Projet Programmation (partie III)
Rapport

Thomas Filasto

05/12/2022 - 24/01/2023

Affichage

J'ai ajouté à print_int les fonctions print_string et print_bool. Pour print_string, il me suffit d'appeler printf puisque cette fonction affiche la chaîne de caractère placée dans %rdi. Pour print_bool, j'utilise deux fonctions auxiliaires : une qui affiche "true" et l'autre qui affiche "false". Dans la fonction principale je fais un saut conditionnel à la fonction qui affiche la valeur placée dans %rdi.

Opérations arithmétiques

Pour effectuer les opérations arithmétiques, j'ai surtout réutilisé les fonctions correspondantes en assembleur. Pour les opérations binaires, je génère le code permettant de placer l'évaluation d'une expression puis j'empile cette valeur. Lorsque les deux expressions ont été évaluées, je dépile leurs valeurs et j'appelle la fonction correspondante en assembleur. J'ai appliqué le même procédé pour Uneg mais je n'ai pas pu le faire pour Unot, j'ai donc dû réécrire un code en assembleur qui fait ce que je souhaitais.

Opérations booléennes

Mon idée de départ pour les opérations booléennes était d'utiliser une première fonction qui testait si la première expression nous permettait de conclure sur la valeur de l'opération complète si c'était le cas je faisait un saut conditionnel à une fonction qui plaçait vrai/faux dans %rdi. Dans le cas contraire, je faisais un saut conditionnel vers une fonction auxiliaire qui déduisait de la valeur de la deuxième expression la valeur de l'opération. Mais lorsqu'il y avait plusieurs opérations booléennes imbriquées, j'avais besoin de beaucoup de fonctions : pour n opérations booléennes $\forall i \in [\![0, n-1]\!]$ je devais créer des fonctions *and_i, or_i, and_aux_i, or_aux_i, true_i* et *false_i*. J'arrivais à le gérer de la manière suivante : j'ai créé une référence que j'incrémentais à chaque fois que je rencontrais un opérateur booléen puis dans la fonction *file* j'ai créé une référence contenant initialement la chaîne de caractère vide puis je faisais une boucle allant de 0 à n-1 dans laquelle je concaténais le code des fonctions mentionnées auparavant à la référence puis je concaténais cette référence au code des fonctions d'affichage. Toutefois lorsque j'ai voulu donner un comportement paresseux à mes opérations j'obtenais des erreurs dont je ne parvenais pas à me défaire et j'ai donc décidé de changer de méthode.
En parallèle j'ai réfléchi à la manière d'implémenter les "if" et je me suis rendu compte que je pouvais implémenter beaucoup plus facilement les opérations booléennes de la même façon : par exemple $e_1 \wedge e_2$ revient à écrire le code de if e_1 then (if e_2 then true else false) else false, ce qui donne en plus immédiatement le caractère paresseux.

Comparaisons arithmétiques

J'ai implémenté l'ensemble des comparaisons arithmétiques de la même manière : j'évalue les deux expressions à comparer en plaçant leurs valeurs sur la pile une fois évaluées pour éviter que celles-ci ne soient modifiées par les effets de bord des autres opérations. Ensuite je les compare à l'aide de l'opération cmpq. Je peux ensuite utiliser les différents drapeaux pour faire les sauts conditionnels adaptés vers un fonction qui place vrai/faux dans %rdi. Comme la seule chose qui changeait selon les comparaisons à effectuer était les sauts conditionnels à effectuer, j'ai utilisé une fonction en caml qui fournissait les sauts conditionnels adaptés à l'opération de comparaison. Il est à noter que j'ai procédé de la même manière pour les tests d'égalité/différence bien que cela ne fonctionne pas sur les chaînes de caractères mais je n'ai de toute façon pas trouvé de moyen de générer du code comparant des objets de ce type.

If et For

Pour gérer le cas des if et for, j'ai du utiliser des sauts conditionnels. Je me suis d'abord occupé des if pour lesquels j'ai procédé de la manière suivante : d'abord je génère le code de la condition du if ensuite je crée une fonction en assembleur qui contient le code du "then", une autre pour le code du "else" et une

autre qui contient la suite du code à générer. Si la condition est vérifiée ce qui se traduit par le fait que %rdi ne contienne pas 0 on fait un saut à la fonction contenant le code du if sinon on fait un saut à la fonction contenant le code du "else". À la fin de ces deux fonctions, on fait un saut à la fonction qui contient la suite du code.
Pour le for, on réutilise le même principe mais en plus simple : on crée une fonction en assembleur dans laquelle on écrit le code associé à la condition, une autre qui contient le code associé au corps de la boucle for et une dernière qui contient la suite du code à générer. Tant que la condition n'est pas vérifiée on fait un saut à la fonction qui contient le code du corps de boucle puis à la fin de la fonction on fait un saut à la fonction qui teste la condition. Si la condition est vérifiée on fait un saut à la fonction qui contient la suite du code.

Gestion des variables

Pour incorporer les variables dans le compilateur, j'ai décidé de les empiler sur la pile en stockant leurs adresses par rapport à rbp. Pour cela, j'utilise une référence *addr* qui part de -8 - l'adresse à laquelle je stocke la première variable - et à laquelle je soustrait l'espace utilisé à chaque nouvelle variable empilée. Au départ, je souhaitais simplement modifier la valeur du champ v_addr des variables utilisées mais en testant mon code, je me suis rendu compte que lorsque je voulais récupérer la valeur de champ, j'obtenais toujours la valeur de départ sans savoir pourquoi. J'ai alors décidé d'utiliser plutôt un dictionnaire *adresses* (avec le module Hashtbl) dans lequel je stockais les couples (v_id, v_addr) pour chaque variable. Toutefois, lorsque j'essayais d'accéder à l'adresse d'une variable, j'obtenais l'erreur Not_found.

