

# Examen de compilation, 13 Janvier 2005

durée: 3h, documents manuscrits et poly du cours autorisés

Fabrice Rastello, Tanguy Risset

**Attention: version corrigée**

## 1 Optimisations

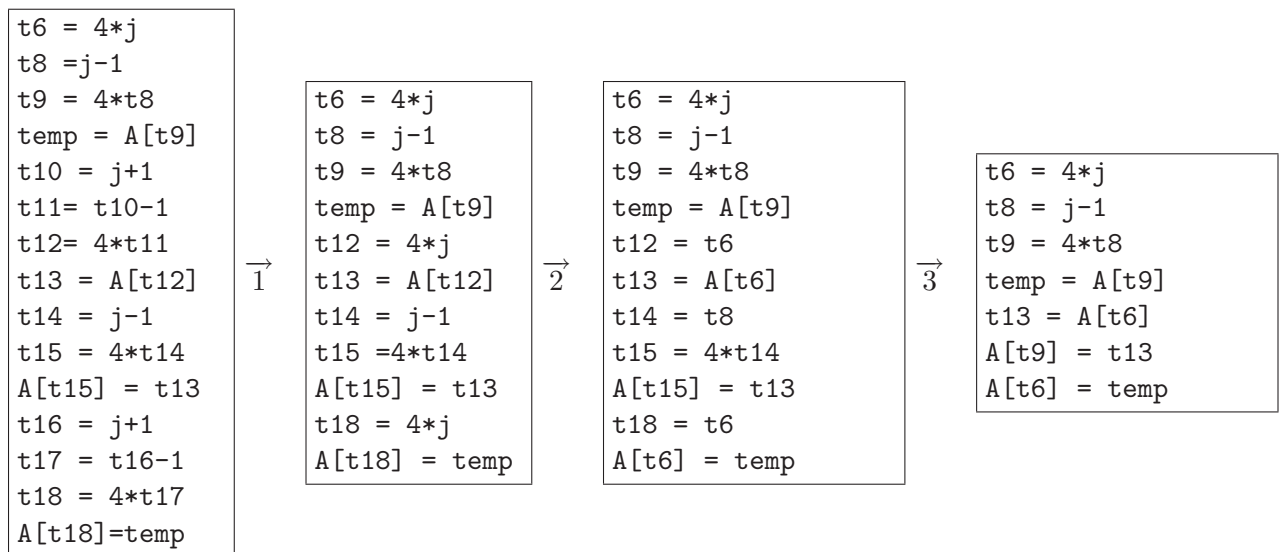


Figure 1: Optimisations successives

On considère les transformations successives appliquées au code assembleur de la Figure 1. On considère que l'on a appliqué des optimisations qui sont parmi les suivantes:

- Propagation de copie (e.g. remplacer  $t1=t2$ ;  $t3=4*t1$  par  $t1=t2$ ;  $t3=4*t2$ ).
- Propagation d'expression (e.g. remplacer  $t1=expr$ ;  $t3=4*t1$  par  $t1=expr$ ;  $t3=4*expr$ ).
- Élimination d'expressions redondantes
- Élimination de code mort.
- Simplification algébrique (e.g. remplacer  $t1+1-1$  par  $t1$ ) associée à une autre transformation.

1. Indiquez pour les étapes 1, 2 et 3 de la Figure 1 quelles optimisations ont été effectuées (on suppose qu'aucune variable, hormis le tableau A, n'est vivante après la fin du bloc). Fournir le code des étapes intermédiaires.

