Un compilateur optimisant pour Petit Scala
Rapport

Nathanaël Courant

17 janvier 2016

1 Utilisation

Les options reconnues par le compilateur sont les suivantes :
— `--parse-only` : Arrêt du compilateur après l'analyse lexicale et syntaxique du fichier source,
— `--type-only` : Arrêt du compilateur après le typage du fichier source,
— `-G` : Pour le déboguage du compilateur, affiche des informations de debug ; de plus, ne rattrape pas une erreur interne mais la laisse passer (pour obtenir un backtrace en cas d'erreur). *Attention* : Cette option peut générer une sortie conséquente sur de gros fichiers.
— `--undefined-null-deref` : Si cette option est passée au compilateur, le résultat de l'appel d'une fonction ou de l'accès à un champ du pointeur `null` est considéré comme indéfini ; cela permet au compilateur de faire certaines optimisations.

2 Choix techniques

2.1 Langages intermédiaires

Les langages intermédiaires utilisés sont très proches de ceux du cours (et de CompCert), étant les suivants :
— `Is` : Sélection d'instructions, *constant folding*, identification d'appels de méthodes qui peuvent être remplacés par des appels explicites (ceci est possible car on compile le programme entier, et requiert l'option `--undefined-null-deref`),
— `Rtl` : Graphe de flot de contrôle,
— `Ertl` : Explicitation des conventions d'appel,
— `Ltl` : Allocation de registres.
On arrive enfin à la production de code assembleur, avec linéarisation du code `Ltl`.

2.2 Sélection d'instructions

On utilise le constructeur `Ebinary3` jusqu'à la phase d'allocation de registres, pour représenter les opérations binaires commutatives comme si elles provenaient de code à trois adresses. En effet, on peut ensuite ajouter deux arêtes de préférence, pour permettre à l'allocation de registres de tenter de minimiser le nombre d'opérations `mov` inutiles qui proviennent du code à deux adresses. Cependant, cela n'est pas suffisant pour obtenir le résultat souhaité sur `fact_rec.scala`, mais je conjecture qu'une bonne heuristique pour choisir quels sommets il faut fusionner permettrait de résoudre ce problème.

2.3 Le *garbage collector* (pas encore implémenté)

Le *garbage collector* n'est pas encore implémenté, cependant, voici des idées pour celui-ci.

Les problèmes rencontrés lorsque l'on veut ajouter un GC sont les suivants :
— Il faut examiner le contenu des registres et de la pile pour trouver les racines,
— Il faut pouvoir discerner un entier d'un pointeur.

L'idée pour cela est la suivante : on va positionner un label dans le code après chaque appel de fonction, avec des données dans le `.data` associant ce label à la configuration de la *stack frame* de la fonction à ce moment précis, et précisant ce que contient la pile et les registres. Plus précisément, chaque registre ou emplacement de pile peut soit contenir un registre de la fonction qui a appelé celle-ci, soit avoir un type (entier ou pointeur) connu à la compilation. Il reste un problème : les fonctions et classes polymorphes. En effet, pour celles-ci, on ne peut pas connaître à l'avance le type de certains registres ou emplacements de pile. Il y a pour cela deux solutions : la première consiste en remplacer chaque classe et fonction polymorphe par plusieurs versions, une pour chaque configuration possible de type des arguments (ce serait quelque chose s'approchant donc de la défonctorisation). Cependant, elle a un défaut : elle peut causer une augmentation exponentielle de la taille du code. Elle peut toutefois être plus efficace que la deuxième solution pour un faible nombre de paramètres de types ($\leqslant 2$ à mon avis). La deuxième solution, plus universelle, est de consacrer un registre (ou emplacement de pile, voire plus si la fonction a plus de 64 paramètres de type...) pour donner les arguments de type de la fonction, et de manière similaire, consacrer un mot dans les objets polymorphes pour préciser leur type. Pour appeller une fonction polymorphe, on lui passe donc le type de ses arguments, et de l'objet `this` dans un registre. Il reste encore une subtilité : *a priori*, un objet de type `Any` peut contenir soit un entier, soit un pointeur... Cependant, comme Petit Scala n'autorise pas de *downcast*, un tel objet ne pourra jamais être utilisé pour son contenu par la suite, et il n'est pas non plus possible de tester son égalité avec un autre objet (`eq` demandant des objets de type plus petit que `AnyRef`). Ainsi, il est correct de collecter ces objets, si ils ne sont utilisés nulle part ailleurs.

Actuellement, l'implémentation du GC se limite à transformer les informations de typage en informations de typage pour le GC (i.e, si une expression est un entier, un pointeur où un paramètre de type) dans `Is`. Il faudrait encore continuer à passer cette information dans les langages suivants, produire les descripteurs de *stack frames*, et écrire le GC lui-même.

3 Problèmes connus

— L'allocation de registres peut prendre longtemps sur de grosses entrées : ainsi, le compilateur met entre 10 et 15s à s'éxécuter sur `quine.scala`. En effet, l'algorithme d'allocation de registres est implémenté de manière assez naïve, ce qui cause de très nombreuses vérifications du critère de George.

— Une suite d'instructions comme `mov #1 #2`, `mov #1 #3`, puis une utilisation des registres `#2` et `#3` cause les registres `#2` et `#3` comme étant déclarés comme interférant, et par conséquent les empêche d'obtenir le même registre.