J'ai heureusement trouvé ce qui posait problème : je rencontre les nouvelles variables dans les TEblock et dans le cas de filtrage des TEblock dans la fonction expr, j'utilise une fonction auxiliaire récursive *aux* qui parcourt la liste d'expressions du TEblock et crée un code différent selon que l'élément rencontré soit de la forme TEvars ou non. J'utilisais ensuite cette fonction dans *(aux vl al env) ++ (seq el env)* mais l'opérateur ++ de la librairie x86-64 évalue d'abord la partie la plus à droite du coup j'évaluais la partie où je recherche l'adresse de la variable dans le dictionnaire avant celle où je l'ajoute. J'ai alors pu résoudre le problème en évaluant d'abord *aux vl al env* que j'ai stocké dans une variable *code* avant de faire *code ++ (seq el env)*.

On a besoin de récupérer la valeur d'une variable lorsqu'on rencontre TEident. Pour cela, je récupère l'adresse de la variable dans le dictionnaire *adresses* puis je déplace la valeur stockée à l'adresse *addr(%rbp)*. J'ai eu du mal à écrire le code assembleur qui me donnait le résultat voulu mais j'ai finalement réussi. Un point important fût de compter le nombre de variables empilées dans chaque fonction afin de toutes les dépiler avant le *ret* pour éviter d'obtenir une erreur *segmentation fault* lors de l'exécution du code assembleur. En fait, comme le premier code exécuté est celui de la fonction qui se retrouve le plus haut dans la pile d'appel, la meilleure solution a consisté à compter le nombre de nouvelles variables créées à l'aide d'une référence *nombre_vars* que je remet à 0 lors des appels à function_.

Pointeurs

Mon idée de départ pour les pointeurs était de stocker la valeur vers laquelle on souhaitait avoir un pointeur sur la pile comme si c'était une variable en utilisant des id négatifs pour ne pas interférer avec des variables et lorsqu'on voudrait déréférencer le pointeur, on n'aurait qu'à déplacer la valeur stockée à cette adresse dans %rdi. Il faudrait également incrémenter le compteur de variables pour que les valeurs soient dépilées à la fin de la fonction. Toutefois, en essayant d'écrire un test, je me suis rendu compte que pour créer un pointeur, il fallait que ce soit un pointeur vers une variable. J'ai donc pu simplement réutiliser l'adresse de la variable pointée par le pointeur.

Néanmoins, j'ai rencontré des difficultés pour générer du code correct. En fait, je souhaitais faire un code pour le déréférencement de la forme $< \&a >$ $(\%rbp)$ mais le problème était que l'adresse était calculée par le code assembleur et placée dans %rdi, je ne pouvais donc pas l'écrire telle quelle dans le code que je générer. J'ai alors décidé de récupérer l'adresse de la tete de la pile puis d'effectuer une soustraction pour obtenir

l'adresse de la valeur pointé et enfin de déplacer le contenu stocké à cette adresse. J'ai vite remarqué qu'il ne fallait pas faire une soustraction mais une addition car les adresses sont négatives mais j'ai eu beaucoup de mal à écrire un code assembleur qui fonctionnait car je ne savais pas bien comment manipuler les adresses en assembleur. Après quelques tâtonnement, j'ai finalement réussi à écrire un code qui fonctionnait et j'ai pu l'adapter pour implémenter l'opérateur de déréférencement.

Appels de fonction

Pour gérer les appels de fonction, il m'a d'abord fallu écrire le code pour TEreturn. Pour cela, je génère le code associé à l'expression du TEreturn et je la place dans %rax. Néanmoins, j'ai eu des problèmes dans les cas où la fonction prend en entrée des paramètres. Pour gérer ces cas, j'ai décidé de stocker des variables correspondant à chaque paramètre de la fonction. Je devais prendre en compte le fait que dans le dictionnaire *adresses*, les clés sont les champs v_id des variables et par conséquent s'il y a plusieurs appels à une même fonction, les valeurs d'un paramètre dans chaque appel sont stockés au même endroit dans la pile.

Dans le cas des fonctions prenant des paramètres, mon compilateur ne fonctionne pas. En fait, le code assembleur produit est bien celui que je souhaite mais je ne comprends pas pourquoi lorsque je compile le code assembleur je n'obtiens pas le résultat attendu.

Autres remarques

- Il y a plusieurs parties du code fourni que je n'ai pas comprises (et par conséquent que je n'ai pas utilisées), c'est notamment le cas du type *env* - en particulier je n'avais aucune idée de ce que pouvait représenter le champ ofs_this -, du type mk_bool et de la fonction compile_bool.
- Dans le filtrage de la fonction expr, il y a plusieurs cas où je ne comprenais pas pourquoi distinguer différents cas d'un même constructeur de expr_desc et pourquoi en ignorer certains, notamment pour *TEassign* où le cas des listes de longueur plus grande que 1 ne pouvait pas être traité puisque le motif sur lequel on filtrait était TEassign(_,_), j'ai donc changé le motif afin de pouvoir traité ce cas.
- Je n'ai pas traité le cas des structures.