Compilation

TD 6 : Représentations intermédiaires – forme SSA

G. Iooss & C. Alias

La forme SSA décrit, à l'aide d'annotations, le flot de données de l'ensemble du programme, permettant des optimisations bien plus puissantes que les DAGs. Le but de ce TD est d'étudier la construction et l'application de la forme SSA à la simplification du code intermédiaire.

Exercice 0. *Échauffement*

Considérez le programme suivant:

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

Questions:

- Construisez manuellement le CFG de ce programme.
- Passez ce programme sous forme SSA. Quel est la signification de chaque fonctions ϕ qui apparaissent dans votre programme?

Exercice 1. *Optimisations inter-blocs*

Considérons le programme suivant:

```
a = 1;
b = 2;
e = 3;
while (e<11) {
  if (a<10) {
    c = b*a;
    e = e+1;
  } else {
    d = b;
    t = d;
    c = t*a;
    e = e-3;
  }
  e = e+1;
}
```

Questions:

- Calculez le CFG de ce programme. Passez sous forme SSA, puis éliminez les expressions redondantes pour chaque blocs.
- Comment diminuer encore plus le nombre d'instructions de ce programme? La méthode de calcul des expressions redondantes (vue en cours) est-elle suffisante pour faire cela?

L'élimination des expressions redondantes étant limitée à des blocs de base, il se peut que l'on manque des opportunités d'optimisation supplémentaires (cf question précédente). Nous allons donc étudier une méthode d'élimination des variables redondante, également basée sur la forme SSA[1].

Instructions et rang: Suppose qu'on a extrait le CFG $\mathcal{G}$ du programme considéré et qu'on l'a mis sous forme SSA. On appelle *backedge* une arête du CFG qui revient en arrière dans le programme (typiquement à la fin d'une boucle). Par la suite, on s'intéressera à $\mathcal{G}'$ le CFG $\mathcal{G}$ duquel on a retiré les backedges.

On associe à chaque instructions un *rang*. Pour cela, on construit le DAG du programme SSA[2] et le rang d'une instruction est égale à sa profondeur. Par exemple, dans le programme de l'exercice 0, le rang de l'instruction "`b = 2`" est 0, celui de "`c = b*a`" est 1 et celui de "`c = c + a`" est 2.

Questions:

- Calculez intuitivement les rangs des instructions du programme au début de cet exercice.
- Donnez un algorithme raisonnant sur $\mathcal{G}'$ pour calculer les rangs des instructions.
- Prouvez que si on ordonne les instructions d'un basic block selon leurs rangs croissants, le programme fait toujours les mêmes calculs.

Remonter des instructions: L'idée de l'algorithme est de faire remonter une par une et progressivement les instructions, de telle façon à déclencher au passage la détection de calcul redondant. Pour cela, il nous faut pouvoir faire passer des instructions d'un basic-block à un autre.

Questions:

- Considérons une instruction de rang 0 situé dans un basic-block admettant au moins un parent. Quels sont les règles de réécriture du CFG et leur conditions de validité pour les cas suivants:
  - (Cas fictif) Le basic-block de l'instruction admet un unique parent et ce dernier n'a qu'un fils.

[1] Article correspondant: "Global Value Numbers and Redundant Computations", par Barry K. Rosen et Mark N. Wegman

[2] Exactement comme pour l'exercice 2 du TP

  – (Join) Le basic-block de l'instruction possède deux parents.
  – (Fork) Le basic-block de l'instruction possède un parent qui possède un autre fils.
  – (Loop header) Le basic-block de l'instruction est le premier d'un corps de boucle.

- Dans le cas d'une instruction de rang différent de 0, quel autres contraintes faut-il prendre en compte avant de remonter une instruction?

Algorithme: L'algorithme en pseudo-code non détaillé est le suivant:

```
Mettre le programme sous forme SSA
Calculer les rangs de tous les instructions
Éliminer les calculs redondants locaux à chaque basic-blocks

Pour (R = 0, 1, ... N) faire {
    Pour chaque nœud n du CFG, en partant de ceux les plus loin dans G', faire:
        Pour chaque instruction S de rang R:
            Remonter le plus possible S dans le CFG.
    Éliminer les calculs redondants locaux à chaque basic-blocks
}
```

Questions:

- Appliquez manuellement l'algorithme sur le programme suivant:

```
READ(A,B,C,D,L,M,S,T) {
  while (X < 100) {
    if (D > 42) {
      L = C + B;
      M = L + 4;
      A = C;
    } else {
      D = C;
      L = D * B;
      S = A * B;
      T = S + 1;
    }
    X = A * B;
    Y = X + 1;
  }
}
```

Exercice 2. *Frontière de dominance*

Le but de cet exercice est de construire un algorithme alternatif pour le calcul d'une frontière de dominance. L'exercice suivant reprendra ensuite l'idée de cet algorithme, ainsi que certains résultats montrés ici, pour construire un algorithme linéaire de placement des ϕ-fonctions dans un graphe de flot de contrôle.

