Travail d'Etude et de Recherche

TER – Compilation

Dimitri Belopopsky, Jérémie Irr

22 Novembre 2017

Introduction

Le Travail d'Etude et de Recherche (TER) a pour objectif de mettre les étudiants en contact avec le milieu de la recherche. En effet, il s'agit d'un travail comprenant une partie développement/programmation ainsi qu'une partie consacrée à la recherche bibliographique, sur un thème lié à l'informatique.

Nous avons, pour notre TER, approfondi les concepts étudiés en cours de Compilation au premier semestre. Ce rapport décrit et détaille les différentes étapes de la construction de ce travail d'étude et de recherche.

Dans une première partie, nous détaillerons la conception du compilateur qui a servi de base à notre projet. Une version simplifiée d'un compilateur du langage C vers AMD64. Après avoir décrit la structure du code et justifié nos choix d'implémentation, nous expliquerons le fonctionnement du compilateur en détaillant ses quatre phases de compilation : analyse lexicale, analyse syntaxique, typage et production de code.

Dans une deuxième, la recherche bibliographique s'est porté sur une fonctionnalité de compilation, les Optimisations Simples, en particulier *l'inlining*, le *constant folding* et *le constant propagation*. Après avoir détaillé les différents concepts de ces optimisations, mis en avant leurs limites, nous expliquerons comment nous avons implémenté une version simple de cette fonctionnalité, les difficultés rencontrées et la méthode utilisée.

Le TER se conclura par une soutenance.

1. Conception du compilateur
1.1. Description du code

- ast.mli :

Arbre de syntaxe abstraite, contient la grammaire du code.
La structure « *info* » est utilisé dans le main pour donner la position en cas d'erreur.

- lexer.mll :

L'analyseur lexical du code.

- parser.mly :

L'analyseur syntaxique du code.

- typing.ml :

Typage, vérification des types et de leur compatibilité.

- compile.ml :

Contient toutes les fonctions nécessaires à la production de code. Utilise la bibliothèque des fichiers *amd64.ml* et *amd64.mli*.

- main.mli :

Contient principalement la fonction « *report_loc* » de localisation des erreurs. Une erreur (lexicale ou syntaxique) est signalée et localisée le plus précisément possible (ligne et charactères).

- amd64.ml + amd64.mli :

Auteurs : 2015 Jean-Christophe Filliatre (CNRS)
2017 Kim Nguyêñ (Université Paris-Sud)
Fragment de bibliothèque pour l'écriture de programmes X86-64. L'intérêt de cette bibliothèque est de grandement simplifier la production de code.

- tests :

Dossier contenant tous les tests réalisés pour le compilateur.

1.2.Fonctionnement

Nous allons maintenant détailler les différentes phases de compilation de notre projet, au travers d'un exemple :

```
1  extern int printf() ;
2
3  int main() {
4      printf(« Hello, World ! ») ;
5      return 0 ;
6  }
```

FIGURE 1 - EXEMPLE FONCTIONNEMENT

1.2.1. Analyse lexicale

Durant cette étape, il s'agit de d'analyser un fichier texte et de repérer tous les mots clés, symboles, caractères, chaines de caractères, nombres, etc., qui se trouvent dans ce fichier, avant de tout envoyer à l'analyseur syntaxique.

Dans notre cas, on utilise *Lexer* qui va *parser* selon des règles que nous lui avons dictées dans le fichier « *lexel.mll* ».

Une fois que *l'ast* est correct, on navigue dans l'arbre et on transforme chaque nœud en du code assembleur équivalent :

```
1  EXTERN INT printf LPAR RPAR SEMI
2
3  INT main LPAR RPAR L_SQ_BRACKET
4      printf LPAR CONST_STRING(« Hello, World ! ») RPAR SEMI,
5      RETURN 0 SEMI
6  R_SQ_BRACKET
```

FIGURE 2 - PHASE ANALYSE LEXICALE

1.2.2. Analyse Syntaxique

L'analyse syntaxique prend, d'une part, les *tokens* de l'analyseur lexicale, mais aussi l'ordre dans lequel ils apparaissent dans le texte. Il compare cela à un ensemble de règles qui constituent la grammaire du langage qu'on essaye de compiler. Si le *token* ou la succession de *token* forment une règle, ces *tokens* sont convertis en leur équivalence dans l'arbre de syntaxe abstraite (AST). On crée

un arbre qui représente le fichier et toutes les relations des différentes règles de grammaire entre elles.