**Correction:**

- (a) Étape 1:
- Propagation d'expression et simplification algébrique:  $t_{11}=t_{10}-1$  devient  $t_{11}=j$
  - Propagation d'expression,  $t_{12}:= 4*t_{11}$  devient  $t_{12}:= 4*j$
  - élimination de code mort:  $t_{10}$  et  $t_{11}$  éliminés.
  - pareil pour  $t_{16}, t_{17}, t_{18}$
- (b) Étape 2:
- Élimination d'expression redondance:  $t_{12}=4*j$  devient  $t_{12}=t_6$ ,  $t_{18}=4*j$  devient  $t_{18}=t_6$ ,  $t_{14}=j-1$  devient  $t_{14}=t_8$
- (c) Étape 3
- Copy propagation  $A[t_{12}]$  devient  $A[t_6]$ ,  $t_{15}=4*t_{14}$  devient  $t_{15}=4*t_8$ ,
  - Élimination d'expression redondance:  $t_{15}=4*t_8$  devient  $t_{15}=t_9$ ,
  - Copy propagation  $A[t_{15}]$  devient  $A[t_9]$ ,  $A[t_{18}]$  devient  $A[t_6]$
  - Élimination de code mort,  $t_{12}, t_{14}, t_{15}, t_{18}$  éliminés.

```

t1 = t0
t2 = 4
t3 = t1 * t1
t4 = t2 + 2
t5 = t0 ^ 2 //t0 puissance 2
t6 = t5 + t3
t7 = t4 * t6
```

Figure 2: Code à optimiser pour la question 1.2

2. Appliquez maintenant vous-même une séquence de ces transformations sur le code de la Figure 2 de façon à minimiser la taille du code résultant. On suppose que seule la variable  $t_7$  est vivante après le bloc.

**Correction:**

- (a) Copy propagation:  $t_3 = t_1 * t_1$  devient  $t_3 = t_0 * t_0$ ,
- (b) dead code elimination:  $t_1 = t_0$  supprimé
- (c)  $t_4 = t_2 + 2$  devient  $t_4 = 6$
- (d) dead code elimination:  $t_2 = 4$  supprimé
- (e) algebraic simplification  $t_5 = t_0 ^ 2$  devient  $t_5 = t_0 * t_0$
- (f) redondant expression elimination:  $t_5 = t_0 * t_0$  devient  $t_5 = t_3$

- (g) on a alors
- ```

t3 = t0 * t0
t4 = 6
t5 = t3
t6 = t5 + t3
t7 = t4 * t6
```

- (h) Constant propagation:  $t_7=t_4*t_6$  devient  $t_7=6*t_6$

- (i) copy propagation  $t6=t5+t3$  devient  $t6=t3+t3$ .
- (j) dead code elimination  $t4 = 6$  et  $t5 = t3$  supprimé.

(k) soit le code: 

|                |
|----------------|
| $t3 = t0 * t0$ |
| $t6 = t3 + t3$ |
| $t7 = 6 * t6$  |

- (l) éventuellement (mais les compilateur ne vont pas le trouver): remplacer  $t3 + t3$  par  $2*t3$  et remplacer  $t6$  par sa valeur: 

|                |
|----------------|
| $t3 = t0 * t0$ |
| $t7 = 12 * t3$ |

## 2 Enregistrement d'activation

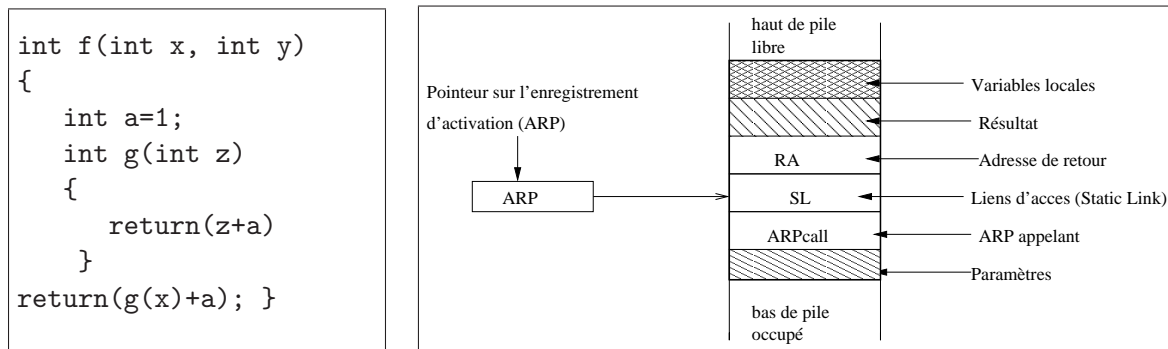
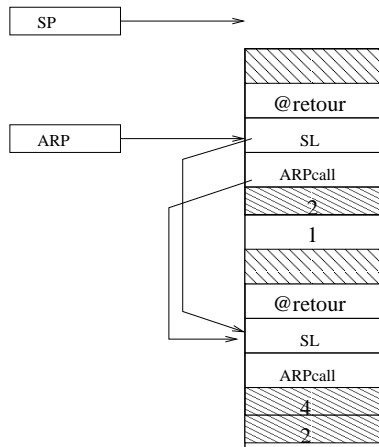


