Projet de Compilation

Grégoire DHIMOÏLA

23 janvier 2023

Résumé

Ce rapport présente mon second projet du cours de Programmation I, dans lequel j'ai implémenté un compilateur de NanoGo. Nous verrons en particulier les parties dédiées au typage et à la production de code assembleur.

1 Typage

A l'entrée de cette phase, on assure que le code go est valide en terme de lexique et de syntaxe. On fournit également un arbre syntaxique représentant les instructions à effectuer, et dans quel ordre.

Le but de cette phase est de prendre cet arbre syntaxique, et de produire un arbre typé, assurant que le code est valide sur ce point, ainsi que de poser des bases qui seront utiles dans la phase de compilation.

1.1 Choix d'implémentation

Pour les environnements de structure et de fonction, j'ai choisi d'utiliser de simples tables de hachages, contrairement au gros module Env qui a été fournit pour gérer les variables, par simplicité et parce que cela suffisait amplement à l'utilisation que j'en ai fait.

La principale raison pour laquelle une simple table de hachage suffit est que toutes les structures et toutes les fonctions sont déclarées globalement, alors que pour les variables, les blocs peuvent s'imbriquer, ou s'appeler les uns les autres, ce qui introduit de la localité et des subtilités sur le fait qu'une variable soit utilisée ou non, sur sa portée, ...

Pour savoir si une structure est récursive, j'ai choisit un algorithme qui s'inclue dans celui de calcul de la taille d'une structure, ce qui a le bon goût d'être à la fois élégant et très efficace puisqu'on ne parcours qu'une fois chaque structure. Les tailles des structures sont initialisées à -1, et lorsqu'on calcule pour la première fois la taille d'une structure, on la met à -2 jusqu'à avoir fini le calcul. Ce calcul est récursif. Ainsi, dès que l'on croise une structure de taille -2, on sait qu'on a un cas de structure récursive.

1.2 Difficultés et échecs

— Je n'ai pas implémenté le traitement de nil. Ceci peut se faire par quelques disjonction de cas en plus et l'usage du type Twild. Cependant je n'ai pas réussi à déterminer comment traiter ce type, puisqu'il doit être égal à n'importe quel autre mais un seul à la fois...

— Le traitement des "_" a été plutôt difficile, car je ne savais pas vraimant à quel endroit il fallait s'en occuper, et comment. Il s'avère que la plupart du temps, le traitement des "_" était plus ou moins le même que le traitement des variables qui ne sont pas dans le même bloc, donc au final la situation s'est débloquée assez naturellement.

— Un autre problème rencontré a été la gestion des return. En effet, les cas comme

```
{
...
if b then
    return
...
}
```

présentent un problème de retour pariel que je ne savais pas comment gérer, j'ai donc du rajouter un champ dans les expr spécifique au retours pour détecter les retours partiels.

2 Production de code assembleur

On suppose maintenant que l'on possède un arbre bien typé, et on s'intéresse à la production de code machine qui executera effectivement les instructions demandées par l'utilisateur au travers du code go.

Contrairement au projet précédent, on ne cherchera aucunement à rendre ce code efficace. Tout ce qu'on demande est qu'il produise la bonne sortie.

2.1 Détour sur une segfault, une mamie et l'histoire du temps qui passe

"On t'a déjà dit que tu as une tête à claques ?" - Une mamie dans le métro au jeune moi. Une étude américaine a réussi à démontrer qu'en fait cette phrase était un message adressé au travers du temps à notre cher ami Windows. Sa condition de mamie étant par ailleurs une représentation métafantasmatique du temps qui pèsera sur mes épaules comme manifestation du fardeau des longs combats perdus d'avance contre une telle machiavélie digitale. En voici un court résumé :

— Segfault
— Segfault
— Segfault
— ...
— Bon, passons sur Linux.
— OH BAH TIENS, BIZARREMENT CA MARCHE BEAUCOUP MIEUX !

Ainsi s'acheva l'histoire du message qui traversa le temps. Il vécut heureux, et eut beaucoup d'enfants.

2.2 Choix d'implémentation

— Comme une grande partie de l'évaluation du projet sera faite au travers d'affichages sur la ligne de commande, une attention toute particulière a été apportée à la gestion des affichage. Je n'ai par ailleurs pas mis de fonction d'affichage par défaut, typiquement, print_int ou print_string, car puisqu'il faut gérer les fonctions d'affichage personnalisées pour les structures, autant ne pas polluer inutilement le code et ne mettre que les fonctions utiles. Les fonctions déjà générées sont stockées dans un environnement spécial pour ne pas les générer plusieurs fois inutilement.
— J'ai commencé par implémenter tous les cas simples et les cas de bases, tel que les opérations arithmétiques sur des constantes, puis j'ai construit petit à petit de quoi gérer des cas plus complexes. L'idée étant que si les cas de base fonctionnent et que les compositions de cas de base fonctionnent, alors tout fonctionne. Le problème est qu'il faut penser à tous les cas qui pourraient avoir été ommis.
— Par soucis de simplicité du code, tout ce que l'on stockera dans la pile sera de taille 8. En particulier, pas de structure dans la pile. Il est donc important de faire la distinction entre une structure, qui est un pointeur vers l'endroit de la mémoire qui contient effectivement cette structure, et un pointeur vers une structure, qui pointe alors vers un endroit de la pile où est stocké le pointeur vers la vraie structure.
— Par soucis de simplicité du code, toutes les fonctions doivent être traitées de la même manière, et en particulier, une fonction qui ne renvoie qu'un seul élément le renverra dans la pile et non pas dans rax.
— La remarque sur la représentation des structures est à nuancer : si un champ d'une structure A contient une structure B, alors cette instance de la structure B est effectivement stockée à l'intérieur de la structure A, et n'est donc plus traitée comme un pointeur vers la mémoire. Ceci pose un problème lors de l'affichage de a.b que je n'ai pas réglé.
— Un problème un peu spécifique maintenant : dans mon implémentation, si on retourne l'adresse d'une variable locale, ça pointe vers un endroit dans la pile qu'on peut écraser. Le vrai go traite ce cas en allouant une case mémoire, et en renvoyant le pointeur vers cette case. Je ne sais pas ce qu'il en est du nano go.
— Pour les TEassign, je n'ai pas compris à quoi servait la disjonction de cas, ni pourquoi aucun cas ne comprenait de listes lvl/el, simplement des listes à un élément et des liste de listes

([lvl]/[el]), j'ai donc pris la libertié de modifier ce cas là.

2.3 Difficultés et échecs

— La sémantique demande d'évaluer les arguments d'une fonction de droite à gauche. Ceci n'est pas respecté pour le cas de la fonction print.
— Je ne gère pas l'affichage de fonctions qui ont plusieurs retours. Il suffit d'évaluer la fonction, puis de parcourir la pile et la liste fn_type simultanément pour savoir comment afficher ces fonctions.