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.

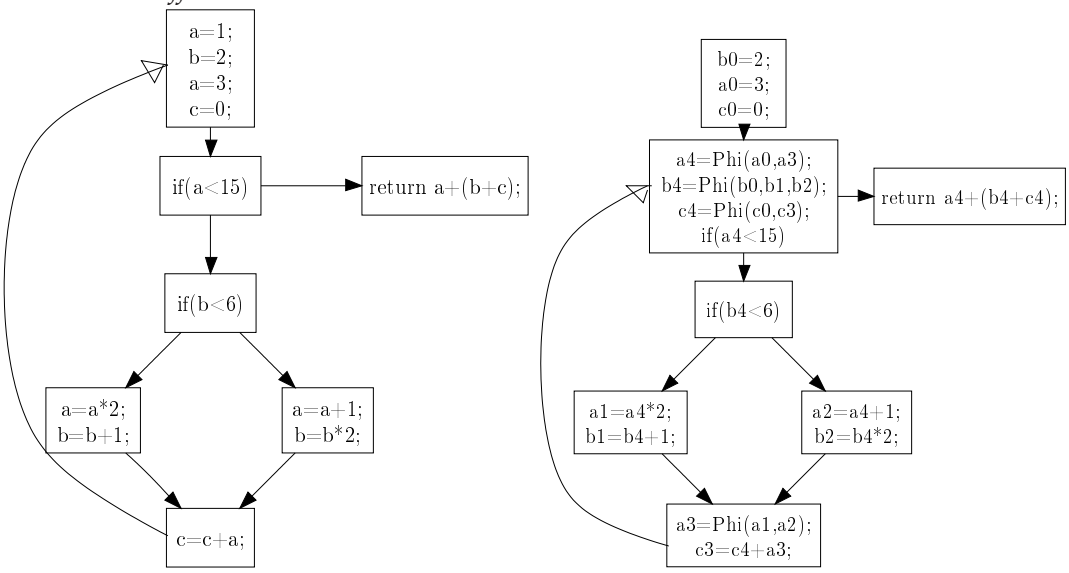
1 Passage en SSA

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);
```

Solution:

Dans un 1er temps, on crée l'arbre CFG du code puis on rajoute les fonctions ϕ là où on en a besoin, c'est à dire aux endroits où l'on a un def qui dépend de plusieurs valeurs précédentes, sur des chemins différents.



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.

Solution: *On note qu'il faut placer les ϕ sur les frontières de dominances. Mais en rajoutant des ϕ, on rajoute des variables, il faut donc itérer ce processus aussi pour les variables que l'on crée. L'algorithme consiste donc à calculer les frontières de dominances des noeuds contenant du CFG, rajouter les ϕ et itérer le processus. On peut facilement écrire un algorithme de point fixe pour calculer les positions des ϕ de la forme :*

calculer les frontières de dominances.

$L \leftarrow \{\phi\}$ correspondant à ces frontières.

Tant que L change, calculer les nouvelles frontières et rajouter les ϕ correspondant dans L.

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

Solution: *Ici, les seules frontières intéressantes à calculer sont celles des noeuds "a=a*2 ;b=b+1 ;" et "a=a+1 ;b=b*2 ;". Ils ont la même frontière , le noeud "c=c+a ;", donc il faut rajouter un ϕ au début de ce noeud. On itère sur la variable nouvellement créée par le ϕ : sa frontière de dominance est le noeud "if a<15". On y rajoute les ϕ nécessaires.*

On retrouve bien les ϕ placés "à la main".

