Compilation – TP 12 : Optimisations

Université Paris Diderot – Master 1

(2014-2015)

Cette feuille de TP vous donne les étapes à suivre pour implémenter les optimisations vues en cours sur le langage RETROLIX. Il s'agit de compléter les modules `RetrolixConstantPropagation`, `RetrolixCSE` et `RetrolixDeadcode`.

Le code source correspondant à ces travaux pratiques se trouve sur le GIT, dont on rappelle l'URL :

`http://moule.informatique.univ-paris-diderot.fr:8080/Yann/compilation-m1`

On rappelle que vous devez faire des *commits* réguliers (à chaque modification de votre code) pour que nous puissions suivre votre avancement.

1 Élimination du code mort

Le programmeur et, plus vraisemblablement, le compilateur peuvent introduire des instructions qui ne seront jamais exécutées ou dont le calcul ne contribue en rien au résultat final L'optimisation appelée "élimination du code mort" sert à supprimer de telles instructions.

Exercice 1 (Élimination du code mort)

1. *Dans quelle mesure l'analyse de vivacité permet-elle de déterminer si une affectation d'une rvalue est inutile ?*
2. *Quand une instruction n'est pas atteignable depuis le début d'un bloc, alors il s'agit aussi d'une instruction inutile. Complétez la fonction* `RetrolixDeadCode.unreachable` *qui permet de calculer l'ensemble des étiquettes des instructions que l'on ne peut pas atteindre depuis l'entrée du bloc d'instructions.*
3. *Complétez la fonction* `RetrolixDeadCode.eliminate` *pour supprimer les instructions inutiles.*

□

2 Propagation des constantes

On peut définir une analyse statique qui calcule pour chaque point du programme l'ensemble des définitions qui l'atteignent. Quand une définition de la forme "$x \leftarrow c$" (où x est une variable et c est une constante) est la seule définition de x qui atteint une instruction qui utilise x alors on peut substituer la valeur de x par la constante c.

Remarquez qu'une fois cette transformation effectuée, la définition de x peut devenir du code mort qui peut être supprimé grâce à l'optimisation précédente.

Remarquez aussi qu'une fois qu'une constante est propagée, il est parfois possible d'effectuer des calculs au moment de la compilation (comme par exemple déterminer la valeur d'un test). Ce calcul peut lui aussi faire apparaître du code mort.

Exercice 2 (Propagation des constantes)

1. *Complétez la fonction* `RetrolixConstantPropagation.reachable_definitions` *qui détermine pour chaque point du programme les étiquettes des définitions qui atteignent ce point. On utilisera les définitions suivantes :*

$$\begin{aligned} out[n] &= gen[n] \cup (in[n] \setminus kill[n]) \\ in[n] &= \bigcup_{p \in pred[n]} out[p] \end{aligned}$$

où $kill[n]$ associe à chaque nœud n l'ensemble des définitions que l'instruction en n rend inaccessibles.

2. *Complétez la fonction* `RetrolixConstantPropagation.simplify` *qui utilise le résultat de la fonction précédente pour réécrire le programme en propageant les constantes (et en simplifiant le programme ainsi obtenu).*

□

3 Élimination des sous-expressions communes

On peut définir une analyse statique qui calcule pour chaque point du programme l'ensemble des valeurs symboliques qui l'atteignent. Quand on croise une instruction de la forme "$x \leftarrow e$" (où x est une variable et e est une expression arithmétique) et que le calcul de e est déjà contenu dans une *lvalue* y alors on peut remplacer l'instruction par "$x \leftarrow y$".

Exercice 3 (Élimination des sous-expressions communes)

1. *Complétez la fonction* `RetrolixCSE.value_numbering` *qui détermine pour chaque point du programme l'ensemble des valeurs symboliques accessibles en ce point. On utilisera l'algorithme* `ValueNumbering` *vu en cours.*
2. *Complétez la fonction* `RetrolixCSE.simplify` *qui utilise la fonction précédente pour factoriser les calculs redondants dans le programme* RETROLIX *pris en entrée.*

□