École Polytechnique

3ème année

INF564

Compilateur du mini c

Auteurs:
Ismael Ouattara
Philippe SAGBO

Superviseur:
Jean-Christophe Filliâtre

March 14, 2021

Abstract

L'objectif de ce projet est de pouvoir construire un compilateur optimisant pour le language mini C tout en implémentant les différentes représentations interne du code source. Ainsi en plus de code source et du code assembleur, ce projet met en lumière le typage du code source ainsi que la création des codes RTL, ERTL et LTL.
Ce projet a été réalisé grâce au language Java.

Contents

1 Typing

Fichier créé: Typing.java
Cette partie qui est la première (en ne tenant pas compte de l'analyseur lexical et semantique) à pour but de transformer un Ptree en Ttree. Le typage que nous avons implémenté se base sur l'utilisation des piles en Java. Ce choix nous a permis de mieux gérer les blocs, expressions et appels imbriqués.
Cependant, le typage effectué n'a pas tenu compte des types **Tvoidstar** et **Ttypenull**. Nous avons eu du mal à comprendre comment utiliser ces deux types.
Il faut ajouter que cette partie a été la plus difficile pour nous.

```
gabi@kali:~/Documents/Ecole/INF564/mini-c/tests$ ./run -2 ../mini-c
Test de ../mini-c


Partie 2
mauvais ........................
bons ..........................................................
Typage  : 92/92 : 100%
```

2 ToRTL

Fichier créé: ToRTL.java
Ici il s'agit de transformer notre Ttree en RTLtree.
Dans cette partie nous commençons à avoir des commandes qui ressemblent à celles du langage assembleur x86-64. On considère une fonction comme étant un graphe directionnel ayant pour sommet des labels et pour arête des instructions (et éventuellement des conditions). Pour transformer notre Ttree en Rtltree, nous parcourons notre programme dans le sens inverse.

Aussi, afin d'optimiser certaines opérations faisant intervenir en grande partie des constantes entières, nous avons implémenter une fonction **reduction** qui prend en paramètre une expression et la réduit autant que possible. Par exemple un code source contenant l'instruction **return 40+2;** se transformera en une instruction **return 42;**. Cependant, cette réduction n'est pas parfaite étant donnée qu'elle se base uniquement sur les opérations du typage. En effet, l'instruction **return 2+n+3;** ne peut pas se transformer aisément en une instruction **return n+5;** ou **return 5+n;**

Au niveau de la construction du langage RTL, l'affectation des variables n'a pas été optimale. En effet, en examinant le code RTL de certains programmes on tombera plusieurs fois sur des copies d'une même valeur dans 2 registres différents. En effet, cela est dû au fait que l'un des 2 registres de destination est directement lié à la variable locale correspondante à la valeur à copier et le second registre est automatiquement crée une fois qu'on appelle une fonction **visit()**.

```
gabi@kali:~/Documents/Ecole/INF564/mini-c/tests$ cat resultat
Test de ../mini-c


Execution normale
-----------------
..........................................................
------------------------------
...
Compilation:
Compilation : 62/62 : 100%
Comportement du code : 62/62 : 100%
```

3 ToERTL

Fichier créé: ToERTL.java
Cette partie du projet a été la plus facile pour nous car il fallait juste apporter un plus au niveau des instruction *call* et lors des définitions de fonction.

1. Pour les instructions *call* il faut sauvegarder les valeurs des paramètres dans des registres (dédiés à cela) ou sur le tas (dans le cas où nous avons plus de paramètres que de registres).
2. Pour les définitions de fonctions on alloue de l'espace sur le tas, on récupère les paramètres dans les registres dédiés (et éventuellement sur le tas s'il y'en a beaucoup) et avant de terminer la fonction on désalloue l'espace allouer sur le tas. (*En fait le tas n'a pas une très grande capacité, donc si on peut sauver de la mémoire il faut le faire ;)*)

```
gabi@kali:~/Documents/Ecole/INF564/mini-c/tests$ cat resultat_ertl
Test de ../mini-c


Execution normale
-----------------
..........................................................
------------------------------
...
Compilation:
Compilation : 62/62 : 100%
Comportement du code : 62/62 : 100%
```

4 ToLTL

Fichier créé: ToLTL.java, Liveness.java, Coloring.java, Interference.java
Dans cette partie on est très proche du langage assembleur, la grande différence est que nous travaillons toujours avec des labels (pas de linéarité pour l'instant).

La modification majeure de cette partie est l'allocation de registres physiques aux pseudos registres. Lors de la transformation en Rtltree et Ertltree nous avons supposé avoir une infinité de registre ce qui n'est évidemment pas le cas (*:(la vie aurait été tellement simple*; Nous devons donc trouver un moyen (efficace) d'allouer nos registres physiques à nos pseudos registres et c'est cela que la classe Coloring fait. Pour cela on introduit une notion de **couleur** qui est soit un registre physique,

soit une allocation sur la pile. Mais avant tout, il faut savoir qu'elles sont les registres qui ne peuvent avoir la même couleur(interférences) et celles qui devrait(de préférence) avoir la même couleur(préférences). L'attribution de couleur se fera donc en suivant une règle de priorité.

L'autre modification à faire c'est de fonctionner maintenant avec des couleurs à la place des pseudos registres mais il faut tenir compte du faite que certaines opération ne peuvent pas être faite sur le tas et donc utiliser un registre temporaire (**%r11** et **%r15**) dans une telle situation.

