Compilation de **Petit Go**

Lucas Tabary-Maujean — ENS Paris-Saclay
lucas.tabary@ens-paris-saclay.fr

25 janvier 2022

Résumé

On propose dans ce rapport une présentation du second projet de Programmation de première année au département d'informatique de l'ENS Paris-Saclay. Le but de ce projet est de réaliser un compilateur capable de produire du code assembleur `x86_64` à partir du langage Petit Go, dérivé simplifié du langage Go. On détaille par la suite la réalisation des différentes parties du projet.

Notons tout d'abord que les analyseurs lexicaux et syntaxiques étaient fournis avec le sujet et n'ont pas été modifiés.

1 Phase de typage

1.1 Principes

La structure de la phase de typage respecte globalement le format proposé par le sujet. J'ai toutefois fait le choix de modifier la structure proposée pour l'enregistrement des fonctions et des structures, privilégiant une structure de données statique construite à partir du foncteur `Map.Make`. De manière cohérente, on construit alors des environnements de variables de la forme :

```
type struct_env = structure M.t
type fun_env = function_ M.t

module Env = struct
  type e = {
    vars: var M.t;
    functions: fun_env;
    structures: struct_env;
    return_values: typ list;
    returns: bool;
    depth: int;
  }
  ...
end
```

Ces environnements, qui permettent de juger de la licéité de l'utilisation des variables, nécessite notamment d'en juger la portée. L'introduction d'un modèle statique permet d'introduire la notion de *profondeur* d'un environnement, caractérisant les imbrications dans des blocs successifs de code : chaque environnement est dupliqué en en augmentant strictement la profondeur à l'entrée d'un nouveau bloc, afin de traiter efficacement de la portée et de l'effet de *shadowing* des différentes variables.

D'un point de vue algorithmique, on peut relever la présence de deux fonctions s'appuyant sur une logique de parcours en profondeur, d'une part pour détecter les constructions cycliques de structures, et, d'autre part pour construire de façon mémoïzante la fonction `sizeof` qui détermine la taille en mémoire de tout type utilisé.

1.2 Réalisation

Le typage de ce compilateur réalise l'ensemble des tâches demandées. On détaille par la suite les différences qui ont été introduites.

Aucune sémantique n'est donnée pour les procédures, même si celles-ci sont présentes en Go. En remarquant que le type de retour de telles fonctions est aussi `tvoid = Tmany []`, autoriser les procédures donne ainsi une caractérisation par le type de ce qu'est une instruction.

La sémantique du *wildcard* a été étendue pour mieux correspondre à celle du langage Go complet.

Plus précisément, `_` ne peut être utilisé comme *r-value*. De plus, il n'a pas besoin d'être initialisé par une structure de la forme `var _ = ...`, car il correspond à une unique variable `wildvar`, systématiquement défini, et de type `Twild`, qui est le seul type à contourner le système de typage.

La génération des visuels pour les arbres de la syntaxe abstraite est réalisée par le fichier `pretty.ml`, généreusement mis à disposition de tous les élèves par Vincent Lafeychine.

Les expressions typées `TEvars` contiennent aussi les éventuelles expressions assignées aux variables (qui valent alors, soit `<nil>`, soit une nouvelle structure allouée sur le tas). Cela rend la création du code assembleur plus simple, et évite d'avoir à considérer des méthodes peu élégantes comme le remplacement de déclarations de variables par un bloc contenant la déclaration de ces variables, puis leur assignation.

2 Phase de production de code

2.1 Principes

Là encore, on respecte globalement la structure suggéré par l'énoncé. Une différence significative est cependant l'absence du fichier `rewrite.ml`, c'est-à-dire qu'aucune règle de réécriture n'est appliquée à l'arbre de la syntaxe abstraite typé produit à l'issu de la phase de typage. Ce choix est personnel, les réécritures du TAST me limitant trop. Pour le reste, il s'agit de construire récursivement un ensemble d'instructions traduisant les opérations associées à l'arbre. Pour cela, on utilise conjointement plusieurs fonctions mutuellement récursives :

- `expr env` évalue une expression dans un environnement donné ;
- `expr_address env` évalue l'adresse d'une *l-value* ;
- `expr_stack env` évalue une liste d'expression et les rajoute successivement à la pile ;
- de manière plus détachée, `expr_print env` évalue l'affichage de différents objets, en lien avec la fonction `add_print_capability` qui rajoute les fonctions nécessaires à afficher un type donné.

En combinant ces différentes fonctions, la production de code recouvre les différents aspects demandés, notamment, la gestion des fonctions à paramètres et valeurs de retour sur la pile, éventuellement multiples.

2.2 Réalisation

Le code proposé réalise la plupart des tâches demandées. Comme déjà indiqué, les fonctions sont entièrement prises en charge, à l'exception de l'affichage d'une fonction à plusieurs valeurs de retours. Les structures sont aussi gérées, et elles sont systématiquement localisées sur le tas. En l'absence d'un ensemble de règles de réécriture, cela nécessite de traiter à part les structures, mais certains de ces traitements auraient été de toute manière présents. Moralement, le production de code perçoit un type `Tstruct s` comme suit :

Une variable de type `Tstruct s` stockée sur la pile contient l'adresse d'une variable sur le tas, et est telle que son appel d'évaluation engendre la création d'une copie d'elle-même sur le tas. Par ailleurs, la comparaison de deux variables de type `Tstruct s` équivaut à la comparaison bit-à-bit du contenu des adresses qu'elles indiquent.

Pour relever les différences avec le modèle proposé, on peut observer la différence des attributs stockés dans les environnements considérés lors de l'évaluation. On peut aussi remarquer l'utilisation d'un cadre (pointeur de *frame*) autour des appels aux fonctions `C`, afin de préserver un alignement de la pile à 16 octets. Par exemple :

```
1 pushq   %rbp
2 movq    %rsp, %rbp
3 andb    $0xF0, %spl
4 xorq    %rax, %rax
5 call    printf
6 leave
```

2.3 Optimisations

Le code produit est normalement correct, mais inefficace. Il s'appuie lourdement sur la pile et engendre des déplacements de valeur inutiles entre les différents emplacements de la mémoire. Une première idée d'amélioration relativement simple à mettre en place consisterait à indiquer en plus comme argument de la fonction `expr` un registre de destination.

Il est cependant encore moins facile d'implémenter une utilisation efficace des différents registres, car cela nécessite de maintenir dans l'environnement la connaissance des différents registres utilisés ou non.

3 Gestion du projet

Le projet a été développé en s'appuyant sur un dépôt Gitlab fourni par le Crans.

Les différents tests sont disponibles dans le dossier `tests/`, et peuvent être réalisés avec le script `correctness.sh`, qui permet d'effectuer des tests de typage ainsi que de comparaison de résultat avec des exécutions fournies par `Go`.

Le code fourni avec le projet se compile avec la commande `make`, depuis le dossier `src/`. La compilation utilise `dune` et `menhir`, et nécessite une version de OCaml suffisamment récente. Notamment, une version trop ancienne ne permet pas l'utilisation de certaines fonctions du module `Hashtbl` et du foncteur `Map.Make`.