

# Traçage de Code

POUILLER Jérôme

TESSARI Marco

RAMIREZ Guillaume

EPITA - Septembre 2003

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Qu'est ce que le traçage de code ?	3
1.2	Etat de l'art	3
1.3	Qu'est ce qu'un décompilateur ?	5
<b>2</b>	<b>De l'assembleur à la représentation</b>	<b>6</b>
2.1	Désassemblage	6
2.2	Pourquoi une représentation intermédiaire	6
2.3	Présentation de différentes représentations intermédiaires	6
<b>3</b>	<b>Analyse du Dataflow</b>	<b>8</b>
3.1	But de l'analyse du <i>dataflow</i>	8
3.2	Elimination des registres	8
3.3	Les temporaires mortes	9
3.4	Repérage des saut conditionnels	9
<b>4</b>	<b>Traitement des Fonctions</b>	<b>10</b>
4.1	Repérage des <i>Basic Blocks</i> et des <i>Function Blocks</i>	10
4.1.1	Construction des Basic Blocks	10
4.1.2	Création du Call Graph	11
4.2	Repérage des signature des fonctions	12
4.2.1	Fonctions prédéfinies	12
4.2.2	Les autres fonctions	12
4.3	Problèmes sur le repérage des fonctions	12
<b>5</b>	<b>Inlining des instructions</b>	<b>12</b>
5.1	Propagation de copie des temporaire	12
5.2	Elimination des temporaire inutile	13
5.3	Variables locales	13
<b>6</b>	<b>Reconnaissance des structures de controles</b>	<b>14</b>
6.1	Repérage des boucles	14
6.2	Repérage des structures conditionnelles	14
6.3	Finalisation	14
<b>7</b>	<b>Analyse de type</b>	<b>15</b>
7.1	Présentation	15
7.2	Inférence de type	15
7.3	Exemple	17
7.4	Structures contre Tableaux	19
<b>8</b>	<b>Conclusion</b>	<b>21</b>
8.1	Etat des lieux	21
	<b>Références</b>	<b>22</b>

# 1 Introduction

## 1.1 Qu'est ce que le traçage de code ?

Le traçage de code se rapproche beaucoup de l'ingénierie inversée (reverse engineering). C'est une technique permettant de déterminer l'utilité et le fonctionnement d'un programme. Le programme est alors assimilée à une boîte noire dont on essaye de trouver qu'elle est la fonction qui permet à partir d'une certaine entrée, d'obtenir la sortie constatée.

Pour ce faire on utilise des 'rétrotéchniques' qui nous permettent de passer d'un code compréhensible par la machine, mais pas par l'homme, à un code lisible par l'homme mais plus par la machine. Il s'agit de décompiler ou de désassembler par exemple.

Le traçage de code peut être utiliser pour un grand nombre de raisons :

- Analyser et comprendre un code inconnu (pour déterminer un algorithme par exemple),
- Récupérer du code source perdu (on ne possède plus qu'un exécutable),
- Déterminer l'existence de virus ou de code parasite dans un programme,
- Déterminer des failles de sécurité, ou des erreurs (bogues),
- Déterminer les protections logicielles,
- Comprendre le fonctionnement d'un logiciel insuffisamment documenté, pour, par exemple, faire un logiciel qui interopère avec le logiciel tracé.

Ceux-ci ne sont que des exemples d'applications, on peut en trouver bien d'autres. Leur nombre fait du traçage de code une pratique très courante.

Pourtant le traçage de code pose des problèmes legaux. Les logiciels sont soumis aux droits d'auteurs qui considèrent l'oeuvre logicielle comme une oeuvre de l'esprit ; les éléments de forme et la structure de l'oeuvre originale sont protégeables. Plus particulièrement en France, le droit d'auteur ainsi que le droit pénal et la loi Godfrain donnent une qualification pénale au fait de pénétrer dans un système, élément de la propriété d'autrui. Par ces lois les programmes sont protégés des techniques de traçage de code. Une exception est faite toutefois si le but est d'assurer l'interopérabilité, c'est-à-dire l'articulation des logiciels les uns avec les autres. Pour le cas de la décompilation en France on pourra se reporter à [Bas02], avec un exemple d'application [Dus00] et sur l'ingénierie inverse dans le monde [eAF95]).

En résumé les techniques de traçage de code ne doivent être utilisées que sur des logiciels dont on est l'auteur, sur des logiciels dont l'accès aux sources nous est autorisé, ou alors à des fins de création d'un logiciel voulant échanger des données avec un logiciel trop peu documenté.

## 1.2 Etat de l'art

Le traçage de code devient de plus en plus indispensable, et ce retrouve à tous les niveaux. Les développeurs de microprocesseurs ont bien compris cela, et sur les proces-

seurs actuels, tel que le PowerPC ou le Pentium d'Intel, on trouve un certain nombre de fonctions facilitant le développement de logiciels. Ils intègrent des capacités étendues de traçage de code en temps réel, des broches supplémentaires permettent de suivre les changements d'adresses, d'autres options incluent le contrôle du processeur (point d'arrêt, démarrage, etc...), des accès aux registres et à la mémoire ainsi qu'un outil de mise au point. L'avantage d'une telle méthode est que l'on peut travailler sur n'importe quel exécutable en temps réel sans besoin de prétraitement, et surtout il est utilisable sur des architectures multipipeline dont l'assembleur et l'enchaînement des instructions est souvent dur à comprendre. Pour plus d'informations sur le PowerPC voir [IBM]) et [Int] pour le Pentium. Ces fonctions sont exploitées par des logiciels comme les débogueurs, par exemple GDB qui utilise des points d'arrêts hardware, ou par des interfaces hardware tel Nexus. Elles ne sont donc jamais utilisées directement par un développeur lambda.

