [1] est une équipe de recherche commune à l'Inria et au Laboratoire de l'Informatique du Parallélisme (LIP). Elle est localisée à l'École Normale Supérieure de Lyon (ENS Lyon).

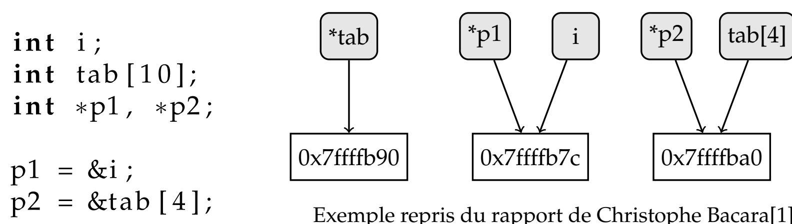
L'objectif de Compsys est le développement de techniques de compilation, plus précisément d'optimisations de codes, appliquées aux domaine des systèmes embarqués de calcul (embedded computing systems). Compsys s'intéresse à la fois aux optimisations bas niveau (back-end) pour les processeurs spécialisés et aux transformations de haut niveau (principalement source à source) pour la synthèse d'accélérateurs matériels. Ces optimisations s'attachent à favoriser le parallélisme, et la localité des données. Pour réaliser ces optimisations, l'équipe utilise souvent des informations *statiquement* découvertes lors de phases d'analyse dans les phases d'optimisation qui suivent.

Les analyses de pointeurs se placent dans ce contexte. Les optimisations développées par Compsys utilisent souvent le *modèle polyédrique*. et dans le cas général ne gèrent pas les pointeurs. Les analyses d'alias permettraient d'appliquer les analyses polyédriques à une classe plus importante de problèmes.

Note sur le modèle polyédrique Le modèle polyédrique [2] est un modèle mathématique qui sert à étudier les boucles imbriquées. Dans cette représentation, les bornes de boucles sont des fonctions affines des indices des boucles englobantes et les fonctions d'accès au tableaux sont également des fonctions affines de ces indices. Cela permet de représenter de représenter des espaces d'itération (valeur des variables des boucles imbriquées) par des polyèdres et de calculer des informations sur celles-ci (dépendance de données, calcul d'ordonnancement, placement de tâches parallèles, optimisation de code, . . .).

1.2 Contexte – analyse statique des *alias* pour la compilation

Deux variables sont dites *alias* si elles se réfèrent à deux zones mémoire qui se chevauchent, voire qui se confondent. Il s'agit d'une caractéristique courante des langages de programmation impératifs, conséquence des concepts de *pointeurs* (en `C`) ou de *références* (en `C++` ou en `Java`). La figure 1 montre comment deux variables `*p1` et `i` peuvent être des *alias* en C (car elles pointent, en fait, sur la même adresse). Or, si l'*aliasing* fait la force de ces langages, elle complexifie aussi l'analyse des programmes et, par conséquent, l'implémentation des optimisations.



Exemple repris du rapport de Christophe Bacara[1]

FIGURE 1 – Un programme C et les adresses de quelques unes de ses *l-values*

Par exemple, une analyse des expressions disponibles – préalable à une élimination des expressions redondantes – ne peut se contenter, après la modification d'une variable, d'invalider uniquement les expressions contenant cette variable, elle doit également invalider toutes les expressions utilisant ses possibles *alias*. Or, en l'absence de connaissances particulières sur les *alias*, les *alias* d'une référence (ou d'un pointeur déréférencé) peuvent être n'importe quelle variable, obligeant l'analyseur à invalider toutes les expressions et empêchant quasiment toute optimisation. Dans le

1. Compilation et systèmes embarqués. Cette description de l'équipe vient de son rapport d'activité 2014
2. Ce texte est repris de l'HDR de Pierre Boulet, chapitre 2 : `http://www.lifl.fr/~boulet/HdR/index.html`

cas de la figure 2, le compilateur ne peut éliminer le second calcul de $a + b$ car il ne peut garantir que `*p` n'est ni un *alias* de `a` ni un *alias* de `b`.

```
int a, b, *p;
/* ... */
*p = a + b;
return a + b;
```

FIGURE 2 – Est-ce que l'expression $a + b$ doit être recalculée ?

Aussi, pour prévenir ce genre de scénario, la plupart des compilateurs optimisants pour les langages susmentionnés implémentent aujourd'hui une *analyse des alias*. Celle-ci a pour but de fournir au compilateur des connaissances sur les couples de variables qui :

- sont **nécessairement** des *alias* – *must-alias* ;
- peuvent **possiblement** être des *alias* – *may-alias* ;
- ne sont **jamais** des *alias* – *cannot-alias*.

On se concentre en particulier sur les *may-alias*. En pratique, ces connaissances sont dérivées d'une *analyse de pointeurs* qui consiste à déterminer, pour chaque pointeur p, l'ensemble des zones mémoire $PT(p)$ sur lesquelles il peut pointer. En effet, partant de là, si l'on détermine que les ensembles $PT(p_1)$ et $PT(p_2)$ sont disjoints alors l'on est certain que les pointeurs p_1 et p_2 ne sont jamais des *alias* (*cannot-alias*), sinon ils peuvent être des *alias* (*may-alias*).

