Rapport de projet
Projet COCass

Thomas MICHEL

17 janvier 2021

1 Parties réalisées

J'ai implémenté l'intégralité des fonctionnalités qui semblaient être attendues pour cette partie du projet. Tous mes tests donnent un code assembleur qui, une fois compilé, semble s'exécuter en accord avec la sémantique du `C--` . Certaines parties de la sémantique divergeaient du comportement obtenu par un même code dans le langage `C` et d'autres étaient laissées floues. Pour ces dernières, j'ai donc choisi de ne pas nécessairement me conformer au langage `C` (C'est par exemple le cas de la réallocation de variables qui est décrite plus bas).

2 Approche suivie lors du développement

J'ai réalisé un squelette de code explorant la totalité de l'AST en m'inspirant du code de `cprint.ml`. Chaque fonction créée renvoie le code approprié pour au type en entrée(par exemple `string_of_dec` renvoie le code relatif à la liste de déclaration en entrée). J'ai ensuite procédé de manière incrémentale en implémentant les différentes fonctionnalités une par une et en testant les nouvelles fonctionnalités au fur et à mesure. L'ordre suivi pour implémenter le compilateur a été le suivant :

- Les déclarations de fonctions
- Le code `C_RETURN` et les constantes
- Déclaration et utilisation de variables locales puis globales
- Opérations arithmétiques binaires
- Les comparaisons
- Les branchements logiques (`if ... else ...`)
- L'appel à d'autres fonctions (positionnement des arguments, alignement de la pile)
- Les chaînes de caractères
- Les tableaux
- Ajout d'éléments annexes (extérieurs à la sémantique) permettant de compiler un plus grand nombre de programmes

Tout au long du projet, j'ai utilisé le logiciel de gestion de version Git qui m'a permis de travailler sur la partie suivante en conservant la possibilité de corriger mes erreurs et de tester de nouvelles fonctionnalités sans craindre de perdre des données.

3 Choix effectués au cours de la réalisation du projet

3.1 Gestion des variables et des chaines de caractères

3.1.1 Les variables locales

Les variables locales sont toujours déclarées sur le dessus de la pile. En `C--` les déclarations de variables se produisent au début de `C_BLOCK`. Ces `C_BLOCK` marquent aussi la limite de la portée des variables. A partir de ce constat, j'ai choisi d'implémenter la gestion des variables comme une pile à l'aide d'une liste `OCaml`. Chaque variable est identifiée par un nombre unique qui correspond également à son ordre d'introduction dans le programme. A la fin d'un block, on replace le pointeur de sommet de pile à sa position avant d'entrer dans le block afin d'utiliser un minimum de mémoire. La structure du code d'un `C_BLOCK` est donc le suivant (en notant `n` le nombre de nouvelles variables déclarées dans le bloc) :

```
sub $(8 x n), %rsp
#code du block
add $(8 x n), %rsp
```

Cette implémentation peut être améliorée en ne déplaçant le pointeur de sommet de pile qu'une seule fois par fonction. Cela permettrait de diminuer le nombre d'instructions exécutées et donc d'augmenter la vitesse d'exécution. Elle a cependant l'avantage de limiter l'usage de la pile au minimum. Cela peut être important pour certains programmes notamment ceux utilisant la récursivité. Cette implémentation permet également de simplifier le code global et surtout la gestion de la portée des variables. Enfin, l'allocation en début de chaque `C_BLOCK` correspond mieux à ce qu'indique la sémantique du `C--` que la proposition d'amélioration précédente.

L'accès à une variable locale est directement déduit de son identifiant à l'adresse `$(id+1)(%rbp)`. Les variables globales sont stockées au sommet de la pile contenant les variables locales et sont repérées par un champ booléen `is_global`. L'accès se fait simplement grâce à `global_nomdelavariable(%rip)` lorsque la variable est détectée comme étant globale. L'ajout de variables à la liste des variables à la manière d'une pile permet de gérer automatiquement la portée des variables. Ainsi, si plusieurs variables ont le même nom, la recherche de l'identifiant s'arrête dès qu'une variable est trouvée dans la liste. Cette première occurrence correspond à la variable ayant la portée la plus réduite. Cette méthode permet de déclarer des variables locales ayant le même nom que des variables globales (on peut donc définir la variable globale `x` et la variable locale `global_x`). Cela permet également la redéfinition des variables. Cette possibilité n'est pas permise dans le langage `C` mais la sémantique de `C--` ne dit rien à ce sujet. La redéfinition est permise dans de nombreux langages et elle est directement permise par mon implémentation de compilateur `C--` .