Figure 3: Code C pour la question 2 et convention utilisée pour l'enregistrement d'activation

On considère le pseudo-code représenté sur la Figure 3. On suppose que l'on génère du code assembleur Iloc pour une machine utilisant une convention d'appel classique. L'enregistrement d'activation d'une procédure est illustré sur la droite de la Figure 3. Pour lever toute ambiguïté on précise que l'ARP pointe sur *le début* de la case contenant le liens d'accès. Donc, pour charger l'ARP appelant dans  $r_1$  on exécute l'instruction  $loadAI\ r_{arp}, 4 \Rightarrow r_1$  (les registres sont des registres 32 bits). On suppose qu'aucun registre n'est sauvegardé par défaut dans la pile lors d'un appel. On dispose d'un nombre infini de registres plus les registres standards:  $r_{arp}$  pointant sur l'enregistrement d'activation de la procédure courante,  $r_{sp}$  pointant sur le sommet de la pile. On suppose aussi disposer d'un registre  $r_{pc}$  qui représente l'adresse de la prochaine instruction à exécuter ainsi que des symboles relogeable  $@f$  et  $@g$  contenant les adresses du code des procédures  $f$  et  $g$  ( $@f$  et  $@g$  sont traités comme des constantes).

1. Lors de l'exécution de  $f(2,4)$ , dessinez l'état de la pile juste avant l'exécution de l'instruction `return` de  $g$ . On détaillera le contenu de chaque case des enregistrements d'activation de  $f$  et  $g$ .

**Corrigé:**



2. Écrire (et commenter abondamment) le code Iloc réalisant l'instruction `return(g(x)+a)` de la fonction `f`.

**Corrigé:**

```
// mise en place du parametre x\\
loadAI Rarp,12 -> r1 \\
store R1 ->Rsp \\
subI Rsp,4 ->Rsp \\
// mise en place de l'ARP appelant
store Rarp -> Rsp\\
subI Rsp,4 ->Rsp \\
// mise en place du lien d'accès
store Rarp -> Rsp\\
// mise en place du nouvel ARP
i2i Rsp -> Rarp
subI Rsp,4 ->Rsp \\
// mise en place de l'adresse de retour
store Rpc -> Rsp\\
subI Rsp,4 ->Rsp \\
// appel de G\\
jumpI -> @g
```

```
// recuperation du resultat\\
load Rsp -> R1 \\
//mise a jour de Rarp
//recuperation de a
loadAI Rarp,4 -> Rarp\\
//pop des l'AR de G
addI Rsp,16 ->Rsp \\
// recuperation de a\\
loadAI Rarp,-8 -> R2 \\
//calcul du resultat
add R1,R2 -> R3 \\
storeAI R3 -> Rarp, -8\\
// mise en place du retour de f
addI Rarp,12 -> Rsp\\
loadAI Rarp,-4 -> Rret\\
jump Rret
```

### 3 Ordonnancement

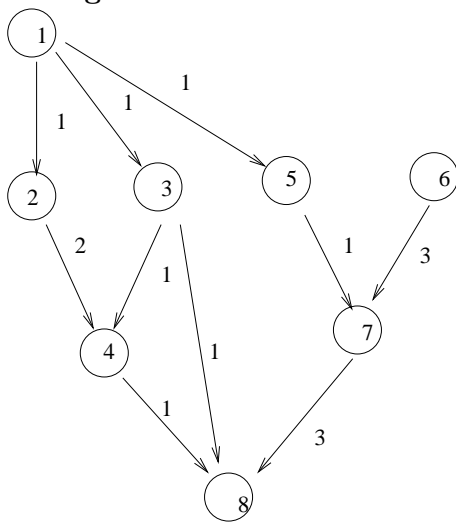
On considère le code de la Figure 4. On suppose que les opérations d'addition (*add* et *addI*) durent un cycle, la multiplication dure deux cycles et les chargements (*loadI*, *loadAI* ou *loadAO*) durent trois cycles. les opérateurs ne sont pas pipelinés.