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

Solution: *On se rend compte que cet algorithme va créer des ϕ inutile. Par exemple , il ne faut pas mettre de ϕ si la variable en question n'est pas* LiveIn *au début du noeud, car elle ne sera pas utilisée après.*

2 Sortie de SSA

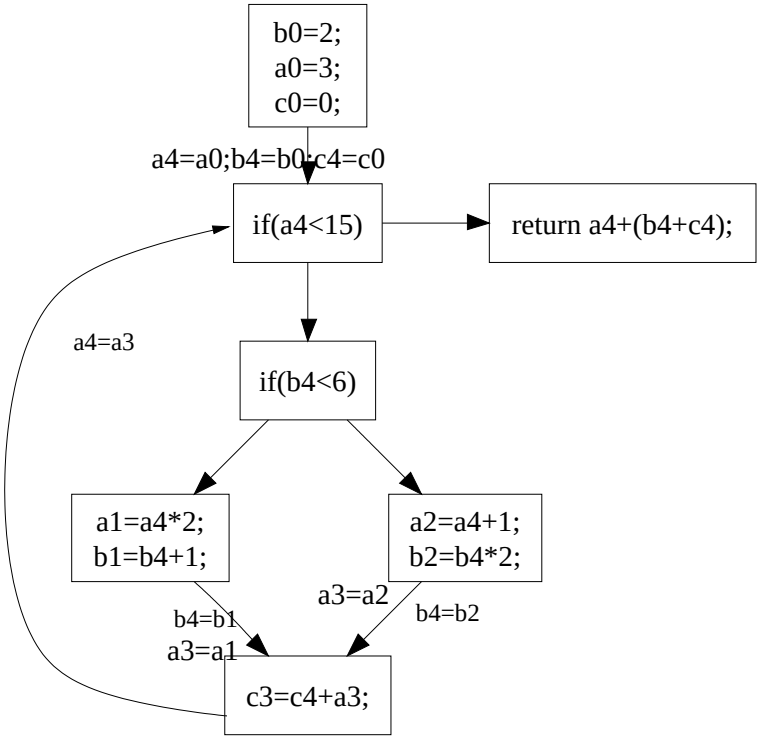
Les fonctions ϕ ne sont évidemment pas des instructions machine. Avant de traduire un programme sous forme SSA en langage machine, il est donc nécessaire de remplacer ces fonctions tout en gardant la même sémantique.

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

Solution: *Pour supprimer les ϕ, on peut les remplacer par des copies sur les arètes entrantes. Cela nous fait sortir de la forme SSA mais en contre partie, on rajoute de nombreuses copies, et on* split *certaines arètes. Ceci peut être ennuyeux par exemple pour les arètes de retour des boucles de types* while *ou* for*. Sur certaines architectures (embarquées par exemple), il est possible de faire des boucles en* hardware *s'il on peut sémantiquement trouver le nombre d'itération. Rajouter des noeuds sur les arètes de retour empecherait cette optimisation qui peut être très bénéfique (pas de temps perdu entre la fin et le début de la boucle).*

Q 2.2 Sortez de SSA votre programme.

Solution:



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 ?

Solution: *Il y a quelques opérations qui ont l'air mortes, mais sont utiles quand même : par exemple des opérations qui metent en jour des paramètres "call-by-reference" ou des affections par pointeur.*