Notre exemple, après analyse syntaxique (chaque « ligne » de la liste est un arbre syntaxique valide):

```
[
Dfun(Int, printf, None, None),
Dfun(Int, main, None, Some (
    Dfun(int, printf, Econst(Cstring(« Hello, World ! »)),  None),
    Sreturn (Some Econst(Cint(Signed, Int, 0))
    )
]
```

FIGURE 3 - PHASE ANALYSE SYNTAXIQUE

1.2.3. Typage

Une fois *l'AST* construit, la phase de *parsing* du fichier est terminée. Le code est représenté sous une forme d'arbre sur laquelle on peut travailler. Le langage sur lequel nous travaillons est un sous ensemble de C (on pourrait appeler ce langage C--), qui est un langage fortement typé, c'est à dire que le typage de toutes les variables et instructions est connu à la compilation et doit faire sens pour la bonne exécution du programme. Il s'oppose aux langages faiblement typés (javascript, python) ou le typage est déduit au moment où l'on en a besoin. Il y a également des langages qui ont un typage fort mais déduit à la compilation et qui n'est pas donné explicitement dans le code.

Tout d'abord, la phase de typage est d'associer chaque élément de *l'AST* à un type. Par exemple, un nombre entier est associé à *int*, un flottant à *double*, etc.
Ensuite, on vérifie que les opérations sur les variables et les fonctions sont correctement typées et leur utilisation fait sens. Par exemple, une fonction qui prend un *int* en paramètre ne doit pas recevoir une chaîne de caractère.
Enfin, l'étape la plus importante est le casting. Dans certains cas, des types peuvent être « transformés » en d'autres. L'exemple le plus évident est les *int* et les *double* qui peuvent être intervertis sans trop de problème (à part le troncage de la partie flottante). Un exemple, moins évident, est le type char qui représente un caractère unique qui est en fait le code ascii qui sert à le représenter. Il se comporte comme un nombre entier.

1.2.4. Production de code

Une fois que *l'ast* est correct, on navigue dans l'arbre et on transforme chaque nœud en du code assembleur équivalent. Après avoir validé le lexique, la syntaxe, le typage, etc., on produit le code :

```
.text
    .globl main
main:
#On rentre dans la fonction main
    pushq %rbp
    mov %rsp, %rbp
    mov $__label__string_00001, %r10
    pushq %r10
    popq %rdi
    mov $0, %rax
    call printf
    mov %rax, %r10
    andq $-1, %r10
    pushq %r10
    popq %r10
main_fin:
    popq %rax
    mov %rbp, %rsp
    popq %rbp
    ret
.data
__label__string_00001:
    .string "Hello, World!"
```

FIGURE 4 - PRODUCTION DE CODE

2. Recherche

Dans un compilateur, l'implantation d'optimisations simples a pour objectif d'obtenir des programmes plus efficaces : moins chronophages, plus compacts et consommant moins de mémoire. L'optimisation se fait par le choix de bonnes extensions qui permettent d'obtenir des résultats intéressants. C'est le cas des extensions d'*inlining*, de *constant folding* et de *constant propagation*. Les petits gains sur des opérations répétées peuvent rapporter beaucoup à terme et il est particulièrement intéressant d'optimiser des instructions qui reviennent souvent.

2.1.Concept

2.1.1. Inlining

L'inlining, ou extension inline, est une optimisation du compilateur qui remplace un appel de fonction par le corps de celle-ci (figure ci-dessous).

```
1  int prod(int x, int y) {
2      if (x == 0 || y == 0)
3          return 0;
4      else
5          return x * y;
6  }
7
8  /* Sans inlining */
9  int f(int x, int y) {
10     return prod(x,y) + prod(x,y+1);
11 }
12
13 /* Avec inlining */
14 int f(int x, int y) {
15     int temp = 0;
16     if (x == 0 || y == 0) temp += 0; else temp += x*y;
17     if (x == 0 || y+1 == 0) temp += 0; else temp += x*(y+1);
18     return temp;
19 }
```