1. Dessinez le graphe de dépendance que l'on peut utiliser pour ordonnancer le code de la Figure 4 (les noeuds de ce graphe représentent les instructions). On étiquettera chaque arc du graphe par le temps qui doit s'écouler entre les dates d'exécution des deux extrémités de l'arc.

|   |               |                            |
|---|---------------|----------------------------|
| 1 | <i>add</i>    | $r_x, r_y \Rightarrow r_0$ |
| 2 | <i>mult</i>   | $r_0, r_0 \Rightarrow r_4$ |
| 3 | <i>addI</i>   | $r_0, 2 \Rightarrow r_5$   |
| 4 | <i>add</i>    | $r_5, r_4 \Rightarrow r_6$ |
| 5 | <i>add</i>    | $r_0, r_0 \Rightarrow r_1$ |
| 6 | <i>loadI</i>  | $@a \Rightarrow r_2$       |
| 7 | <i>loadAO</i> | $r_1, r_2 \Rightarrow r_3$ |
| 8 | <i>loadAO</i> | $r_6, r_5 \Rightarrow r_1$ |

Figure 4: Code à ordonnancer (question 3)

**Corrigé:**



2. Quel est le temps minimal d'exécution de ce code en supposant que l'on a autant d'unités fonctionnelles que nécessaire (i.e. sans contrainte de ressources)?

**Corrigé:**

Le chemin critique du graphe est de longueur 6, donc on ne peut faire en moins de  $6+3=9$  cycles.

3. Donner un ordonnancement optimal en supposant que l'on ait deux unités fonctionnelles pouvant exécuter toutes les opérations.

**Corrigé:**

| <i>cycle</i> | <i>FU1</i> | <i>FU2</i> |
|--------------|------------|------------|
| 1            | 6          | 1          |
| 2            | 2          | 3          |
| 3            | 5          |            |
| 4            | 7          | 4          |
| 5            |            |            |
| 6            |            |            |
| 7            | 8          |            |

4. Donner un ordonnancement optimal en supposant que l'on ait une seule unité fonctionnelle (montrer qu'il est optimal).

**Corrigé:**

| <i>cycle</i> | <i>FU1</i> |
|--------------|------------|
| 1            | 6          |
| 2            | 1          |
| 3            | 5          |
| 4            | 7          |
| 5            | 2          |
| 6            | 3          |
| 7            | 4          |
| 8            | 8          |

Il y a une opération par cycle, on ne peut faire mieux.

5. On introduit précisément les dénominations suivantes:

- Un ordonnancement est *optimal sans contrainte de ressource* si son temps d'exécution est le même que celui trouvé à la question 3.2.
- L'ordonnancement *au plus tôt* est l'ordonnancement optimal sans contrainte de ressources pour lequel chaque noeud est exécuté le plus tôt possible.
- L'ordonnancement *au plus tard* est l'ordonnancement optimal sans contrainte de ressources pour lequel chaque noeud est exécuté le plus tard possible.

- (a) Calculer les ordonnancement au plus tôt et au plus tard pour le code de la Figure 4. Expliquer comment vous les calculer.

**Corrigé:**

| <i>cycle</i> | <i>ASAP</i> | <i>ALAP</i> |
|--------------|-------------|-------------|
| 1            | 1,6         | 6           |
| 2            | 2,3,5       | 1           |
| 3            |             | 5           |
| 4            | 4,7         | 2,7         |
| 5            |             | 3           |
| 6            |             | 4           |
| 7            | 8           | 8           |

- (b) L'*écart* d'une instruction est la différence entre ses dates d'exécutions dans l'ordonnancement au plus tôt et l'ordonnancement au plus tard (quantité toujours positive). Calculer l'écart pour les instructions du code de la Figure 4

**Corrigé:**

| <i>instruction</i> | <i>ecart</i> |
|--------------------|--------------|
| 1                  | 1            |
| 2                  | 2            |
| 3                  | 3            |
| 4                  | 2            |
| 5                  | 1            |
| 6                  | 0            |
| 7                  | 0            |
| 8                  | 0            |

- (c) Calculer un ordonnancement avec l'algorithme du *List scheduling* donné en cours en supposant que l'on ait une seule unité fonctionnelle et que la stratégie pour choisir une opération prête est: *choisir celle dont l'écart est maximal*. Détailler le déroulement de l'algorithme pour calculer cet ordonnancement. Expliquer intuitivement la qualité du résultat.