Ce qui nous amène à parler d'autres outils de traçage de code que sont les débogueurs. Le but d'un débogueur est de permettre de suivre ce qui se passe "à l'intérieur" d'un programme de façon dynamique, c'est à dire quand il s'exécute, ou de savoir ce qu'il faisait lors d'un arrêt volontaire ou involontaire. Pour une utilisation optimale d'un débogueur l'exécutable doit être compilé en mode "debug". Ceci permet d'ajouter des informations tels que des symboles (noms de variables, de fonctions, ...) et une correspondance entre la ligne assembleur et le code source. Pour plus de renseignement sur sa structure interne voir [Dev03]. Ainsi lors de son utilisation interactive, le débogueur pourra faire le lien entre le code machine et le code source. Les informations sont donc facilement exploitables par le développeur. Les débogueurs les plus connus sont GDB <sup>1</sup> et SoftIce <sup>2</sup>.

Mais bien souvent nous n'avons pas les sources donc l'utilisation d'un débogueur s'avère délicate. On peut alors se servir d'outils comme des traceurs d'appels système comme la commande 'strace' ou d'une bibliothèque dynamique 'ltrace'. Ceux-ci sont basés autour de l'appel système 'ptrace' qui permet de contrôler l'exécution d'un processus depuis le processus parent. L'utilisation de ces commandes reste très simple, il suffit de lancer la commande avec en paramètre la commande que l'on veut exécuter. Nous aurons alors une trace de tous les appels système ou de bibliothèque externe. Il est à noter que ceci n'est nullement interactif comme avec un débogueur. Sur la trace des appels systèmes on obtiendra des informations de haut niveau comme la valeur des paramètres, ainsi que la valeur de retour des fonctions, dans un format facilement lisible par l'homme.

Mais parfois on aimerait pouvoir faire des modifications à l'exécutable. On passe alors par un désassembleur pour transformer le code machine en hexadécimal en langage assembleur lisible par l'homme. Le plus gros problème du désassemblage est déterminer les parties qui sont du code et les parties qui sont des données, car sur les machines actuelles, tout est représenté de la même manière. On trouve donc une grande différence entre des désassembleurs qui se contentent de traduire les instructions comme la commande 'objdump -D' où l'on ne distingue pas le code des données et un désassembleur commercial comme IDAPro <sup>3</sup>. Celui-ci sépare automatiquement les données du code et reconnaît les points d'entrée des bibliothèques communément utilisées par les compilateurs. Le dé-

---

<sup>1</sup>[<http://sources.redhat.com/gdb/>]

<sup>2</sup>[<http://www.compuware.com>]

<sup>3</sup>[<http://www.datarescue.com/idabase/ida.htm>]

assemblage est une opération statique, c'est-à-dire qu'on désassemble l'ensemble d'un exécutable sans l'exécuter, il faut alors le lire pour en comprendre son fonctionnement. Ceci requiert une bonne connaissance de l'assembleur et la lecture de nombreuses lignes de code. Ceci est d'autant plus handicapant que l'assembleur dépend du processeur. Par contre on peut alors modifier et réassembler le code pour obtenir un nouvel exécutable.

En résumé, actuellement si on veut tracer le code d'un programme il faut soit posséder les sources pour pouvoir utiliser un débogueur, soit on se limite au traçage des appels système et bibliothèques dynamiques, voir au pire aux instructions assembleur. De plus si on veut pouvoir modifier l'exécutable on doit passer par de l'assembleur aussi. Cela n'est vraiment pas pratique car de moins en moins de personnes maîtrisent l'assembleur. De plus il faut lire un grand nombre de lignes donc cela requiert beaucoup de temps.

### 1.3 Qu'est ce qu'un décompilateur ?

En analysant l'état actuel du traçage de code on s'aperçoit qu'il manque un outil qui permettrait de passer d'un exécutable en langage machine vers en langage de haut niveau, idéalement un langage connu par l'utilisateur, avec un minimum de lignes. C'est ce que fait un décompilateur. Son fonctionnement est très similaire à un désassembleur, c'est-à-dire qu'il agit d'une façon statique sur un exécutable, sauf que le format de sortie diffère.

Le décompilateur suis un processus similaire à celui d'un compilateur, mais en sens inverse. Les différentes étapes sont :

- Décoder le fichier binaire,
- Décoder les instructions machines en langage assembleur, comme un désassembleur,
- Faire une analyse sémantique de base pour reconnaître des types de bas niveau comme les `long`, et pour simplifier les instructions,
- Charger les informations dans une forme intermédiaire qui sera utilisée par la suite indépendamment du langage assembleur source,
- Analyser le flot de données pour enlever de la représentation intermédiaire les aspects les plus bas niveaux comme les registres et les références à la pile,
- Analyser le flot de contrôle pour reconnaître les boucles et les sauts conditionnels, ainsi que leur degré d'imbrication,
- Analyser le typage pour retrouver les types du langage haut niveau,
- Générer le code de haut niveau depuis le code intermédiaire.

Dans cet article nous allons nous intéresser exclusivement à la décompilation. Dans un premier temps nous allons voir les différentes formes intermédiaires existantes. Puis nous allons nous intéresser aux techniques à base de graphes pour l'analyse de flot de données et de contrôles. Enfin nous allons voir comment appliquer l'inférence de type pour la reconstruction de type. Nous allons conclure sur les différents compilateurs existants.

## 2 De l'assembleur à la représentation

### 2.1 Désassemblage

Le désassemblage consiste à partir d'un programme déjà compilé, donc compréhensible uniquement par la machine, d'en retrouver une représentation intermédiaire. C'est-à-dire une représentation entre le code source original du concepteur incompréhensible directement par la machine et le code binaire auquel ne peut être associé qu'un seul et unique code assembleur.

