Homework Assignment #3

Due Monday, March 6, at the start of lecture

1. Put the following program in SSA form (you may draw a control flow graph to illustrate your solution):

```
x := 0;
do {
   x := x + 1;
   z := x;
   y := 0;
   if (...) {
      y := 1;
   }
   w := y + z;
} while (...);
print(x, y, z, w);
```

2. Give an algorithm for constant propagation that exploits def/use chains to work faster than the propagation-based algorithm presented in class. What is the time complexity of your algorithm, assuming def/use chains are already constructed? How, if at all, would converting the program to SSA form before constructing def/use chains help your analysis?

3. Give an algorithm for dead assignment elimination that exploits def/use chains to work faster than the propagation-based algorithm that used live variables analysis presented in class. Your algorithm should not miss any optimization opportunities found by the best live variables-based algorithm presented in class. What is the time complexity of your algorithm, assuming def/use chains are already constructed? How, if at all, would converting the program to SSA form before constructing def/use chains help your analysis?

4. Consider integer range analysis, with the dataflow information at each program point taking the form of a map from each integer variable in scope at that point to a pair (*lo*,*hi*), where *lo* and *hi* are integer constants, with the meaning that at that program point, the value *v* of each integer variable satisfies $lo \le v \le hi$. You'd like to design an analysis that can prove statically that the assertions in the following examples are all true:

```
int i = 10;
while (i < 20) {
   assert(i >= 10 && i <= 19);
   i = i + 1;
}
```

```
int j = 20;
while (10 < j) {
   assert(j >= 11 && j <= 20);
   j = j - 1;
}
```

   You can assume that the simple flow functions for *X* `:=` *C* and *X* `:=` *Y* *OP* *Z* defined in class are part of your analysis.

   a. Define the *IRA* lattice formally, specifying at least the set of elements and the partial

ordering relation over them.

b. Define the flow function for compare-and-branch instructions of the form `if (`*`I`*` < `*`J`*`) goto `*`L1`*` else goto `*`L2`* where *I* and *J* are integer variables or constants. Such compare-and-branch instructions have one incoming control flow edge and two outgoing control flow edges, for the true- and false-outcome successors, respectively. Thus, you are to define the function $IRA_{\texttt{if (}I\texttt{ < }J\texttt{) goto }L1\texttt{ else goto }L2}(\text{in}) \rightarrow (\text{out}_{L1}, \text{out}_{L2})$. Exploit the different knowledge gained about the truth of *I*<*J* in each of the successor edges to refine the possible ranges of *I* and *J*.

c. Define the flow function for non-loop-head merges. For simplicity, assume that a merge is a node with exactly two incoming control flow edges and one outgoing control flow edge. Thus, you are to define the function $IRA_{\texttt{merge}}(\text{in}_1, \text{in}_2) \rightarrow \text{out}$.

d. Define the flow function for loop-head merges. For simplicity, assume that a loop-head merge is a node with exactly two incoming control flow edges (for the loop entry edge and the loop back edge, respectively) and one outgoing control flow edge. Thus, you are to define the function $IRA_{\texttt{loophead}}(\text{in}_{\text{entry}}, \text{in}_{\text{backedge}}) \rightarrow \text{out}$. Include sufficient widening to ensure that optimistic iterative analysis completes in a reasonable amount of time, but be sufficiently precise to identify that the assertions in the examples above are true.

e. Illustrate the results of running your analysis on the CFG derived from the C example above. Treat each assertion statement as a single CFG node that has no run-time effect. Show each step of iterative analysis separately.

f. Illustrate the results of running your analysis on the following example:

```
int j = 20;
do {
   assert(j >= 10 && j <= 20);
   j = j - 1;
} while (j >= 10);
```

Does your analysis succeed in proving the assertion true? If not, what sort of "loop normalization" step might you invoke to transform your program into an equivalent one in which the assertion can be proved true?

5. Consider extending your integer range analysis from the previous question to be interprocedural.

a. Explain what kind of information would be computed by a bottom-up callee summary vs. a top-down caller summary? What intraprocedural analysis information would be improved by each kind of summary?

b. Discuss the tradeoffs among versions of interprocedural integer range analysis that are (1) context-oblivious, (2) context-insensitive, (3) context-sensitive keyed by call strings, and (4) context-sensitive keyed by calling context. Explain how precise the results of the different versions are likely to be, and long different versions are likely to take to compute their results. Give an example program which illustrates the differences, at least in precision. Which approach would you pick for a practical compiler, if any?