En plus, l'algo suivant élime aussi des boucles infinies sans input/ouput. Comma ca les instructions après les boucles infinies sont executées. C'est une change du comportement de programme qui est indésirable pour le plupart des applications.

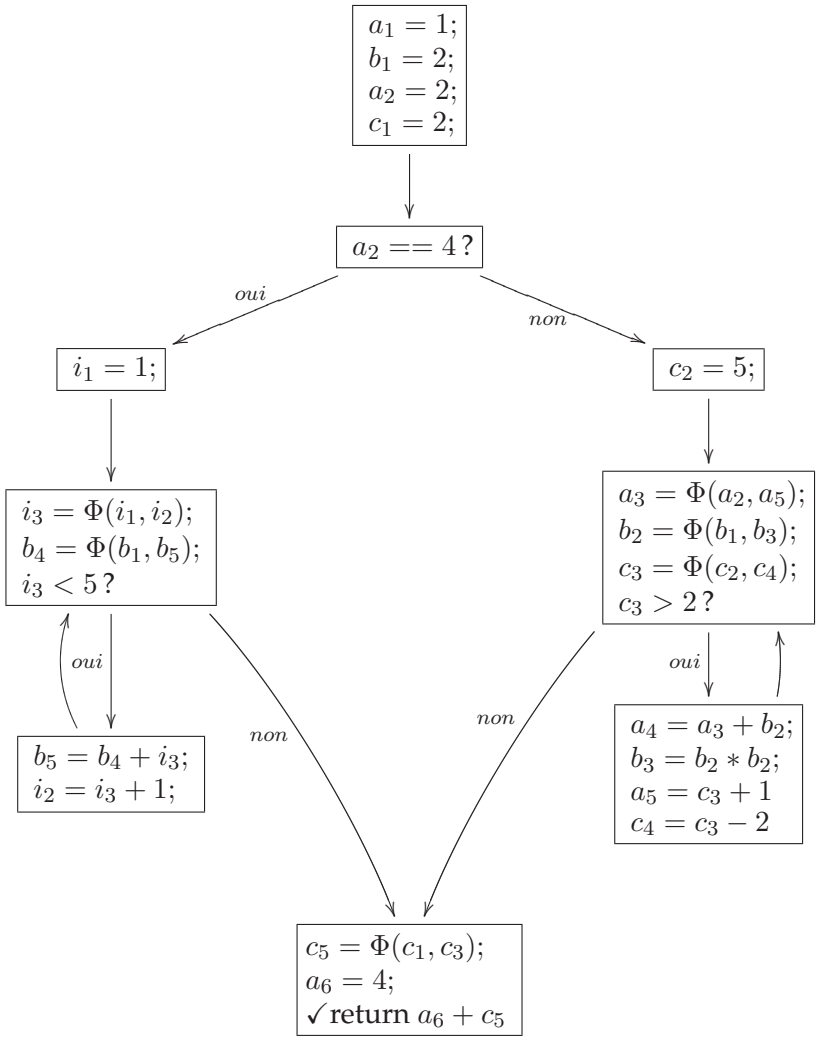
Q 4.2 Donnez un algorithme pour l'élimination de code mort travaillant sur une représentation intermédiaire sous forme SSA. L'algorithme marquera d'abord les opérations *a priori* utiles et remontera le long de leurs dépendances.

Solution:
Il faut qu'on sache les opérations utiles :
– *opération input et output*
– *store en mémoire*
– *return du fonction*
– *call d'une autre fonction qui peut avoir des side-effects*
D'abord l'algo marque toutes cettes opérations live. Puis il marque toutes les opérations live qui definent une variable utilisée par un autre opération live.
Il faut aussi marquer toutes les opérations If-Then-Else qui branchent à une opération live comme live et ajoute aussi les variables utilisées par le If à la worklist.
L'algorithme itère jusqu'à ce que la workliste soit vide.

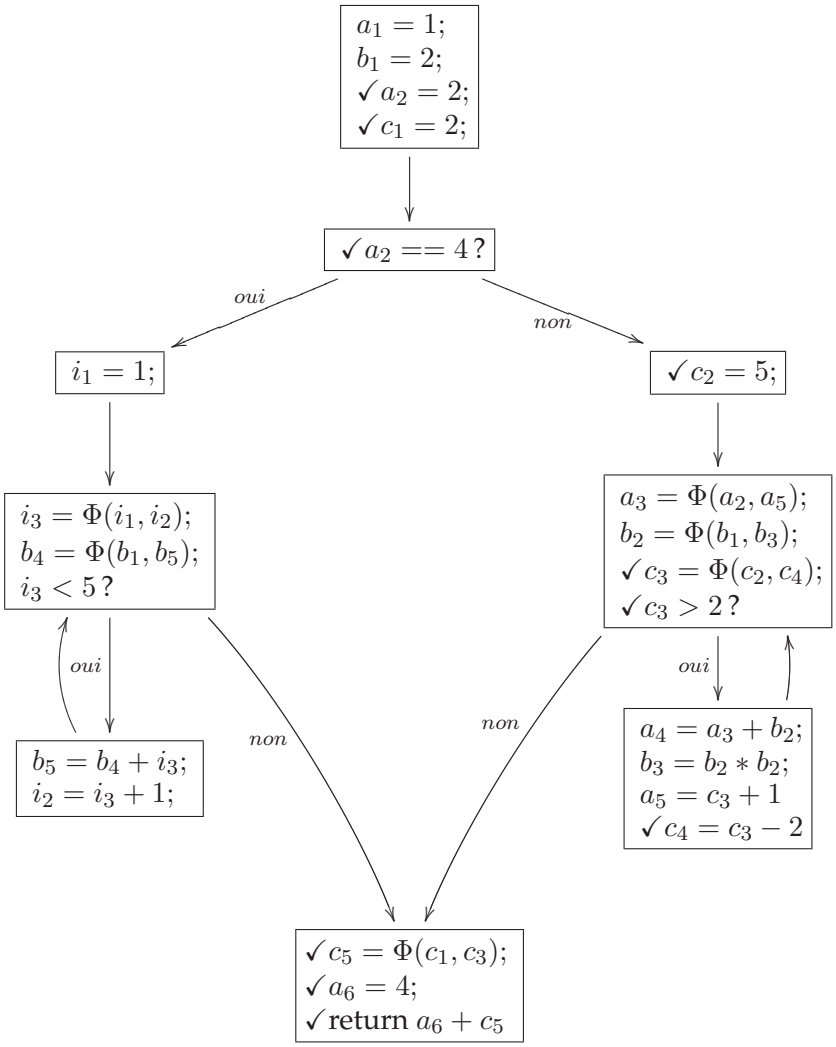
Q 4.3 Appliquez votre algorithme sur le programme suivant :

```
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;
```

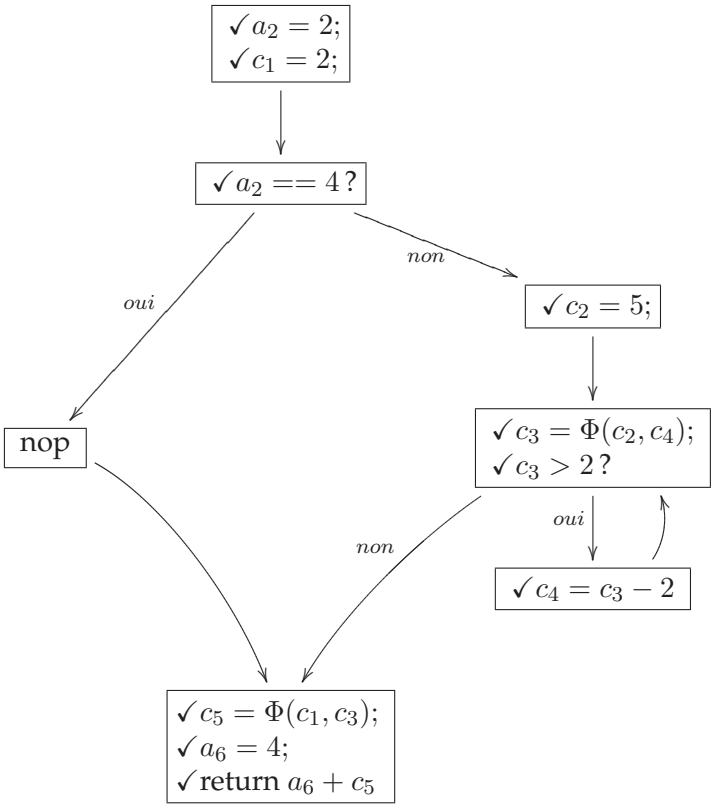
D'abord on produit le graphe de flot de controle et on marque les opérations utiles ✓ (c'est qu'une seule).