### 2.2 Pourquoi une représentation intermédiaire

Le fait de transformer le code d'un programme compilé, donc binaire, en sa représentation code assembleur unique permet le passage d'un langage incompréhensible à l'homme à un langage permettant à celui-ci des analyses dessus. Mais le but est bien évidemment de ne pas s'arrêter à ce code assembleur, mais d'essayer de remonter vers une représentation visant le code source original.

### 2.3 Présentation de différentes représentations intermédiaires

- La représentation assembleur. Elle est donc la représentation de base car ayant une relation bijective avec le code machine.
- La représentation RTL. Ou Register Transfer Level est une des représentation intermédiaire utilisée par GCC entre autre. Appelée également "code 3 adresses", elle fait parti des langages intermédiaires les plus utilisés, c'est un langage impératif dont les constructions de base correspondent directement à des instructions du processeur, mais opèrent sur des pseudo-registres en nombre arbitraire.

Exemple de code RTL :

```
signal r124_Q : std_logic_vector( 3 downto 0 );
?
P1 : process
begin
?
wait until clk_bit'event and clk_bit = '1';
if ( n420 = '1' ) then
r124_Q <= (Sy_1_vector(3), Sy_1_vector(2), Sy_1_vector(1),
Sy_1_vector(0));
r142_Q <= (Sx_1_vector(3), Sx_1_vector(2), Sx_1_vector(1),
Sx_1_vector(0));
end if;
?.
End process;
?
```

- La représentation SSA. Ou Single Static Assignment est une forme de représentation intermédiaire dans lequel chaque variable du code à une seule et unique défini-

tion. Code original :

```
i = 0  
i = i + 1  
j = func(i)  
i = 2
```

Code SSA :

```
i0 = 0  
i1 = i0 + 1  
j0 = func(i1)  
i2 = 2
```

## 3 Analyse du Dataflow

### 3.1 But de l'analyse du *dataflow*

Le but de l'analyse du *dataflow* est de passer du niveau assembleur à un langage possédant des structures de contrôle. Pour cela, nous allons analyser le Dataflow, construire un *graphe de contrôle*, puis, analyser ce graphe de contrôle. Nous nous sommes fortement inspiré des algorithmes présentés par A. Appel [App98], de la thèse de Cristina Cifentes [eAF95] et de nos connaissances acquise lors du projet Tiger pour écrire cette partie.

### 3.2 Elimination des registres

En éliminant les registres, nous allons avoir une représentation à peu près équivalente au LIR du compilateur Tiger de A. Appel [App98].

On veut transformer les registres en temporaires dont la durée de vie sera la plus courte possible. Effectivement, plus la durée de vie est courte, plus facile seront les manipulations et meilleurs seront les résultats des algorithmes suivants. On commence par transformer le dataflow en graphe orienté. Chaque noeud de ce graphe correspond à une instruction et un arc indique la possibilité de passer d'une instruction à l'autre. A chaque instruction, on associe l'ensemble des temporaires utilisées ("*use*") et définie ("*def*"). Par exemple, dans l'instruction `add ax, bx`, `ax` et `bx` sont utilisés et `ax` est définie. On rappelle que les temporaires sont, soit des registres, soit des variables sur la pile. C'est à dire que `ax`, `[sp + 2]` et `[sp + 6]` sont trois temporaires. On associe ensuite un ensemble de temporaires "*liveIn*" et "*liveOut*" initialement vides. On parcourt ensuite, le graphe en appliquant à chaque noeud  $n$  les équations suivantes :

$$\begin{aligned} \text{liveIn}(n) &= \text{use}(n) \cup (\text{liveOut}(n) - \text{def}(n)) \\ \text{liveOut}(n) &= \bigcup_{s \in \text{successeurs}(n)} \text{liveIn}(s) \end{aligned}$$

On parcourt le graphe jusqu'à ce qu'il n'y est plus de changements. On peut ainsi connaître la durée de vie de chaque variable.

On commence alors à renommer les temporaires. La durée de vie des temporaires doit être la plus courte possible. On commence donc par la première instruction et on donne le même nom à la temporaire tant que celle si est vivante et qu'elle n'est pas utilisée et définie dans la même instruction. Sinon, on change de nom. On en profitera pour modifier l'écriture de l'assembleur pour une syntaxe plus proche d'un langage haut niveau. Par exemple :

```
add cx, ax
    def: cx
    use: cx, ax
    livein: bx, ax, cx
    liveout: bx, cx
mov ax, bx
    def: ax
```

```

    use: bx
    livein: bx, cx
    liveout: cx
mov [sp + 8], cx
    def: [sp + 8]
    use: ax
    livein: cx
    liveout:

```

donne

```

t3 = t2 + t1
t5 = t4
t6 = t3

```

On remarque que cette méthode peut avoir des problèmes avec certaines architectures, gestion de la pile ou optimisation du processeur. Effectivement, on considère que lorsque le compilateur n'a pas de assez de registre, il stocke ses temporaires sur la pile. Sur certaines architectures n'ayant pas de mode d'adressage indexé ou bien par soucis d'optimisation, on peut gérer la pile autrement (en utilisant `push` et `pop` par exemple). Notre méthode peut alors devenir moins satisfaisante.

Nous nous sommes ainsi débarrassé de la notion de registres. Il existe d'autres méthodes pour calculer les durées de vie des variables. Nous avons montré ici une méthode dérivée de A. Appel [App98]. Dans sa thèse Cristina Cifentes [eAF95] utilise une méthode plus complexe que Appel, mais, elle n'apporte pas grand intérêt. Vous pouvez trouver une implémentation de cet algo dans le compilateur Tiger de A. Appel [App98] dont une implémentation est faite par le LRDE<sup>4</sup>

### 3.3 Les temporaires mortes

Avant de continuer l'analyse du Graphe de Contrôle, nous pouvons éliminer les temporaires dont la vie est nulle (c'est à dire, une temporaire définie, mais, jamais utilisé). Cette optimisation est normalement faite par le compilateur. Elle est tout de même utile dans le décompilateur, car, après plusieurs transformations, des temporaires mortes peuvent apparaître. Cette optimisation simplifiera les étapes suivantes.