**Corrigé:**

- (a)  $cycle = 0, Ready = \{6, 1\}, Active = \emptyset$
- (b) Choisir instruction 1,  $Active = \{1\}, Ready = \{6\}$
- (c)  $cycle = 1, Active = \emptyset, Ready = \{6, 2, 3, 5\}$
- (d) Choisir 3,  $Active = \{3\}, Ready = \{6, 2, 5\}$
- (e)  $cycle = 2, Active = \emptyset, Ready = \{6, 2, 5\}$
- (f) Choisir 2,  $Active = \{2\}, Ready = \{6, 5\}$
- (g)  $cycle = 3, Active = \{2\}, Ready = \{6, 5\}$
- (h) Choisir 5,  $Active = \{2, 5\}, Ready = \{6\}$
- (i)  $cycle = 4, Active = \emptyset, Ready = \{6, 4\}$
- (j) Choisir 4,  $Active = \{4\}, Ready = \{6\}$
- (k)  $cycle = 5, Active = \emptyset, Ready = \{6\}$
- (l) choisir 6,  $Active = \{6\}, Ready = \emptyset$
- (m)  $cycle = 6$  pas d'opération prête
- (n)  $cycle = 7$  pas d'opération prête
- (o)  $cycle = 8, Active = \emptyset, Ready = \{7\}$
- (p) choisir 7,  $Active = \{7\}, Ready = \emptyset$
- (q)  $cycle = 9$  pas d'opération prête
- (r)  $cycle = 10$  pas d'opération prête
- (s)  $cycle = 11, Active = \emptyset, Ready = \{8\}$
- (t) choisir 8

Temps d'exécution: 13 cycles. Ce choix est mauvais car les variables avec l'écart le plus important sont celle qui sont moins contraintes, la bonne solution est évidemment de prendre celle dont l'écart est le plus faible.

## 4 Forme SSA

### 4.1 Forme maximale, semi-pruned, minimale

La forme SSA maximale est obtenue à l'aide de l'algorithme suivant:

