Rapport PetitC

Vladimir Ivanov Elias Caeiro

15 janvier 2023

1 Syntaxe et Typage

Le typeur transforme l'arbre abstrait de syntaxe du fichier en une liste de fonctions et un environement. Toutes les variables (même les variables locales de fonctions differentes pas imbirquées l'une dans l'autre) et toutes les fonctions sont indexées par des ids distincts, ainsi, on évite tout problème de conflit de noms.

— L'environement assossie à chaque fonction sa signature, les ids de ses fonctions parents (les fonctions dans lesquelles elle est imbirquée) et sa profondeur (0 pour les fonctions top level, 1 pour les fonctions imbirquées, 2 pour les fonctions imbriquées das les fonctions imbirquées). Il associe au variables leurs types et les profondeurs des fonctions dans lesquelles elles sont définies.

— Chaque fonction est decrite par son id, larbre *typé* de son corps (qu'avec des instructions, sans les fonctions qui y sont imbriquées), les ids de ses variables locales (mais pas ceux des variables de ses parents ou enfants) et les ids de ses arguments, qui sont considérés aussi comme variables locales.

2 L'assembleur

Une instruction est représentée par un type `inst`, on a alors une liste d'instructions et labels par fonction. Les labels ne sont unique qu'au sein des fonctions, on les préfixe par les ids des fonctions quand on transforme notre représentation de l'assembleur en texte. Pas toutes les instructions qui peuvent être construites par le type `instr` sont légales, mais toutes celles qu'on produit le sont.

On définit les regles de typage ne Coq. Notons que les équivalences de pointeurs et déréferencements de `void*` n'y sont pas conformes au sujet, mais au comportement qu'on a observé dans clang et gcc.

3 Génération de code

Nous procédons de la manière naturelle, en traversant l'arbre abstrait de syntaxe de chaque fonction et en génèrant de l'assembleur. Les seules choses notables relevent du

fait qu'on fait de la programmation purement fonctionnelle, ainsi, on utilise une monade d'état et on passe en argument à chaque fonction génératrice de code des instructions assembleur à ajouter à la fin de celles qu'elle produit. La majorité de ses fonctions renvoient des `option` car on fait des operations qui peuvent échouer, telles que la lecture dans un dictionnaire. On devrait pouvoir prouver qu'elles renvoient toujours `Some` sur une entrée bien typée et ainsi eliminer les `option`.

4 Sémantique

Nous définissons une sémantique à petits pas avec continuations pour PetitC. Nous définissons un petit pas d'une paire état - continuation vers une autre paire état - continuation. Un état consiste en le contenu de la mémoire, les charactéres qui ont été affichés et les pointeurs, ainsi que les pointeurs vers les emplacements en mémoire des variables de la portée (notons qu'on ne pourait pas juste garder leurs valeurs car le programmeur peut acceder à leurs adresses). Quand nous faisons un appel, ces pointeurs sont mémorisés dans la continuation (ceci est pertinant pour les appels récursifs, où ces pointeurs changent). Nous n'avons pas défini la sémantique pour la boucle for.

La sémentique est plus forte que celle de C. Par exemple, nous fixons un ordre d'évaluation pour qu'elle devienne déterministe (bien qu'on puisse penser qu'il est de gauche à droite, il ne l'est pas car le typeur transforme `x > y` en `y < x`, par exemple). Nous définissons aussi des comportements pour des situations où ils sont indéfinis en C, par exemple, ajouter un entier à un pointeur lui ajoute toujours 8 fois cet entier octets (alors qu'en C c'est indéfini pour `void*`). En revanche, dans notre sémentique, la division par 0 et le déréferencement d'un pointeur nul sont des comportements indéfinis, alors qu'il y a des tests qui testent qu'ils échouent, car (des gens sur internet ont dit que) c'est un comportement indéfini en C.

Nous avons prouvé en Coq que la sémantique de PetitC (sans les boucles for) est déterministe.

Nous définissons aussi la sémantique du fragment de l'assembleur qu'on utilise, mais nous ne prouvons rien dessus. La définir était beaucoup plus simple car on peut définir un petit pas d'un état (mémoire et sortie) vers un autre, pas besoin de continuations ni de contextes. Nous n'avons rien prouvé sur cette sémantique.

5 Tentative de Certification du Générateur de Code

Nous avons essayé de certifier la génèration de code mais n'avons beaucoup avancé.

— Pour cela, il est nécessoire de définir l'équivalence de mémoire. Deux états de la mémoire sont équivalents s'ils sont les mêmes à permutation des ids des blocs sur le tas et des valeurs sur la pile. Ainsi, nous ne deverons pas prouvé que le code assembleur produit le même état de la mémoire que le code PetitC (ce qui est faux vu notre définition de la sémantique), mais qu'ils produisent des états équivalents. Pour formaliser l'équivalence d'états, nous disons qu'un ensemble de pairse de pointeurs correspondants sont équivalents si pour chacun d'entre eux, s'il pointe

vers un bloc ou une valeur sur la pile, cette valeur ou les éléments de ce bloc sont soit égales soit correspondants si c'est des pointeurs. Nous n'autorisons pas de mettre une valeur du tat sur la pile car ça simplifie la définition. Nous disons alors que la deux états mémoire accessibles depuis les variable locales sont équivalents si l'on peut appareiller en paires de pointeurs correspondants les parties de ces états mémoire accessibles depuis ces variables.

— A cause des appels récursifs, toute inducition structurelle sur l'arbre typé pour prouver que la génération de code est correcte est vouée à l'échec. Nous faisons ainsi une récurence forte sur le nombre de petits pas pour passer d'un état initial à un état final dans la sémantique de PetitC.

— Nous prouvons la backward simulation, qui équivaut la la forward simulation des que la sémantique est déterministe, mais qui est plus simple à prouver.

6 Structure du Projet

Le typeur, le lexeur, le parseur, l'afficheur de l'assembleur et l'interface entre tous les modules sont écrits en OCaml. L'arbre typé, la représentation de l'assembleur, et le générateur de code sont écrits en Coq et extraits vers OCaml.