Puis on ajoute les variables qui sont utilsées par les fonctions utiles, c'est à dire a_6, c_5 à la worklist. Ensuite on marque les opérations qui définissent une variable de la liste utiles et ajoute les variables utilisées par ces opérations. On procède jusque la worklist est vide.



Maintenant la deuxième passe enlève toutes les opérations non-marquées :



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.

On fait un pass du CFG (SSA) en mettant les définitions des constants dans une table de hachage à la clé de nom de variable et cherchant les variables utilisées dans cette table. Si on en trouve, on les remplace par sa valeur et si possible applique des opérations pour les simplifier. L'algo est ceci-là :

```
HashTable constants;

procedure simplify( a <- a_1 op a_2 ) {
    if constants.find(a_1) then
        replace(a_1, constants.valueOf(a_1));
    if constants.find(a_2) then
        replace(a_2, constants.valueOf(a_2));
    if isConstant(a_1) && isConstant(a_2) then
        constants.add(a, eval(op, a_1, a_2) );
```

```
            replace(a_1 op a_2, eval(op, a_1, a_2));
    }

    procedure main() {
        for each (instruction I) in dominance order do {
            simplify ( I )
            if I of the form ( a <- b ) and isConstant(b) then
                constants.add(a,b)
        }
    }
```

Les affections inutiles sont enlevées par l'algo d'elimination de code mort plus tard.

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 ?