### 3.4 Repérage des saut conditionnels

On aimerait maintenant avoir des instructions de sauts conditionnels sous une forme pour humaine (ie : `jcond (t1 > t2) ll`). Sur la plupart des processeurs, les sauts conditionnels dépendent de l'état du registre de 'flags'. Nous devons donc repérer les sauts conditionnels puis repérer quel instruction met à jour le flag utilisé. Pour cela, on calcule la durée de vie du flag grâce au même algorithme que pour la durée de vie des registres. Par exemple :

---

<sup>4</sup>Laboratoire de Recherche et Développement d'EPITA

```
t2 = t1 - 2
push t2
jnz l1
```

En calculant la durée de vie du flag  $Z^5$ , on remarque que c'est l'instruction  $t2 = t1 - 2$  qui a modifié ce flag. On peut alors écrire `jcond (t2 != 0) l1`

## 4 Traitement des Fonctions

### 4.1 Repérage des *Basic Blocks* et des *Function Blocks*

Pour les étapes suivantes, nous avons besoin de repérer les Basic Blocks et les Function Blocks. Un Function Blocks commence par un label et fini par un `return`. Un Basic Block commence par un label et fini par un `return`, un saut, un label ou un appel de fonction. On commence par construire les Basic Blocks.

#### 4.1.1 Construction des Basic Blocks

Au début, on considère l'ensemble du code comme un seul Block. On parcourt le code, puis, à chaque saut, `return`, ou appel de fonction, on coupe le block en deux. On vérifie ensuite que l'adresse à laquelle on saute est bien le début d'un Basic Block, sinon, on coupe en deux Basic Blocks. Enfin, on met à jour les arcs entre les Basic Blocks que l'on vient de créer. Considérons l'exemple suivant :

```
t1 = t2
fun1:
  t3 = t1
  call fun1
  t4 = t3 + t1
l1:
  t5 = t1 + 5
  jcond (t1 < t3) l1
  ret
t6 = t2
```

on parcourt le code, puis, lorsque l'on arrive l'instruction `call`, on divise le Basic Block en deux :

```
t1 = t2                                block 1
fun1:
  t3 = t1                                block 2
  call fun1                              block 2
  t4 = t3 + t1                          block 3
l1:
  t5 = t1 + 5                            block 3
  jcond (t1 < t3) l1                    block 3
```

---

<sup>5</sup>Le flag Z signifie "Est ce que le resultat est Zéro"

```

ret                                block 3
t6 = t2                            block 3

```

On continue de parcourir le code, puis, on arrive sur l'instruction `jcond` :

```

t1 = t2                            block 1
fun1:
t3 = t1                            block 2
call fun1                          block 2
t4 = t3 + t1                       block 3
l1:
t5 = t1 + 5                        block 4
jcond (t1 < t3) l1                block 4
ret                                block 5
t6 = t2                            block 5

```

Enfin, on arrive sur l'instruction `ret` :

```

t1 = t2                            block 1
fun1:
t3 = t1                            block 2
call fun1                          block 2
t4 = t3 + t1                       block 3
l1:
t5 = t1 + 5                        block 4
jcond (t1 < t3) l1                block 4
ret                                block 5
t6 = t2                            block 6

```

Le résultat est un graphe ressemblant au graphe construit pour le calcul de la durée de vie des temporaire mais, avec beaucoup moins de noeuds. Avec les Basic Blocks, le graphe garde les informations sur la structure du programme, mais, nous savons qu'il n'y a pas d'arc à l'intérieur des Basic Block et ainsi, chaque Basic Block peut être considéré comme une instruction.

#### 4.1.2 Création du Call Graph

On peut maintenant regrouper les Basic Blocks en Function Blocks. On utilise la même méthode que pour les Basic Blocks, mais, cette fois, on ne repère que des appels de fonctions et les instructions `return`. Pour l'exemple précédent, nous obtenons :

```

t1 = t2                            block 1
fun1:
t3 = t1                            block 2
call fun1                          block 2
t4 = t3 + t1                       block 2
l1:
t5 = t1 + 5                        block 2
jcond (t1 < t3) l1                block 2
ret                                block 2
t6 = t2                            block 3

```

## 4.2 Repérage des signature des fonctions

Nous voulons maintenant transformer les instructions `call` en appels de fonctions plus facile à lire. Nous devons tout d'abord retrouver la signature de chaque fonction.

### 4.2.1 Fonctions prédéfinies

Dans le cas des fonctions appartenant à des bibliothèques standards ou des built-ins, on possède déjà le code source et les signatures de celles-ci. Il suffit de retrouver le code de la fonction pour savoir son emplacement et trouver les endroits où elle est appelée. De même, on peut utiliser la table des symboles des bibliothèques dynamiques utilisée pour nommer les fonctions.

### 4.2.2 Les autres fonctions

Nous voulons maintenant connaître quels registres servent d'arguments à un Function Block. Pour reprérer un argument passé par les registres, on regarde simplement les temporaires utilisées dans un Function Block mais jamais définies dans le Function Block. Ces temporaires sont forcément des arguments de la fonction. On repère ensuite les registres qui sont définis dans la fonction et utilisés dans le code suivant l'appel de la fonction. Lorsque nous avons la signature de nos fonctions, nous pouvons remplacer les instructions `call` par la signature des fonctions. Nos fonctions sont maintenant bien independantes du reste du code.

