Rapport projet de TDL

Korantin Auguste Etienne Lebrun Maxime Arthaud

mai 2014

Table des matières

1 Introduction

Nous avons décidé de nous éloigner un peu du sujet, puisque nous n'avons pas utilisé EGG, mais une modification de ce dernier faite par Maxime (avec de nombreux ajouts intéressants, mais une syntaxe légèrement différente, et donc incompatible). De plus, nous avons souhaité compiler non seulement en TAM, mais aussi en x86.

Notre compilateur est modulaire et contient donc des classes « machines » capables de générer du code assembleur. Nous avons donc une machine TAM, et une machine x86.

Ainsi, nous pouvons compiler les programmes en assembleur x86, et les faire tourner sur nos propres machines !

Nous disposons aussi d'un petit système de préprocesseur permettant d'inclure d'autres fichiers.

Pour pouvoir nous amuser un peu, nous avons aussi développé une librairie TAM qui propose quelques fonctions utiles, pour afficher du texte... en appelant une simple fonction (au lieu de mettre du code assembleur). De même, nous disposons d'une petite librairie x86, développée entièrement par nos soins et qui effectue des appels systèmes (via une interruption processeur), afin de lire, d'écrire, de faire de l'allocation dynamique... Nous avons même codé une implémentation de la commande « cat » en MOC.

2 EGG

2.1 Typage

Pour gérer les types, nous avons une interface TTYPE. Cette interface a une méthode $getSize()$ pour obtenir la taille du type, en fonction de la machine.

Plusieurs méthodes gèrent la consistance des types :

constructFrom(TTYPE other) : retourne vrai si l'on peut construire le type grâce à partir du type *other*

comparableWith(TTYPE other) : retourne vrai si l'on peut comparer les 2 types

binaryUsable(TTYPE other, String op) : retourne vrai si l'on peut utiliser l'opérateur binaire *op*

unaryUsable(String op) : retourne vrai si l'on peut utiliser l'opérateur unaire *op*

isCastableTo(TTYPE other) : retourne vrai si l'on peut caster le type vers le type *other*

testable() : retourne vrai si le type peut être utilisé dans un if

equals(TTYPE other) : permet de comparer 2 types.

Notre système de vérification est assez souple, comme le C. Par exemple, on peut caster un pointeur d'un type vers un pointeur d'un autre type quelconque.

2.2 Table des symboles

Notre table des symboles peut contenir 3 types d'objets : des variables, des fonctions et des classes.

Les fonctions ont un nom, et la liste des types des paramètres. Lors de l'appel d'une fonction (règle $F \rightarrow ident\ opar\ ARGS\ cpar$), on vérifie son existence dans la table des symboles. Pour chaque argument, on vérifie que le type est compatible. Avec du recul, il aurait été mieux de générer la liste des types des arguments dans le MOC.egg et de faire la vérification après en Java.

2.3 Partie 2

La partie 2 consiste à ajouter la gestion des classes. Nous avons une classe TCLASS, qui contient une référence vers la classe parente, la liste des attributs et des méthodes, sans inclure les attributs et les méthodes de la classe parente. Quand on essaie d'appeler une méthode, on appelle la méthode $findCallableMethod$ de TCLASS, qui va regarder dans la classe et si besoin dans les classes parentes. Les classes sont dans la table des symboles, il n'est pas nécessaire de mettre les attributs et les méthodes dans la table.

Une méthode $getVTable()$ permet de générer la liste des méthodes de la classe, avec l'offset de chaque méthode.

Nous avons du gérer spécialement la méthode init, qui est statique mais crée implicitement un objet. Nous gérons aussi le mot clef *super* qui force un appel sans résolution tardive.

3 La machine TAM

3.1 Partie 1

3.1.1 gestion des chaînes de caractères

Un des premiers problèmes que nous avons eu à résoudre est la gestion des chaînes de caractères. Nous voulions en effet qu'elles soient compatibles avec les char*, ce qui n'est pas possible avec les fonctions proposées par TAM, qui ne gère que des chaînes statiques. La règle $F \rightarrow string$ appelle donc la fonction $genString()$ de la classe MTAM. Celle-ci génère un code qui sera exécuté au début du programme, et qui positionne la chaine en mémoire en début de pile avec des LOADL . Le code renvoyé par $genString()$ pose sur la pile un pointeur vers le premier caractère de la chaîne.

3.1.2 sous-classes TAM Parameters et Variable Locator

Ces classes sont nécessaires car le traitement des variables et des paramètres diffère en fonction du langage cible.

Le $ParametersLocator$ permet de positionner les paramètres lors de l'appel d'une fonction. Le premier paramètre se situe en -1[LB], les suivants sont en dessous.

Le *VariableLocator* est initialisé à partir d'un offset initial, et alloue les positions en décalant l'offset en fonction de la taille des variables. À chaque nouveau bloc, un nouveau *locator* est généré à partir du précédent en lui communiquant son offset courant. De cette manière, les variables locales au sous-bloc pourront être écrasées par le premier bloc.