1.3 État de l'art des analyses de pointeurs

Les analyses de pointeurs sont souvent, pour des raisons d'efficacité, *insensibles au flot de données*, c'est-à-dire qu'elles ignorent l'ordre d'exécution des instructions dans un programme et sont incapables, par exemple, de fournir des informations sur depuis quand deux pointeurs sont des *alias* et jusqu'à quand ils resteront des *alias*.

Motif d'instruction	Contrainte *Andersen*	Contrainte *Steensgaaard*
`a = b`	$PT(a) \subset PT(b)$	$PT(a) = PT(b)$
`a = &b`	$b \in PT(a)$	
`a = *b`	$\forall x \in PT(b),\ PT(a) \subset PT(x)$	$\forall x \in PT(b),\ PT(a) = PT(x)$
`*a = b`	$\forall x \in PT(a),\ PT(x) \subset PT(b)$	$\forall x \in PT(a),\ PT(x) = PT(b)$

FIGURE 3 – Contraintes générées par les deux algorithmes

Généralement, ces techniques parcourent séquentiellement le code du programme – ou d'une procédure du programme – et, lorsqu'elle rencontre une instruction digne d'intérêt (comme une affectation de pointeurs), génère une nouvelle contrainte. À la fin, il suffit de résoudre le système de contraintes ainsi construit pour obtenir le résultat. Deux célèbres algorithmes fonctionnent de cette façon : l'algorithme de *Andersen*[2] et l'algorithme de *Steensgaaard*[3].

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

FIGURE 4 – Application des deux algorithmes sur un même code source C

Les contraintes sont souvent représentés par un graphe comme ceux montrés figure 4. L'algorithme de *Steensgaaard* est plus rapide (complexité quasi-linéaire) que son concurrent, grâce au remplacement de la contrainte d'inclusion par une contrainte d'unification (cf. figure 3). Cette

contrainte est réalisée en fusionnant les nœuds lorsqu'ils sont pointés par le même nœud. En pratique, cela aboutit au fait que chaque nœud n'a qu'un arc sortant et à une analyse finale moins précise que celle fournie par *Andersen*, comme le montre la figure 4 : sur cet exemple, l'algorithme de *Steensgaaard* ne distingue plus entre `py` et `px` et perd, du coup, la connaissance spécifique que `px` pointe sur `x` et `py` pointe sur `y`. Il s'agit de faire un choix entre précision et vitesse.

Ces algorithmes posent le problème de ne pas gérer suffisamment l'arithmétique des pointeurs (par exemple `py = px + 4`). En effet, généralement, cette situation est gérée en assimilant deux pointeurs faisant référence à deux emplacements distincts dans une même zone mémoire comme étant des *alias* (comme, dans l'exemple, `py` et `px`). En d'autres termes, les zones mémoire sont traitées comme étant des unités indivisibles.

1.4 Objectif du stage

L'équipe *Compsys* (en particulier, Laure Gonnord, Maroua Maalej – doctorante – et Fernando Pereira – collaborateur invité) est en train de concevoir un nouvel algorithme d'analyse de pointeurs, plus précis, capable de gérer l'arithmétique des pointeurs et, éventuellement, de distinguer deux pointeurs faisant référence à deux intervalles distincts dans une même zone mémoire.

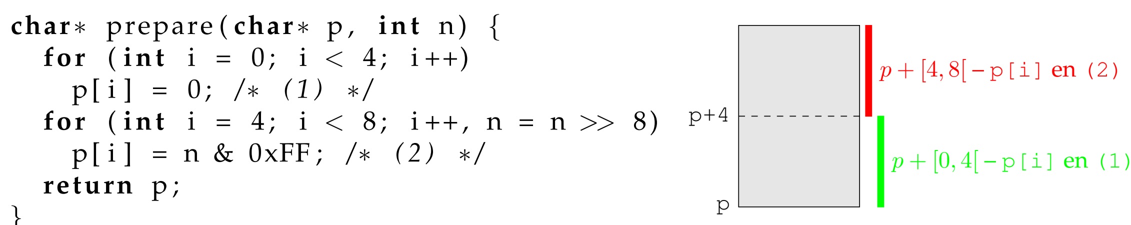


FIGURE 5 – `p[i]` ne pointe jamais sur la même position en `(1)` et `(2)`

Dans la figure 5, l'on voit une situation où une analyse considérant chaque zone mémoire comme une unité indivisble échouerait à détecter que `p[i]` en `(1)` et `p[i]` en `(2)` ne pointent jamais sur les mêmes adresses : elle se contenterait de constater que `p+i` est – dans les deux cas – un pointeur sur le tableau `p`. Par contre, une analyse prenant en compte les intervalles où évoluent `i` aboutirait à un meilleur résultat : elle observerait que `p[i]` en `(1)` n'évolue que dans l'intervalle $p + [0, 4[$ tandis que `p[i]` en `(2)` ne balaie que l'intervalle $p + [4, 8[$; or, ces deux ensembles sont évidemment disjoints, comme le montre le schéma à droite.