Des optimisations possibles sont des simplifications de par exemple
- des additions avec 0
- des multiplication avec $c \in 0, 1$
- des divisions par 1

Il faut faire attention parce que la semantique ne doit pas être dependant de l'état de la machine. Considerer l'exemple suivant dans lequel la précision d'evalution de la division est signifiquante :

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]

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

$z = w + j$
$q = a * b$
$\gamma = @v + z$
$*\gamma = q$

Code en applicant le value-numbering (à chaque expression est affectée une valeur) :

$s = i * 10 \ \rightarrow 1$
$t = s + j \ \rightarrow 2$
$\alpha = @A + t \ \rightarrow 3$
$a = *\alpha \ \rightarrow 4$
$u = i * 10 \ \rightarrow 1$ //$s = i * 10$, donc $u = s$
$v = u + j \ \rightarrow 2$ //de même $t = s + j$, donc $v = t$
$\beta = @B + v \ \rightarrow 5$
$b = *\beta \ \rightarrow 6$
$w = i * 10 \ \rightarrow 1$ //$s = i * 10$, donc $w = s$
$z = w + j \ \rightarrow 2$ //$t = s + j$ et $w = s$, donc $z = t$
$q = a * b \ \rightarrow 7$
$\gamma = @v + z \ \rightarrow 8$
$*\gamma = q \ \rightarrow 9$

Code simplifié :

$s = i * 10$
$t = s + j$
$\alpha = @A + t$
$a = *\alpha$
$u = s$ //La valeur de s est attribué à u
$v = t$ //La valeur de t est attribué à v
$\beta = @B + t$
$b = *\beta$
$w = s$ //La valeur de s est attribué à w
$z = t$ //La valeur de t est attribué à z
$q = a * b$
$\gamma = @v + z$
$*\gamma = q$

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.

Solution:

Avant l'algorithme un peu d'explication.
La table de hachage nous permet de détecter les redondances, pour celà il faudrait utiliser une table

de hachage et calculer la clé de hachage pour chaque expression. La clé de hachage d'une expression est calculée à partir des numéros des opérandes. La table de hachage est au debut vide. On calcule les numéros des opérandes qu'on sauvegarde dans la table de valeur. Si une entrée dans la table de valeur existe déjà, le nombre qui lui est assignée, reste inchangé. Sinon l'ajouter et lui assigner un numéro. Toute variable utilisée est définie plus haut et donc a un numéro qui lui est assigné. Pour une expression donnée, on peut calculer la clé de hachage pour l'expression et utiliser cette clé pour contrôler les redondances dans la table de hachage. Si une entrée existe avec une clé déjà enregistrée dans la table de hachage, celle-ci est redondante.

L'algorithme est le suivant :

pour toute *affectation* a *de la forme "*$x \leftarrow y$ *op* z*" dans* b **faire**
  $expr \leftarrow \langle VN[y]$ *op* $VN[z] \rangle$
  si $expr$ *peut être simplifiée en* $expr'$ **alors**
    remplacer a *par* $x \leftarrow expr'$
    $expr \leftarrow expr'$
  si $expr$ *se trouve dans la table de hachage avec le numéro* v **alors**
    $VN[x] \leftarrow v$
    si $VN[name[v]] = v$ **alors**
      remplacer *la partie droite de a avec* $name[v]$
  sinon
    $v \leftarrow$ *valeur numerique suivante*
    $VN[x] \leftarrow v$
    ajouter $expr$ *à la table de hachage avec le numéro* x
    $name[v] \leftarrow x$

Q 6.3 Faites le tourner sur l'exemple donné.

Solution:

On considère que chaque expression est un bloc.

