Compilation

TD 6: Intermediate representations – SSA form

A. Isoard & C. Alias

credits: G. Iooss

While the SSA form has a lot of interesting properties, it is sometimes needed to build it efficiently (for Just-in-time compilation for example). The objectives of this TP is to define, build and optimize this intermediate form in an efficient way.

Exercise 0. *Warm-up*

Consider the following program:

```
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:

- Build by hand the CFG of this program.
- Translate this program into its SSA form. What is the meaning of each ϕ functions that appears in your program?

Exercise 1. *Inter-block optimisations*

Consider the following program:

```
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:

- Compute the CFG of this program. Put it into SSA form, then eliminate redundant expressions for each blocks.

- How to reduce even more the number of instructions? Is the method of redundant expressions (seen in the lecture) enough to achieve that?

As elimination of redundant expressions is done per basic block, we might miss opportunities of optimization (as seen in the previous question). Lets study a new method for elimination of redundant variables[1], still based on the SSA form.

Instructions and rank: We suppose that we extracted $\mathcal{G}$, the program CFG, and that we put it under SSA form. We call *back-edge* an edge of the CFG that go back in the program (typically at the end of a loop). We will work on $\mathcal{G}'$, the CFG $\mathcal{G}$ from which we removed that said back-edges.

We associate to each instructions a *rank*. For that, we build the DAG of the program in SSA and the rank of an instruction is equal to its depth. Fir example, in the program of exercise 0, the rank of "`b = 2`" is 0, the one of "`c = b*a`" is 1, and the one of "`c = c + a`" is 2.

Questions:

- Compute intuitively the rank of the instructions of the program at the beginning of this exercise.
- Give an algorithm over $\mathcal{G}'$ that computes the rank of the instructions.
- Prove that if we schedule the instructions of a basic block under increasing rank, the program still do the same computation.

Pull up of the instructions: The idea of the algorithm is to pull up, each instruction, progressively, such that we trigger the detection of redundant computation. To do that, we need to be able to move instruction from one basic block to an other.

Questions:

- Consider an instruction of rank 0 inside a basic-block admitting at least a parent. What are the rewriting rules of the CFG, and their validity conditions for the following cases:
  - (Fictive case) The basic-block of the instruction only have a single parent, and this parent only have a single child.
  - (Join) The basic-block of the instruction have two parents.
  - (Fork) The basic-block of the instruction have a single parent that possess an other child.
  - (Loop header) The basic-block of the instruction is the first of a loop.
- In the case of an instruction of rank different than 0, what other constraint should we take into account before pulling up an instruction?

Algorithm: The algorithm in pseudo-code without details is the following:

```
Put the program under SSA form
Compute the rank of every instruction
Eliminate redundant computation inside each basic-block

For (R = 0, 1, ... N) do {
    For each node n of the CFG, by begining from the further away in G', do:
        For each instruction S of rank R:
            Pull up S as far up as possible inside the CFG.
    Eliminate redundant computation inside each basic-block
}
```

Questions:

[1] For more details, see: "Global Value Numbers and Redundant Computations", by Barry K. Rosen and Mark N. Wegman

- Apply manually the algorithm on the following program:

```
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;
  }
}
```

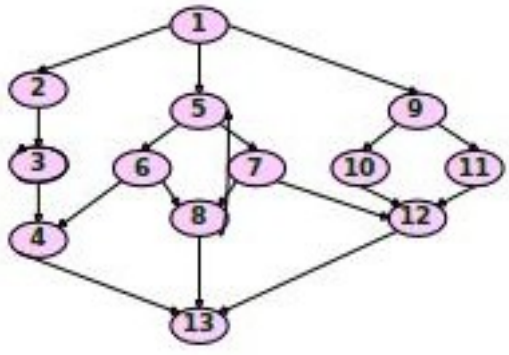
Exercise 2. *Dominance frontier, efficiently*

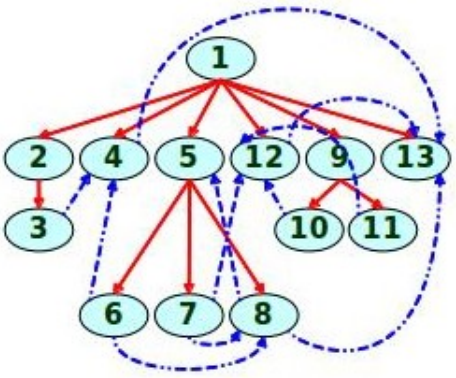
The goal of this exercise is to build an alternate algorithm for the computation of dominance frontier. The next exercise will then take the idea of this algorithm, and some of the results showed here, to build an algorithm of linear time complexity for placing ϕ-functions inside a CFG.