FIGURE 5 - EXEMPLE D'INLINING

Les intérêts d'une telle extension sont multiples. *L'inlining* supprime les appels et les retours de fonction pour les remplacer par des exécutions de code qui seront plus adaptées à l'optimisation. Nous pouvons rappeler qu'un appel de fonction correspond à la vérification des paramètres de l'appel, la mise en registres de ces paramètres, la sauvegarde de valeur interne à la fonction, l'exécution du corps de la fonction, le retour du résultat, la libération des registres, etc. Autant de tâches qui, pour la plupart, ne sont pas effectuées après *l'inlining* de la fonction.

Par ailleurs, plusieurs études ont déjà décrit, *l'inlining* comme une partie essentielle d'un compilateur C optimisé. Dans un article (Allen & Cocke, 1976), il est mis en avant plusieurs raisons évidentes pour la mise en place *d'inlining*. Premièrement, le problème de l'abaissement variable devient moins coûteux après *inlining*. Deuxièmement, l'optimiseur de code peut fonctionner sur les effets réels de la fonction. Troisièmement, *l'inlining* d'appels de fonction contenus dans les boucles peut augmenter les possibilités de vectorisation. Par ailleurs, nous allons voir que plusieurs techniques d'amélioration du code peuvent être appliquées après *l'inlining* de ce dernier, dont l'amélioration dans l'attribution des registres, l'ordonnancement du code, l'élimination des sous-expressions communes, mais aussi la propagation constante et l'élimination des codes morts (morceaux de code qui sont appelés mais dont le résultat n'est jamais utilisé).

2.1.2. Constant folding

Le *constant folding* est aussi une optimisation de compilateur. C'est un processus de reconnaissance et d'évaluation des expressions constantes au moment de la compilation, plutôt que de leur exécution au lancement du programme. On parle ici d'expressions simples, comme l'entier 1 ou 2, mais cela peut aussi être aussi n'importe quelle variable dont la valeur est connue au moment de la compilation (figure ci-dessous). L'intérêt est la réduction de la répétition de calcul.

```
/* Sans constant folding */
print_int (1 + 2);

/* Avec */
print_int (3);
```

FIGURE 6 - CONSTANT FOLDING

2.1.3. Constant propagation

Le *constant propagation* est lui aussi une optimisation de compilateur. C'est un processus de substitution des valeurs de constantes connues au moment de la compilation (figure ci-dessous).

```
/* Sans constant propagation */
let x = 2;
print_int (x);

/* Avec */
let x = 2;
print_int (2);

/* Ou même */
print_int (2);
```

FIGURE 7 - CONSTANT PROPAGATION

La variable *x* est connue au niveau de la fonction *print_int* ; le code est donc directement produit avec la valeur de cette variable. L'intérêt est de ne pas effectuer l'accès mémoire qui récupère la valeur de *x* dynamiquement.

2.1.4. Complémentarité

D'une part, le *constant folding* et le *constant propagation* sont généralement associés pour obtenir le plus de réductions et de simplifications dans le code. Pour cela, on procède à des itérations en alternant les deux optimisations jusqu'à obtenir le code le plus simple possible (figure ci-dessous).

```
/* Code initial */
int a = 1;
int b = 2 - (a / 1);
int c;

c = b * 4;
if (c < 5) {
   c = c - 2;
}
return c * (3 * a);
```

```
/* Après plusieurs itérations */
int c;
if (true) {
   c = 2;
}
return c * 3;

/* Et même */
return 6;
```

FIGURE 8 - COMPLEMENTARITE CONSTANT FOLDING/PROPAGATION

D'autre part, nous allons voir que l'exécution ces deux extensions peut être bloquée dans certains cas, comme par exemple en présence d'un appel de fonction. Or, nous avons vu que *l'inlining* permet de supprimer les appels de fonction au profit du seul corps de ces fonctions, laissant alors place à des nouvelles optimisations, en particulier celles de *constant folding* et de *constant propagation* (figure ci-dessous).

```
/* Sans inlining, le code n'est pas optimisable */
int x;
x = add(1,2);
print_int(x);

/* Avec inlining */
int x;
x = 1 + 2;
print_int(x);
```

```
/* On obtient avec association */
print_int(3);
```

FIGURE 9 - COMPLEMENTARITE DES TROIS EXTENSIONS

Combinées ensemble, ces trois extensions deviennent un outil très puissant d'optimisation, ce qui les rend fortement complémentaires entre-elles.