1. $s^3 = i^1 * 10^2$
   - $VN[s] = 3$ *et ajouter* $i^1 * 10^2$ *à la table de hachage*
2. $t^5 = s^3 + j^4$
   - $VN[t] = 5$ *et ajouter* $s^3 + j^4$ *à la table de hachage*
3. $\alpha^7 = @A^6 + t^5$
   - $VN[\alpha] = 7$ *et ajouter* $@A^6 + t^5$ *à la table de hachage*
4. $a^8 = *\alpha^7$
   - $VN[a] = 8$ *et ajouter* $*\alpha^7$ *à la table de hachage*
5. $u^3 = i^1 * 10^2$
   - $VN[u] = 3$, $u^3 = s^3$ *car* $i^1 * 10^2$ *existe déjà dans la table de hachage*
6. $v^5 = u^3 + j^4$
   - $VN[v] = 5$, $v^5 = t^5$ *car* $u^3 = s^3$ *et* $s^3 + j^4$ *existe dans la table de hachage*
7. $\beta^{10} = @B^9 + v^5$

– $VN[\beta^{10}] = @B^9 + v^5$

8. $b^{11} = *\beta^{10}$

– $VN[b] = 11$ *et ajouter* $*\beta^{10}$ *à la table de hachage*

9. $w^3 = i^1 * 10^2$

– $VN[w] = 3$, $w^3 = s^3$ *car* $i^1 * 10^2$ *existe déjà dans la table de hachage*

10. $z^5 = w^3 + j^4$

– $VN[z] = 5$, $z^5 = t^5$ *car* $w^3 = s^3$ *et* $s^3 + j^4$ *existe dans la table de hachage*

11. $q^{12} = a^8 * b^{11}$

– $VN[q] = 12$ *et ajouter* $a^8 * b^{11}$ *à la table de hachage*

12. $\gamma^{13} = @v^5 + z^5$

– $VN[\gamma] = 13$ *et ajouter* $@v^5 + z^5$ *à la table de hachage*

Programme simplifié

$s = i * 10$
$t = s + j$
$\alpha = @A + t$
$a = *\alpha$
$u = s$
$v = t$
$\beta = @B + v$
$b = *\beta$
$w = s$
$z = t$
$q = a * b$
$\gamma = @v + z$
$*\gamma = q$

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.

En faite il faut ajouter des tests selon les différentes opérations. Par exemple pour les opérations commutatives, on pourra trier les opérandes par leurs valeurs numériques (cf cours page 98).

Un algorithme pour le value numbering sous SSA à base de la structure des dominateurs est l'algorithme suivant [1]

1. L'algorithme est présenté dans *Briggs, Cooper, Simpson : Value Numbering, Software : Practice and Experience 27, 6 (June), 701–724*

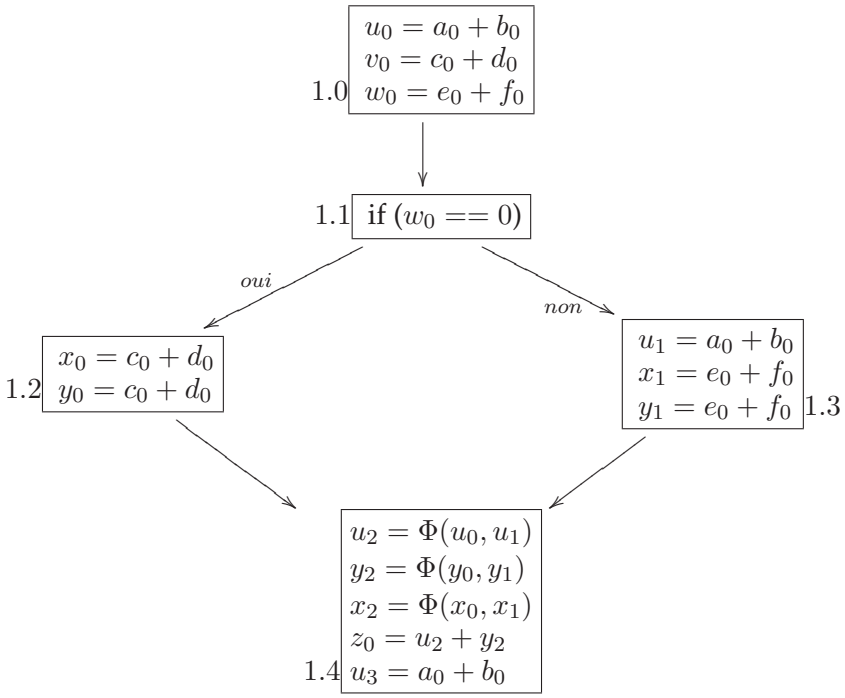
procédure DVN(bloc b**)**
marquer le début d'une nouvelle portée
pour toute fonction ϕ p de la forme "$\eta \leftarrow \phi(\ldots)$" dans b **faire**
si p est dépourvue de sens **ou** redondante **alors**
$VN[n] \leftarrow p$
supprimer p
sinon $VN[n] \leftarrow n$
ajouter p à la table de hachage
pour toute affectation a de la forme "$x \leftarrow y$ op z" dans b **faire**
écraser y par $VN[y]$ et z par $VN[z]$
$expr \leftarrow \langle y \text{ op } z \rangle$
si $expr$ peut être simplifiée en $expr'$ **alors**
remplacer a par $x \leftarrow expr'$
$expr \leftarrow expr'$
si $expr$ se trouve dans la table de hachage avec le numéro v