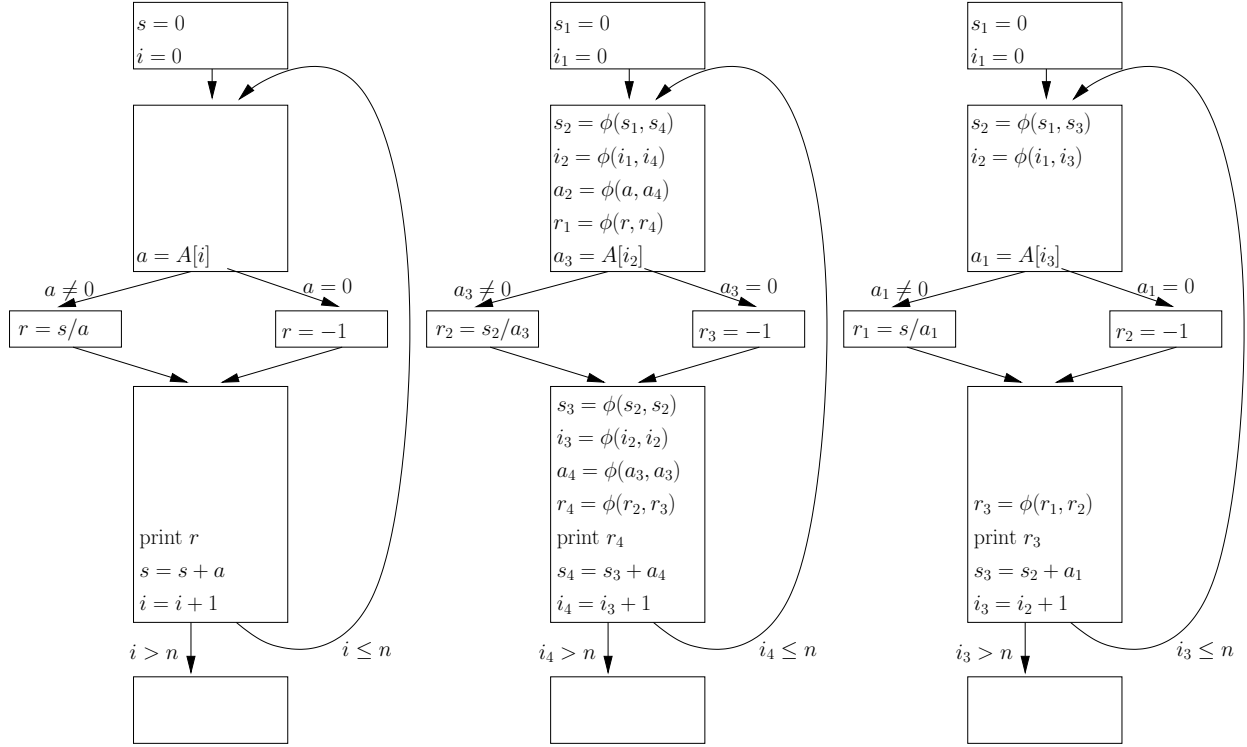
1. *insertion de fonctions  $\phi$* : à chaque point de confluence (noeud ayant plusieurs prédécesseurs) du CFG, insérer des fonctions  $\phi$  pour tous les noms de variables utilisées dans la procédure. Par exemple, pour la variable  $y$  et un noeud ayant deux prédécesseurs, on rajoute  $y = \phi(y, y)$ .
2. *Renommage*: après avoir inséré les fonctions  $\phi$  calculer les relations entre les définitions et les utilisations de variables (on suppose que toute variable est initialisée). Indexer l'ensemble des variables définies ( $y_1, y_2$ , etc.) et renommer les utilisations en conséquence.

Soit le pseudo-code suivant:

```
{On suppose que  $0 \leq n$ }  
a = 0  
r = 0  
s = 0  
for i = 0 to n  
  a = A[i]  
  if a  $\neq$  0  
    then r = s/a  
    else r = -1  
  print r  
  s = s + a  
endfor
```

1. Donner le graphe de flot de contrôle (CFG) de ce code: on ne vous demande pas d'écrire de l'assembleur, on reste en pseudo-code.
2. Passez en forme SSA maximale.
3. Donnez la forme SSA dite minimale, c'est à dire avec un nombre minimum de fonctions  $\phi$ .

Corrigé:



## 4.2 Dominance, dominance stricte, dominance immédiate

On considère le CFG d'une procédure dont le noeud racine est nommé entry. On dit que A domine B ( $A \in \text{Dom}(B)$ ) si tout chemin de entry à B (inclus) contient A. La notion de dominance définit un ordre partiel sur les noeuds dont le diagramme de Hasse est un arbre. Le dominateur immédiat d'un noeud B est noté  $i\text{Dom}(B)$ .

On calcule par analyse data-flow l'ensemble des dominateurs de chaque noeud L à l'aide de la formule suivante ( $\text{pred}(L)$  représente l'ensemble des noeuds prédécesseurs de L):

$$\text{Dom}(L) = \{L\} \cup \left( \bigcap_{P \in \text{pred}(L)} \text{Dom}(P) \right) \quad (1)$$

On note  $\mathcal{L}$  l'ensemble des noeuds du CFG. On considère deux initialisations possibles pour les équations data-flow:

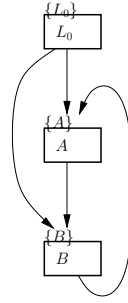
$$\begin{cases} \forall L \in \mathcal{L}, \text{Dom}(L) = \mathcal{L} \\ \forall L, \text{Dom}(L) = \emptyset \end{cases}$$

1. L'une est correcte, l'autre pas. Pourquoi?

**Corrigé:**

$L_0$  est le noeud entry.

