Projet Compilateur C–

Izumi Watanabe

Décembre 2019

1 Résumé

Le fichier `compile.ml` contient une fonction `compile out decl_list`, qui prend en argument un `out_channel` (où l'on va écrire le code assembleur) et une liste de déclarations (de type `Cparse.var_declaration list`). Elle écrit le code assembleur correspondant à `decl_list` dans le channel `out`.

Pour cela, le code se divise en trois parties :
- La définition d'un type enregistrement `environment`, qui va servir à stocker les données nécessaires à la compilation tout au long de cette dernière
- Une fonction `first_decl_list` qui fait un premier parcours de `decl_list`, pour repérer en avance les chaînes de caractères et les noms des fonctions
- Les fonctions principales qui s'appellent récursivement pour parcourir l'arbre `decl_list` et compiler le code.

2 Le type `environment`

Le type `environment` est composé de 8 enregistrements, tous mutables à l'exception de `strings` (qui est une table de hachage) :
- `stack_size` qui contient la taille du stack frame ;
- `local_var` qui permet d'associer à chaque variable locale, sa position par rapport à la base de la pile ;
- `strings` qui associe à chaque chaîne de caractères, le numéro du label qui est utilisé pour la stocker ;
- `functions` qui stocke les noms des fonctions définies dans le code ;
- `string_count`, `if_count`, `while_count`, `cmp_count` qui comptent respectivement le nombre de chaînes de caractères, de blocs `CIF` ou `EIF`, de blocs `CWHILE` et d'opérations de comparaison.

Pourquoi rendre les enregistrements mutables ? C'est surtout pour rendre le code plus simple, car on n'aura pas à créer une nouvelle variable du style `new_env` à chaque fois que l'on modifie l'environnement, et on n'aura pas à passer `env` en argument de chaque fonction. De plus on sera très souvent amené à modifier l'environnement, car il évolue constamment au fur et à mesure de la compilation. Enfin, il y a peu de risques de mauvaise manipulation car il n'y a qu'une seule création d'environnement dans la fonction principale `compile`.

Cette déclaration d'environnement est suivie de la définition de plusieurs procédures qui modifient l'environnement. Elles seules seront utilisées dans la fonction `compile`, dedans on ne manipulera plus les composants d'un environnement individuellement.

Voici des précisions sur le stockage des variables dans `local_var`. On procède ainsi :
— Lorsque l'on rentre dans un nouveau `CBLOCK` (portion de code délimitée par `{` et `}`), on empile un `None` dans `local_var` ;
— Lorsque l'on doit déclarer une variable locale `x`, on augmente la taille `stack_size` de la pile, et on empile le couple `Some (x, stack_size)` dans `local_var` ;
— Pour chercher la position une variable locale `x`, on cherche l'occurrence de `x` qui a été stockée en dernier (donc celle que l'on trouve en premier dans le parcours de `local_var` ;
— Enfin, lorsque l'on quitte un `CBLOCK`, on supprime toutes les variables de `local_var` jusqu'à tomber sur un `None`.

Ainsi, pour le code suivant :

```
int main () {
  int a;
  {
    int b;
    int a;   //(1)
  }          //(2)
```

L'état de `local_var` lorsque l'on a compilé jusqu'au repère `(1)` est :
`[Some ("a", 3); Some ("b", 2); None; Some ("a", 1); None]` ;
et celui lorsque l'on a compilé jusqu'au repère `(2)` est :
`[Some ("a", 1); None]`.

(Remarque : si l'on gère bien la pile, on a juste besoin de connaître ***`var_count`*** *et de regarder la position de* ***`x`*** *dans* ***`local_var`*** *pour retrouver l'adresse de* ***`x`****. L'implémentation que j'ai utilisée a été définie à un moment où je gérais la pile autrement, et j'ai préféré ne pas prendre le risque de la changer.)*

3 La fonction principale `compile`

Elle comporte tout d'abord une fonction `first_decl_list` (avec ses 3 fonctions auxiliaires) qui réalise un premier parcours pour stocker les chaînes de caractères, et sauvegarder le nom des fonctions définies dans le code.