2.2.Limites

2.2.1. Blocages

La simplification de *constant folding* n'est pas toujours possible : c'est le cas quand au moins un des paramètres n'est pas directement une valeur (figure ci-dessous). Certains de ces blocages peuvent être enlevés par le *constant propagation* qui peut remplacer *x* par sa valeur connue (exemple1), et d'autres par *l'inlining* des appels de fonction (exemple 2). Néanmoins, il n'est pas toujours possible d'éviter ces blocages.

```
/* Exemple 1 */
print_int ( 1 + x );
/* Exemple 2 */
print_int ( 1 + f(x) );
```

FIGURE 10 - BLOCAGE CONSTANT FOLDING

Le même effet de blocage se produit pour le *constant propagation* (figure ci-dessous). Dans le premier exemple, la valeur de *x* appelée par la fonction *print_int()* est ambigüe : on ne peut pas effectuer la simplification du code. Dans le deuxième exemple, on ne connait pas l'effet de la fonction *f()* sur la variable *x* ; il est donc aussi dangereux de simplifier le code. Dans certains cas, *l'inlining* peut débloquer certaines de ces situations, comme nous l'avons montré précédemment.

```
/* Exemple 1 */
if b then x := 1 else x := 2; print_int(x);
/* Exemple 2 */
x := 1; f(x); print_int(x);
```

FIGURE 11 - BLOCAGE CONSTANT PROPAGATION

L'inlining, quant à lui, n'est pas toujours possible, notamment pour les fonctions de récurrence. En effet, *l'inlining* récursif des appels ne se termine pas. Il existe diverses solutions pour résoudre ce blocage, telles que l'analyse des graphes d'appels des fonctions ou *l'inlining* d'une quantité limitée de la fonction.

2.2.2. Inconvénients

L'Inlining est une optimisation importante qui peut avoir des effets négatifs sur la performance. De par la duplication du corps de fonction sur chaque appel, la taille du code augmente considérablement. Cette augmentation a un impact sur la performance de la mémoire cache des instructions. En effet, à petite dose, *l'inlining* augmente grandement la vitesse pour un coût mémoire restreint, mais utilisé avec excès, la vitesse est impactée négativement à cause de la consommation du cache des instructions de *l'inlining* et nécessite un espace mémoire trop conséquent. Cet impact sur la performance est donc, avant tout, un problème pour les grosses fonctions qui sont utilisées à plusieurs reprises. Ce problème est compliqué à corriger car le seuil de rentabilité, au-delà duquel *l'inlining* réduit les performances, est difficile à déterminer et dépend en général d'une quantité donnée précise.

Néanmoins, nous avons vu en cours que, même si ces phases d'optimisation peuvent être gourmandes au moment de la compilation, pour apporter uniquement des modifications mineures, le programme s'exécute plusieurs fois et certaines de ses instructions un très grand nombre de fois. Le compilateur, lui, ne s'exécute qu'une seule fois.

2.3.Implémentation

Pour l'implémentation d'une fonction, nous avons choisis l'extension *inline*, dans une version simplifiée. Quand le compilateur *inline* une fonction particulière, l'exécution de l'opération, en elle-même, est habituellement simple : le compilateur calcule simplement les arguments, les stocke dans des variables correspondant aux arguments de la fonction et insère le corps de la fonction sur le site d'appel. Un exemple d'inlining (Godbolt), C++ vers x86-64.

Par ailleurs, nous avons dû gérer le cas du blocage, vu précédemment, en cas de fonction récursive. Après avoir rejeté l'idée de gérer ce cas par *l'inlining* d'un nombre limitée de fonction, ce qui aurait pu rendre nos compiler instable, nous avons décidé de ne pas effectuer *d'inlining* dans le cas de fonction récursive. Il a donc fallu implémenter une détection de récursivité des fonctions dans notre compilateur.

