Projet Programmation 1

Compilation de NanoGo

Louis Lachaize

Pour le 24 janvier 2023

Le code de ce projet est disponible à l'adresse `https://github.com/louiiiiiis/Projet-Prog-1-NanoGo`. Ce document détaille les principales étapes du projet, les difficultés rencontrées et les différences par rapport à ce qui a été demandé.

Le but est de compiler un fragment de Go en code assembleur `x86-64`. Les étapes d'analyses lexicale et syntaxique sont fournies avec le sujet et n'ont pas été modifiées. Ces étapes permettent d'aboutir à un arbre syntaxique abstrait que l'on doit correctement typer, puis compiler. Pour cela le squelette des fichiers est fourni, et à compléter.

1 Typage

L'obectif de cette étape de typage est de parcourir l'AST généré par le parseur afin de vérifier s'il est bien typé. Ainsi, suivant le théorème de Milner[1], aucune erreur ou résultat imprévu ne pourra être généré lors de la compilation à proprement parler. Cette étape permet également de préparer un arbre syntaxique typé et relativement agréable à parcourir lors de la gérération du code assembleur car il contient bon nombre d'informations (comme la portée des variables par exemple).

1.1 Adaptation au squelette fourni

La première difficulté de ce projet, et peut-être la plus longue, a été la compréhension du squelette déjà fourni avec la source. En effet, plusieurs choix d'implémentations n'auraient pas été les miens en partant d'un sujet vierge comme l'utilisation de modules ou encore la définition d'environnement. J'ai essayé au maximum de compléter le code fourni sans trop m'éloigner de l'esprit de celui-ci, mais j'ai tout de même changé certains points. J'ai notamment utilisé des tables de hachage pour les environnements, qui sont simples d'utilisation et permettent de repérer rapidement les doublons dans les noms d'objets par exemple.

1.2 Environnements

La principale difficulté du typage a été la gestion des environnements. Il faut un environnement pour les variables, un pour les structures et un pour les fonctions. Cependant, les fonctions et les structures sont mutuellement récursives quelque soit leur ordre d'appel,

[1] "Well-typed programs do not go Wrong"

autrement dit on peut utiliser une fonction qui n'est définie que plus tard dans le code. En conséquence, les environnements sont difficiles à initialiser. Pour cela j'ai réalisé un algorithme en deux passes : la première passe consiste à n'ajouter que le nom de l'objet dans l'environnement, pas son contenu. Lors de la deuxième passe, on complète le contenu ce qui est possible car tous les objets se trouvent maintenant dans l'environnement (ou du moins leur nom). Concrètement, j'ai même utilisé deux tables de hachage différentes lors des deux passes :

- Lors de la première passe, on garde un arbre syntaxique non typé, et l'environnement contient des objets non typés (on ne touche pas au contenu de ces objets, on s'intéresse seulement à leur nom pour l'instant).
- Lors de la deuxième passe, on transforme le contenu non typé des objets en contenu typé, en l'ajoutant dans un nouvel environnement : cet environnement contient tout ce qu'il faut pour construire l'arbre syntaxique typé, et son remplissage "au fur et à mesure" du parcours permet de ne pas recalculer plusieurs fois la même chose : avant de type le contenu d'un objet on vérifie si cela n'a pas déjà été fait et enregistré dans l'environnement.

2 Compilation en assembleur `x86-64`

Ma mauvaise gestion du temps sur ce projet ne m'a pas permis de terminer le compilateur, mais seulement une version minimale car j'ai préféré traiter bien quelques cas plutôt que moyennement tous les cas. Voici donc les commandes prises en charge par mon compilateur et celles qui ne le sont pas (globalement dans l'ordre dans lequel j'ai fonctionné ou prévoyais de fonctionner) :

Traité
Opérations arithmétiques et logiques
Boucles if, for
Appels à fmt.Print (int, bool, string)
Utilisation de variables
Appels de fonction
Utilisation de pointeurs et d'adresses

Non traité
Fonctions renvoyant 2 éléments ou plus
Utilisation de structures
Tests d'égalité hors entiers

Par ailleurs, la sémantique n'est pas toujours rigoureusment respectée : les arguments d'une fonction par exemple sont censées être évaluées de droite à gauche mais cela n'a de toute façon pas d'importance car leur calcul n'a pas d'effet de bord.

2.1 Utilisation de la pile

L'utilisation de la pile n'est pas très compliquée car les seuls types utilisés (entiers, booléens vus comme des entiers, pointeurs, chaînes de caractères vus comme des pointeurs vers la data) occupent tous 8 bits. Les registres `%rsp` et `%rbp` se déplacent donc de 8 en 8. La pile sert à stocker les résultats intermédiaires aux calculs, les variables et les arguments de fonctions.

Environnement et gestion des variables

Je n'ai pas réellement compris l'environnement proposé par le sujet, je l'ai donc redéfini. J'ai utilisé pour cela une table de hachage qui à l'identificateur d'une variable (`var.v_id`, unique) associe son offset dans la pile (par rapport à l'origine `%rbp`). Lors de la déclaration d'une variable, on cherche une adresse disponible, on ajoute dans la table de hachage l'offset correspondant, puis on place arbitrairement la valeur `0` à son adresse dans la pile afin de réserver cet espace. Lors de l'appel d'une variable ou de l'assignation d'une valeur, il suffit d'accéder à son adresse dans la pile.

Appels de fonction

Lors de l'appel d'une fonction, on commence par placer tous ses arguments sur la pile. Ensuite on stocke (toujours sur la pile) la valeur courante du registre `%rbp` qui indique l'"origine" de la pile, pour pouvoir la remettre à zéro (*ie* à la même valeur que `%rsp`). La fonction peut alors utiliser une zone de la pile "vide", qui lui est réservée, pour s'exécuter. Par contre, quand ses arguments sont utilisés, il faut aller les chercher au dessus de la pile (*ie* avec un offset positif). Une fois l'appel terminé, on dépile les arguments qu'on avait mis sur la pile, et on redonne la bonne valeur à `%rbp` qui correspond la zone de la pile réservée à l'appelant.

2.2 Problème avec l'opérateur infixe ++

Le module `X86_64` permet de générer le fichier assembleur ligne par ligne en les concaténant grâce à l'opérateur infixe `++`. Cependant, l'ordre de priorité d'`OCaml` pour les opérateurs infixe nous pose problème : dans l'expression `a ++ b`, `b` est évalué en premier. En particulier lorsque `a` ou `b` a un effet de bord (par exemple définir l'adresse d'une variable dans l'environnement), celui-ci n'est pas réalisé dans l'ordre voulu (certaines variables sonr donc utilisées sans encore avoir été définies dans l'environnement). Ma première idée a été de redéfinir un nouvel opérateur infixe :

```
let (+++) a b =
    let val_a = a
    in val_a ++ b
```

mais cela ne résoud pas le problème puisque c'est toujours `b` qui va être évalué en premier dans l'expression `a +++ b`. Il faut donc lourdement remplacer dans le code toutes les expressions `a ++ b` dans lesquelles `a` ou `b` a un effet de bord par

```
let val_a = a in a ++ b
```

3 Tests

Un dossier de tests se trouve également avec le projet. Chaque test se concentre sur un point particulier du compilateur de manière à ce que le code assembleur généré soit plus court et plus lisible. Les fichiers de test ont des noms assez explicites.