L'objet du stage était initialement de construire un prototype LLVM permettant d'expérimenter ces dernières recherches, d'en examiner les résultats et de les comparer – caractéristiques du graphe construit, vitesse de l'analyse, gain de performances sur les exécutables optimisés, etc. – aux algorithmes actuels.

L'objectif du stade a dérivé de la construction d'un prototype LLVM vers la réalisation d'un traducteur de *bytecode* LLVM vers le langage spécifique accepté par un prototype en Prolog déjà existant (réalisation 1). Par ailleurs, pour permettre comparer la performance de ce nouvel algorithme par rapport aux analyses de pointeur déjà existantes, j'ai également écrit une *passe* LLVM effectuant l'analyse de Steensgaaard (réalisation 2).

Pendant la durée de mon stage, j'ai eu l'opportunité de recevoir les conseils de Fernando Pereira, professeur invité, enseignant-chercheur en compilation à l'Université de Minas Gerais (Brésil).

Remarque importante Durant ce stage, j'ai eu l'opportuné de suivre les cours d'une école de recherche à destination des étudiants de Master Informatique de l'ENS Lyon. Cette école a duré une semaine (24h de cours et TP), et a été l'occasion de découvrir la place cruciale qu'a l'analyse statique dans le domaine de la construction de compilateurs. De plus, les TPs ont été l'occasion de me familiariser avec le *framework* LLVM. [3]

3. Le programme et le contenu de l'école de recherche sur l'analyse statique et la compilation peut être trouvé à la page `http://laure.gonnord.org/pro/research/compil_research_school.html`.

2 Réalisation 1 : Un traducteur C vers Prolog

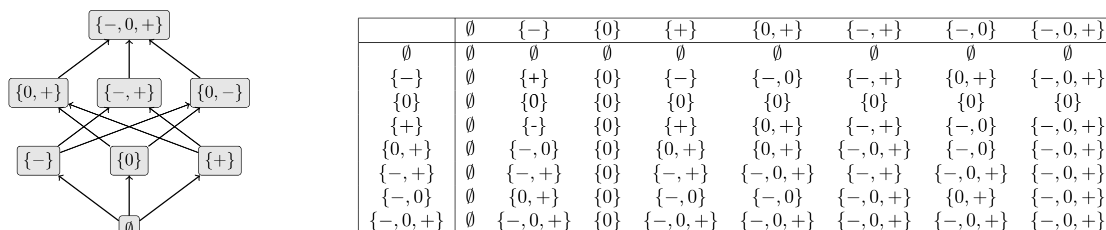
2.1 Contexte théorique

2.1.1 Interprétation abstraite et analyse d'intervalle

L'*interprétation abstraite* est une méthode d'analyse statique, formalisée et popularisée par Patrick Cousot et Radhia Cousot, consistant à évaluer un programme en utilisant des valeurs abstraites, prises un treillis, en lieu et place des valeurs concrètes. Cette évaluation aboutit ainsi à associer à chaque variable du programme une valeur abstraite, ce qui permet de déduire quelques propriétés utiles sur lesdites variables et, partant, sur le programme entier. Cela nécessite de redéfinir les opérations du langage en termes de variables abstraites : l'on parle alors de *sémantique abstraite*, par opposition à la *sémantique concrète*.

Afin de garantir que l'analyse termine, l'interprétation abstraite impose que les valeurs des variables *croissent* dans le treillis. Cette contrainte s'exprime dans la sémantique abstraite par le fait que l'affectation d'une valeur dans une variable enregistre dans cette variable la borne supérieur de l'ancienne et de la nouvelle valeur au lieu d'y stocker simplement la nouvelle valeur. Ainsi, le nombre de modification de chaque variable est borné par la hauteur du treillis, ce qui prouve la terminaison dans le cas où la hauteur du treillis est fini. Dans le cas contraire, si le treillis est de hauteur infini, on a également recours au *widening*, qui consiste à affecter une valeur suffisamment grande dans une variable pour empêcher que celle-ci puisse croître à nouveau.

Un exemple simple d'interprétation abstraite consiste à réduire les valeurs entières à leur signe (positif, négatif, nul). Le treillis correspondant est dessiné sous la forme d'un diagramme de Hasse à la figure 6, accompagné de la sémantique abstraite de la multiplication sous la forme d'une table de calcul.