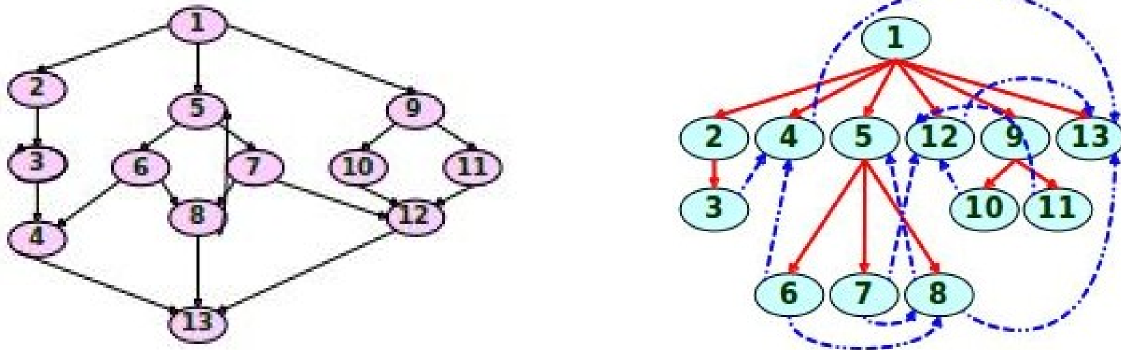
Étant donné un graphe de flot de contrôle et deux nœuds x et y, on note:

- x **dom** y pour indiquer que x domine y.
- x !**dom** y pour indiquer que x ne domine pas y.
- x **sdom** y pour indiquer que x domine *strictement* y (càd x **dom** y et $x \neq y$).
- $x =$ **idom** (y) (ou x **idom** y) pour indiquer que x est un parent de y dans l'arbre de dominance.
- $SubTree(x)$ l'ensemble des nœuds dominés par x.

- $x.level$ le *niveau* de x, càd sa profondeur dans l'arbre de dominance.

- $DF(x)$ la frontière de dominance du nœud x.

Dans cet exercice comme dans l'exercice suivant, on ne travaillera sur une extension du graphe de flot de contrôle appelée DJ-graphe. Un DJ-*graphe* possède le même ensemble de noeud que le graphe de flot de contrôle, et deux types d'arcs appelés D-arcs et J-arcs. Un D-*arc* est un arc de l'arbre de dominance, et un J-arc est un arc du graphe de flot de contrôle qui n'est pas un D-arc. La figure suivante montre un graphe de flot de contrôle et le DJ-graphe correspondant. Les D-arcs sont pleins et les J-arcs sont en pointillés :



Dans la suite, on considère un graphe de flot de contrôle $G = (N, E)$ et son DJ-graphe $DJ = (N_{dj}, E_{dj})$. Le cardinal d'un ensemble X est noté $|X|$.

Questions:

- Montrer que $|N_{dj}| = |N|$ et que $|E_{dj}| < |N| + |E|$.

- Soit x un noeud du DJ-graphe et $y \in DF(x)$. En utilisant **idom** (y), montrer que $x.level \geq y.level$.

- En déduire que z est dans $DF(x)$ si et seulement si il existe $y \in SubTree(x)$ tel que $y \to z$ soit un J-arc et $z.level \leq x.level$. Pour la partie $\Leftarrow$, on distinguera le cas $z \in SubTree(x)$ de $z \notin SubTree(x)$.

Ces remarques nous amènent à construire l'algorithme suivant, le pseudo-code `y -> z == J_arc` testant s'il existe un J-arc de y vers z :

```
DF(x) {
  DF = {}
  foreach y in SubTree(x) do
    if((y->z == J_arc) and
       (z.level <= x.level))
      DF = DF U {z}
}
```

Question

- Utiliser cet algorithme pour calculer $DF(5)$ dans le graphe de flot de contrôle donné ci-dessus. On détaillera les étapes de l'algorithme.

- Avec cet algorithme, quelle est la complexité au pire de la méthode de mise sous forme SSA vue en cours ?

Exercice 3. (Bonus) *Calcul efficace de la forme SSA*

Cet exercice propose d'étudier un algorithme de placement des ϕ-fonctions linéaire en le nombre de nœuds du graphe de flot de contrôle. Les notions, idées et résultats de l'exercice précédent seront exploitées. Cette partie n'est donc pas indépendante.