3.1.2 Les variables globales

Les variables sont des entiers stockés sur 64 bits. Je déclare donc les variables globales à l'aide de

```
global_nomdelavariable: .long 0
```

Cette déclaration convient car chaque variable globale a un nom qui lui est unique. En C, seules les fonctions qui se trouvent après la définition d'une variable globale peuvent accéder à cette dernière. Ce comportement est bien adapté à la gestion des variables par mon implémentation du compilateur détaillée précédement. Cependant, la sémantique du `C--` indique que l'accès aux variables globales est indépendant de l'ordre de déclaration des variables et des fonctions. Il est ainsi possible d'accéder à une variable globale déclarée après dans le code `C--` . Cela a donc nécessité une adaptation de ma gestion des variables. Pour cela j'effectue simplement un premier parcours de la liste de déclarations passée en argument de la fonction `compile` afin de collecter l'ensemble des variables globales. Cette liste est ensuite passée en paramètre des fonctions de parcours de l'AST comme liste initiale de variables. On a bien la propriété sur la portée des variables décrite précédemment car les variables globales se trouvent alors toujours au bout de la liste des variables.

3.1.3 Les chaînes de caractères

Bien que le code assembleur produit pour accéder à des chaînes de caractères soit assez proche de celui produit pour les variables globales, leur gestion est assez différente. J'ai d'abord envisagé de faire en sorte que les différentes fonctions utilisées lors du parcours renvoient une liste contenant l'ensemble des chaînes de caractères utilisées. Afin de ne pas trop alourdir le code, j'ai décidé de plutôt utiliser une référence à une liste qui est remplie par effet de bord au fur et à mesure du parcours. Cette solution, pouvant paraître inélégante, épargne de larges ajouts de code diminuant la lisibilité finale du fichier `compile.ml`. Chaque chaine de caractères se voit attribuer un nom qui est unique grâce à la fonction `genlab` fournie avec le code initial (c'est également celle-ci qui permet de générer tous les labels destinés au code assembleur).

3.2 Evaluation des expressions

En accord avec la sémantique du langage `C--` , les expressions sont évaluées de droite à gauche. Les étapes intermédiaires sont sauvegardées sur la pile et dépilées au besoin. La gestion des expressions parenthésées s'adapte très bien à la structure de pile mais nécessite que le pointeur de sommet de pile indique toujours un espace inutilisé de la mémoire. C'est pourquoi le pointeur est toujours déplacé juste après la dernière variable locale. De cette manière, les valeurs intermédiaires peuvent être gérées facilement grâce aux instruction `push` et `pop`. Ainsi, l'évaluation de l'expression (a+b)*c donne un code de la forme :

```
...              // évaluation de l'expression c
push %rax        // sauvegarde de la valeur  de l'expression c
...              // évaluation de l'expression b
push %rax        // sauvegarde de la valeur de l'expression b
...              // évaluation de l'expression a
pop %rcx         // récupération la valeur de l'expression b
add %rcx, %rax   // addition de a et b
pop %rcx         // récupération la valeur de l'expression c
imul %rcx, %rax // multiplication de c et (a+b)
```

L'avantage de cette implémentation est qu'elle est simple et extensible. Il est possible de mettre une expression sous cette forme immédiatement grâce à la structure d'arbre de l'AST et cette implémentation est bien adaptée à des expressions de taille arbitrairement longue.

Cependant, le défaut que l'on peut trouver à cette manière de procéder est qu'elle est assez inefficace en termes de temps d'exécution. En effet, l'accès à la mémoire lors des appels à `push` et à `pop` est plus lent que l'utilisation directe de registres. Si l'optimisation de la durée d'exécution était l'un des objectifs de ce projet, il faudrait peut-être envisager d'évaluer les expressions en minimisant l'usage de la pile.

