CS243 Winter 2004 Midterm solutions

February 18, 2004

1 True / False

a. Applying constant propagation before lazy code motion can improve the applicability of lazy code motion

   Answer: **True**

   Example: see Figure 1

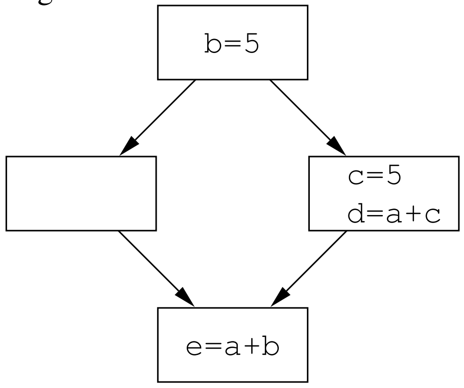

Figure 1: Applying constant propagation before lazy code motion can help

b. Applying lazy code motion before constant propagation can improve the applicability of constant propagation

   Answer: **True**

   Example: see Figure 2

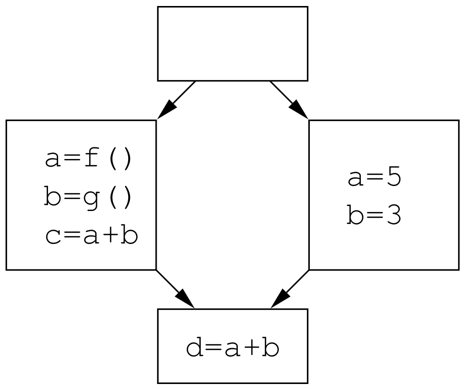

Figure 2: Applying lazy code motion before constant propagation can help

c. If reverse post-order is used, the data-flow algorithm for constant propagation in at most $d + 2$ iterations, where d is the loop depth of the input program.

   Answer: **False**

   As explained in class, the bound is only valid when cycles don't modify the flow of information (which is not the case for constant propagation).

d. A reverse control flow graph is like a control flow graph, except that the directions on the edges are reversed. That is, given a control flow graph $G =< N, E >$, $G' =< N, E' >$ is a reverse control flow graph of G if $(n_1, n_2) \in E$ iff $(n_2, n_1) \in E'$.

   True or false: G' is reducible if G is reducible.

   Answer: **False**

2 Natural loops and reducibility

a. Natural loops

   There are three back edges in the graph:

   $2 \rightarrow 1$, $2 \rightarrow 3$ and $4 \rightarrow 3$.

   The corresponding natural loops are respectively: $\{1, 2, 3, 4, 5\}$, $\{2, 3\}$, $\{2, 3, 4, 5\}$.

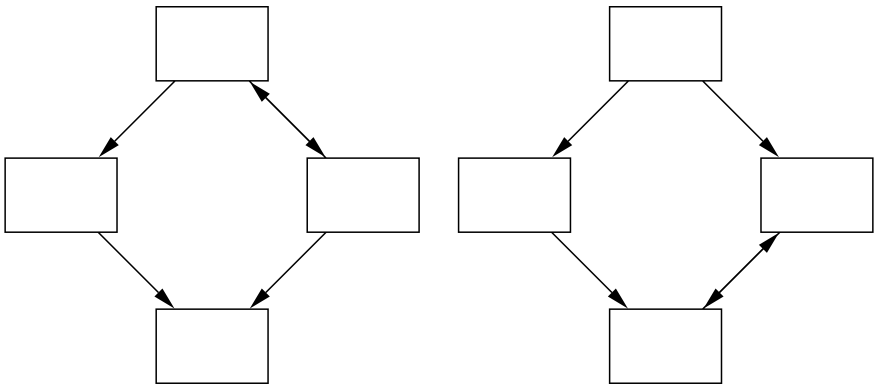

Figure 3: A reducible control flow graph and the corresponding non-reducible reverse control flow graph

b. Reducibility

   If we remove the back edges, there is still a cycle between 4 and 5, so the graph is not reducible.

3 Lazy code motion

see Figure 4.

4 Method inlining

This question had a few subtleties.

The idea here is to know for each variable all the possible classes it can be an instance of. Given this knowledge we may in some cases statically determine what method is invoke by a virtual call.

For a class $class$ we'll call $sub(class)$ the set of all subclasses (including $class$) of $class$ (we already know the class hierarchy).

For a given variable x of type $class$, the lattice will be all the possible subsets of $sub(class)$. It is a usual set lattice, with meet being $\cup$.

For all variables we'll just take the usual product lattice, producing a function F from variables the sets of subclasses of the corresponding delcared type.

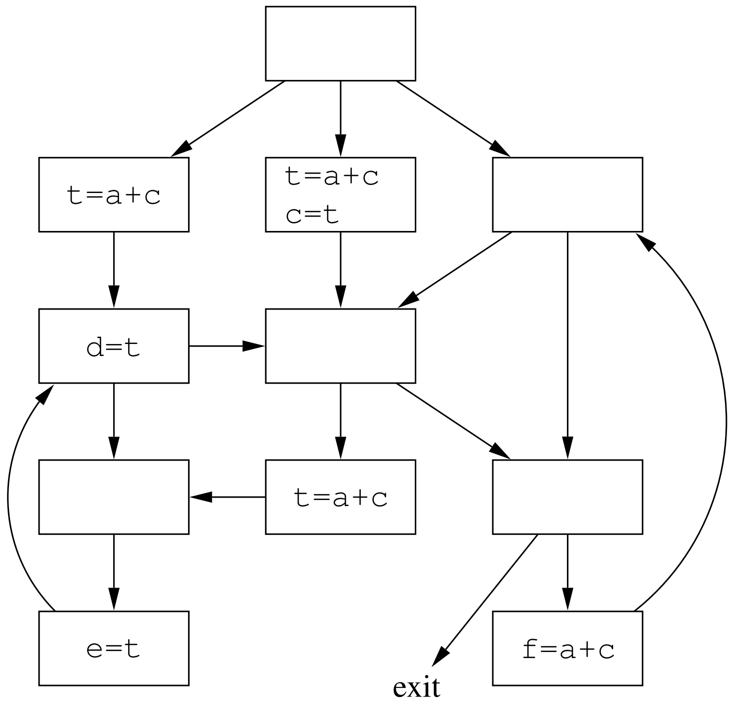

Figure 4: Result of lazy code motion

The initialization it $\top$, which maps variables to $\emptyset$. $\bot$ maps each variable a to $sub(class)$ where $class$ is the declared type of a.

The transfer function (let's call it T) is pretty simple, it includes the results of part *b* (which will be explained later).

- `a=b`: $T(F)(a) = F(b)$; for $x \neq a$, $T(F)(x) = F(x)$;
- `a=new C`: $T(F)(a) = \{C\}$; for $x \neq a$, $T(F)(x) = F(x)$;
- `a = b.method()`: $T(F)(a) = sub(c)$ where c is the return type of `method`. For $x \neq a$, $T(F)(x) = F(x)$;
- `a = b.field`: $T(F)(a) = sub(c)$ where c is the declared type of `field`. For $x \neq a$, $T(F)(x) = F(x)$;
- *All other cases*: $T(F) = F$.

This transfer function is monotone and distributive.

Convergence follows from the monotonicity and finiteness of the lattice.

Explanation of the `field` transfer: you can't keep track of a field as you would a variable. Since you get them from pointers, there may be aliasing, so when you modify one field you may very well modify the same field for another variable. For this reason we just give up and only take the declared type information.

Once we have the result of our analysis, we can do a static lookup for `a.n()` if there is only one definition of `n` in the set of possible classes for variable `a`.