On étend la définition de la frontière de dominance $DF(S)$ à un ensemble de nœuds S:

$$DF(S) = \bigcup_{x \in S} DF(x)$$

. On définit la frontière de dominance itérée $IDF(S)$ pour un ensemble de nœuds S comme la limite de la suite croissante:

$$\begin{aligned} DF_1(S) &= DF(S) \\ DF_{i+1}(S) &= DF(S \cup DF_i(S)) = DF(S) \cup DF(DF_i(S)) \end{aligned}$$

Question:

- Expliquez en quoi l'algorithme de mise sous forme SSA vu en cours revient à calculer plusieurs frontières de dominance itérées.

Comme dans l'exercice précédent, on ne travaille pas sur le graphe de flot de contrôle, mais sur son DJ-graphe. Notre algorithme manipule une structure de données appelée *banque*. Une banque est un tableau dont chaque case contient une liste de nœuds du DJ-graphe. Les nœuds sont insérés de telle façon que la case i ne contient que des nœuds de niveau i.

On considère une unique banque `bank` (variable globale). L'opération d'insertion, `InsertNode(n)` ajoute le nœud `n` à la liste `bank[n.level]`. L'opération de Recherche/suppression, `GetNode()` cherche la dernière case de `bank` non vide (la case d'indice le plus grand dont la liste est non vide), et retourne (puis supprime) son premier nœud.

Chaque nœud est représenté par un enregistrement:

```
struct {
  visited: { Visited, NotVisited }; //already traversed
  inphi: {InPhi, NotInPhi};         //already added to IDF
  initial: {Initial,NotInitial};    //belong to the initial set
  level: integer;
}
```

Initialement:

- n.visited = NotVisited, node.inphi = NotInPhi, n.initial = NotInitial,
- n.level = profondeur dans l'arbre de dominance

Voici maintenant l'algorithme qui nous intéresse. Cet algorithme calcule la frontière de dominance itérée d'un ensemble S de nœuds passé en paramètre. Le résultat pourra être directement exploité pour placer les ϕ-fonctions dans le graphe de flot de contrôle.

```
IDF_main(S) {
  for each node n in S {
    InsertNode(n)
    n.initial = Initial
  }

  while(x = GetNode() != NULL) {
    CurrentRoot = x;
    x.visited = Visited;
    Visit(x);
  }
}
```

```
Visit(x) {
  for (each node y successor of x) {
    if(x -> y == J_arc and              //
        y.level <= CurrentRoot.level and // y in DF(x)...
        y.inphi == InPhi) {              // ...and not already added to IDF
      IDF = IDF U {y};
      if(y.initial != Initial)  //do not re-process initial nodes
        InsertNode(y);
    }

    if(x -> y == D_arc and
        y.visited != Visited) { // not already visited
      x.Visited = Visited
      Visit(y);
    }
  }
}
```

Initialement, on insère les nœuds de l'ensemble S dont on cherche la frontière de dominance itérée. Ensuite, pour chaque nœud x par ordre de niveau décroissant (en allant plus haut dans l'arbre de dominance), on appelle Visit(x). Visit(x) parcours en profondeur le DJ-graphe en partant de x, en évitant d'explorer les nœuds déjà visités lors d'un appel précédent (voir le `for each` et son deuxième `if`).

Suivant l'idée développée dans l'exercice précédent, le premier `if` collecte les nœuds dans la frontière de dominance de x (voir les commentaires). Ces nœuds sont ajoutés à l'ensemble résultat `IDF`, puis à la banque *via* `insertNode()`, pour être traités dans un appel ultérieur à `Visit()`.

Questions:

- Détailler les étapes de l'algorithme sur le calcul de $IDF(\{6, 9\})$ sur le DJ-graphe donné dans l'exercice précédent.
- Expliquer pourquoi un nœud est toujours inséré (*via* `InsertNode()`) à un niveau $\leq$ Current-Root.level.
- Soient x et y deux nœuds insérés dans la banque, puis supprimés par GetNode(). Montrer que si $y.level > x.level$, alors Visit(y) est appelé avant Visit(x). On distinguera, au moment de la suppression de x, le cas où y est dans la banque, et le cas où y n'est pas dans la banque.
- Montrer que quand Visit(x) est appelé avec $x = CurrentRoot$, tous les nœuds de $DF(x)$ sont dans IDF. on prendra garde au fait que $SubTree(x)$ n'est pas toujours entièrement parcouru.
- Montrer que quand l'algorithme termine, $IDF(S) =$ `IDF`. On pourra montrer l'inclusion $\subset$ par induction sur la définition de $IDF(S)$.
- Montrer que chaque noeud est visité (par un appel a Visit()) au plus une fois. Pour çà, on pourra montrer qu'un noeud ne peut pas être visité à la fois dans le while de IDF_main et dans le deuxième if de Visit().
- En déduire la complexité en temps de l'algorithme.