3.1.3 lib TAM

Nous avons décidé d'écrire une petite librairie standard, principalement pour gérer les méthodes sur les chaînes de caractères et malloc. C'est elle aussi qui contient l'assembleur inline "CALL (LB) f_main HALT " qui permet de lancer le programme principal. Elle est donc inclue dans tous les tests de TAM à l'aide du #include qui est remplacé par le préprocesseur. L'inclusion de la boucle while à la définition du langage a permis de la rédiger en grande partie en Moc.

3.2 Partie 2

La principale difficulté de cette partie 2 a été la gestion de la liaison tardive. Nous avons donc implémenté un système de table de méthodes virtuelles. Le code qui génère la vtable est ajouté au code d'initialisation lors de la création de la classe correspondante. Il est constitué d'autant de LOADA que de méthodes , suivis des labels correspondants, ceux de la classe mère pour les méthodes qui n'ont pas été redéfinies. Le premier champ d'une instance de classe est un pointeur vers la vtable de la classe réelle. Lors d'un appel de méthode, on ajoute à celui-ci la position de la méthode voulue dans la vtable. On charge la valeur avec un LOADI puis on appelle la méthode avec un CALLI. CALLI n'empilant que deux valeurs, on doit rajouter une valeur sur la pile avec un LOADL 0.

4 La machine x86

La machine x86 est chargée de générer du code assembleur x86 (qui devra être compilé par nasm). Au final, elle est très proche de la machine TAM et je ne m'étendrai pas sur le sujet. Toutefois, un point diffère complètement : là ou TAM est une machine à pile pure, en x86 nous devons gérer les registres.

4.1 Gestion des registres

Plusieurs choix se sont offerts à nous pour gérer les registres en x86 :

Tout d'abord, nous aurions pu tout mettre sur la pile, à la façon de TAM. Lors d'un calcul, il suffisait de faire comme TAM en effectuant des pop, de faire le calcul, et de tout remettre sur la pile. Ce système aurait été très simple, mais nous aurions toujours utilisé un ou deux registres, et notre but était d'aller un peu plus loin que la simple machine à pile.

A l'autre extrême, il aurait été possible dans l'idéal de stocker certaines variables uniquement dans des registres, sans jamais les mettre en mémoire, et d'utiliser tous les

registres de manière intensive. C'est ce que font les vrais compilateurs, mais il s'agit de quelque chose d'extrêmement complexe.

Au final, nous avons retenu une solution qui est relativement simple, tout en tirant déjà plus partie des registres : les variables sont toutes stockées sur la pile, mais lors d'un calcul tous les résultats intermédiaires pourront être stockés dans divers registres, et n'auront jamais besoin d'être mis sur la pile.

Pour cela, nous utilisons une pile qui retient les registres contenant les résultats des derniers calculs. Ainsi, un calcul verra ses résultats intermédiaires stockés dans différents registres. Cette solution marche plutôt bien, tire vraiment partie des registres et n'a pas été trop dure à mettre en oeuvre.

4.2 lib x86

Pour faciliter le travail, nous avons développé une petite bibliothèque qu'il suffit d'inclure, et qui se charge de générer un squelette de programme en assembleur, avec une dizaine de fonctions utiles pour permettre à nos programme de faire des choses : lire, écrire, quitter, afficher du texte...

Son développement fut très intéressant, puisque nous avons tout fait nous-même, sans faire appel à une bibliothèque tierce comme la libc. Pour communiquer avec le système d'exploitation, il suffit d'utiliser l'instruction « int » permettant de faire une interruption, et de mettre les bonnes données dans les registres, pour dire au système d'exploitation ce que l'on attend de lui.

Ainsi, pour afficher du texte nous avons une fonction print, qui fait elle-même appel à une fonction write, qui effectue un appel système :

```
int write(int fd, char* buf, int nbytes) {
    asm("
        ; write syscall
        mov eax, 4
        mov ebx, %fd
        mov ecx, %buf
        mov edx, %nbytes
        int 0x80 ; the result is returned into eax : perfect
    ");
}

void print(char *s) {
    write(1, s, strlen(s));
}
```

5 Tests

Nous avons une série de tests pour le système de types. Celui-ci est fait en python, et peut être lancé avec *python3 test.py*

6 Conclusion

Ce projet aura été l'occasion d'effectuer un petit compilateur jouet, et nous aura permis de nous rendre compte que c'était une tâche assez longue, mais réalisable.

Toutefois, nous avons aussi mis le doigt sur le fait que s'il est simple de réaliser un petit compilateur qui fonctionne, arriver à lui faire générer du code rapide et optimisé doit être une tâche extrêmement difficile.

Le fait d'arriver à générer du x86 a donné un côté très amusant au projet, puisque nous pouvions lancer nos programmes sur nos machines, et même utiliser un debugger comme gdb pour comprendre certains bugs.

Arriver à effectuer des appels système pour avoir un programme qui fait de vraies actions sur nos ordinateurs était aussi particulièrement sympathique !