1 Dominators

The notion of *dominance* is useful for many code analyses that operate on the flow graph. This questions asks you to define, compute, and exploit dominators.

1. Let d and n be nodes of a flow graph. Define the meaning of "d dominates n."

2. Let d *dom* n denote the relation "d dominates n." Which of the following properties does the *dom* relation have? Justify your answers. If it does not have a property, say why.

   - transitivity,
   - reflexivity,
   - symmetry,
   - anti-symmetry,
   - equivalence.

3. A straightforward representation of the *dom* relation requires N^2 space, where N is the number of nodes in the flow graph. Describe a more economical dominator representation that is commonly used in compilers. What are its space requirements?

4. Dominators are typically computed on the flow graph, using a dataflow-analysis-like algorithm, or more efficiently, using the Lengauer-Tarjan algorithm. For simple programs, the dominators can be computed even without constructing the flow graph, using a syntax-directed translation.

   Consider the simple imperative language defined below. (Assume the usual control-flow semantics for the language.)

$$
\begin{array}{rcl}
program & \rightarrow & cmd \\
cmd & \rightarrow & Id := exp \\
 & | & \text{while (} exp \text{) do } cmd \text{ od} \\
 & | & \text{if (} exp \text{) do } cmd \text{ od} \\
 & | & cmd; cmd \\
exp & \rightarrow & Id \\
 & | & Int
\end{array}
$$

   Give a syntax-directed *definition* (i.e., a system of attribute equations) that computes for each assignment the set of assignments that dominate it (in the flow graph of the program). Assume that assignments can be uniquely identified by the attribute $Id.num$ of their left-hand-side variable.

   Illustrate your answer using an example.

2 Multiple Inheritance

This question asks you to discuss the complications with implementing multiple inheritance. Discuss your answers in the context of a concrete programming language, e.g., C++ or "Extended" Java (you are free to select which language to use).

1. Using a small example program, illustrate the use of multiple inheritance.

2. What are the complications of implementing multiple inheritance?

3. Describe an object layout that supports multiple inheritance. Describe how your layout supports (a) member field access; and (b) virtual function call.

   Is the cost of these two operations higher than with the typical (i.e., single-inheritance) layout? Why?

3 Structural Induction

1. Consider the recursive data type *tree*, defined as follows:

   A *tree* is either *empty* or it is a *node* consisting of two subtrees and an integer.

   Now consider the two versions of function *sum*, given below. Both compute the sum of integers in the tree. Function *sum1* traverses the tree postorder; function *sum2* traverses the tree inorder.

   ```
   let rec sum1(T) =
      cases (T) of
      empty: 0
      node(T1, k, T2): sum1(T1) + k + sum1(T2)

   let rec sum2(T) = sum3(T, 0)

   let rec sum3(T, a) =
      cases (T) of
      empty: a
      node(T1, k, T2): sum3(T2, k + sum3(T1, a))
   ```

   Prove, using structural induction on T, that for all trees T, $sum1(T) = sum2(T)$. Be sure to justify each step of your proof.

2. Consider rewriting programs $sum1$ and $sum2$ into Java methods called `J1` and `J2`, shown below.

   ```
   public class Tree {
      public Tree left, right;
      public int  k;
   };
   public class Qual {
      ...
      static public int J1 (Tree T) {
         if (T == null) return 0;
         else return J1(T.left) + T.k + J1(T.right);
      }
      static public int J2 (Tree T) { return J3(T, 0); }
      static public int J3 (Tree T, int a) {
         if (T == null) return a;
         else return J3(T.right, T.k + J3(T.left, a));
      }
   }
   ```

   Discuss challenges you will face when applying structural induction to prove that `J1` and `J2` are equivalent Java methods. In particular, discuss what you will be able to prove about these two methods; and what program analysis and/or user annotations may be necessary to carry out the proof.

4 Memoization

This is a question about the memoizing optimization.