Il n'y a pas unicité du point fixe, sur cet exemple avec  $\emptyset$  comme initialisation ca converge vers ca:



Preuve de correction de l'autre possibilité (initialisation à  $\mathcal{L}$ ):

- $Dom(L)$  décroît à chaque assignation (récurrence sur les assignations. attention à l'initiation de la récurrence) donc converge.
- On a l'invariant suivant: si  $D$  est un dominateur de  $L$  alors  $D \in Dom(L)$  à chaque étape d'assignation. On le prouve en montrant que si  $D \neq L$  est un dominateur de  $L$  alors pour tout  $P$  predecesseur de  $L$ ,  $D$  est un dominateur de  $P$ . En effet soit un chemin  $\mathcal{C}$  de  $L_0$  à  $P$ .  $L_0 \xrightarrow{\mathcal{C}} PL$  est un chemin de  $L_0$  à  $L$ , donc  $D \in \mathcal{C}$ . Finalement on vérifie aisément que la relation d'invariance est vérifiée initialement.
- Reste à montrer que  $Dom(L)$  point fixe de l'équation (1) ne contient *que* des dominateurs de  $L$ . On montre que si  $D$  n'est pas dans un chemin de  $L_0$  à  $L$ , alors  $D \notin Dom(L)$ . On raisonne par récurrence sur la longueur des chemins. On vérifie aisément la propriété sur un chemin de taille 0 ( $L_0$ ). Pour un chemin de taille  $n \geq 1$   $L_0 \xrightarrow{\mathcal{C}} QL$  (Pour  $n = 1$ ,  $Q = L_0$ ) tq  $D \notin \mathcal{C}$ ;  $D \notin L_0 \xrightarrow{\mathcal{C}} Q$ , donc (HYP)  $D \notin Dom(Q)$ . Or  $Q$  est un prédécesseur de  $L$  et (Equation (1))  $Dom(L) \subset \{L\} \cup Dom(Q)$  donc  $D \notin Dom(L)$ .

2. On dit que le noeud  $A$  domine *strictement* le noeud  $B$  ( $A \in sDom(B)$ ) si tout chemin de entry au *début* de  $B$  *traverse*  $A$ . Exprimez  $sDom$  en fonction de  $Dom$ . **Corrigé:**  
 $sDom(A) = Dom(A) - A$

### 4.3 Frontière de dominance

Soit  $L$  un noeud du CFG. La frontière de dominance de  $L$  notée  $DF(L)$  est l'ensemble des noeuds non strictement-dominés par  $L$  mais ayant un prédécesseur dominé par  $L$ . La Figure 5 illustre cette propriété.

L'algorithme de calcul de la frontière de dominance (dont l'un des but de cette partie est de prouver la correction) donné dans le Cooper est le suivant:

```

for all nodes  $L$ 
   $DF(L) = \emptyset$ 
for all nodes  $M$ 
  if  $M$  has several predecessors then
    foreach predecessors  $P$  of  $M$ 
       $L \leftarrow P$ 
      while  $L \neq iDom(M)$ 
         $DF(L) \leftarrow DF(L) \cup \{M\}$ 
         $M \leftarrow iDom(M)$ 

```

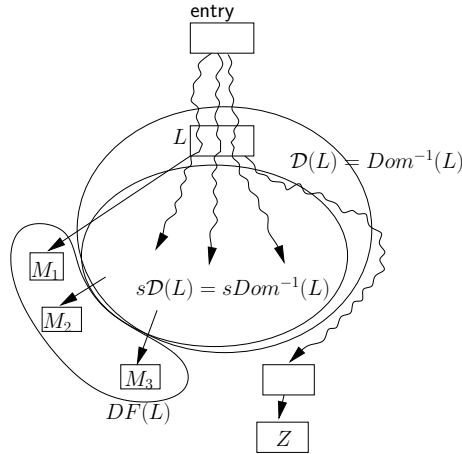
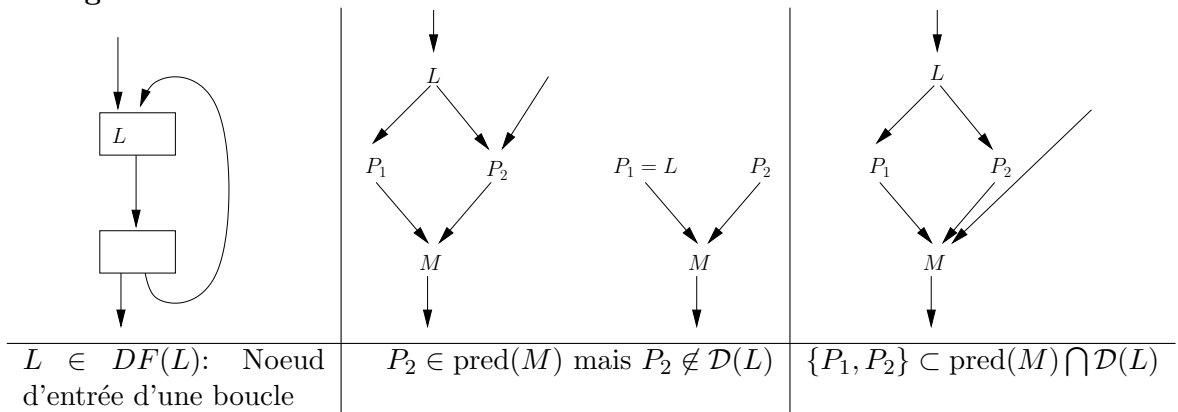


