RAPPORT PROJET INF564

11 Mars 2022

Manas UTESHEV
Vladimir OGORODNIKOV

1 Difficultés rencontrées et leurs solutions

Lors de la réalisation de l'étape *LTL* nous avons été contraints de redéfinir la méthode *hashCode()* dans une classe *Operand* afin de pouvoir utiliser les structures *HashMap* et *HashSet* en Java. De plus, lors de la réalisation de l'étape final *x86-64 assembly* nous avons constaté que comme les instructions logiques, telles que *sete, setne, etc.*, nécessitent un registre sur 8 bits et pas sur 64 bits. Pour résoudre ce problème nous avons enforcé l'utilisation d'une paire de registres *%rax* et *%al* au niveau de ERTL. Nous avons aussi modifié la méthode *set(Register r, boolean b)* dans une classe Java *Machine*, car sa version initiale ne faisait pas différence entre les pseudoregistres et les registres physiques. En outre, nous étions aussi obligés de modifier les classes Java de l'arbre de typage et l'arbre de syntaxe afin de propager les champs *Loc location*, qui sont exploités pour produire les messages d'erreurs explicites.

Notre code passe tous les tests fournis lors des TDs. Ils nous ont bien aidé à trouver nos erreurs et ainsi, à les corriger d'une manière efficace.

1.1 Implémentation d'une classe java CompilerError

La classe abstraite Java *CompilerError*, que nous avons créé dans notre projet, possède trois sous-classes *SyntaxError*, *LexicalError* et *TypeError*. Elles étaient utilisées afin de bien distinguer nos erreurs produites au niveau de la compilation et ainsi, les afficher sous forme de messages explicites.

2 Extensions réalisées

2.1 Instructions break et continue

Dans notre projet nous avons effectué une extension, qui nous a permis de traiter les instructions *break* et *continue*. Dans le cas où on rencontre ces instructions en dehors d'une boucle, notre code va léver une erreur *SyntaxError*.

2.2 Boucle for

Nous avons aussi implémenté le traitement d'une boucle *for(A ; B ; C)*. Dans notre code vous allez voir que toutes les parties de la tête d'une boucle *for*, qui sont A, B et C, sont obligatoires et ne peuvent pas être vides. De plus, les instructions *break* et *continue* fonctionnent bien à l'intérieur d'une boucle *for*.

2.3 Opérateur binaire %

L'opérateur binaire %, qui calcule le reste de la division, a été aussi implémenté dans notre code. Comme il est assez proche de l'opérateur de division, on a utilisé la même technique que dans la division, i.e. nous avons forcé l'utilisation des registres *%rax* et *%rdx* pour l'opérateur *%*.

2.4 Optimisation de compilation

Afin de optimiser le code produit par notre compilateur nous avons ajouté l'évaluation de constantes entières pendant la compilation. Pour cela nous avons implémenté une classe Java *ConstEvalOptimizer*, qui prend un arbre de typage et produit ensuite un autre arbre de typage, qui sera passé aux étapes suivantes.