Pour réaliser cette implémentation nous avons adopté la structure de code suivante :

- Création d'un « *analysis.ml* »

La structure « *fun_analyse* » que l'on rajoute à la définition de *Dfun* et qui va servir à stocker des informations sur la fonction, notamment si elle est récursive ou non.

- Dans « typing.ml »

Pour vérifier si une fonction est récursive, il suffit de savoir si dans la définition d'une fonction il y a un *Ecall* vers une fonction portant le même nom. En stockant le nom de la fonction avant de descendre dans la description du bloc, on peut vérifier très facilement si elle est récursive ou non.

- Dans « compile.ml »

La partie la plus compliqué de *l'inline* est de transformer les appels *Ecall* en bloc de code de la fonction équivalente. Il faut donc d'une part stocker le code de tel sorte à pouvoir le générer plusieurs fois à cause des labels (explications plus loin car on ne peut pas générer le code une fois et le coller à la place d'un *Ecall*). Ensuite à chaque moment où on veut faire un appel de fonction et que la fonction qu'on veut *inliner* n'est pas récursive, on génère son code et on le met à la place de l'appel.

Quand on génère du code et qu'il y a des conditions, on génère des labels qui vont servir pour les sauts dans le code. Tous les labels devant être uniques. La solution est de faire une *Hashtbl* avec comme clé le nom de la fonction, avec un *n-uplet* ayant d'une part la fonction partiel *compile_bloc* prête à être appliqué à un bloc, et le bloc de la fonction. Il ne reste plus qu'à assembler au besoin pour avoir du code *inliner*.

Conclusion

Ce Travail d'Etude de Recherche avait pour but de nous sensibiliser au milieu de la recherche. En effet, il nous a permis de conforter plus particulièrement nos connaissances dans le compilateur ainsi que dans les optimisations simples qui peuvent lui être apportées.

Nous avons commencé par l'implémentation d'un compilateur (dans sa version simplifiée), ce qui nous a permis d'accroitre nos connaissances dans la structure d'un compilateur, mais aussi, dans son fonctionnement au travers des différentes phases qui le constituent (l'analyse lexicale et syntaxique, le typage et la production de code).

Le travail de recherche est apparu comme un réel défit. En effet, nous n'avions jamais réalisé auparavant de recherche bibliographique dans le domaine de l'informatique. Ce TER nous a permis d'approfondir les concepts d'optimisations étudiés en cours, en particulier ceux *d'inlining*, de *constant propagation* et de *constant folding* et de nous rendre compte de l'existence d'une réelle complémentarité entre eux. Une complémentarité qui s'étend à d'autres fonctions d'optimisation et qui fait d'elles un moteur d'optimisation très puissant.

C'est pour l'ensemble de ces raisons que ce travail a été vraiment bénéfique et formateur pour nous.

Table des illustrations

Bibliographie

Allen, F. E., & Cocke, J. (1976, Mars). A Program Data Flow Analysis Procedure. *Journal of the Association for Computing Machinaty, 19-3*, 137-147.

Balabonski, T. (4 novembre 2016). *Compilation et langages : optimisations.*

Collberg, C. (2011). *Principles of Compilation - Optimization IV.* University of Arizona.

Filliâtre, J.-C. (2016-2017). *Langages de programmation et compilation.* Ecole Normale Supérieure.

Godbolt. (s.d.). GCC compiler, C++ vers x86-64. *https://godbolt.org/g/MLUsMe*.

Hwu, W.-m. W., & Chang, P. P. (1989). Achieving High Instruction Cache Performance with an Optimizing Compiler. *Proceedings of the 16th Annual Symposium on Computer Architecture*.

Hwu, W.-m. W., & Chang, P. P. (1989). *Inline Function Expansion for Compiling C Programs.* University of Illinois.

Hwu, W.-m. W., & Chang, P. P. (1989). Trace Selection for Compiling Large C Application Programs to Microcode. *Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitectures*.

Jelvis, T. (2016). Operations Research in Haskell at Target. *Quora.com*.

Serrano, M. (1997). *Inline expansion: when and how ?* University of Geneva.

Wikipedia. (s.d.). Dead Code. *https://en.wikipedia.org/wiki/Dead_code*.

Wikipedia. (s.d.). Inline expansion. *https://en.wikipedia.org/wiki/Inline_expansion*.