Figure 5:  $\mathcal{D}(L)$  représente l'ensemble des noeuds dominés par  $L$ .  $s\mathcal{D}(L)$  représente l'ensemble des noeuds strictement dominés par  $L$ . Les noeuds  $M_i$  sont dans la frontière de dominance de  $L$ : ils ne sont pas strictement dominés par  $L$ , mais ont un prédécesseur dominé par  $L$ .  $Z$  n'a pas de prédécesseur dominé par  $L$ .

1. Notez que  $L$  peut être lui même dans  $DF(L)$ . Donnez un exemple de graphe de flot de controle où c'est le cas.
2. Les noeuds de  $DF(L)$  sont nécessairement des noeuds de confluence c'est à dire ayant plusieurs prédécesseurs. Justifiez pourquoi.
3. Un noeud de  $DF(L)$  n'a pas nécessairement tous ses prédécesseurs dans  $\mathcal{D}(L)$ . Donner un exemple illustrant ce point.
4. Par contre un noeud de  $DF(L)$  peut avoir plusieurs prédécesseurs dans  $\mathcal{D}(L)$ . Donner un exemple illustrant ce point.

**Corrigé:**



Soit  $M \in DF(L)$ ,  $P$  un prédécesseur dominé par  $L$ .  $P$  ne peut être unique prédécesseur car sinon  $P$  domine  $M$  et donc  $L$  domine  $M$ . Les noeuds de  $DF(L)$  sont donc nécessairement des noeuds de confluence.

5. Prouvez la correction de l'algorithme de calcul de la frontière de dominance donné ci-dessus.

**Corrigé:**

La dominance est une relation d'ordre partiel dont le diagramme de Hasse est un arbre si bien que l'ensemble des dominateurs stricts de  $P$  peut être parcouru en remontant l'arbre de dominance par les dominateurs immédiats.

Entry n'a pas de prédécesseur donc  $M \neq \text{Entry}$ . Par le processus  $M \leftarrow iDom(M)$  (sans condition d'arrêt) on rejoindrait irrémédiablement le noeud Entry. On tombe donc nécessairement sur un noeud  $D$  dominant  $P$  et dominant strictement  $M$ . Soit un tel noeud  $D$ ,  $iDom(M)$  allant de Entry à  $M$  et donc sur tout chemin allant de  $D$  à  $M$  et en particulier sur tout chemin allant de  $D$  à  $P$  (puisque'il est différent de  $M$ );  $iDom(M)$  domine donc  $P$ . Aussi, pour cette même raison, dans la boucle while, l'ensemble des noeuds qui précèdent strictement  $iDom(M)$  ne dominant pas  $P$ , et l'ensemble des noeuds qui succèdent largement  $iDom(M)$  dominant  $P$ .

6. En tenant compte de l'une des remarques précédentes, améliorez la condition d'arrêt de l'algorithme.

**Corrigé:**

La condition devient  $L \neq iDom(M)$  and  $M \notin DF(L)$ . On peut aisément implémenter pour chaque  $L$  l'ensemble  $DF(L)$  comme une pile: l'union est un push; le test  $M \in DF(L)$  vérifie simplement si  $M$  est en haut de la pile.