Semaine 2 (Compilateur) - Projet IFT3150

Philippe Caron

29 août 2017

1 Objectif

La deuxième semaine était principalement consacrée à l'écriture d'un compilateur lua-IR dont le jeu d'instrution était à adapté aux particularité du langage lua tout en se rapprochant au maximum de l'assembleur. En ce sens, les principaux défis étaient :

- Trouver un moyen de d'évaluer les destination PUIS les valeurs
- Déterminer quand les variables étaient en position de retourner plusieurs éléments et quand elles étaient «tronquées»
- Bien associer les appels de fonctions
- Détecter l'appel terminal et la fermeture

2 Le compilateur

Le compilateur fonctionne de manière très analogue au préprocesseur. En fait, les deux auraient facilement pu être combinés, mais la quantité de code rendrait vraiment ce fichier ingérable. L'avantage qu'ils soient séparés est que dans le compilateur, la fonction `nextexpr` est vraiment très simple.

2.0.1 nextexpr

Maintenant, toute les expression sont parenthèsées et séparées par au moins un caractère «blanc». La fonction `nextexpr` fait essentiellement juste compter les séparateur et retourne la chaîne dès qu'elle rencontre un espace et que l'équilibre de parenthèse est atteint.

2.0.2 evaluate et translate

Les fonctions `evaluate` et `translate` travaillent de concert pour produire le code IR. Le but de `evaluate` est tout d'abord de séparer les expressions en 3 catégories principales :

- **Une expression locale (set)** : Une expression précédée par le mot réservé "local" qui aura une influence sur la pile.
- **Une expression globale/accès (chg)** : Une expression précédée d'aucun mot réservé qui n'aura pas d'impact sur la pile.
- **Une expression retournée (ret)** : Une expression précédée par le mot réservé "return" qui conduira à la sortie d'une fonction.

Avec le mécanisme actuel d'assignation, les deux seconde expressions sont en fait une extension de la première.

set ... stack

Le mode **set** fait tous simplement évaluer successivement toute les expression passée après le «=» et les empiles sur la pile d'exécution. Si il n'y a pas d'expression alors il ne fait que réserver l'espace sur la pile.

chg ... place ... stack

La nature de ce mode vient du fait qu'en Lua les variables destinataires ne doivent pas être changée tant que toute la séquence de valeurs n'est pas évaluée. La manière la plus logique était donc d'empiler toutes les destinations, puis toute les valeurs et simplement les remplir par correspondance. En rétrospective, si seulement les valeurs avaient été sur la piles, puis distribuées par la suite à leurs destinataires, il n'y aurait pas eu d'adresse qui traîne sur la pile, et le masque de transfert n'aurait probablement pas été nécessaire.

ret ... stack

Comme set, ret empile les variables sur le stack, mais aulieu de réserver l'espace occupé par celui-ci, il fait plutôt pointer %rbx dessus et quitte la fonction.

translate

Cette fonction va traduire chacune des expression en code IR, la tâche est plutôt facile étant donné que le parenthèsage est bon. Pour de plus amples détails consulter le jeu d'instruction IR dans le manuel.

2.0.3 Fonctions d'environnement et compile

La fonction `compile` est la fonction-mère du fichier, au même titre que dans le préprocesseur elle sert simplement à diriger l'exécution dans la bonne direction. Les fonctions d'environnement ont également un rôle analogue.

3 Fermetures

Comme pour le préprocesseur, le compilateur tente de produire du code en repassant le moins possible sur lui-même. Pour ce qui est des fermetures, cela signifie qu'une variable peut «naître» comme une variable locale, puis être «enfermée» par la suite dans l'environnement de fermeture.

3.1 Processus de détection de fermeture

Lorsque que le compilateur entre dans une fonction, il possède une représentation interne des niveau d'accès de chaque variable. S'il détecte qu'une fonction tente de sortir de son niveau d'accès, alors il enferme la variable au niveau où se fait l'accès. Puisque l'environnement de fermeture a préséance sur l'environnement local, une fois qu'une variable est enfermée, elle masque son équivalent local et toutes les autres références à cette variable seront faites à l'environnement de fermeture.

Une fois sorti de la fonction, la variable demeure enfermée (puisqu'elle l'a été au niveau accédé et non au niveau de la fonction). Maintenant tous les niveau inférieurs ou égal qui voudront référer à cette variable obtiendront la fermeture.

Si une autre fonction crée une fermeture depuis la même variable, il n'y a pas de conflit en raison de la structure en arbre de l'environnement de fermeture. Voir le manuel pour plus de détails.

4 Appel terminal

Lorsque que le compilateur évalue une fonction, il tente d'abord de détecter son nom. S'il le trouve il le passe en paramètre à la fonction d'évaluation de fonction. À mesure que la fonction est évaluée, le nom est mis à NIL l'expression n'est pas en position terminale, et conservé sinon. Ainsi la propagation terminale est effectuée. Lorsque l'on tombre sur un "return", le nom de la fonction est comparé au nom en paramètre, si celui-ci est nil, il n'y a aucune chance que ce soit vrai, sinon, la comparaison détermine s'il sagit vraiment d'un appel terminal. L'appelle terminal ne fonctionne pas sur les fonction à arguments variables.

5 Particularités

5.1 Détection des troncation

Lorsque le dernier élément d'une séquence est parenthèsée, le compilateur ajoute la directive IR `trunc` qui signal au compilateur-IR d'ignorer le stack retourné, à l'inverse, si cette valeur n'est pas parenthèsée, il envoir la directive `struct` qui signale au compilateur de tenter de développer la structure pour obtenir les autres valeurs de retour.

5.2 Format d'appel

Le langage Lua permet l'appel de string et de tables sans parenthèses, combinés avec les indexation, cela signifie que le compilateur doit toujours regarder «dans le futur» afin de s'assurer que le prochain élément n'est pas un paramètre potentiel. En raison de la signifiation de la parenthèse, le préprocesseur ne peut pas non-plus envelopper le tout entre parenthèse, il faut donc à chaque fois toujours vérifier et ajouter les éléments tant que ceux-ci peuvent être des paramètres.

6 Librairie

Le compilateur gère l'inclusion automatique de librairie. En compilant les librairie il crée un fichier dans lequel il note toutes les variables globables disponibles et dans quelle librairie les trouver. En compilant un fichier normal, il consulte la liste et détermine des librairie à être inclue dans le programme.

7 Conclusion

Le compilateur s'est avérer très fiable, et le code qu'il produit est suffisemment proche du code assembleur pour que le compilateur-IR aie la tâche relativement simple. Sa capacité à choisir quel librairie inclue est également très intéressante, car dans le cas d'une librairie statique comme ici, si on décide de toujours inclure toute les librairies disponibles les fichiers exécutable risque de devnir énormes inutilement. Tout comme pour le préprocesseur, le compilateur souffre d'un codage un peu lourd, et désagréable à lire, mais ceci est un peu un effet secondaire de Lua. En effet même si ce langage possède beaucoup d'exceptions et de formes spéciales, il souffre malgré tout d'une expressivité relativement pauvre.