	$\emptyset$	$\{-\}$	$\{0\}$	$\{+\}$	$\{0,+\}$	$\{-,+\}$	$\{-,0\}$	$\{-,0,+\}$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\{-\}$	$\emptyset$	{+}	$\{0\}$	$\{-\}$	$\{-,0\}$	$\{-,+\}$	$\{0,+\}$	$\{-,0,+\}$
$\{0\}$	$\emptyset$	$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$
$\{+\}$	$\emptyset$	{-}	$\{0\}$	$\{+\}$	$\{0,+\}$	$\{-,+\}$	$\{-,0\}$	$\{-,0,+\}$
$\{0,+\}$	$\emptyset$	$\{-,0\}$	$\{0\}$	$\{0,+\}$	$\{0,+\}$	$\{-,0,+\}$	$\{-,0\}$	$\{-,0,+\}$
$\{-,+\}$	$\emptyset$	$\{-,+\}$	$\{0\}$	$\{-,+\}$	$\{-,0,+\}$	$\{-,+\}$	$\{-,0,+\}$	$\{-,0,+\}$
$\{-,0\}$	$\emptyset$	$\{0,+\}$	$\{0\}$	$\{-,0\}$	$\{-,0\}$	$\{-,0,+\}$	$\{0,+\}$	$\{-,0,+\}$
$\{-,0,+\}$	$\emptyset$	$\{-,0,+\}$	$\{0\}$	$\{-,0,+\}$	$\{-,0,+\}$	$\{-,0,+\}$	$\{-,0,+\}$	$\{-,0,+\}$

FIGURE 6 – Treillis des valeurs abstraites de signe et table de multiplication

Un autre exemple d'interprétation abstraite consiste à abstraire les valeurs entières par des intervalles : il s'agit de l'*analyse d'intervalle* (*range analysis* en anglais). La sémantique abstraite associée correspond à l'arithmétique d'intervalles, abondemment décrite sur Internet [4]. Contrairement à l'exemple des signes, le treillis des intervalles est infini, il est donc nécessaire de recourir au *widening* pour garantir que l'analyse termine bien. Il s'agit de remplacer les bornes gênantes (i.e. les bornes supérieurs qui croissent ou les bornes inférieurs qui décroissent au fil des itérations) par l'infini, de sorte qu'elle ne puisse plus varier. Cela se fait donc au détriment de la précision de l'analyse.

2.1.2 L'analyse de pointeurs proposée par CompSys

Le nouvel algorithme d'analyse de pointeurs en cours de conception par CompSys s'appuie sur l'interprétation abstraite et sur une analyse d'intervalle (qui est préalablement fournie) pour déterminer la portion d'espace mémoire pointée par chaque pointeur et, ce faisant, si deux pointeurs sont susceptibles d'être des *alias* (ou non). Il se fonde sur une représentation particulière des pointeurs.

Un pointeur peut être représenté de deux manière distinctes : soit comme une simple adresse, soit comme le couple formé par un identifiant de zone mémoire, d'une part, et un décalage (*offset*) dans cette zone mémoire, d'autre part. La première représentation correspond au fonctionnement

4. Sur l'arithmétique d'intervalles, voyez l'article Wikipédia : `http://en.wikipedia.org/wiki/Interval_arithmetic`

de la plupart des machines modernes. Malheureusement, elle se prête difficilement à l'analyse statique.[5] La seconde représentation est plus féconde. En effet, elle peut être abstraite de la façon suivante :
— l'identifiant de zone mémoire est réduit au numéro de l'instruction ayant alloué cette zone ;
— le décalage dans la zone mémoire est abstrait par un intervalle.

Ainsi, si un programme compte n sites d'allocation distincts, les valeurs abstraites de pointeurs sont des n-uplets d'intervalles, où le i-ième élément ($0 \leq i < n$) correspond à (une surapproximation de) l'intervalle d'*offset* balayée par le pointeur dans les zones mémoire allouées par la i-ième instruction d'allocation (ou l'intervalle vide si ce pointeur ne pointe jamais sur une zone mémoire alloué par cette instruction).

En utilisant une analyse d'intervalle (cf. supra) préalable, l'algorithme est capable de calculer une sémantique abstraite assez précise de l'arithmétique des pointeurs (i.e. les opérations du type `p + i` où `p` est un pointeur et `i` un entier) en sommant chaque intervalle d'*offset* de `p` avec l'intervalle de `i` calculé par ladite analyse d'intervalle.

Un prototype en Prolog a été développé par l'équipe CompSys pour permettre de tester l'algorithme proposé au fur et à mesure de son développement.