alors $VN[x] \leftarrow v$
supprimer a
sinon $VN[x] \leftarrow x$
ajouter $expr$ à la table de hachage avec le numéro x
pour tout successeur s de b **faire**
ajuster les noms des variables SSA dans les fonctions ϕ dans s
pour tout fils c de b dans l'arbre des dominateurs **faire**
DVN(c)
nettoyer la table de hachage en sortant de la portée

On considère le programme exemple suivant :

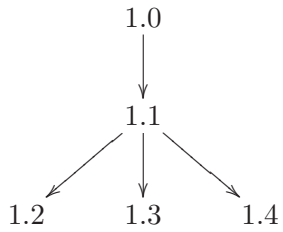
```
u = a + b;
v = c + d;
w = e + f;
if (w == 0) {
  x = c + d;
  y = c + d;
} else {
  u = a + b;
  x = e + f;
  y = e + f;
}
z = u + y;
u = a + b;
```

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.

Forme SSA :



Arbre de dominance :



1. Dans le bloc 1.0 de l'arbre de dominance nous avons :

   - $VN[u_0] = u_0$, $VN[v_0] = v_0$ et $VN[w_0] = w_0$
   - ajouter $VN[a_0] + VN[b_0]$, $VN[c_0] + VN[d_0]$ et $VN[e_0] + VN[f_0]$
   - il n'y a rien à ajuster, passer au prochain fils.

2. Dans le bloc 1.1 w_0 reste inchangé.

3. Dans le bloc 1.2 nous avons :

   - $x_0 = v_0$
   - $VN[c_0] + VN[d_0]$ qui est déjà dans la table de hachage donc,
   - $VN[x_0] = v_0$
   - et on supprime $v_0 = c_0 + d_0$ dans le bloc 1.1
   - **même procédé pour** y_0
   - L'algorithme nous permet immediatement de remplacer x_0 et y_0 par v_0 dans les successeurs du bloc 1.2.
     Donc $x_2 = \Phi(v_0, x_1)$ et $y_2 = \Phi(v_0, y_1)$

4. Dans le bloc 1.3 pareil que dans 1.2 nous avons :

   - $u_1 = u_0$
   - $VN[a_0] + VN[b_0]$ est déjà dans la table de hachage donc,
   - $VN[u_1] = u_0$
   - et on supprime $u_0 = a_0 + b_0$
   - $x_1 = w_0$
   - $VN[e_0] + VN[f_0]$ est déjà dans la table de hachage donc,
   - $VN[x_1] = w_0$
   - et on supprime $w_0 = e_0 + f_0$
   - procédé identique pour y_1
   - remplacer u_1 par u_0 et x_1, y_1 par w_0 dans les successeurs du bloc 1.3. Donc $u_2 = \Phi(u_0, u_0)$, $x_2 = \Phi(v_0, w_0)$ et $y_2 = \Phi(v_0, w_0)$

5. Dans le bloc 1.4 nous avons :

   - $u_2 = \Phi(u_0, u_0)$ est depourvue de sens $VN[u_2] = u_0$ et $u_2 = \Phi(u_0, u_0)$ est supprimée. Nous pouvons remarquer ici que à partir de l'arbre de dominance étant donné que 1.3 ne domine pas sur 1.4 u_2 est égale à u_0.
   - $x_2 = \Phi(v_0, w_0)$ est redondant car $x_2 = y_2$, on peut alors supprimer x_2.
   - nous avons une nouvelle expression $VN[u_0] + VN[x_2]$
   - $VN[z_0] = z_0$ et $VN[u_0] + VN[x_2]$ est ajouté à la table de hachage.
   - Par la suite nous avons $u_3 = u_0$
   - $VN[a_0] + VN[b_0]$ se trouve déjà dans la table de hachage donc $u_0 = a_0 + b_0$ est supprimmer.

Graphe simplifié :