## 4.3 Problèmes sur le repérage des fonctions

Dans un code généré par un compilateur courant, les algorithmes précédants ne doivent pas poser de problèmes. Néanmoins, le langage machine permet certaine pratiques qui ne passera pas avec nos algorithmes. On peut par exemple trouver des Basic Block appelés quelques fois avec `call` et d'autres fois avec `jump`. Nous ne pouvons alors pas décider si le Basic Block est une fonction ou non. Il est aussi possible qu'un saut arrive sur un Block interne à une fonction ou bien qu'une fonction sorte sans l'instruction `return`. Il peut aussi y avoir des fonctions retournant deux registres. Ce sont des pratiques qu'on ne devrait pas trouver dans un code généré, mais, en théorie, elles sont faisables.

## 5 Inlining des instructions

### 5.1 Propagation de copie des temporaire

Nous voulons maintenant 'inliner' les instructions. C'est à dire passer de

```
t4 = t2 + t3
t1 = fun1(t4)
t5 = t4 - t1
```

à

```
t4 = (t2 + t3)
t5 = t4 - fun1(t4)
```

On regarde pour cela la durée de vie des temporaires. Lorsqu'une temporaire est utilisée qu'une fois, on peut alors, la substituer à sa définition. Reprenons l'exemple précédant :

```
t4 = t2 + t3    ; def = t4    use = t2, t3
t1 = fun1(t4)  ; def = t1    use = t4
t5 = t4 - t1   ; def = t5    use = t4, t1
```

On remarque que `t1` n'est utilisé qu'une fois, on peut donc la substituer à sa définition :

```
t4 = (t2 + t3)
t5 = t4 - fun1(t4)
```

## 5.2 Elimination des temporaire inutile

Nous avons, lors de l'élimination des registres, ajouter beaucoup de temporaires pour que la durée de vie des temporaires soit la plus courte possible. Cela nous a permis de faire beaucoup d'optimisations sur le code. Nous devons maintenant faire l'étape inverse et supprimer les temporaires inutiles. Par exemple :

```
t4 = (t2 + t3)
t5 = t4 - fun1(t4)
```

donne

```
t4 = (t2 + t3)
t4 = t4 - fun1(t4)
```

car `t5` et `t4` peuvent être dans la même temporaire sans gêner le fonctionnement du programme. Dans sa thèse Cristina Cifentes garde la notion de registres pendant toute la phase d'analyse du Dataflow et évite cette phase d'élimination des temporaires inutiles, mais, cela complexifie l'analyse du Dataflow.

## 5.3 Variables locales

Après tous ces traitements, les registres utilisés pour stocker des résultats intermédiaires ont normalement été supprimés. On peut donc dire que les registres restant sont des variables locales. On change alors les noms des registres. On crée des nouveaux noms en fonction de la vie de la donnée. Par exemple :

```
t4 = (t2 + t3)
t4 = t4 - fun1(t4)
```

donne

```
loc1 = (t2 + t3)
loc1 = loc1 - fun1(loc1)
```

## 6 Reconnaissance des structures de controles

C'est sûrement l'une des parties les plus difficiles d'un décompilateur. Nous avons maintenant du code lisible mais, mais, il est encore difficile de comprendre la structure de l'algorithme utilisé car il n'y a pas pas encore de structures de controle<sup>6</sup>. On doit donc retrouver la structure de l'algorithme utilisé.

### 6.1 Repérage des boucles

D'après la thèse de Cristina Cifentes [eAF95], nous commençons tout d'abord par repérer les structures de boucle (ex : `while`, `for`, `repeat . . . until`). Pour cela, on repère les arcs en arrière<sup>7</sup>. Un arc en arrière est typique d'une boucle. On regarde ensuite, si aucun arc arrive à l'intérieur de la boucle. Si c'est le cas, nous ne sommes pas en présence d'une boucle. Effectivement, à part avec un `goto`, on ne peut pas rentrer au milieu d'une boucle. On regarde ensuite si il y a une condition juste avant l'arc en arrière ou bien à l'arrivée de l'arc en arrière. Le premier cas est synonyme d'un `repeat . . . until` et le second d'un `while`. Si il n'y a pas de condition ni avant, ni après l'arc en arrière, on se trouve sûrement en présence d'une boucle sans condition d'arrêt (qui se finit sûrement avec un `break`). Les arcs partant de l'intérieur de la boucle et sortant de la boucle peuvent être représentés avec des `break` ou des `goto`.

### 6.2 Repérage des structures conditionnelles

Pour repérer les structures conditionnelles, on repère les arcs en avant pour trouver des structures `if . . . then` et les arcs croisés pour repérer les structure `if . . . then . . . else`. On trouve ensuite les conditions imbriquées. On élimine alors l'imbrication en transformant la condition à l'aide de OU ou de ET logiques. Les conditions sont généralement optimisées par le compilateur. Il peut alors être intéressant d'utiliser un programme de transformation d'expression logique<sup>8</sup> pour rendre la condition plus lisible.

### 6.3 Finalisation

On applique ainsi plusieurs fois les repérages précédents. Cela nous permet de fusionner les Basic Blocks en intégrant les structures de contrôle et ainsi de simplifier le graphe. Si nous ne trouvons plus de structure de controle dans le graphe, nous utilisons `goto` sur l'un des arcs pour débloquer la situation. L'algorithme s'arrête quand tous les Basic Blocks sont fusionnés. Nous obtenons alors un code ne possédant plus de `jcond`, dont l'appel des fonctions est lisible et dont les expressions sont inlinées. Néanmoins, ce code ne contient aucun typage !