On a ensuite une fonction `compile_decl_list` qui réalise le parcours principal, en s'aidant des fonctions auxiliaires suivantes :

— `compile_mon_op`, `compile_bin_op`, `compile_cmp_op`, `compile_expr`, `compile_decl`, `compile_code`, dont les noms sont assez explicites, et qui traitent chacun des noeuds de l'arbre retourné par le parseur en s'appelant récursivement ;
— `compile_call`, `pull_args` qui sont des sous-fonctions des fonctions définies plus haut (pour faciliter la lecture) ;
— `compile_two_expr` et `compile_bool` qui font de légères optimisations, dans le but de rendre le code assembleur plus lisible.

Lorsqu'elles évaluent une commande, toutes les fonctions ci-dessus placent le résultat dans le registre `%rax`, pour que la fonction qui les a appelée sache où retrouver le résultat.

Les précisions nécessaires pour comprendre chacune de ces fonctions sont données ci-dessous.

`compile_mon_op`

Lorsque l'expression est de la forme `a[i]++`, on se sert du fait que la compilation de `a[i]` laisse l'adresse de `a[i]` dans le registre `%rdx` à la sortie. On peut donc directement agir sur cette adresse.

`compile_two_expr`

Elle évalue dans l'ordre l'expression `e2`, puis `e1`, et les place respectivement dans `%rcx` et `%rax` en vue d'une utilisation par `compile_bin_op` ou `compile_cmp_op`. Elle n'utilise pas la pile lorsque `e1` peut être évaluée sans utiliser d'autres registres que `%rax` : c'est-à-dire lorsque `e1` est de type `VAR`, `CST` ou `STRING`.

`compile_cmp_op`

Elle prend parmi ses arguments une chaîne de caractères `jump_dest` qui est le label de l'adresse où l'on doit sauter lorsque la comparaison s'évalue à Faux. Elle écrit donc l'opération contraire de celle qui est demandée.

`compile_bool`

Elle évalue un booléen `e`, en vue d'une utilisation dans une instruction `EIF`, `CIF` ou `CWHILE`. Pour cela on évalue directement `e` si elle est de type `CMP`, pour s'éviter des branchements conditionnels inutiles.

`compile_call`

On commence par décaler le pointeur de pile `%rsp` pour qu'il soit aligné sur 16 octets au moment du `call`. Pour cela, il faut qu'un nombre pair de cases soient occupées sur le stack frame à ce moment-là. On doit donc considérer la parité de la taille de la pile, et celle du nombre d'arguments à empiler.

On fait attention à évaluer tous les arguments et à les stocker temporairement sur la pile, avant de les envoyer dans les registres. En effet, par exemple lors l'évaluation de `f(g(), x)`, l'évaluation de `g()` peut modifier la valeur de `%rsi`.

Si la fonction à appeler n'est pas définie dans le code, on ajoute le suffixe `@PLT` au nom de la fonction.

Enfin, si la fonction retourne un entier 32 bits (c'est le cas pour les fonctions de la librairie C sauf celles mentionnées dans `quad_functions`), on convertit la sortie avec `cltq`.

`compile_decl`

Lorsque l'on entre dans la déclaration d'une fonction, on efface l'ancienne liste de variables locales et on déclare les arguments dans l'environnement. Du côté du code assembleur, on réalise les tâches suivantes :

— Empiler l'ancienne valeur de `%rbp` ;
— Stocker le pointeur de pile dans `%rbp` ;
— Réserver de l'espace dans la mémoire pour les arguments de la fonction.

Les opérations inverses seront réalisées lors d'un `CRETURN` grâce à la commande assembleur `leave`.

Lorsque l'on a fini de compiler une déclaration de fonction, on ajoute les commandes `leave` et `ret` pour s'assurer que l'on sort bien de la fonction à l'exécution. Ce n'est pas demandé dans la sémantique mais cela permet d'éviter des erreurs lorsque l'on veut coder des procédures.