Static Single Assignment, optimisations de compilation

bogdan.pasca@ens-lyon.fr

16 nov 2010

Note : Thanks to Boissinot Benoit for his contribution to this subject

Commençons toujours par rappeler quelques définitions :

- Un programme (un CFG) est sous forme *Single Static Assignment* (*SSA*) si chaque variable n'a *textuellement* (statiquement) qu'une seule définition. *Dynamiquement*, c'est-à-dire lors de l'exécution du programme, une même variable peut avoir de multiples affectations.
- SSA est en fait une *représentation intermédiaire*.
- Il existe plusieurs formes de SSA :
  - *strict SSA* : toute variable utilisée est nécessairement définie ;
  - *pruned SSA* : forme simplifiée et optimisée par un calcul de vivacité.
- On appelle *passage en SSA* la transformation de la représentation normale d'un programme à la représentation sous forme SSA. Cette opération renomme des variables et introduit les fonctions ϕ dont l'utilité sera montrée plus loin.
- *Sortir de SSA* consiste à transformer un programme sous SSA en programme normal. Les fonctions ϕ doivent alors être remplacées par des instruction de copie.
- Les fonctions ϕ servent à *"choisir"* la bonne version d'une variable renommée *dépendante du flot de contrôle du programme*. Celles-ci seront abordées plus en détail pendant le TD.

1 Passage en SSA

Q 1.1 Passez manuellement le programme suivant sous forme SSA. Placez des fonctions ϕ aux endroits nécessaires et comprenez leur sens.

```
a = 3;
b = 2;
c = 0;
while (a < 15) {
  if (b<6) {
    a = a * 2 ;
    b = b + 1 ;
  } else {
    a = a + 1 ;
    b = b * 2 ;
  }
  c = c + a
}
return a + (b + c);
```

Q 1.2 Etablissez un lien entre les frontières de dominance des variables et les endroits pour le placement des fonctions ϕ. Construisez-en un algorithme.

Q 1.3 Appliquez votre algorithme sur le programme ci-dessus.

Q 1.4 Considérez maintenant aussi une condition sur la vivacité pour le placement des fonctions ϕ pour obtenir du pruned SSA.

2 Sortie de SSA

Les fonctions ϕ ne sont évidemment pas des instructions machine. Avant de traduire un programme sous forme SSA en langage machine, il est donc nécessaire de remplacer ces fonctions tout en gardant la même sémantique.

Q 2.1 Proposez une solution. Quels sont les problèmes rencontrés ? Quelles solutions éventuelles peut-on apporter ?

Q 2.2 Sortez de SSA votre programme.

3 Optimisations sous SSA

L'utilité de *Static Single Assignment* est sa parfaite maniabilité lors des optimisations. Certaines contraintes pour une optimisation disparaissent ou deviennent faciles à calculer.

4 Elimination de code mort

Une opération est *morte* si elle définit une variable jamais utilisée.

Q 4.1 Pourquoi peut-il s'avérer dangereux d'éliminer certaines opérations bien qu'elles soient mortes ?

Q 4.2 Donnez un algorithme pour l'élimination de code mort travaillant sur une représentation intermédiaire sous forme SSA. L'algorithme marquera d'abord les opérations *a priori* utiles et remontera le long de leurs dépendances.

Q 4.3 Appliquez votre algorithme sur le programme suivant :

```
a = 1;
b = 2;
a = 2;
c = 2;
if (a == 4) {
  for (i=1;i<5;i++) {
    b = b + i;
  }
} else {
  c = 5;
  while (c > 2) {
    a = a + b;
    b = b * b;
    a = c + 1;
    c = c - 2;
  }
```

```
}
a = 4;
return a + c;
```

5 Propagation de constante simple

Le principe de la propagation de constante est de remplacer chaque utilisation d'une variable v définie à partir d'une constante c par la valeur c de la constante. Les $v \leftarrow \phi(c_1, \ldots, c_n)$ avec $c_1 = \cdots = c_n = c$ peuvent être remplacées par $v \leftarrow c$ de la même façon.

Q 5.1 Donnez un algorithme pour la propagation de constante simple sous SSA se servant d'une liste de travail et d'une table de hachage. C'est bien d'être linéaire en la taille du graphe.

Q 5.2 Dans l'optique de la propagation de constante, d'autres optimisations sont possibles. Auxquelles pensez-vous ? Quels sont les problèmes liés ? Peut-on modifier l'algorithme de propagation de constante simple pour qu'il prenne en compte ces cas-là ?

Q 5.3 Appliquez votre algorithme sur votre programme déjà optimisé par l'élimination de code mort.

6 Value numbering

Le *value numbering* est une technique pour éviter de recalculer d'expressions dans un code. Il consiste principalement à attribuer à chaque expression un numéro correspondant à une classe d'équivalence. Deux expressions auront donc le même numéro si elles peuvent être prouvées égales. Les expressions redondantes que l'on peut éliminer seront celles qui ont le même numéro qu'une expression disponible au nœud correspondant du graphe de flot de contrôle.

Q 6.1 Appliquez à la main un value numbering sur le programme suivant et simplifiez-le (A et B sont de dimension 10*10) :
V[i * 10 + j] <- A[i, j] * B[i, j]

$$
\begin{aligned}
s &= i * 10 \\
t &= s + j \\
\alpha &= @A + t \\
a &= *\alpha \\
u &= i * 10 \\
v &= u + j \\
\beta &= @B + v \\
b &= *\beta \\
w &= i * 10 \\
z &= w + j \\
q &= a * b \\
\gamma &= @v + z \\
*\gamma &= q
\end{aligned}
$$

Q 6.2 Donnez un algorithme pour le *value numbering* fonctionnant sur un bloc de base – non nécessairement sous forme SSA. Utilisez pour cela :
- une table de hachage qui attribue à chaque variable et expression un numéro ;
- un tableau `VN` qui fait correspondre à chaque variable un numéro ;
- un tableau `name` qui donne le nom de variable à laquelle correspond un numéro donné.

Q 6.3 Faites le tourner sur l'exemple donné.

Q 6.4 Convainquez-vous et surtout votre TDman que vous sauriez faire si on vous demandait d'étendre l'algorithme pour la gestion de la commutativité de certaines opérations et d'autres identités algébriques.

Q 6.5 Faites tourner l'algorithme sur cet exemple – que vous aurez évidemment préalablement passé sous forme SSA et dont vous aurez calculé l'arbre de dominance. Comprenez chaque étape de calcul et les différents cas d'optimisations.