---

<sup>6</sup>On appelle une structure de controle un mot clef du langage servant à orienter le code (par contradiction aux fonctions/procédures qui servent à faire une action). Typiquement, ce sont les boucles et les conditions

<sup>7</sup>On peut facilement calculer la nature d'un arc en faisant un parcours profondeur du graphe

<sup>8</sup>Projet Ovide fait dans le cadre pédagogique d'EPITA fait ca très bien, par exemple

## 7 Analyse de type

### 7.1 Présentation

L'analyse de type dans un décompilateur s'ajoute à tout le travail précédent. Le but de cette étape est d'ajouter des informations de type au code de haut niveau généré jusque là. Le code que l'on a est du C non typé, il ne manipule que des types que l'assembleur connaît, c'est à dire des mots machine (courts, simples ou doubles). A partir de cela on aimerait récupérer tout le confort de typage d'un langage de haut niveau, par exemple en C, les structures, les tableaux, et les unions.

Dans le processus de compilation les informations de type sont présentes dans l'analyse sémantique (avant la transcription en représentation intermédiaire). Par la suite elles disparaissent, l'assembleur n'étant pas typé. Les informations de types peuvent toute fois être sauvegardés pour diverses utilités comme pour l'implémentation d'un ramasse miettes, ou encore pour la vérification de code (comme dans l'IDL de .NET de Microsoft [Mic], ou dans Java de Sun [Sun]). Dans ce cas l'analyse de type en est grandement facilitée, voir triviale.

Pour l'instant il n'existe pas de décompilateur permettant la reconnaissance de type. Ceci car il y a très peu de compilateur opérationnels, de plus le travail sur les système de typage est récents (voir ML [Roc] ou Haskell [GHC]). Les documents abordants le sujet sont aussi très rares. La référence en la matière étant aujourd'hui le travail de Alan Mycroft "Type Based Decompilation"[Myc99]. C'est d'ailleurs de celui ci que nous allons nous inspirer, les idées principales ainsi que les exemples proviennent directement de A. Mycroft.

Pour réaliser l'analyse de type, nous allons nous baser sur la forme SSA, grâce à laquelle nous allons générer des contraintes de type, puis nous ferons tourner l'algorithme d'inférence de type de Hindley-Milner [Mil78]. Grâce à celui ci nous allons récupérer des informations de type qui nous permettrons de décrire le code C non typé et de le retravailler afin de le rendre plus proche d'un code humainement écrit.

### 7.2 Inférence de type

L'inférence de type est le procédé permettant de retrouver le type d'une variable à partir de ces utilisations. Elle est surtout utilisée dans les langages fonctionnels (comme ML et Haskell). C'est Hindler et Milner qui ont indépendamment découvert une méthode d'inférence de type à la compilation. L'inférence de type est basée sur la preuve automatique de théorèmes utilisée en logique.

L'algorithme Milner-Hindley propose de reduire le probleme d'inférence de type à un problème d'unification. Nous avons un ensemble de types définis (`int`, `bool`, ...) et de type variables. A partir du code nous allons produire un certain nombre de variables à type variable, et des contraintes sur ces variables, celui-ci est appelé l'environnement. Grâce à cet environnement nous allons pouvoir unifier le type d'une variable avec l'envi-

ronnement en lui donnant un type. Une fois cette variable typée, nous allons pouvoir faire une substitution dans l'environnement et générer de nouvelles contraintes de types qui pourront alors nous servir pour l'unification. Le détail de l'algorithme dépasse le cadre de cet article et peut être trouvé soit dans Milney [Mil78] ou Appel [App98] dans le chapitre 'Polymorphic Types'.

Dans notre cas, contrairement à dans un compilateur, le typage est forcé. L'unification doit toujours réussir. C'est donc cette phase qui sera la principale différence avec l'algorithme d'origine.

Le typage du langage C ne satisfait pas tous les besoins que nous avons pendant la reconstruction de type, donc comme le propose A. Mycroft, nous utilisons donc notre propre structure de type. La règle 't' est pour les types, 's' pour les champs d'une structure et 'r' pour les registres.

$$\begin{aligned} t &::= \text{char} | \text{short} | \text{int} | \text{ptr}(t) | \text{array}(t) | \text{mem}(s) \\ s &::= n_1 : t_1, \dots, n_k : t_k \text{ avec } n_i \in N \text{ et } k > 0 \\ r &::= \text{int} | \text{ptr}(t) \end{aligned}$$

La notation  $\text{mem}(s)$  représente une `struct C`. Les  $n_i$  décrit l'octet où se trouve le champ dans une structure. On peut noter que les unions peuvent être simulées par  $\text{mem}(0 : t_1, \dots, 0 : t_k)$ .

Grâce à cette notation on va pouvoir décrire toutes les instructions en ajoutant des contraintes. Voici quelques exemples, proposées par A. Mycroft, en adoptant la notation 'rX' pour un registre et 'tX' son type associé.

instruction	contraintes générées
<code>mov r4, r6</code>	$t6 = t4$
<code>ld.w 6[r3], r5</code>	$t3 = \text{ptr}(\text{mem}(6 : t5))$
<code>xor ra2, r1b, r1c</code>	$t2a = \text{int}, t1b = \text{int}, t1c = \text{int}$
<code>add r2a, r1b, r1c</code>	$t2a = \text{ptr}(\alpha), t1b = \text{int}, t1c = \text{ptr}(\alpha) \vee$ $t2a = \text{int}, t1b = \text{ptr}(\alpha'), t1c = \text{ptr}(\alpha') \vee$ $t2a = \text{int}, t1b = \text{int}, t1c = \text{int}$
<code>ld.w (r5)[r0], r3</code>	$t0 = \text{ptr}(\text{array}(t3)), t5 = \text{int} \vee$ $t0 = \text{int}, t5 = \text{ptr}(\text{array}(t3))$
<code>mov #42, r7</code>	$t7 = \text{int}$
<code>mov #0, r7</code>	$t7 = \text{int} \vee t7 = \text{ptr}(\alpha')$

On note que l'opération '+' est surchargée et donne lieu à des contraintes disjonctives. En effet l'addition peut soit s'appliquer sur les pointeurs soit sur des entiers, contrairement au `xor` qui lui ne s'applique que sur des entiers. De même ceci s'applique aux instructions de chargement et de sauvegarde indexées, ainsi qu'à la constante 0 qui signifie généralement le pointeur `null`.