For a given CFG and two nodes x and y, we write:

- x **dom** y to indicate that x dominate y.
- x **!dom** y to indicate that x does not dominate y.
- x **sdom** y to indicate that x *strictly* dominate y (ie. x **dom** y and $x \neq y$).
- $x =$ **idom** (y) (or x **idom** y) to indicate that x is a parent of y in the dominance tree.
- $SubTree(x)$ is the set of nodes dominated by x.
- $x.level$ is the *level* of x inside the dominance tree.
- $DF(x)$ is the dominance frontier of x.

In this exercise, as in the next one, we will work on an extension of the CFG called DJ-graph. A *DG-graph* is build on the same set of nodes than the CFG, and possess two types of edges called D-edges and J-edges. A *D-edge* is an edge of the dominance tree, and a J-edge is an edge of the CFG that is not a D-edge. The following figure show a CFG with its associated DJ-graph. The D-edges are full while the J-edges are dashed:





We consider a CFG $G = (N, E)$ and its DJ-graph $DJ = (N_{dj}, E_{dj})$. The cardinality of a set X is written $|X|$.

Questions:

- Show that $|N_{dj}| = |N|$ and that $|E_{dj}| < |N| + |E|$.
- Let x be a node of the DJ-graph and $y \in DF(x)$. By using **idom** (y), show that $x.level \geq y.level$.
- Deduce that z est is in $DF(x)$ if and only if there exists $y \in SubTree(x)$ such as $y \rightarrow z$ is a J-edge and $z.level \leq x.level$. For the $\Leftarrow$ part, we distinguish between the case $z \in SubTree(x)$ and $z \notin SubTree(x)$.

Those remarks bring us to the following algorithm (`y -> z == J_edge` tests if there exists a J-edge from y to z):

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

Questions

- Use this algorithm to compute $DF(5)$ in the CFG given below. Detail the steps of the algorithm.
- With this algorithm, what is the worst case complexity of SSA transformation as shown in the lecture?

Exercise 3. (Bonus) *SSA form, efficiently*

This exercise proposes to study an algorithm for the placing of ϕ-functions that has a linear time complexity in the number of nodes of the CFG. We will use the results from the previous exercise.

We extend the definition of dominance frontier $DF(S)$ to a set of nodes S:

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

. We define the iterated dominance frontier $IDF(S)$ for a set of nodes S as the limit of the following growing sequence:

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

Question:

- Explain why the SSA transformation algorithm from the lecture is equivalent to compute multiple iterated dominance frontier.

As in the previous exercise, we will work on the DJ-graph. Our algorithm manipulate a data structure called *bank*. A bank is an array from which each cell contains a list of nodes of the DJ-graph. The odes are inserted such that the cell i contains only nodes of level i.

We consider a unique `bank` (global variable). The insertion operation, `InsertNode(n)` add the node `n` to the list `bank[n.level]`. The search/delete operation, `GetNode()` seach for the last non empty cell of `bank`, and return (then delete) its first node.

Each node is represented by a record:

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

Initially:

- n.visited = NotVisited, node.inphi = NotInPhi, n.initial = NotInitial,
- n.level = depth in the dominance tree

There is the algorithm that compute the iterated dominance frontier of a set S of nodes. The result can be directly used to place the ϕ-functions inside the CFG.

```
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);
    }
  }
}
```

Initially, we insert the nodes from S that we want the iterated dominance frontier. Then, for each node x by increasing level, we call Visit(x). Visit(x) do a depth-first processing of DJ, starting in x, but avoid exploring nodes that has already been explored from a previous call (see the `for each` and its second `if`).

Following the idea of the previous exercise, the first `if` collect the nodes in the dominance frontier of x. Those nodes are added in the result set `IDF`, then to the bank *via* `insertNode()`, to be processed in a later call to `Visit()`.

Questions:

- Detail the steps of the algorithm over $IDF(\{6, 9\})$ on the DJ-graph from the previous exercise.
- Explain why a node is always inserted (*via* `InsertNode()`) at a level $\leq$ CurrentRoot.level.
- Let x and y two nodes inserted inside the bank, then suppressed by GetNode(). Show that if $y.level > x.level$, then Visit(y) is called before Visit(x). We distinguish, at the deletion of x, between the case where y is in the bank, and the case where y is not.
- Show that when Visit(x) is called with $x = CurrentRoot$, every nodes of $DF(x)$ are in IDF. We take into consideration that $SubTree(x)$ is not always entirely processed.
- Show that when the algorithm ends, $IDF(S) =$ `IDF`. We can show the inclusion $\subset$ by induction over the definition of $IDF(S)$.
- Show that each node is processed (by a call to Visit()) at most one time. For that, we can show that a node cannot be processed in the while loop of IDF_main and in the second if of Visit().
- Deduce the time complexity of the algorithm.