Compilation – TP 11 : De Retrolix à MIPS

Université Paris Diderot – Master 1

(2014-2015)

Cette feuille de TP vous donne les étapes à suivre pour terminer le compilateur ciblant l'architecture MIPS32 (Linux).

Le code source correspondant à ces travaux pratiques se trouve sur le GIT, dont on rappelle l'URL :

`http://moule.informatique.univ-paris-diderot.fr:8080/Yann/compilation-m1`

On rappelle que vous devez faire des *commits* réguliers (à chaque modification de votre code) pour que nous puissions suivre votre avancement.

1 Une allocation de registres réaliste

Pour que l'allocation de registres puisse être utilisée sur l'architecture MIPS32 sous Linux, il faut absolument implémenter les conventions d'appel. On rappelle que les conventions d'appels sont décrites dans la spécification de MIPS et servent à rendre possible l'appel de code de bibliothèques (compilées avec d'autres compilateurs) depuis le code compilé par notre compilateur.

Exercice 1 (Implémentation des conventions d'appels)

1. *Modifier la compilation des appels de fonction pour prendre en compte les conventions de passage des arguments et la récupération du résultat de l'appel.*
2. *Modifier la compilation des corps de fonction pour respecter les conventions de passage des arguments et la récupération du résultat de l'appel de cette fonction.*
3. *Modifier la compilation des corps de fonction pour qu'une fonction commence par sauvegarder tous les registres "callee-save" avant de commencer son calcul, et finisse par restaurer tous ces registres avant de retourner à son appelant.*

□

Exercice 2 (Rajout des registres physiques dans le graphe d'interférence)

1. *Pourquoi doit-on prendre en compte les registres physiques dans le graphe d'interférence ?*
2. *Étendre l'analyse de vivacité pour qu'elle prenne en compte les registres physiques. Attention à bien prendre en compte les registres utilisés et définis par un appel de fonction. Comment s'assurer que les registres "caller-save" sont bien sauvegardés par l'appelant d'une fonction ?*

□

Exercice 3 (Précoloriage) *1. Modifier l'algorithme de coloriage du graphe en s'assurant que les registres physiques sont précolorés. (Pour rappel, les nœuds précolorés ne sont pas simplifiables.)*

□

Exercice 4 (Variables globales) *1. Modifier votre coloriage pour vous assurer que les variables globales ne sont jamais représentées par des registres.*

Exercice 5 (Branchement de l'allocation de registres)

1. *Compléter la fonction* `RegisterAllocation.colorize_graph g` *pour utiliser le module générique de coloriage de graphe.*
2. *Compléter la fonction* `RegisterAllocation.register_allocation` *pour modifier le programme en utilisant effectivement le registre physique alloué à chaque variable et en laissant inchangée les variables qui seront représentées en pile.*
3. *Implémenter le module* `RetrolixKillMove` *qui s'occupe de remplacer toutes les instructions de la forme* `MOVE X,X` *par des commentaires.*
4. *Implémenter le module* `RetrolixCompression` *qui transforme le programme en supprimant tous les commentaires (et en mettant à jour les sauts en conséquence).*

□

2 Structure d'un programme MIPS pour GCC

La production d'un fichier assembleur compatible avec le compilateur GCC nécessite le respect des contraintes suivantes :

- *les conventions d'appel doivent être respectées ;*
- *le point d'entrée du programme doit être une fonction* `main` *;*
- *les variables globales doivent être déclarées.*

Exercice 6 (Structuration du programme)

1. *Complétez la définition de* `main`*. On doit s'assurer que cette fonction initialise correctement les variables globales du programme.*
2. *Complétez la définition de* `code`*. Le code compilé du programme doit contenir l'ensemble des définitions du programme source ainsi que le code du* `main`*.*

□

3 La pile

Dans cette dernière passe, on doit expliciter les opérations d'extension, de réduction, de lecture et d'écriture qui s'appliquent sur la pile du programme. Il s'agit de mettre à jour le registre `sp` *et de l'utiliser comme base pour calculer la position des variables locales en mémoire.*

Exercice 7 (Implémentation des routines de modification de la pile)