### 7.3 Exemple

Mycroft nous propose d'illustrer l'algorithme avec le code suivant :

```

; struct A { int hd; struct A *tl; };
;
; int f(struct A *x)
; {
;     int r = 0;
;     for ( ; x != 0; x = x->tl)
;         r += x->hd;
;     return r;
;
f:
        mov     #0, r1
        cmp     #0, r0
        beq     L4F2
L3F2:
        ld.w   0[r0], r2
        add    r2, r1, r1
        ld.w   4[r0], r0
        cmp    #0, r0
        bne    L3F2
L4F2:
        mov    r1, r0
        ret

```

C'est une fonction qui somme une liste écrite de façon itérative. A partir du code assembleur, nous allons essayer de retrouver le code source initial, indiqué en commentaires. On notera que la structure A est définie récursivement, ce qui pourrait sembler être un problème.

La première étape de notre reconstruction de type est de mettre le code assembleur sous la forme SSA, afin de pouvoir générer les contraintes de type. Voici ce qu'on obtient :

f :		tf = t0 ← t99
	mov r0, r0a	t0 = t0a
	mov #0, r1a	t1a = int ∨ t1a = ptr(α <sub>1</sub> )
	cmp #0, r0a	t0a = int ∨ t0a = ptr(α <sub>2</sub> )
	beq L4F2	
L3F2 :	mov φ(r0a, r0c), r0b	t0b = t0a, t0b = t0c
	mov φ(r1a, r1c), r1b	t1b = t1a, t1b = t1c
	ld.w 0[r0b], r2a	t0b = ptr(mem(0 : t2a))
	add r2a, r1b, r1c	t2a = ptr(α <sub>3</sub> ), t1b = int, t1c = ptr(α <sub>3</sub> ) ∨ t2a = int, t1b = ptr(alpha <sub>4</sub> ), t1c = ptr(alpha <sub>4</sub> ), ∨ t2a = int, t1b = int, t1c = int
	ld.w 4[r0b], r0c	t0b = ptr(mem(4 : t0c))
	cmp #0, r0c	t0c = int ∨ t0c = ptr(α <sub>5</sub> )
	bne L3F2	
L4F2 :	mov φ(r1a, r1c), r1d	t1d = t1a, t1d = t1c
	mov r1d, r0d	t0d = t1d
	ret	t99 = t0d

Les  $\alpha_x$  sont différents types que l'algorithme d'inférence doit déterminer.

On lance alors l'algorithme d'inférence de type. Celui ci n'est pas d'écrit en détail, mais on va tomber sur une inconsistance :

$$t0c = t0b = ptr(mem(4 : t0c)) = ptr(mem(0 : t2a))$$

Pour résoudre cela on introduit :

$$\begin{aligned} & \text{struct } G \{ t2a \text{ m0}; t0c \text{ m4}; \dots \} \\ t0c &= ptr(mem(0 : t2a, 4 : t0c)) = ptr(\text{struct } G) \end{aligned}$$

On obtient donc les solutions suivantes :

$$\begin{aligned} t0 &= t0a = t0b = t0c = ptr(\text{struct } G) \\ t99 &= t1a = t1b = t1c = t1d = t2a = t0d = int \\ tf &= ptr(\text{struct } G) \leftarrow ptr(int) \end{aligned}$$

ou

$$\begin{aligned} t0 &= t0a = t0b = t0c = ptr(\text{struct } G) \\ t2a &= int \\ t99 &= t1a = t1b = t1c = t1d = t0d = ptr(\alpha_4) \\ tf &= ptr(\text{struct } G) \leftarrow ptr(\alpha_4) \end{aligned}$$

La deuxième solution est due à la surcharge de l'opérateur '+' dans l'ajout des éléments de la liste. Le type  $ptr(\alpha_4)$  correspond au `void` du C. Les deux cas sont bien sur plausible, mais intuitivement on preferara le premier cas. Pour lever cette ambiguïté on peut imaginer une priorité dans les types, avec les pointeurs ayant la plus faible. Une autre singularité à noter c'est que la structure G contient des champs indéfinis, c'est à

dire qu'ils sont inutilisés dans cette fonction. Mycroft propose alors d'ajouter un champs optionnel qui permet de bloquer de la place pour eventuellement d'autres champs.

Le programme final obtenu, à partir de la reconnaissance de type et du l'analyse du flot de données et de contrôle est le suivant :

```
struct G { int m0; struct G *m4; Tpad m8; };
int f(struct G *x)
{
  int r = 0;
  if (x != 0)
  {
    do
    {
      r += x->m0;
      x = x->m4;
    }
    while (x != 0);
    return r;
  }
}
```

Le résultat obtenu est satisfaisant, il est proche de celui d'origine<sup>9</sup>, et facilement compréhensible.

## 7.4 Structures contre Tableaux

Différencier les structures des tableaux est une tâche difficile. Nous sommes obligé de faire des choix, pour automatiser le processus, qui ne sont pas forcément les bons. On pourrait par contre imaginer une interface demandant de choisir entre plusieurs types possibles.