2.1.3 Détails du prototype Prolog

L'analyse implémentée en Prolog s'effectue sur une abstraction de programme sous forme d'un langage d'assemblage dont les principales instructions au moment de la rédaction du rapport[6] sont :
— `add(Dst, Src0, Src1)` : Enregistre la somme de `Src0` et `Src1` dans `Dst` ;
— `sub(Dst, Src0, Src1)` : Enregistre la différence de `Src0` et `Src1` dans `Dst` ;
— `mul(Dst, Src0, Src1)` : Enregistre la produit de `Src0` et `Src1` dans `Dst` ;
— `div(Dst, Src0, Src1)` : Enregistre la quotient de `Src0` par `Src1` dans `Dst` ;
— `move(Dst, Src)` : Copie le contenu de `Src` dans `Dst` ;
— `alloc(Address, Size)` : Alloue `Size` et enregistre l'adresse dans `Address` ;
— `leq(Pred, Src0, Src1)` : Enregistre 1 dans `Pred` si `Src0` $\leq$ `Src1`, 0 sinon ;
— `neq(Pred, Src0, Src1)` : Enregistre 1 dans `Pred` si `Src0` $\neq$ `Src1`, 0 sinon ;
— `bnz(Pred, Label)` : Saute à l'instruction `Label` si `Pred` est non-nul ;
— `jmp(Label)` : Saute inconditionnellement à l'instruction `Label` ;
— `const(Dst, Constant)` : Enregistre une constante `Constant` sous le nom `Dst` ;
— `load(Address, Dst)` : Charge le contenu de la cellule pointée par `Address` dans `Dst` ;
— `store(Address, Src)` : Mémorise `Src` dans la cellule pointée par `Address` ;
— `halt` : Arrête la machine virtuelle ;
— `phi(Dst, [Src0, ... SrcN])` : Une ϕ-fonction qui affecte à `Dst` l'un des `Src`[7].
— `intersect(Dst, Src, Min, Max)` : Enregistre, dans le cadre de l'analyse d'intervalle, l'intersection de la valeur abstraite de `Src` avec l'intervalle [`Min` ; `Max`] dans `Dst`.

```
?- prog(8, I), evalProg(I, 0, 0, [0, 0, 0, 0, 0, 0], [], N, M, S).
I = [const(one, 1), const(minus_one, -1), const(five, 5), alloc(call, 5),
     add(add_ptr, call, five), phi(xl_0, [call, incdec_ptr]),
     phi(xu_0, [add_ptr|...]), leq(cmp, xl_0, xu_0), bnz(..., ...)|...],
N = 5,
M = [1, 1, 1, -1, -1, -1],
S = [ (cmp, 0), (xu_0, 2), (xl_0, 3), (incdec_ptr1, 2), (incdec_ptr, 3),
      (xu_sigma, 3), (xl_sigma, 2), (cmp, 1), (..., ...)|...]

?- prog(8, I), rangeFixedPoint(I, [], RT).
I = [const(one, 1), const(minus_one, -1), const(five, 5), alloc(call, 5),
      add(add_ptr, call, five), phi(xl_0, [call, incdec_ptr]),
      phi(xu_0, [add_ptr|...]), leq(cmp, xl_0, xu_0), bnz(..., ...)|...],
RT = [ (xu_sigma, range(0, 5)), (xl_sigma, range(0, 5)),
       (xu_0, range(-1, 5)), (xl_0, range(0, 6)),
       (incdec_ptr1, range(-1, 4)), (incdec_ptr, range(1, 6)),
       (add_ptr, range(5, 5)), (call, range(..., ...)), (..., ...)|...]
```

FIGURE 7 – Exécutions concrète puis abstraite du programme de la figure 5

5. Les adresses dépendent fortement du contexte (système) et peuvent varier grandement d'une exécution à l'autre. Par ailleurs, elles sont fortement dépendantes du contexte du programme : une variable locale n'aura pas la même adresse dans le pile en fonction de la chaîne d'appelants.

6. Le jeu d'instruction n'a pas encore été stabilisé, et ne le sera sans doute jamais, s'agissant d'un prototype.

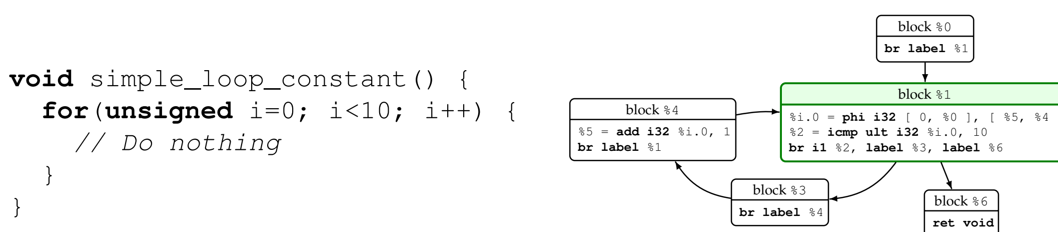
7. Les ϕ-fonctions sont expliquées dans la section 2.2

Le prototype est capable à la fois d'exécuter concrètement un programme écrit dans le langage ci-dessus mais aussi de l'interpréter abstraitement. Dans ce cas, il utilise l'instruction `intersect`, qui est sinon inutile, et ignore les conditionnements des branchements, qui sont pourtant nécessaire pour l'exécution concrète. La figure 7 montre le résultat de ces deux exécutions sur un même programme : s'agissant de l'exécution concrète, le contenu final de la mémoire `M` (qui ne contient ici que 6 cases), le nombre `N` de cellules allouées et les valeurs `S` des registres virtuels sont affichés ; s'agissant de l'exécution abstraite, les intervalles `RT` des registres virtuels sont indiquées.

Le traducteur que j'ai dû écrire devait être capable à la fois de générer du code pouvant être exécuté concrètement mais aussi interprété abstraitement par le prototype Prolog.

2.2 Contexte technologique – LLVM

Le traducteur s'appuie sur LLVM. LLVM est une infrastructure de compilation, constituée de plusieurs *front-ends* [8], optimiseurs et *back-ends* [9], organisée autour d'un langage intermédiaire commun – le *bytecode* LLVM. Cette représentation intermédiaire utilise une quantité illimitée de registres virtuels et suit la forme SSA (Static Single Assignment), ce qui signifie que chacun de ces registres (ou variable) n'est affecté que par une seule instruction [10]. Un module LLVM [11] est constitué de fonctions qui sont elles-mêmes décomposées en blocs, qui correspondent aux noeuds dans son graphe de flot de contrôle.



