Assignment 6 (Written/Programming): Dataflow Analysis

17-654/17-754: Analysis of Software Artifacts
Jonathan Aldrich (`jonathan.aldrich@cs.cmu.edu`)

Due: Tuesday, March 6, 2007 (10:30 am)

100 points total

Turn in a file named <username>-17654-A6.{zip}, where username is your Andrew id. The zip file should contain the file answers.xxx (the answers to text questions in txt, pdf, or Word (doc) format), each of the .java files you wrote, and output.xxx (either analysis output in .txt format or a screenshot in some common graphics format). At the top of answers.txt, state your name, Andrew id, and how long you spent on the assignment.

Assignment Objectives:

- Precisely define two analyses using lattices, abstraction functions, and flow functions.
- Derive the control flow graph of a program.
- Simulate analysis execution on a program using the worklist algorithm.
- Implement a dataflow analysis in a code framework built based on the concepts of flow functions and lattices.

1 Lock Analysis (35 points)

After graduating from CMU, you have been hired by CrystalSoft, a (fictional) company devoted to bringing the benefits of the Crystal Java analysis tools to C programmers. Your first task is to get a simple lock analysis up and running.

You quickly observe that C presents a harder problem than Java, because there's no built-in synchronization statement. Thus it's easy to make simple errors that can't happen in Java, like locking a lock when you enter a function and forgetting to unlock it when you return from that function. Your first task, therefore, is to design an analysis that can detect simple errors like deadlock, which will occur if the programmer tries to lock the same lock twice. For this assignment, you need only consider one thread running at a time–believe it or not, double-locking errors due to a single thread that forgets to unlock a lock have been found in the Linux kernel, causing the system to hang.

You study the problem first in the context of the WHILE language. You model locks with two new kinds of statements:

- lock(x) locks the variable x
- unlock(x) unlocks the variable x

For the purposes of data flow analysis over the control flow graph, you can assume these statements turn into similar three address code statements, lock(x) and unlock(x). You decide you will base your analysis on a tuple lattice, with one element of the tuple for each lock variable in the program.

Question 1.1 (10 points).

> Design a lattice for a single variable. Your lattice should be able to represent both locked and unlocked states. Define the lattice by giving (a) the set of lattice elements and (b) the ordering relation between them, (c) the top element and (d) the bottom element.

Question 1.2 (4 points).

> What is the initial analysis information before the first statement of each function? Justify your choice (more than one answer may be correct, so long as it is justified).

Question 1.3 (5 points).

Define the flow functions for your analysis, using the notation given in class. Naturally, you will need to include flow functions for the new `lock(x)` and `unlock(x)` statements.

Question 1.4 (8 points).

Assume you had an implementation of your lock analysis (from above). Explain how you would identify deadlock errors based on the results of this analysis. A deadlock occurs when a program locks a variable that is already locked. Specifically, describe (a) the While AST element that is associated with the error, (b) what condition on the analysis information at that location indicates a definite deadlock error, and (c) whether your condition is based on the analysis information immediately before or after the AST element. Finally, (d) explain what you would change about the condition to find a *possible* deadlock error (e.g. in cases where the analysis is too imprecise to tell if there is definitely an error).

There is a corresponding double-unlocking error that could be found, but finding it is not required for this assignment.

Question 1.5 (8 points).

> Simulate your analysis on the following program, using the worklist algorithm. Use the notation from the lecture 12, slide 46 ("Example of Worklist"), so you have 4 columns: the first describing to which statement you are applying the flow function (with 0 at the beginning to show the dataflow values at the entry of the CFG), the second column showing the statements on the worklist, and the last two columns showing lock lattice values for each variable (x and y—the variable b is not relevant since it is not a lock).