1. *Comment doivent être implémentées les extensions et les réductions de la pile ? Complétez les fonctions* `allocate_stack_frame` *et* `free_stack_frame` *en conséquence.*
2. *Sachant que les variable locales à stocker dans la pile sont représentées par la liste* `locals`*, choisissez une façon de positionner chaque variable dans le bloc d'activation. Complétez la fonction* `variable_address` *en conséquence.*

□

4 Compilation des instructions

À l'aide des primitives définies dans les deux exercices précédents, vous devriez pouvoir compléter la fonction de compilation des instructions de Retrolix *vers* Mips*. Vous pouvez ignorer l'instruction* `TailCall`*.*

Pour implémenter les fonctions `block_create`*,* `block_get` *et* `block_set`*, on va utiliser un petit environnement d'exécution qui implémente ces fonctions en C dans un fichier nommé* `runtime.c`*.*

Exercice 8 (Compilation des instructions)

1. *Implémentez les fonctions* `block_create`*,* `block_get` *et* `block_set` *(dans le fichier* `runtime.c`*).*

2. *Implémentez une fonction* `print_int` *qui affiche un entier et saute une ligne (dans le fichier* `runtime.c`*).*
3. *Complétez la fonction* `instruction` *du module* `RetrolixToMIPS` *en ne traitant pas le cas* `TailCall`*. N'oubliez pas d'utiliser* `load_rvalue` *pour éviter toute explosion du nombre de cas à traiter.*

□

5 Comment tester le code compilé ?

Pour tester votre code compilé, il faut se préparer un environnement d'émulation d'une machine Linux-MIPS32 comme suit :

1. *Installer* `qemu` *sur votre système et s'assurer que* `qemu-system-mips` *est bien accessible.*
2. *Téléchargez les images qemu suivantes :*
   — *https://people.debian.org/~aurel32/qemu/mips/debian_wheezy_mips_standard.qcow2*
   — *https://people.debian.org/~aurel32/qemu/mips/vmlinux-3.2.0-4-4kc-malta*
3. *Lancez la commande suivante :*

```
qemu-system-mips \
   -M malta -kernel vmlinux-3.2.0-4-4kc-malta \
   -hda debian_wheezy_mips_standard.qcow2 \
   -append "root=/dev/sda1 console=tty0" \
   -net user,hostfwd=tcp::10022-:22 -net nic
```

4. *Connectez-vous en root (mot de passe : root) et installez* `gcc` *avec la commande* `apt-get install gcc`*.*
5. *Copiez votre* `runtime.c` *sur la machine à l'aide de la commande :*

```
scp -P 10022 runtime.c root@localhost:
```

6. *Compilez votre* `runtime.c` *sur la machine à l'aide la commande :*

```
gcc -Wall -c runtime.c
```

Ensuite, compiler puis exécuter un programme `prog.fopix` *sur la machine émulée peut se faire à l'aide d'une unique commande :*

```
../flap --gcc true -s fopix -t mips -! retrolix prog.fopix && \
scp -P 10022 prog.mips root@localhost:prog.S && \
ssh -p 10022 root:root@localhost 'gcc -o prog runtime.o prog.S && ./prog'
```

6 Comment débugguer le code compilé ?

Pour débugguer le code compilé, il faut commencer par installer `gdb` *sur la machine émulée à l'aide de la commande* `apt-get install gdb`*.*

Puis, on lance le programme dans gdb à l'aide de la commande :

```
% gdb ./prog
```

On peut poser un point d'arrêt sur le `main` *en faisant :*

```
(gdb) b main
```

Puis, lancez le programme à l'aide de la commande `run` *(ou simplement* `r`*).*
On active le désassemblage automatique à l'aide de la commande :

```
(gdb) set disassemble-next-instruction on
```

On peut voir la valeur d'un registre à l'aide la commande :

```
(gdb) info registers sp
```

Enfin, on avance instruction par instruction dans l'exécution, à l'aide de la commande `stepi` *; on saute un appel de fonction sans rentrer dedans à l'aide de la commande* `nexti` *; et on continue l'exécution jusqu'au prochain point d'arrêt à l'aide de la commande* `cont`*.*

Lisez le manuel de `gdb` *pour plus de renseignements.* □