Discuss your answers in the context of a language that contains (i) procedures whose arguments are passed by value; (ii) global variables accessible from all procedures; and (iii) *no* pointers.

1. What is memoizing? How can it improve the execution speed of a program? Give an example of a *simple* program whose performance is significantly improved by memoizing it.

2. What are the tradeoffs in memoizing a program? How can you estimate whether memoizing will be effective or not?

3. A function is called *side-effect-free* if it does not write any global variables. A function is called *pure* if it neither reads nor writes any global variables.

   It is often required that the memoized (sub)program be a pure function. Why is this requirement imposed?

4. It is clear that some "impurities" are benign. For example, a side-effect-free procedure that reads a global variable only at the very beginning of the procedure can be easily converted into a pure procedure.

   Suggest how a side-effect-free procedure can be analyzed to determine if parts of it can be memoized. In particular,

   (a) Define dataflow analysis that will compute the set of *pure* instructions in a procedure. An instruction is pure if its arguments depend only on the values of procedure arguments.

   Give the lattice of dataflow facts, its meet operator, and the flow functions.

   Is your dataflow problem a bitvector problem (i.e., is it *separable*)?.

   (b) Show how the result of the dataflow analysis can be used to find a region in the procedure that is memoizable.

5 Low-Level Optimizations

The quality of compiler optimizations is influenced both by the language design (i.e., the interface between the programmer and the compiler) and also by the target instruction set (i.e., the interface between the program and the computer). This question discusses the impact of the latter.

Some modern instruction-set architectures provide support for run-time memory disambiguation. For example, in the Transmeta Crusoe processors, the following two instructions are available:

- *load-and-protect:* The `ldp` x, y instruction
  1. loads into the register x the value from the address in register y, and also
  2. records the address and size of the data loaded in a finite-size table.
- *store-under-alias-mask:* The `stam` x, y instruction
  1. stores into the address in register y the content of register x, and
  2. if the store overwrites previously loaded data, an exception is raised and the run-time system takes *corrective action*.

The ldp/stam pair may be used, among other things, to enable more aggressive instruction scheduling, by allowing scheduling loads prior to stores.

1. The descriptions of `ldp` and `stam` omit some details. Describe what other instructions may be needed and/or what other information must be stored in hardware tables to make the run-time memory disambiguation disambiguation work.

2. Describe how the ldp/stam pair can be used to implement removal of redundant loads and redundant expressions (i.e., redundant sequences of instructions). Illustrate the optimization on the following code. Also show the corrective-action code.

   ```
   ld  x, y
   add x, x, 1      // x = x + 1
   st  z, w
   ld  v, y
   add v, v, 1      // v = v + 1
   ```

   If you need to make assumptions on how the corrective-action code behaves, use those you wrote down in Problem 1 above. Note that `ld` and `st` are the usual load and store operations.

3. The run-time disambiguation instructions must be used conservatively. If the run-time corrective action is taken too frequently, the optimization may have no benefit. Describe some practical methods for deciding when to use the ldp/stam instructions.

4. The important benefit of the ldp/stam pair is that it decreases the dependence of the optimizer on pointer analysis. Describe some optimization that requires pointer analysis but is *unlikely* to benefit from the ldp/stam instructions.

6 SSA

1. What is Static Single Assignment (SSA) form?

   Explain how a program involving only scalar variables can be placed in SSA form. Illustrate your technique on the following simple program.

```
a = b + 1;
while (a > 0) {
 if (b > 0)
      a = a+b;
 else a = a-b;
}
print(a,b);
```

2. One of the advantages of SSA form is that it greatly simplifies the analysis necessary to implement various optimizations.

   Explain what *copy propagation* is. If a program is *not* in SSA form, what analyses are necessary to determine whether copy propagation may be applied.

   If a program *is* in SSA form, how are the rules of validity for copy propagation simplified? Illustrate your answer using the following code fragment, copy propagating the initial assignment to a:

```
a=b;
if (a>0)
     a=a+1;
else b=a+1;
c=a;
```