3.3 Gestion des arguments de fonctions

Dans un premier temps, j'ai implémenté l'appel des fonctions de façon à ce que les arguments soient évalués un par un, de droite à gauche puis directement placés dans leur position finale (celle qu'ils auraient au moment de l'instruction `call`). Cependant, cette méthode provoquait des problèmes de réécriture sur les arguments déjà placés durant l'évaluation. Par exemple, c'est le cas lors de la division sur 64 bits pour laquelle le diviseur placé dans `%rax` est étendu sur 128 bits à l'aide du registre `%rdx` (qui contient peut-être un argument déjà calculé). Pour pallier à ce problème, j'ai décidé de d'abord stocker tous les résultats d'évaluation des arguments sur la pile. Les arguments sont ensuite déplacés dans le registre approprié. Le schéma de l'appel à une fonction est donc le suivant pour 7 arguments :

```
...              //Evaluation de l'argument 7
push %rax        //rax contient l'argument 7
...
...              //Evaluation de l'argument 1
push %rax        //rax contient l'argument 1
pop %rdi         //Place l'argument 1 dans le registre qui lui est
   attribué
...
pop %r9          //Place l'argument 6 dans le registre qui lui est
   attribué
xor %rax, %rax   //Met le registre rax à 0
call function_name
```

Le registre `rax` doit stocker le nombre de registres vectoriels utilisés par les arguments d'une fonction à nombre variable d'arguments (comme `printf` par exemple). Le `C--` ne nécessite pas de tels registres, c'est pourquoi rax est mis à 0 avant l'appel à une fonction.

La récupération des arguments par la fonction appelée s'effectue de la manière attendue en stockant sur la pile les valeurs des registres et de la pile dans l'ordre convenu (d'abord les 6 registres puis les valeurs stockées sur la pile dans l'ordre inverse). Ces arguments sont ensuite gérés de la même manière que toutes les autres variables locales sur la pile.

4 Difficultés rencontrées

4.1 Alignement de la pile

La plupart des fonctions de la bibliothèque standard requiert que le sommet de la pile soit à une adresse multiple de 16 (alignement de la pile). Dans le cas contraire, l'appel à l'une de ces fonctions (c'est par exemple le cas avec `printf`) peut aboutir à une erreur de segmentation lors de l'exécution du programme. La raison est que l'utilisation de certaines instructions de type SIMD (Single Instruction, Multiple Data) nécessite un tel alignement. Bien que de telles instructions ne soit pas explicitement écrites par mon implémentation du compilateur, elles sont utilisées par des fonctions externes susceptibles d'être appelées par le programme compilé. L'alignement global ne pouvant être connu, la norme prescrit que chaque fonction rétablisse l'alignement avant d'appeler une autre fonction. L'alignement est ainsi conservé d'appel en appel et peut donc être géré plus simplement lorsqu'une instruction SIMD est utilisée.

Cette règle n'est cependant pas universelle. Il semble en effet que celle-ci soit dépendante du processeur de l'ordinateur et du compilateur utilisé. En effet, un tel alignement n'est pas nécessaire sur mon ordinateur personnel. Mais suite à l'annonce des machines de références (celles de la salle machine de l'ENS Paris-Saclay), je me suis rendu compte que le code renvoyé par mon compilateur produisait des erreurs de segmentation. L'origine de telles erreurs a été difficile à trouver car je n'avais aucune indication sur la partie de mon implémentation faisant défaut. J'ai cependant pu retracer l'erreur à l'utilisation de l'instruction `call printf` et finalement, après recherche, j'ai découvert l'existence de la règle d'alignement décrite plus haut. Cette difficulté ne faisait l'objet d'aucune indication et était difficile à déceler. En effet, certains programmes ne provoquent pas d'erreur de segmentation car leur pile est naturellement alignée du fait de leur nombre de variable locale.