[lock(x)]$_1$;
if [$b > 0$]$_2$
  then [lock(y)]$_3$
  else [skip]$_4$;
[lock(x)]$_5$;
if [$b > 0$]$_6$
  then [unlock(y)]$_7$
  else [skip]$_8$;
[unlock(x)]$_9$;

2 Valid Pointer Analysis Specification (25 points)

Now it is time to precisely define a valid pointer analysis for WHILE. A valid pointer is a pointer which it is safe to dereference (i.e. it is not null and does not point to garbage). Assume that WHILE has been extended with the following syntactic construct:

$$\begin{array}{rcl} a & ::= & \ldots \\ & | & \&x \end{array}$$

The $\&x$ expression takes the address of some variable x, resulting in a valid pointer. For the present, we will assume WHILE has a C-like pointer model, where regular integer variables can hold pointer values. We represent null with the constant 0. We also assume that integer constants–including zero–are not valid pointers. For the purposes of data flow analysis over the control flow graph, you can assume that this new expression type turns into a three address code statement of the form $y = \&x$.

Question 2.1 (10 points).

Design a lattice for a single variable. Your lattice should be able to represent both definitely valid and definitely not valid states, as well as possibly valid. Define the lattice by giving (a) the set of lattice elements and (b) the ordering relation between them, (c) the top element and (d) the bottom element.

Question 2.2 (5 points).

What is the initial analysis information before the first statement of each function? Justify your choice (more than one answer may be correct, so long as it is justified).

Question 2.3 (10 points).

Define the flow functions for your analysis, using the notation given in class. Naturally, you will need to include flow functions for the new expression form $\&x$, as well as for integer constants such as 0 (which are invalid pointers).

3 Valid Pointer Analysis Implementation (40 points)

Next, you will implement your valid pointer analysis for the Java programming language using Crystal's dataflow analysis capability. (You will need to download an updated Crystal release from Blackboard.)

In Java, integer variables are separate from pointer variables ("references"), so your implementation need not keep track of whether integer variables and expressions are valid or not (you may if you wish; if so, these are all invalid). However, pointer variables are either valid or `null` (with `null` taking the place of 0 in WHILE). Thus, your analysis should track the null-ness of local variables, but should assume that values in fields, arrays, and method parameters could be either null or non-null.

You should implement your analysis by defining a new subclass of LatticeElement which describes a valid pointer lattice for a single variable, and using this with the TupleLatticeElement class to build a tuple lattice for valid pointer analysis. You should subclass AbstractTransferFunction to define your pointer analysis flow function as well as the initial and bottom analysis info. Finally, you should write a subclass of AbstractCrystalMethodAnalysis that runs Crystal's FlowAnalysis class with your transfer function and reports invalid pointer dereferences. Your analysis should produce exactly one warning in the Eclipse errors window for each of the errors marked with comments in the test file TestNull.java and no additional warnings. *Hint:* Valid pointer analysis works a lot like the zero analysis presented in class, so you may find it helpful to look at the zero analysis sample implementation that comes with Crystal.

AbstractTransferFunction allows you define your flow function over a form of *3-Address-Code* (3AC) that represents the (far more complicated) Java AST in terms of a small set of atomic operations on variables. 3AC looks a lot like assembly instructions and is implemented in `edu.cmu.cs.crystal.tac`. You will have to familiarize yourself with the various 3AC instructions that appear as arguments to the `transfer` methods in AbstractTransferFunction. Notice that there are several types of variables (some of which can never be `null`) that are represented as subclasses of `Variable`.

To make the assignment more feasible to grade, all the classes you write must be put in the package edu.cmu.cs.crystal.asst6. The client analysis class (the one that inherits from AbstractCrystalMethodAnalysis and needs to be registered in CrystalPlugin) must be named NullPointerAnalysis.java.

Question 3.1 (10 points).

Run your analysis on TestNull.java. Turn in a screenshot of the problems window, or the text produced by your analysis if you wrote to the errors window. When capturing the screenshot, resize the window if necessary to show all the errors. Do **not** change TestNull.java.

Question 3.2 (30 points).

Turn in your analysis code. Your code should follow the basic design described above. Remember to use package edu.cmu.cs.crystal.asst6 and client class name NullPointer-Analysis.java.

Important note: Your code must be robust. For example, it should not throw unexpected exceptions when analyzing valid Java code (these exceptions will show up on the Crystal console). We will be running your code on a large codebase to check its robustness, and we recommend you do so as well. One simple approach is to run your analysis on the Crystal codebase.