Comme vu précédemment, on a :

```
ld.w 6[r3], r5    t3 = ptr(mem(6 : t5))
ld.w (r5)[r0], r3 t0 = ptr(array(t3)), t5 = int ∨
                    t0 = int, t5 = ptr(array(t3))
```

Cela vient du fait que nous considérons un accès fait grâce à une constante comme un accès à un champ d'une structure, et un accès fait par une variable comme un tableaux. Cette décision vient du fait qu'on utilise la plus part du temps un indice pour accéder aux tableaux, tandis que les accès à un champ d'une structure sont générés par le compilateur avec des décalages constants.

Un code du type :

```
ld.b 0[r0], r1
ld.b 48[r0], r2
ld.w (r5)[r0], r3
```

---

<sup>9</sup>Pour transformer la boucle `do...while` en `for` on peut utiliser un postprocesseur

Aura comme resultat :

```
union G { struct { char m0; char pad1[47]; char m48; } u1;
          int u2[];
        } *r0;
```

On voit tout de suite que le resultat n'est pas très bon, et qu'il est difficile de bien interprété un tel code. De plus il est impossible de connaître la taille du tableau<sup>10</sup>.

Mais pire encore, il est impossible de faire la différence entre différentes structures :

```
struct S1 { int a; int b[4]; int c; int d[4]; } x1;
struct S2 { int a; int b[4]; } x2[2];
struct S3 { struct S2 c; struct S2 d; } x3;
```

Donc, en général, le mélange de structures et de tableaux est très difficile à gérer.

---

<sup>10</sup>A moins que le binaire contienne des informations pour vérifier l'intégrité du code, comme pour le *bound checking*

## 8 Conclusion

### 8.1 Etat des lieux

Ainsi donc comme nous l'avons vu précédemment les techniques utilisées pour le traçage de code sont rodées, mais on peut aller encore plus loin. Le traçage de code est donc bien utile dans pas mal de cas (cf intro) mais il faut savoir que son utilisation est illégale dans le cas d'utilisation sur des programmes protégés. Le traçage de code ne doit donc être utilisé que pour travailler sur son propre code d'une façon légale. Ce nouvel outil d'aide aux développeurs se généralise de plus en plus et maintenant même les constructeurs se mettent à faciliter le traçage de code au point de vue hardware.

- décompilateurs existants (dcc, SourceAgain )
  - SoftIce, développé par la société compuware. Est un débogueur avancé pour les programmes sous Windows. Ce débogueur ne s'appuie pas sur les ressources Windows pour travailler et est donc immunisé aux erreurs Windows.
  - GDB, est le plus connu des débogueur Unix. Il permet d'examiner ce qui se passe quand le programme s'arrête et même de changer des choses dans le programme afin de corriger un bug et de continuer dans l'exécution du programme.
  - SourceAgain, est actuellement le plus développé des décompilateur pour Java. Il est capable d'analyser correctement et de décompiler les plus complexe contrôle de flot en Java, et d'en reproduire un code Java recompilable à chaque fois.
  - Dcc, est un décompilateur développé par Cristina Cifuentes. Il permet de décompiler des executables pour les plateformes i386 et DOS de programmes écrits en C. Il est basé sur sur les techniques standart d'optimisation et de théorie des graphes.
- les résultats obtenus  
Les résultats obtenues à l'aide de ces divers décompilateurs sont dans l'ensemble bien plus que satisfaisant lorsqu'il s'agit de code simple, mais peut s'averer divergeant du code initial dans le cas de code plus complexe lorsque l'on tente de remonter vers des représentations intermédiaires de plus haut niveau.

## Références

- [App98] A. Appel. *"Modern Compiler Implementation"*. 1998.
- [Bas02] Séverine Basset. *Le point sur la décompilation*. Publié sur le web, 25 juin 2002. [<http://www.droit-ntic.com>].
- [Dev03] GDB Developers. *GDB Internals Manual*. Publié sur le web, 2003. [<http://sources.redhat.com/gdb/>].
- [Dus00] Séverine Dusollier. *Protection juridique des mesures techniques anti-piratage : le cas DVD*. Publié sur le web, janvier 2000. [<http://www.droit-technologique.org>].
- [eAF95] C. Cifuentes et A. Fitzgerald. "reverse engineering of computer programs : Comments on the copyright law review committee's final report on computer software protection.". *"Journal of Law and Information Science"*, 6(2) :241–276, Dec 1995. Peut être trouvé sur [<http://www.csee.uq.edu.au/~crisrina/jlis95.ps>].
- [GHC] Glasgow haskell compiler. [[haskell.cs.yale.edu/ghc/](http://haskell.cs.yale.edu/ghc/)].
- [IBM] IBM. *PowerPC 750FX Microprocessor User's Manual*. Peut être trouvé sur [<http://www.ibm.com>].
- [Int] Intel. *IA-32 Intel Architecture Software Developer's Manual, Volume 3 : System Programming Guide*. Chapitre 5. Peut être trouvé sur [<http://www.intel.com>].
- [Mic] Microsoft. *ECMA and ISO/IEC C# and Common Language Infrastructure Standarts*. Peut être téléchargé en PDF sur [<http://msdn.microsoft.com/net/ecma/>].
- [Mil78] R. Milner. A theory of polymorphism in programming. *JCSS*, 1978.
- [Myc99] A. Mycroft. Type-based decompilation. *Lecture Notes in Computer Science*, 1576, 1999. Peut être trouvé en PostScript sur [<http://www.cl.cam.ac.uk/users/am/papers/esop99.ps.gz>].
- [Roc] INRIA Rocquencourt. Caml, un dialecte de ML. [<http://caml.inria.fr>].
- [Sun] Sun Microsystems. *The Java Virtual Machine Specification*. Peut être téléchargé en PDF, HTML et PostScript sur [<http://java.sun.com/doc/>].
- [WW01] Ray Weiss William Wong. *PowerPC On-Chip Debug Hardware Resources*. Publié sur le web, 2001. [<http://www.elecdesign.com>].