Il faut remarquer aussi que l'opération **get_param** sur le tas sera modifier par des opérations sur **%rbp** et **%rsp** sans oublier l'allocation (**resp.** la désallocation) de mémoire en début (**resp.** fin) de l'exécution d'une fonction.

```
gabi@kali:~/Documents/Ecole/INF564/mini-c/tests$ cat resultat_ltl
Test de ../mini-c


Execution normale
-----------------
..........................................................
-----------------------------
...
Compilation:
Compilation : 62/62 : 100%
Comportement du code : 62/62 : 100%
```

5 Lin

Fichier créé: Lin.java Ici, il s'agit de transformer le code LTL en code assembleur grâce à la linéarisation. En dehors des différentes astuces présentées dans le cours, deux fonctions semblent sortir de l'ordinaire: *this.visit(Lmunop)* et *this.visit(Lmbinop)*.

1. **Lmunop** Afin de convertir **Msete** RTL en assembleur, nous utilisons ici l'instruction **sete** de l'assembleur. Cependant, le fait que cette instruction prenne uniquement une opérande de taille 1 octet, nous oblige à donc gérer les cas des différents registres. Le travail fait sur le registre **rax** est donc fait sur le registre **al**.

2. **Lmbinop** L'utilisation de l'instruction **set** est similaire à celle de la fonction **this.visit(Lmbinop)**.

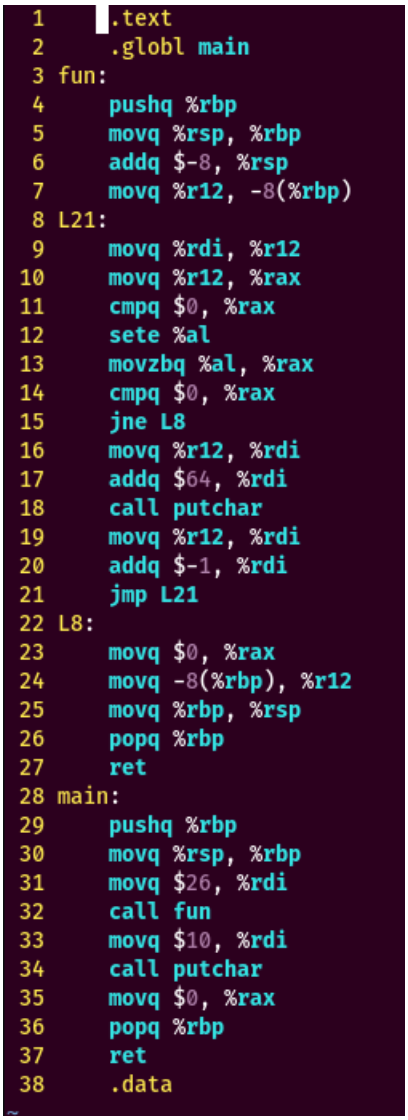
```
gabi@kali:~/Documents/Ecole/INF564/mini-c/tests$ ./run -3 ../mini-c
Test de ../mini-c


Partie 3
Execution normale
-----------------
.................................................
ECHEC : mauvaise sortie pour exec/shadow1.c
.......
Execution conduisant ?un ?chec
------------------------------
../run: line 266: 365702 Segmentation fault      ./a.out > out
../run: line 266: 365733 Floating point exception./a.out > out
../run: line 266: 365764 Floating point exception./a.out > out

Compilation:
Compilation : 62/62 : 100%
Code produit : 62/62 : 100%
Comportement du code : 61/62 : 98%
```

6 Extension

Comme extension à ce projet, nous avons utilisé l'appel terminal afin de convertir le code ERTL au code LTL.
Le code **rec_term.c** est un exemple d'utilisation d'un appel terminal.

```
gabi@kali:~/Downloads/Sagbo-Ouattara$ ./mini-c rec_term.c
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
gabi@kali:~/Downloads/Sagbo-Ouattara$ gcc rec_term.s -o rec_term
gabi@kali:~/Downloads/Sagbo-Ouattara$ ./rec_term
ZYXWVUTSRQPONMLKJIHGFEDCBA
```

Nous constatons bien qu'à la ligne 21, le code assembleur fait un **jmp** et non un **call** qui utiliserait de l'espace sur le tas (*Rappelez-vous, toujours sauvegarder de l'espace du tas si on le peut ;)*).



```
 1     .text
 2     .globl main
 3 fun:
 4     pushq %rbp
 5     movq %rsp, %rbp
 6     addq $-8, %rsp
 7     movq %r12, -8(%rbp)
 8 L21:
 9     movq %rdi, %r12
10     movq %r12, %rax
11     cmpq $0, %rax
12     sete %al
13     movzbq %al, %rax
14     cmpq $0, %rax
15     jne L8
16     movq %r12, %rdi
17     addq $64, %rdi
18     call putchar
19     movq %r12, %rdi
20     addq $-1, %rdi
21     jmp L21
22 L8:
23     movq $0, %rax
24     movq -8(%rbp), %r12
25     movq %rbp, %rsp
26     popq %rbp
27     ret
28 main:
29     pushq %rbp
30     movq %rsp, %rbp
31     movq $26, %rdi
32     call fun
33     movq $10, %rdi
34     call putchar
35     movq $0, %rax
36     popq %rbp
37     ret
38     .data
~
```