Exemple repris du rapport de Gabriel Radanne [4]

FIGURE 8 – Une fonction C et sa représentation LLVM

La figure 8 montre un exemple simple de représentation LLVM. Le bloc `%0` correspond au point d'entrée de la fonction, le bloc `%1` à la condition du `for`, le bloc `%3` au corps de la boucle (qui est vide), le `%4` à l'instruction de passage [12] et le bloc `%6` au point de sortie de la fonction. Comme la variable `i` est affectée deux fois dans le code source (à l'initialisation de la boucle et dans l'instruction de passage), cela aboutit à deux versions distinctes de la variable `i` (selon la définition de la forme SSA). Or, la comparaison peut s'effectuer sur l'une ou l'autre des versions selon les circonstances. Les ϕ-fonctions permettent de résoudre ce problème : l'instruction `phi` dans le bloc `%1` affecte à `i.0` la valeur 0 si l'on vient du bloc `%0` (i.e. l'on vient de rentrer dans la boucle) ou le contenu du registre virtuel `%5` (i.e. le résultat de l'incrémentation) si l'on vient du bloc `%4` (i.e. l'on vient d'itérer).

Chaque bloc LLVM finit par une instruction terminale, qui peut être un branchement inconditionnel (dans les blocs %0, %3 et %4), un branchement conditionnel (dans le bloc %1, dans cet exemple, l'on saute au bloc %3 si %2 est vrai et au bloc %6), un branchement multiple (c'est-à-dire un `switch`, non illustré ici) ou un retour à l'appelant (dans le bloc %6). La condition est séparée du branchement et relève d'une instruction séparée, `icmp`. Dans l'exemple, `%2` est un booléen (i.e. un entier à un 1 bit, i.e. un `i1`) indiquant si `%i.0` est strictement inférieur à 10 en utilisant une comparaison non-signée (`ult` pour *unsigned less than*).

8. Un *front-end* traduit un programme écrit dans un langage de programmation utilisé par des humains (par exemple, le C pour le *front-end Clang*) vers une représentation intermédiaire
9. Un *back-end* traduit la représentation intermédiaire vers du code machine.
10. Attention, même si le programme est sous forme SSA, il est possible qu'un registre virtuel soit affecté plusieurs fois **à l'exécution**, par exemple si l'instruction qui l'affecte est dans une boucle...
11. Un module LLVM est l'équivalent d'une unité de traduction (*translation unit*) en C.
12. L'*instruction de passage* d'un `for` est l'instruction exécutée entre deux itération de la boucle (typiquement, l'incrémentation d'un itérateur).

Les optimisations et les analyses de LLVM prennent la forme de *passes* qui peuvent être exécutées (et même chaînées) en utilisant l'utilitaire `opt`. Celui-ci inclut déjà une collection de *passes* [13], mais possède également un mécanisme d'extension permettant de charger des *passes* tierces, qui prennent alors la forme de simples bibliothèques dynamiques [14]. Il existe différents types de *passes* en fonction des éléments sur lesquelles elles sont appelées (des modules, des fonctions, des boucles naturelles, des régions *single-entry-single-exit*, etc.).

Il est parfois utile de pouvoir distinguer entre deux usages d'une même variable lorsque les chemins suivis pour partir de sa définition vers ses utilisations sont différents et apportent des informations distinctes sur cette variable. La **forme e-SSA** étend la forme SSA en scindant ces variables en différentes versions, de sorte que les connaissances apportées par le chemin soient homogènes pour chacune de ces versions.

Prenons l'exemple d'un *si-alors-sinon* conditionné par un test sur une variable `i`. Dans ce cas, nous savons que, dans le bloc *alors*, `i` est contraint par le test alors que, dans le bloc *sinon*, `i` est contraint par la négation du test. Ces deux connaissances contradictoires ne peuvent pas être associées à la même variable `i`. Dans ce cas, la forme e-SSA divise la variable `i` en deux nouvelles versions, `i.then` et `i.else`, ce qui résout le problème.

2.3 Réalisation et résultats

L'implémentation du traducteur consiste en une *passe* LLVM, d'environ 800 lignes de code, appelée sur chaque fonction. Celle-ci réalise alors les opérations suivantes :

1ère étape Création, pour chaque bloc de code LLVM, d'un bloc de code Prototype ;

2ème étape Recensement de l'ensemble des constantes entières utilisées par la fonction analysée ;

3ème étape Traduction instruction par instruction du code LLVM en code Prototype ;

4ème étape Linéarisation des blocs de code Prototype puis calcul de leurs positions respective ;

5ème étape Écriture du code Prototype dans un fichier.