Pour résoudre ce problème, il faut parvenir à connaître l'alignement de la pile au moment de l'utilisation de `call`. Toutes les valeurs manipulées et stockées sur la pile par le programme compilé étant sur 64 bits (8 octets), seuls deux cas sont possibles : soit la pile est alignée sur un multiple de 16, soit le décalage avec l'état aligné est de 8 octets. Dans le second cas, il suffit donc de décaler le pointeur de sommet de pile de 8 (on lui soustrait 8 pour ne pas écraser les valeurs sur la pile). Dans mon implémentation du compilateur, le pointeur vers le sommet de la pile (registre `rsp`) n'est modifié qu'à quatre occasions :

— allocation d'espace pour une variable locale

— passage d'un argument à une fonction lorsqu'il y a plus de 6 arguments
— stockage d'un résultat intermédiaire lors de l'évaluation d'une expression
— stockage temporaire des arguments d'une fonction sur la pile

Le nombre actuel de variables locales ainsi que le nombre d'arguments de la fonction à appeler sont connus. De plus, on ajoute des arguments sur la pile seulement lorsque la fonction a strictement plus de 6 arguments. L'implémentation de la rectification de l'alignement est donc la suivante :

```
let is_aligned = (nb_args<7 && (var_cnt mod 2 = 0))
                  ||(nb_args>=7 && ((var_cnt + nb_args) mod 2 = 0))
                     in
(* code that correct the alignment *)
let correct_begin = if is_aligned then "" else "    sub $8, %rsp\n"
and correct_end = if is_aligned then "" else "    add $8, %rsp\n" in
```

`correct_begin` et `correct_end` peuvent alors directement être ajoutés au début et à la fin du bloc de code traduisant un appel à une fonction. Le décalage de la pile issu du stockage temporaire de valeurs est traité par l'incrémentation du compteur `var_count`. Celui-ci représente alors le décalage effectif de la pile par rapport à sa position au début de la fonction courante (c'est à dire celle contenue dans le registre rbp)

4.2 Le langage assembleur

L'utilisation du langage assembleur ne m'a pas posé de difficultés particulières car celui-ci a été largement expliqué lors des cours de Programmation 1. Cependant, l'utilisation pratique de ce langage nécessite la connaissance d'un nombre important de mots-clefs et de normes. Les renseignements à leur propos ne sont pas faciles d'accès lorsque l'on ne sait pas exactement ce que l'on recherche. Il pourrait être intéressant d'en avoir un aperçu (même superficiel) en début de projet afin d'orienter les élèves dans une direction qui leur éviterait des heures de recherches infructueuses.

Au cours du projet j'ai été amené à lire une certaine quantité de code assembleur afin de trouver des erreurs. J'ai pu trouver des indices et des combinaisons de mots-clefs qui ont permis à mon code de fonctionner. Cette recherche fastidieuse et souvent peu instructive ne me paraît pas être l'objectif initial de ce projet. Le mise à disposition d'une page "foire aux questions" me semble avoir été une très bonne idée pour empêcher les élèves de rester bloqués su des détails parfois obscurs de la programmation assembleur.

5 Conclusion

La partie de ce projet dont je suis le plus fier est peut-être la gestion des variables et de leur portée. La gestion des variables locales et globales est unifiée par une seule liste qui modélise également leur portée. L'implémentation de la portée des variables s'intègre également très bien au style de programmation fonctionnelle que je me suis efforcé d'adopter tout au long de ce projet.

Je suis également assez fier de l'implémentation générale de mon compilateur. Le code s'inscrit dans un style de programmation fonctionnelle malgré certains compromis en faveur de la clarté du code. Les chaînes de caractères ainsi que la liste des fonctions auraient pu être implémentées dans un style fonctionnel en ajoutant des arguments et des valeurs de retour supplémentaires aux différentes fonctions constituant le parcours de l'AST. L'effet de ce style se retrouve également dans la relative concision du code du fichier `compile.ml`.

Je pense que l'objectif a été atteint : implémentation en OCaml d'un compilateur `C--` respectant la sémantique. La réalisation d'un tel projet est très gratifiante et j'aurai aimé avoir l'occasion de réaliser également la seconde partie (gestion des exceptions), voire d'aller plus loin en réalisant un compilateur optimisant. J'espère pouvoir réaliser ces dernières étapes si j'en ai l'occasion dans le futur.