Assignment 6 (Programming): Dataflow Analysis

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

Due: Thursday, March 23, 2006 (5:00 pm)

100 points total

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

Assignment Objective:

- Implement a more realistic dataflow analysis that tracks typestate in a manner similar to Fugue (though still greatly simplified).

1 Protocol Analysis Implementation (100 points)

In this assignment, you will implement a protocol analysis similar to the one in Fugue.

We have designed an annotation system that describes a protocol of interaction. This annotation system is illustrated by the file SimpleProtocolTest.java. The annotation @States is advisory only; it declares the intended set of states that a given class can be in. While a real implementation of an analysis would verify state change specifications for consistency with this annotation, this is not required in this homework.

The annotation @Creates can appear on a constructor; it states that the object begins life in the specified state when that constructor is called. The annotation @ChangesState specifies that the method requires the receiver

object to be in the from state before the call, and that the receiver will be in the to state after the call. The annotation @InState specifies that the method requires the receiver to be in the given state when the method is called, and it leaves the receiver in that state. If no annotation is given for a method, we assume that method can be called from any state, and that it does not change the state of the reciever.

The annotations described above are defined in the package edu.cmu.cs.crystal2.asst6; you will need to include this package whenever you are testing your analysis in a child Eclipse window.

The analysis you write should track the current state of every variable in the program. Of course, it is possible that due to control flow merges your analysis may not be able to determine what state a variable is in; your analysis should represent this as well.

Your analysis should report a warning whenever a method is called, and the method requires the receiver to be in a certain state, and your analysis cannot tell for sure that the receiver is indeed in that state.

You should assume there is no aliasing between variables. This is, of course, unsound. In fact, it can cause your checker to have both false positives and false negatives in the case when there really is aliasing. However, you are not required to handle the case of aliasing (except for the extra credit, described below).

You should assume that any objects returned from a method or read from a field are in an unknown state. A more sophisticated system would use annotations or interprocedural analysis to make more realistic assumptions, but this is not required for this assignment. You should also assume that the arguments of a method are in an unknown state, when checking the method.

You should, however, track assignments between variables. Specifically, if you see an assignment x = y, you should assume that x is now in the same state as y was before the assignment. Again, you do not have to track the fact that x and y are now aliased and change state together, unless you do the extra credit.

Once you report an error, you are not required to accurately report the state of the variable involved in the error, for control flow after the error. All of the test code ends immediately after an error is detected.

Your implementation should be clean and use the lattice infrastructure of Crystal. You should use a flow analysis to determine what state variables are in, and a tree walker analysis to report errors when the state of a variable does not match the required state of a method. Your treewalker analysis must be called ProtocolAnalysis.java, and all your files must be in

package edu.cmu.cs.crystal2.asst6.

It is possible to structure your analysis in a very similar way to ZeroAnalysis, using a tuple lattice. This is highly recommended. One difference vs. ZeroAnalysis, of course, is that you may have several different states, not just zero or non-zero.

There is no reason your implementation has to be long; the instructor's implementation is about 250 lines of code, a substantial portion of which was copied from ZeroAnalysis.

We have provided a two helper functions in ProtocolUtilities.java. These helper functions take care of figuring out what is the required pre-state and post-state for a method or constructor declaration (as returned by MethodInvocation.resolveMethodBinding() or ClassInstanceCreation.resolveConstructorBinding()).

To help you test your analysis, we have provided the test file SimpleProtocolTest.java. Your analysis should report no errors in the correctUsage function. Your analysis should report exactly one error for each of the 6 badUsage functions. There is no requirement to report or not report errors in the extraCreditFunWithAliasing function, unless you are doing the extra credit.

Question 1.1 (20 points).

> Run your analysis on SimpleProtocolTest.java. Turn in a screenshot of the problems window, or the text produced by your analysis if you wrote to the user console window. When capturing the screenshot, resize the window if necessary to show all the errors.

Question 1.2 (60 points).

> Turn in your analysis code. Your code should follow the basic design described above. Remember to use package edu.cmu.cs.crystal2.asst6 and client class name ProtocolAnalysis.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.

Question 1.3 (20 points).

Run your analysis on SocketProtocolTest.java. To get this to work, you need to put the special annotated version of Socket.java in your project. This version is the same as Socket.java in JDK 1.5.0, except that protocol annotations have been added and a few lines have been commented out to work around Crystal bugs(!).

Turn in a screenshot of the problems window, or the text produced by your analysis if you wrote to the user console window. When capturing the screenshot, resize the window if necessary to show all the errors.

Question 1.4 (EXTRA CREDIT).

For up to 20 points of extra credit, extend your analysis to track aliasing through local variables. You can track aliasing as part of your main Protocol lattice, or you can write a completely separate analysis that tracks aliasing and use the results of this analysis in your protocol analysis.

For full credit, your alias analysis should not report any errors for the extraCreditFunWithAliasing function.

If you are doing the extra credit, you do not need to turn in a separate set of files for with and without the extension. Just turn in one set of files that does both.

Question 1.5 (EXTRA CREDIT).

For another extra credit opportunity, use your analysis to find a protocol error in open source software. To get credit, you must:

- Give the name of the software
- Give the location where you found the software, which must be publically accessible on the internet. The software must not be written by you, nor can it be written specifically for this assignment by anyone else at your request.
- You must turn in the source code, with your added protocol annotations

- You must show the analysis results through an output file
- You must briefly describe the protocol you are checking. The protocol must be real, and not made up by you: either the protocol should be documented in some way in code comments, or you should be able to point to an error (such as an exception thrown) that will occur if the protocol is not followed.
- The error you find must be real, i.e. not a spurious warning that appears because your analysis does not track aliasing or is otherwise too imprecise

The number of points given will vary depending on the error and the software. The minimum is 20 points; the maximum is 40 points for interesting errors found in widely used software.

If you can convince the developer of the software that you have found a real bug, and the developer checks in changes as a result of your bug report, you will get up to 100 points of extra credit. You can provide evidence of this check-in at any time during the semester.

Hint: one source of legitimate errors that is easy to find is resource cleanup errors, such as when you forget to close a file. Finding resource cleanup errors is not required by the assignment, but you can extend your analysis to find these by the extra credit, and it may make finding bugs in software easier (a lot of people don't close files, etc.) To do this, define a "last state" somehow (e.g. the last state in the @States annotation) and ensure that at all return statements from a method, every object is in its "last state"

Note, however, that it may be hard to convince a developer to check in changes because they forgot to close a file; this is not a big deal in most applications (although it can cause problems in some cases). So you may not get 100 points through this route, but 20-40 is probably not too hard.

Collaboration Note. Finding errors in open source apps can be a lot of work. So, ***for extra credit problem 1.5 ONLY (not 1.4) you may work in pairs*** (using the analysis written by either member of the pair–you must each complete your solution before you begin to collaborate). Also, ***only the first pair to find a given***

bug in a given application will get credit for that bug. This is to ensure that the pair who identified the application and the bug gets rewarded for their work. To determine who is first, send the instructor an email as soon as you find a bug, with the project name and file where the bug is found.

Note: the handling of extra credit in this course will be as follows. The instructor will compute all grades without extra credit and decide on boundaries between letter grades based on this information only. Then extra credit will be added and will be used to increase the scores of individuals, perhaps pushing them to a higher letter grade. This methodology is intended to ensure that no-one is penalized for not doing extra credit; it truly is optional.