Le langage reconnu par le prototype ne permettant pas d'utiliser directement des constantes comme opérandes de la plupart des instructions, il est nécessaire de construire au début du programme généré une *constant pool*, c'est-à-dire une série d'instructions servant uniquement à nommer les différentes constantes du programme. L'étape 2 sert à cela.

Par ailleurs, les branchements (`bnz` ou `jmp`) n'utilisent pas de *labels* mais directement l'ordinal de la destination (i.e. `jmp(21)` saute à la 22ème instruction du programme). Or, la position des blocs ne peut pas être connu à l'avance car elle dépend de la taille des différents blocs donc de la quantité d'instruction qui aura été générée dans chacun des blocs. Aussi, la résolution des sauts est laissée en suspens jusqu'à l'étape 4.

L'arithmétique des pointeurs suit, dans LLVM, une logique particulière et utilise une instruction spéciale – `GetElementPtr`, ou `GEP`. Celle-ci permet, à partir d'un pointeur sur un tableau et d'un indice, de calculer un pointeur sur un élément particulier de ce tableau, et, à partir d'un pointeur sur une structure et d'un indice constant, de calculer un pointeur sur un champ particulier de cette structure. Cette instruction permet un chaînage implicite et accepte donc un nombre illimité d'indices [15]. Toutefois, un `GEP` n'effectue jamais de déréférencement [16]. Le traducteur transforme les `GEP` en une série de `mul` et de `add`.

Pour utiliser le traducteur, il faut d'abord compiler le code C vers la représentation intermédiaire

13. Parmi les *passes* fournies par LLVM figurent *mem2reg* qui élève certaines variables automatiques (allouées dans la pile) au rang de registre virtuel, ou encore *instnamer* qui nomme les registres virtuels (qui, sinon, peuvent être anonymes).

14. Pour dire à `opt` de charger une *passe* tierce, il faut lui fournir l'argument `-load` suivi du nom de la bibliothèque dynamique contenant la *passe*.

15. Cela permet d'accéder en une seule instruction à une cellule d'un tableau multidimensionnel ou à un champ d'une structure imbriquée dans une autre structure

16. En LLVM, les instructions de déréférencement sont `load` et `store`.

LLVM en utilisant *Clang* [17], puis utiliser l'optimisateur LLVM pour charger la *passe* et l'exécuter [18]. La figure 9 montre le résultat de la traduction d'un code C par notre traducteur. La génération des instructions `intersect`, utilisée par l'interprétation abstraite, nécessite d'exécuter au préalable la *passe* vSSA [19], qui calcule la forme e-SSA (cf. supra).

```c
#define SIZE 5
void main(void) {
  char *x = malloc(SIZE);
  char *xl = x;
  char *xu = x + SIZE;
  while (xl < xu) {
    *xl = 1, *xu = -1;
    xl++, xu--;
  }
  printf("x[3]=%d\n",x[3]);
  return;
}
```

```
[
  const(const_1, 5),
  const(const_2, 1),
  const(const_3, -1),
  alloc(call, 5),
  add(add_ptr, call, const_1),
  jmp(6),
  phi(ins_xl_0, [call, incdec_ptr]),
  phi(ins_xu_0, [add_ptr, incdec_ptr1]),
  leq(tmp_1, xu_0, xl_0),
  bnz(tmp_1, 17),
  jmp(11),
  store(xl_0, const_2),
  store(xu_0, const_3),
  add(incdec_ptr, xl_0, const_2),
  add(incdec_ptr1, xu_0, const_3),
  jmp(6),
  halt
]
```

FIGURE 9 – Traduction d'un code C.

Il s'agit d'une implémentation de test, insuffisamment validée mais qui se veut prototype de test des propositions d'algorithmes avant une implémentation finale dans LLVM. Ce prototype a été documenté.

3 Réalisation 2 : Expérimentations autour de l'algorithme de Steensgaaard

3.1 Objectif

Pour confronter cette nouvelle analyse des pointeurs aux méthodes déjà existantes, il m'a fallu également implémenter l'algorithme de Steensgaaard dans LLVM. Cette méthode (cf Section 1.3) consiste à assimiler les *points-to set* à des types, les *types de Steensgaaard*, puis à procéder à une *inférence de type*. Cela aboutit à associer à chaque pointeur son type de Steensgaaard, donc l'ensemble des zones mémoire sur lequel il pointe.

Cet algorithme est sensible aux particularités de la représentation intermédiaire LLVM et, en particulier, à la forme SSA. En effet, cette forme peut aboutir à scinder les pointeurs du code originel en plusieurs versions distinctes, qui peuvent ainsi avoir des types de Steensgaaard différents. Elle introduit également des ϕ-fonctions, que l'algorithme original ne gère pas. Par ailleurs, LLVM travaillant sur des registres virtuels, dont on ne peut prendre l'adresse, la contrainte basique [20] de Steensgaaard est éliminée. Cette implémentation a aussi été l'occasion d'examiner ces différences.

3.2 Réalisation et premiers résultats expérimentaux

Mon analyse de Steensgaaard prend la forme d'une `ModulePass` LLVM d'environ 300 lignes qui génère le code GraphViz correspondant au graphe *points-to* du programme fourni. Ce graphe comporte deux genres de noeuds : ceux représentant les registres virtuels (rectangle marron) et ceux figurant les types de Steensgaaard (ellipse verte). Il y apparaît aussi deux espèces d'arcs : ceux en pointillés expose que tel registre est de tel type (i.e. qu'il pointe sur tel ensemble de zones mémoire) tandis que ceux en trait plein indique que tel type est obtenu en déréférançant un registre de tel autre type. La figure 10 montre le résultat de notre *passe* sur un exemple de programme C [21].

17. *Clang* est le *front-end* officiel de LLVM pour les langages de la famille du C. Il suffit de lui passer les options `-c --emit-llvm` pour qu'il s'arrête à la génération du bytecode LLVM au lieu de faire une compilation complète.

18. Dans notre cas, cela consiste à lancer la ligne de commande `opt -load PrologTranslator.dylib -mem2reg -instnamer -prolog-translate prog.bc` où `prog.bc` est le nom du module LLVM à traduire.

19. Nous avons utilisé la *passe* vSSA disponible sur `https://code.google.com/p/range-analysis/`

20. La contrainte basique est celle associée au motif d'instruction `p = &a`.

21. Il s'agit du même exemple que celui donné dans le papier de Steensgaaard[3].

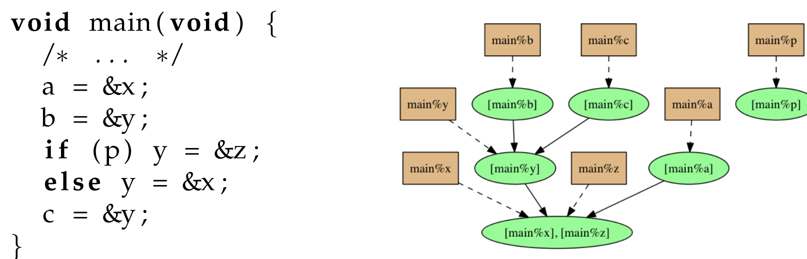
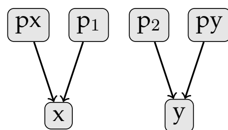


FIGURE 10 – Analyse de Steensgaaard par notre *passe* d'un programme simple

Le *passe* procède en trois étapes : (i) association à chaque pointeur du programme d'un type de Steensgaaard dédié ; (ii) parcours du programme instruction par instruction pour exécuter les contraintes ; (iii) émission du code GraphViz. Les deux seules genres de contraintes à pouvoir apparaître sont les *simples* (i.e. `p = q`) et les *complexes* (i.e. `p = *q`). Ces dernières correspondent aux instructions `load` et `store` tandis que les premières, à défaut d'affectation, apparaît au niveau des ϕ-fonctions. Notre analyse ne cherchant pas à distinguer à l'intérieur des zones mémoire, l'instruction `GEP` (i.e. l'arithmétique des pointeurs) est traitée par une contrainte *simple*.

```
int x, y;
int *p, *px, *py;
px = &x;
p = px; // p_1
py = &y;
p = py; // p_2
```

Steensgaaard SSA



Steensgaaard classique

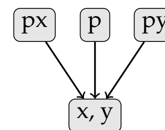


FIGURE 11 – Comparaison entre notre *passe* et Steensgaaard classique sur un exemple

La figure 11 montre que l'algorithme de Steensgaaard exécuté sur la forme SSA utilisée par LLVM est plus précis que sa version originale [22]. En distinguant entre les deux définitions de `p`, il est capable de déterminer que la première version de `p` ne pointe jamais sur `y` et que la seconde version de `p` ne pointe jamais sur `x`. Ce faisant, elle évite de fusionner les noeuds `x` et `y`, ce qui offre incidemment un gain de précision sur `px` et `py` [23].

4 Conclusion

Ce TER m'a permis de découvrir quelques techniques d'analyse statique et le *framework* LLVM qui s'avère extrêmement pratique pour étudier et transformer des programmes. À l'aide de ce *framework* j'ai pu développer deux analyses de programme C, un algorithme existant et une traduction vers un langage *ad-hoc*. Les perspectives pour ces travaux seraient de valider de manière plus approfondie ces deux réalisations.

Sans doute vais-je pouvoir réutiliser ces nouvelles connaissances. J'ai pu également développer mes compétences en C++ et même découvrir un petit peu le milieu de la recherche.

Références

[1] Christophe Bacara. Implémentation dans llvm d'analyses de pointeurs, rapport de stage de L3, 2013.

[2] Lars Ole Andersen. Program analysis and specialization for the c programming language, 1994.

[3] Bjarne Steensgaard. Points-to analysis in almost linear time. Technical report, Microsoft Research, 1996.

[4] Gabriel Radanne. Synthesis of ranking functions using extremal counterexamples, rapport de stage de Master, 2014.

22. En fait, la forme SSA permet de rendre *sensibles au flot de données* une analyse qui ne l'est pas, comme l'algorithme de Steensgaaard.

23. Sur cet exemple en particulier, qui est le même que la figure 4, notre *passe* est même plus précise que l'analyse d'Andersen.